Lucent Technologies
Bell Labs Innovations

5ESS® Switch Software Analysis Guide

5E13 and Later Software Releases

235-600-510
Issue 5.00B
June 2001

Notice

Every effort was made to ensure that the information in this information product was complete and accurate at the time of publication. However, information is subject to change.

This information product describes certain hardware, software, features, and capabilities of Lucent Technologies products. This information product is for information purposes; therefore, caution is advised that this information product may differ from any configuration currently installed.

Mandatory Customer Information

Interference Information: Part 15 of FCC Rules - Refer to the *5ESS*® Switch Product Specification information product.

Trademarks

5ESS is a registered trademark of Lucent Technologies in the United States and other countries
AnyMedia is a registered trademark of Lucent Technologies in the United States and other countries
Common Language is a registered trademark, and CLEI, CLLI, CLCI, and CLFI are trademarks of Bell Communications Research, Inc.
DATAKIT is a registered trademark of Lucent Technologies in the United States and other countries
INTEL is a registered trademark of Intel Corporation
Motorola is a registered trademark of Motorola, Inc.
OneLink Manager is a trademark of Lucent Technologies in the United States and other countries
PowerPC is a trademark of International Business Machines Corporation
SLC is a registered trademark of Lucent Technologies in the United States and other countries
UNIX is a registered trademark of The Open Group in the United States and other countries.

Limited Warranty

Warranty information applicable to the *5ESS*® switch may be obtained from the Lucent Technologies Account Management organization. Customer-modified hardware and/or software is not covered by this warranty.

Ordering Information

This information product is distributed by the Lucent Technologies Customer Information Center in Indianapolis, Indiana.

The order number for this information product is 235-600-510. To order, call the following:

1-888-LUCENT8 (1-888-582-3688) or fax to 1-800-566-9568 (from inside the continental United States)

+1-317-322-6847 or fax to +1-317-322-6699 (from outside the continental United States).

Support Telephone Numbers

Information Product Support Telephone Number: To report errors or ask nontechnical questions about this or other information products produced by Lucent Technologies, call 1-800-645-6759.

Technical Support Telephone Numbers: For initial technical assistance, call the North American Regional Technical Assistance Center (NARTAC) at 1-800-225-RTAC (1-800-225-7822). For further assistance, call the Customer Technical Assistance Management (CTAM) center as follows:

1-800-225-4672 (from inside the continental United States)

+1-630-224-4672 (from outside the continental United States).

Both centers are staffed 24 hours a day, 7 days a week.

Acknowledgment

Developed by Lucent Technologies Customer Training and Information Products.

Lucent Technologies values your comments!

Lucent Technologies
Bell Labs Innovations

5ESS® Switch Software Analysis Guide 5E13 and Later Software Releases

235-600-510 5.00B June 2001

Lucent Technologies welcomes your comments on this information product. Your opinion is of great value and helps us to improve.

1. Was the information product:

	Yes	*No*	*Not applicable*
In the language of your choice?	☐	☐	☐
In the desired media (paper, CD-ROM, etc.)?	☐	☐	☐
Available when you needed it?	☐	☐	☐

Please provide any additional comments:

_____

_____

2. Please rate the effectiveness of this information product:

	Excellent	*More than satisfactory*	*Satisfactory*	*Less than satisfactory*	*Unsatisfactory*	*Not applicable*
Ease of use	☐	☐	☐	☐	☐	☐
Level of detail	☐	☐	☐	☐	☐	☐
Readability and clarity	☐	☐	☐	☐	☐	☐
Organization	☐	☐	☐	☐	☐	☐
Completeness	☐	☐	☐	☐	☐	☐
Technical accuracy	☐	☐	☐	☐	☐	☐
Quality of translation	☐	☐	☐	☐	☐	☐
Appearance	☐	☐	☐	☐	☐	☐

If your response to any of the above questions is "*Less than satisfactory*" or "*Unsatisfactory,*" please explain your rating.

_____

_____

3. If you could change one thing about this information product, what would it be?

_____

_____

4. Please write any other comments about this information product:

_____

_____

Please complete the following if we may contact you for clarification or to address your concerns:

Name: _____ Date: _____

Company/organization: _____ Telephone number: _____

Address: _____

Email address: _____ Job function: _____

If you choose to complete this form online, go to http://www.lucent-info.com/comments
Otherwise fax to 407 767 2760 (U.S.) or +1 407 767 2760 (outside the U.S.) or email comments to ctiphotline@lucent.com

Lucent Technologies values your comments!

Lucent Technologies
Bell Labs Innovations

5ESS® Switch Software Analysis Guide 5E13 and Later Software Releases

235-600-510 5.00B June 2001

Lucent Technologies welcomes your comments on this information product. Your opinion is of great value and helps us to improve.

1. Was the information product:

	Yes	*No*	*Not applicable*
In the language of your choice?	☐	☐	☐
In the desired media (paper, CD-ROM, etc.)?	☐	☐	☐
Available when you needed it?	☐	☐	☐

Please provide any additional comments:

_____

_____

2. Please rate the effectiveness of this information product:

	Excellent	*More than satisfactory*	*Satisfactory*	*Less than satisfactory*	*Unsatisfactory*	*Not applicable*
Ease of use	☐	☐	☐	☐	☐	☐
Level of detail	☐	☐	☐	☐	☐	☐
Readability and clarity	☐	☐	☐	☐	☐	☐
Organization	☐	☐	☐	☐	☐	☐
Completeness	☐	☐	☐	☐	☐	☐
Technical accuracy	☐	☐	☐	☐	☐	☐
Quality of translation	☐	☐	☐	☐	☐	☐
Appearance	☐	☐	☐	☐	☐	☐

If your response to any of the above questions is "*Less than satisfactory*" or "*Unsatisfactory,*" please explain your rating.

_____

_____

3. If you could change one thing about this information product, what would it be?

_____

_____

4. Please write any other comments about this information product:

_____

_____

Please complete the following if we may contact you for clarification or to address your concerns:

Name: _____ Date: _____

Company/organization: _____ Telephone number: _____

Address: _____

Email address: _____ Job function: _____

If you choose to complete this form online, go to http://www.lucent-info.com/comments
Otherwise fax to 407 767 2760 (U.S.) or +1 407 767 2760 (outside the U.S.) or email comments to ctiphotline@lucent.com

Software Analysis Guide

CONTENTS PAGE

1. INTRODUCTION 1-1

2. USING THE PROGRAM LISTINGS 2-1

3. C PROGRAMMING LANGUAGE 3-1

4. DISASSEMBLY/ASSEMBLY LANGUAGE 4-1

5. ASSERT ANALYSIS 5-1

6. GENERIC ACCESS PACKAGE (GRASP)/ENHANCED GRASP 6-1

7. GENERIC UTILITIES. 7-1

8. INTERRUPT ANALYSIS 8-1

9. SINGLE PROCESS PURGE (SPP) 9-1

10. AUDIT ANALYSIS. 10-1

11. DATA COLLECTION AND ANALYSIS 11-1

12. OSDS MONITOR 12-1

13. OSDS OVERLOAD MONITOR 13-1

A1. ENVIRONMENT TO PATHNAME CROSS REFERENCE A1-1

A2. IS25, 3B20, AND 3B21 COMPUTER INSTRUCTION LIST (BY MNEMONIC). A2-1

A3. *Motorola* MC68000 PROCESSOR FAMILY INSTRUCTION SET A3-1

A4. *Intel* 8086 AND 80186 PROCESSOR INSTRUCTION SET. A4-1

A5. *PowerPC* PROCESSOR FAMILY INSTRUCTION LIST (BY MNEMONIC). A5-1

A6. SWITCHING MODULE ASSERT ANALYSIS EXAMPLE AUDIT TRAIL A6-1

GLOSSARY G-1

INDEX I-1

Software Analysis Guide

CONTENTS PAGE

1. INTRODUCTION 1-1
1.1 PURPOSE 1-1
1.2 UPDATE INFORMATION 1-1
1.2.1 General 1-1
1.2.2 Supported Software Releases 1-1
1.2.3 Terminology 1-1
1.3 ORGANIZATION 1-3
1.4 USER FEEDBACK 1-4
1.5 DISTRIBUTION 1-5
1.6 TECHNICAL ASSISTANCE 1-5
1.7 REFERENCES 1-5

1. INTRODUCTION

1.1 PURPOSE

This document contains information and directions for using tools to analyze software-related problems within the *5ESS*® switch. It deals with administrative module (AM), communications module (CM) and switching module (SM) software problems, such as asserts, audits and interrupts, as well as program language usage and program code documentation.

The content assumes the user can read and understand assembly and C programming languages pertaining to the *5ESS* switch. The *Software Analysis Guide* should be used as a problem solving tool. Each component can be used independently to research or investigate a point of interest, or the document can be read from beginning to end as one might read a book. The intention is to facilitate the identification and resolution of problems within the *5ESS* switch.

Use of this guide will help the user become familiar with the structure and content of the program code, the organization, and use of error messages as well as understand the interactive languages which permit the human interface with the switch.

1.2 UPDATE INFORMATION

1.2.1 General

As the *5ESS* switch evolves with new capabilities and features, this document will be updated. This issue includes an updated procedure for accessing the online program listings, Section 2.2.

1.2.2 Supported Software Releases

In accordance with the *5ESS* Switch Software Support Plan, the 5E12 software release is rated Discontinued Availability (DA) as of September, 2000. The information supporting 5E12 and earlier is being removed over time, instead of concurrently, from all documentation.

If you are supporting offices that use a software release prior to 5E13 and you have a need for the information that is being removed, retain the associated pages as they are removed from the paper documents, or retain the earlier copy of the CD-ROM.

1.2.3 Terminology

1.2.3.1 Lucent Electronic Delivery

The Lucent Electronic Delivery system is replacing the Software Change Administration and Notification System (SCANS) as the system used to download software changes to Lucent products. During the transition, both systems will be supported. When products no longer require SCANS, Lucent Technologies will notify any customers still using SCANS of the plans for completing the migration to Lucent Electronic Delivery. The **OneLink Manager**™ *ASM User's Guide*, 235-200-145, describes the Lucent Electronic Delivery system. Documentation currently referencing SCANS will be changed over time, as other technical changes are required.

1.2.3.2 Communication Module Name Change

The term Communication Module (CM) has been changed to the Global Messaging Server (GMS), representing the new portfolio name of this particular module. The current names of the specific types of the GMS (the CM2 and CM3) have not been changed. Where the CM name has been used in a generic way within this information product, the name will be changed to GMS. Where the specific version of GMS (CM2

or CM3) is being described or mentioned, the name will not be changed. However, the GMS name may be added to the description in certain places as a reminder of the change, and that the particular version is a part of the overall portfolio. The following list provides some examples of how you may see these names used together:

- Global Messaging Server (formerly Communication Module)
- GMS (formerly CM) Global Messaging Server-CM2
- GMS-CM2
- Global Messaging Server-CM3
- GMS-CM3.

These name changes will be made over time as other technical changes are required. Also these changes may not be reflected in all software interfaces (input and output messages, master control center screens, and recent change and verify screens). Where the information product references these areas, the names are used as they are within the software interface.

1.2.3.3 Bellcore/Telcordia Name Change

As of March 18, 1999, Bellcore officially changed its name to Telcordia Technologies. Not all pages of this document are being reissued to reflect this change; instead, the pages will be reissued over time, as technical and other changes are required. Customers on standing order for this document may see that, on previous-issue pages, the Bellcore name is still exclusively used.

Customers receiving new orders for this document will see the Telcordia Technologies name used as appropriate throughout the document, and the Bellcore name used only to identify items that were produced under the Bellcore name. Exceptions may exist in software-influenced elements such as input/output messages, master control center screens, and recent change/verify screens. These elements will not be changed in this document until such time as they are changed in the software code. Document updates will not be made specifically to remove historical references to Bellcore.

1.2.3.4 *5ESS*-2000 Switch Name Change

This *5ESS* switch document may contain references to the 5ESS switch, the *5ESS*-2000 switch, and the *5ESS AnyMedia*® Switch. The official name of the product has been changed back to the *5ESS* switch. In the interim, assume that any reference to the *5ESS*-2000 switch or the *5ESS AnyMedia* Switch is also applicable to the *5ESS* switch. It should be noted that this name change may not have been carried forward into software-influenced items such as input and output messages, master control center screens, and recent change/verify screens.

1.2.3.5 National ISDN

National ISDN is an evolving platform in which new features will continue to be introduced for new revenue opportunities, improved operational efficiencies, and for support of specific applications. NI 1, NI 2, and NI 3 represent specific features as documented in Bellcore SRs 1937, 2120, and 2457. The industry is migrating to an additional terminology to more specifically denote the availability of National ISDN features: NI 95, NI 96, etc. A feature is included in a specific version (such as, NI 96) if it is available by the switch vendors by the first quarter of the year.

1.3 ORGANIZATION

The content and organization of the *Software Analysis Guide* is as follows:

- **Introduction**

  Section 1 describes the content and organization of this document. It also describes the reasons for updates and how to obtain assistance.

- **Using the Program Listings**

  Section 2 explains the structure and use of on-line program listings.

- **C Programming Language**

  Section 3 reviews the C programming language and its use in the *5ESS* switch.

- **Disassembly/Assembly Language**

  Section 4 reviews the disassembly/assembly language for the 3B20D and 3B21D computers, *MOTOROLA*[1] MC68000 microprocessor family, and the *Intel*[2] 80186 microprocessor and discusses their use in the *5ESS* switch.

- **Assert Analysis**

  Section 5 explains the function of asserts, the key parts of the printout associated with them, and how to analyze them.

- **Generic Access Package (GRASP)/Enhanced Generic Access Package (EGRASP)**

  Section 6 discusses the use of the interactive language used in the AM.

- **Generic Utilities**

  Section 7 explains the use of the interactive language used in the SMs, the CM, and the peripherals.

- **Interrupt Analysis**

  Section 8 examines the function of an interrupt and discusses the output sent to the receive only printer (ROP).

- **Single Process Purge Analysis**

  Section 9 analyzes the function of a single process purge (SPP) and discusses the output received on the ROP.

- **Audit Analysis**

  Section 10 discusses audits and the messages received on the ROP.

- **Data Collection and Analysis**

  Section 11 discusses the types of data and the manner of data collection necessary to analyze a trouble event so that a complete picture is compiled, particularly an event history.

- **OSDS Monitor**

---

1. Registered trademark of Motorola Inc.
2. Registered trademark of Intel Corporation.

Section 12 explains how the operating system for distributed switching (OSDS) monitor tool is used for gathering performance data and investigating performance problems.

- **OSDS Overload Monitor**

  Section 13 explains how the OSDS overload monitor tool is used for investigating OSDS resource overload problems.

- **Supplementary Information**

  The supplementary sections contain various reference materials, including:

  - **Appendix 1: Environment to Pathname Cross Reference**

    Appendix A1 is provided for use with `upd:ftrc` to find a failing function.

  - **Appendix 2: IS25, 3B20D, 3B21D Computer Instruction Set**

    Appendix A2 is a list of the IS25, 3B20D, and 3B21D Computer opcodes and their functions.

  - **Appendix 3: *MOTOROLA* MC68XXX Microprocessor Family Instruction Set**

    Appendix A3 is a list of the *MOTOROLA* MC68XXX microprocessor family opcodes and their functions.

  - **Appendix 4: *Intel* 8086 and 80186 Processor Instruction Set**

    Appendix A4 is a list of the *Intel* 8086 and 80186 processor opcodes and their functions.

  - **Appendix 5: *PowerPC*[3] Processor Family Instruction List (By Mnemonic)**

    Appendix A5 includes an instruction field summary, a list of split-field notation and conventions, and the entire *PowerPC* instruction set sorted by mnemonic.

  - **Appendix 6: SM Assert Analysis Example Audit Trail**

    Appendix A6 is the contents of the header files needed to construct the data organization for the Assert Analysis Example - SM, Section 5.2.2.

  - **Glossary**

    The Glossary defines terms and expands acronyms used in this document.

1.4 USER FEEDBACK

We are constantly striving to improve the quality and usability of this information product. Please use one of the following options to provide us with your comments:

- You may use the on-line comment form at **http://www.lucent-info.com/comments**
- You may email your comments to **ctiphotline@lucent.com**
- You may print the comment form (located at the beginning of this information product after the Legal Notice) to send us your comments either by fax or mail as follows:

  — Fax to **1-407-767-2760**

  — Mail to the following address:

3. Trademark of International Business Machines Corporation.

Lucent Technologies
Documentation Services Coordinator
240 E. Central Parkway
Altamonte Springs, FL 32701-9928

- You may call the hot line with your comments. The telephone number is **1-800-645-6759**. The hot line is staffed Monday through Friday from 8:00 a.m. to 5:00 p.m. Eastern time.

Please include with your comments the title, ordering number, issue number, and issue date of the information product, your complete mailing address, and your telephone number.

If you have questions or comments about the distribution of our information products, see Section 1.5, Distribution.

1.5 DISTRIBUTION

For distribution comments or questions, either contact your local Lucent Technologies Account Representative or send them directly to the Lucent Technologies Customer Information Center (CIC) in Indianapolis, Indiana.

A documentation coordinator has authorization from Lucent Technologies to purchase our information products at discounted prices. To find out whether your company has this authorization through a documentation coordinator, call **1-888-LUCENT8 (1-888-582-3688)**.

Customers who are not represented by a documentation coordinator and employees of Lucent Technologies should order *5ESS* switch information products directly from CIC.

To order, call the following telephone number:

- **1-888-LUCENT8 (1-888-582-3688)** or fax to **1-800-566-9568** (from inside the continental United States)
- **+1-317-322-6847** or fax to **+1-317-322-6699** (from outside the continental United States).

1.6 TECHNICAL ASSISTANCE

For initial technical assistance, call the North American Regional Technical Assistance Center (NARTAC) at **1-800-225-RTAC (1-800-225-7822)**.

For further assistance, call the Customer Technical Assistance Management (CTAM) center at the following number:

- **1-800-225-4672** (from inside the continental United States)
- **+1-630-224-4672** (from outside the continental United States).

Both centers are staffed 24 hours a day, 7 days a week.

1.7 REFERENCES

More information is found in the following Lucent Technologies documents:

- 235-105-110 - *System Maintenance Requirements and Tools*
- 235-105-210 - *Routine Operations and Maintenance Procedures*
- 235-105-220 - *Corrective Maintenance Procedures*

- 235-600-400 - *Audits Manual*
- 235-600-500 - *Asserts Manual*
- 235-600-700 - *Input Messages Manual*
- 235-600-750 - *Output Messages Manual*

Note: An "x" or "x"s in the last three positions of a release specific document number indicate the digits that change from release to release. Refer to 235-001-001, *Documentation Description and Ordering Guide* for the document number associated with each software release.

Software Analysis Guide

CONTENTS PAGE

2. USING THE PROGRAM LISTINGS 2-1
2.1 INTRODUCTION TO ONLINE PROGRAM LISTINGS. 2-1
2.2 ONLINE ACCESS PROCEDURE 2-1
2.3 THE LISTINGS MENU 2-2
2.4 THE LISTINGS MENU 2-2
2.5 LISTINGS MENU OPTIONS 2-3
2.6 EXAMPLE — USING THE PROGRAM LISTINGS 2-4
2.6.1 Example Introduction 2-4
2.6.2 Determine the Function Name 2-5
2.6.3 Locate the Function 2-6

LIST OF EXHIBITS

Exhibit 2-1 — AM Assert Printout 2-4

Exhibit 2-2 — Breakpoint Listing for Function `INptcomp()` 2-7

Exhibit 2-3 — Disassembly Listing for Function `INptcomp()` 2-8

2. USING THE PROGRAM LISTINGS

2.1 INTRODUCTION TO ONLINE PROGRAM LISTINGS

Program listings consist of C language source code, its corresponding assembly code, and a set of indices for locating code. Online listings are available through a processor at Lucent Technologies, and can be accessed by using the dial-up service provided by Lucent Technologies Web and Media Management (WMM).

Web and Media Management contains various documentation products, including this document and the *5ESS*® switch program listings and source code.

Understanding the organization and use of online program listings is fundamental to the process of analyzing and resolving problems in the *5ESS* switch. This section of the Software Analysis Guide describes the organization of the online listings. The listings support the CNI, RTR, and most of the modular subsystems.

Upon completion of this section, the user should be able to locate the desired documentation in the online listings.

Note: Users have access to the officially sanctioned *5ESS* switch source files. These files are updated periodically to reflect changes made by the software update mechanism. A message of the day (MOTD) is displayed each time the user accesses the online listings environment. Because the MOTD provides information on the status of the online listings environment (such as stable, unstable, etc.), it is important that users read the MOTD at the start of each login session.

2.2 ONLINE ACCESS PROCEDURE

Use the following procedure to access the online listings.

1. If using a modem, dial 1-630-224-6640 to reach the Lucent Technologies Web and Media Management (WMM) *5ESS* Switch Online Switch Program Listings.

   Lucent employees can access the Listing environment via a telnet session to 135.185.130.28.

2. The system will prompt you to: Enter your login and password. (If you do not have a login and password you should contact 1-888-LUCENT-8. Lucent employees should call 1-800-228-6763.)

3. From the Lucent Technologies Document Menu, select *5ESS* Switch Program Listings.

4. From the next menu, ***5ESS* Switch Listings Program** select …… Execute Listings.

   a. The following instructions are displayed:

   ```
   Remember "exit" or "CTRL-D" will take you back to IDS
   ENTER "LIST"  It Will Put You Into The LISTING Menu
   Please remove your files when you are finished.
   ENTER RETURN TO CONTINUE...
   ```

   b. Press <Return>.

5. The Listings Environment menu is displayed; select the desired software release. After the message "Now entering the 5EX(x) and later listings environment," a submenu is presented; again, specify the desired software release.

   Enter your choice from the menu and press <return>.

The following message is displayed:

```
Enter editor desired (less or more)
Enter desired choice (less or more) and <return>.
```

a. The system will then reply:

```
THE ENVIRONMENT SETUP COMPLETE FOR LISTINGS.
Enter RETURN to CONTINUE
```

b. You will then be prompted to make a choice of setting up your environment to use Stackmap and Other Tools. Make your selection. To select Stackmap simply type in y (yes) at the prompt.

The system will then prompt you to select a subsystem from the list that is displayed on the screen.

2.3 THE LISTINGS MENU

The Listings Menu provides options for accessing the online program listings. It uses several environment variables that were set when the "Execute Listings" option was selected.

The menu display looks like:

```
5EX(X)XX.XX (SU LEVEL XX-XXXX) LISTINGS MENU
1) SYMBOL/FUNCTION DEFINITION
2) SOURCE FILE NAME
3) SYMBOL/FUNCTION REFERENCES
4) RTR NAMELIST
5) DGN SOURCE CODE CROSS-REFERENCE
6) POPRULES FILE
7) GENERATE BREAKPOINT/DISASSEMBLY LISTINGS
8) CDB INTERACTIVE MENU (FOR EXPERIENCED USERS)
9) EXIT

Enter Choice (RETURN will exit)-=>
```

To use this menu and its options,

- When prompted for a selection, type its associated number followed by <Return>.
- When prompted for a name, type the name of the product followed by <Return>.
- Press <Return> to display the previous screen. Entering <Return> from the Listings Menu will exit the program.

2.4 THE LISTINGS MENU

The Listings Menu provides options for accessing the online program listings. It uses several environment variables that were set when the "Execute Listings" option was selected.

The menu display looks like:

```
5EX(X)XX.XX (SU LEVEL XX-XXXX) LISTINGS MENU
1) SYMBOL/FUNCTION DEFINITION
2) SOURCE FILE NAME
3) SYMBOL/FUNCTION REFERENCES
4) RTR NAMELIST
5) DGN SOURCE CODE CROSS-REFERENCE
6) POPRULES FILE
7) GENERATE BREAKPOINT/DISASSEMBLY LISTINGS
8) CDB INTERACTIVE MENU (FOR EXPERIENCED USERS)
9) EXIT
```

```
Enter Choice (RETURN will exit)-=>
```

To use this menu and its options,

- When prompted for a selection, type its associated number followed by <Return>.
- When prompted for a name, type the name of the product followed by <Return>.
- Press <Return> to display the previous screen. Entering <Return> from the Listings Menu will exit the program.

2.5 LISTINGS MENU OPTIONS

Several menu options display a list of valid processor types and prompt the user to choose from the list. Select ASM to specify the traditional switching module (SM), and ASM2K to specify the SM-2000.

A brief description of each menu item follows.

- Symbol/Function Definition

  This option prompts for a symbol name. This can be a function, macro, or C variable defined as an extern (that is, a global variable).

  If the symbol or function is found, the system prompts the user to select from a list of path names to read the appropriate file.

- Source File Name

  This option prompts for a file name. If the specified file is located, the system displays a list of file locations and prompts the user to select one for viewing.

- Symbol/Function References

  This option prompts for a symbol name. This can be a function, macro, or C variable defined as an extern (that is, a global variable).

  If the symbol is found, the system prompts the user to select from a list of path names.

- RTR Namelist

  This option provides symbol and address information for the RTR environment. It can also be used to list the RTR Namelist files.

- DGN Source Code Cross-Reference

  This option provides a way to identify the function associated with a specific hardware diagnostic phase.

  This option initiates a series of interactive sub-menus. For *5ESS* switch application hardware, the path name to the diagnostic source code is provided. For RTR diagnostics (such as CU, MHD, etc.) the path name to the breakpoint file is given.

- Population Rules File

  The Automated Static Office Dependent Data (SODD) Audit is a feature that uses the source files of Population Rule Language, Version 5.0 (PRL 5.0) to

perform integrity checks on the office dependent data. Use this option to examine the `checks` section of the PRL 5.0 source code when analyzing SODD audit errors. At the prompt:

```
Please enter Poprule File -=>
```

enter the name of a relation, in lower case, preceded by `RL`, and appended with a `.R`. For example, enter `RLfc_line.R` to display the population rules for relation FC_LINE.

The online PRL 5.0 source code files include the latest software updates to the population rules.

- Generate Breakpoint/Disassembly Listings

  This option prompts the user for additional input, via a sub-menu. The sub-menu enables the user to generate either a breakpoint or disassembly listing, or both. The user is prompted for the subsystem name and the subsystem module name, information that is part of the path name information output in response to a function name query from the Symbol/Function Definition of the Listings Menu. Note that the online listings environment does not support the creation of CNI breakpoint or disassembly listings.

```
//XS845/inteam1z/si_app/5e11_1z/si/INcmpictl/INrecovery.c
                                  ^      ^
                                  |      |
                            subsystem    |
                               name      |
                                      subsystem
                                        module
                                         name
```

- CDB Interactive Menu (For Experienced Users)

  Use this option to gain access to the CDB tool. For information about its use, please refer to Item 5 (CDB1 - Manual Pages) under the *5ESS* Switch On-Line Field Grade LISTINGS sub-menu.

- Exit

  Use this option to exit the Listings Menu.

2.6 EXAMPLE — USING THE PROGRAM LISTINGS

2.6.1 Example Introduction

This section presents an example of how one might use the online program listings to locate and analyze the source code associated with a failing function. While it is beyond the scope of this example to detail all aspects of assert analysis, it does provide an explanation of the steps needed to locate the source code associated with a specific function. See "Assert Analysis," Section 5 for more detail.

Assert messages contain a failing address, one or more stack trace addresses and, as in the case of this AM example, the environment in which the assert fired. This information will be used to identify the failing function. See Exhibit 2-1 for a sample of the AM Assert printout.

Exhibit 2-1 — AM Assert Printout

```
S570-262181 88-09-28 06:03:55 000640
INIT AM LVL=RPI FPUMP DEF-CHK-FAIL= 1275 EVENT=373 COMPLETED
SW-ERR FAIL-ADDR=H'19b2 AM-MODE NORMAL CU 1 TIME 3:52:3
  PROCESS: OSDS=0,0 CALL-INTJ NONE DMERT 262254 EVENT-FLAGS=0'0
FCODE=0
```

```
  REQ-PROC 0 HDW-LVL 0     SPP-COUNTS=0,0   NO-AUD-SCHED

S570-262181 88-09-28 06:03:55 000641
REPT  AM STACK TRACE ENV=FPUMP SRC=DCF EVENT=373
  USER:  000019b2  00000C14  00002844  0000292C  000026F8  0000D66
         00000D42  0000289F

S570-262181 88-09-28 06:03:55 000642
REPT  AM  STACK FRAME ENV FPUMP  SRC DCF EVENT=373
  FUNC ADDR:   H'19b2
  PARAMETERS:  000004FB  00000C14  00040160  00040190  00040190
               00000000  00040000  00200000  00000001  00000000
  LOCAL DATA:  000004FB  00001A26  00040190  000401D4  00000001
               00060014  010002A6  00000001  00000000  00040000
               00200000  00000001  20000000  00000000  00040200
               000401D4  00040208  00000454  00000002  04FB0000
               003CC70B  00040100  00040200  00001A26  000401D4
               00040200  00040214  000022C2  000401D4  00040200
               00040208  00040240  00000000  000000FA  00040208
               00040240  00000001  00000000  00040000  00200000
               00000000  00000000  00000000  00000000  00000000

S570-262181 88-09-28 06:03:55 000643
REPT  AM  STACK FRAME ENV FPUMP  SRC DCF EVENT=373
  FUNC ADDR:   H'c14
  PARAMETERS:  00040110  40000000  00002844  000400D4  00040110
               00040000  00200000  00000001  00000001  00000000
  LOCAL DATA:  000004FB  00000C14  00040160  00040190  00040190
               00000000  00040000  00200000  00000001  00000000
               00040000  00200000  00000001  40000000  000004FB
               00001A26  0004019C  000401D4  00000001  00060014
               010002A6  00000001  00000000  00040000  00200000
               00000001  20000000  00000000  00040200  000401D4
               00040208  00000454  00000002  04FB0000  003CC70E
               00040100  0004020C  00001A26  000401D4  0004020C
               00000000  00000000  00000000  00000000  00000000
```

Note the failing address and environment in the printout and use this information to generate breakpoint and disassembly listings for the failing function.

- Failing address — 0x000019b2
- Environment — FPUMP

2.6.2 Determine the Function Name

The first step is to identify the path name of the environment. Use "Environment to Pathname Cross Reference," Appendix A1 to accomplish this step. Next, the `upd:ftrc` input message is used to identify the failing function:

```
upd:ftrc:fn="/no5text/prc/fpump",addr=h'19b2;
```

The format of the `upd:ftrc` output is:

```
UPD FTRC REPT
OBJECT_FILE=/no5text/prc/fpump
ADDRESS FUNCTION  START    SIZE   OFFSET   TV       FILE      SYMINDEX
------- -------- ------- -------- ------ ----- -------------- --------
   19b2 INptcomp    18dc      138     d6    16 INfpump_c.c    [    32]
```

The offset into the function is calculated by the switch and is included in the `UPD FTRC REPT` output message. The offset is needed so that the user will know the exact point of failure in the function.

2.6.3 Locate the Function

Use the "On-Line Access Procedure," Section 2.2 to access Web and Media Management and examine the source code.

Select option 1 (SYMBOL/FUNCTION DEFINITION) and, when prompted, enter the name of the failing function `INptcomp`.

When prompted, select from the list of valid processor types. Choosing the default (`<Return>`) will cause the data tables for *all* processor types to be searched, significantly increasing the search time. For a more efficient search specify the processor type when known. In this example, AM was specified. The system will now display the name of the file that contains the failing function.

Note the relative path of the source file, in particular the subsystem and subsystem module names. This information will be needed to generate breakpoint and/or disassembly listings. View the file to verify that it contains the function definition.

In this example, the relative path is `.../si/INampump/INfastpump.c`. The function exists in subsystem `si`, subsystem module `INampump`. Note that the subsystem and subsystem module names are case sensitive; this becomes important when breakpoint or disassembly listings are to be created.

Return to the main Listings menu by pressing `<Return>`, then select option 7 (GENERATE BREAKPOINT/DISASSEMBLY LISTINGS).

When prompted, specify the type of listings to generate. For this example, both breakpoint and disassembly listings are requested.

Next, enter a processor type. Since there is no default on this menu, a valid processor type must be specified.

Enter the subsystem and, when prompted, the subsystem module name. Recall that these names are case sensitive and must be entered exactly as displayed by the SYMBOL/FUNCTION DEFINITION search.

The system now displays a list of the files that were generated for the specified subsystem. Depending on the subsystem module, several module product files may have been created. Select the disassembly file for the module product desired. In this example, the file of interest is `.../INfpump.dis`.

Use standard `vi` commands to search the file for the function name; in this instance, the function is `INptcomp()`. (See Exhibit 2-3.)

Once the start of the function has been located, take note of the relative start address. The relative start address of this function is h'1948. To determine the exact point at which the assert fired, the previously calculated offset (h'd6) must be added to this start address: h'1948 + h'd6 = h'1a1e. Thus, h'1a1e marks the relative address of the instruction immediately following the assert macro call (though this doesn't ALWAYS hold true).

An examination of the assembly code reveals that the failure occurred in the instruction set associated with breakpoint [37]. The next step is to locate this breakpoint line in the breakpoint listing for function `INptcomp()` and determine the reason for the assert. (See Exhibit 2-2).

Quit the disassembly file, `INfpump.dis`, and the list of generated files for the subsystem module will be displayed again.

Select the breakpoint file for the module product desired. For this example, the file is `.../INfpump.bp`.

Search the file for the definition of the function `INptcomp()`. Since this file contains code for numerous functions, the first occurrence of function `INptcomp()` may not be its definition, but rather a call to it from another function. Use standard `vi` commands to step beyond these references. Note that the full name of the desired function is `INptcompmsg()`; by convention, only the first eight characters of a function name are used for software releases prior to 5E11(1). As of the 5E11(1) software release, C function/variable names can be longer than eight characters. The `INptcomp()` example is still valid in the 5E11(1) online listing environment; however, the `upd:ftrc` output would have listed the function's name as `INptcompmsg()` and all references to the eight-character name in the example would be replaced with the full function name.

Once the source code of the failing function has been located, search for the desired breakpoint line. In this example the breakpoint line is [37]. The C-language source code can now be analyzed to determine why the assert fired.

Repeat this procedure to locate variables or functions referenced by function `INptcomp()`.

Exhibit 2-2 — Breakpoint Listing for Function `INptcomp()`

```
AM:INFPUMP
@FUNCTION: INptcomp (INfpump_c.o)
/*
* Function:     INptcompmsg()
*
* Description:  This function will format and send the PARTIALLY COMPLETE
*               message to the SM and then clear the control structure that
*               the information was taken out of.
*
* Parameters:   None.
*
* Returns:      None
*
* Calls:        INeventno()     - get an event number
*               INfosmsg()      - format OSDS message
*               INperror()      - pump error handler
*               sendport()      - send a message
*
* Externals:    INpartinfo[]    - section start addresses for sections pumped
*               msgbuf          - FPUMP message buffer
*               ctrl            - FPUMP misc. information area
*/
void
INptcompmsg()
{
        INBCASTMSG      bcastmsg;       /* message to broadcast pump */
        INCNTRLTBL      *partinfo_ptr;  /* pointer to control structure */
        unsigned short  i;              /* useful index */
[6]     partinfo_ptr = &INpartinfo[INcurrent  ^ 1];
[8]     if ( INlinkname == 0 ) {
[9]             msgbuf.btmsg.osds_hdr.realhdr.type &= ~LINKMASK;
        }else{
[11]            msgbuf.btmsg.osds_hdr.realhdr.type |= LINKMASK;
        }
[13]    INlinkname ^= 1;
                /* send the message provided that this is NOT an offline pump */
[16]    if ( ! INISOFFLINEPUMP( ctrl.ptype ) ) {
                /* Fill in the header that the SMs will see. */
[18]            bcastmsg.msghead.type = MGPUFPMP;
[19]            bcastmsg.msghead.priority = INMSGPRIOR;
```

```
[20]            bcastmsg.msghead.length = sizeof( struct mgPUFPMP );

                /* Fill in the body that the SMs will see. */

[23]            bcastmsg.msgs.pufpmp.ccode = INPARTSUCC;
[24]            bcastmsg.msgs.pufpmp.nblocks = partinfo_ptr->no_blocks;
[25]            bcastmsg.msgs.pufpmp.lastbyte = partinfo_ptr->lstbytpmp;

AP#    88/INptcomp    /INptcomp
    FPUMP Handling Process Control Functions   5ESS           PR-5D12010-71
                 AM:INFPUMP   ISSUE 01    PAGE 88
                SEE PROPRIETARY NOTICE ON COVER PAGE

AM:INFPUMP

[26]            bcastmsg.msgs.pufpmp.linknum = INlinkname;
                /* load section start addresses in message */
[29]            for ( i = 0; i < partinfo_ptr->tot_sect; i++ ) {
[30]                    bcastmsg.msgs.pufpmp.strtaddr[i+1] == partinfo_ptr->sect_add
r[i];
[31]            }
                /* store total number of sections in the message */
[33]            bcastmsg.msgs.pufpmp.strtaddr[0] = (unsigned long)partinfo_ptr->tot
_sect;

                /* Broadcast the Partial Complete message to the SMs. */
[36]            if (INfhbcastsend(&bcastmsg,ctrl.pumping_sms,ctrl.bootpid.procno)!=
SUCCESS) {
[37]                    INperror( INPUSDPT );
                }
        }
        /* clear the control structure that was just used */
[43]    partinfo_ptr->lstbytpmp = 0L;
[44]    partinfo_ptr$>no_blocks = 0;
[45]    partinfo_ptr->tot_sect = 0;

        /* Indicate that we are making progress for fast pump */
[48]    if (( ctrl.ptype == INFP ) || ( ctrl.ptype == INBCP )) {
[49]            SIpprog[ INFP ].event_no = INeventno();

                /* save the time that progress is updated */
[52]            INprog_time = getime();
        }
[54] }

AP#    89/INptcomp    /INptcomp

    FPUMP Handling Process Control Functions   5ESS           PR-5D12010-71
                 AM:INFPUMP   ISSUE 01    PAGE 89
                SEE PROPRIETARY NOTICE ON COVER PAGE
                                                              11/ 6/91
```

Exhibit 2-3 — Disassembly Listing for Function `INptcomp()`

```
AM:INFPUMP

@DISASSEMBLY: INptcomp (INfpump_c.o)

                ****  3b DISASSEMBLER  ****

disassembly for INfpump_c.o

 section        .text
INptcomp()
            1948: 7a20                      save    &0x2,&0x0
            194a: 346a 06cb                 addw2   &0x6c,%sp
[6]         194e: 4448 0004 1f80            movzbw  $0x41f8,%r0
            1954: 3221 0000                 xorw2   &0x1,%r0
```

```
            1958: 349a 0540                  umulw2  &0x54,%r0
            195c: 346c 0000 414c 0000        addw2   &0x0414c,%r0
            1964: 1408                       movw    %r0,%r8
[8]         1966: 2bf8 0004 1f60             cmph    $0x41f6,&0x0
            196c: 8106                       bne     +0x6 <197a>
[9]         196e: 2d0c 7fff 0008 0004 5940   andh2   &0x7fff,$0x4594
            1978: 8005                       br      +0x5 <1984>
[11]        197a: 2d1c 8000 0008 0004 5940   orh2    &-0x8000,$0x4594
[13]        1984: 2b28 0004 1f61             xorh2   &0x1,$0x41f6
[16]        198a: 23f8 0004 7763             cmpb    $0x4776,&0x3
            1990: 8246                       be      +0x46 <1a1e>
            1992: 23f8 0004 7766             cmpb    $0x4776,&0x6
            1998: 8242                       be      +0x42 <1a1e>
[18]        199a: 4d0a 5220 a040             movh    &0x522,0x4(%fp)
[19]        19a0: 4300 a060                  movb    &0x0,0x6(%fp)
[20]        19a4: 450a 05c0 a070             movb    &0x5c,0x7(%fp)
[23]        19aa: 4b00 a085                  movh    &0x5,0x8(%fp)
[24]        19ae: 1480                       movw    %r8,%r0
            19b0: 4d00 04e0 a0a0             movh    0x4e(%r0),0xa(%fp)
[25]        19b6: 1480                       movw    %r8,%r0
            19b8: 5500 0480 a5c0             movw    0x48(%r0),0x5c(%fp)
[26]        19be: 4d08 0004 1f60 a600        movh    $0x41f6,0x60(%fp)
[29]        19c6: 4a00 0007                  movh    &0x0,%r7
            19ca: 8011                       br      +0x11 <19ee>
[30]        19cc: 3870                       movzhw  %r7,%r0
            19ce: 5620                       llsw2   &0x2,%r0
            19d0: 1080                       addw2   %r8,%r0
            19d2: 14a1                       movw    %fp,%r1
            19d4: 346a 0101                  addw2   &0x10,%r1
            19d8: 693a 001e 7002             addh3   &0x1,%r7,%r2
            19de: 3822                       movzhw  %r2,%r2
            19e0: 5622                       llsw2   *0x2,%r2
            19e2: 1021                       addw2   %r2,%r1
            19e4: 5500 0000 1000             movw    0x0(%r0),0x0(%r1)

AP#    90/INptcomp    /INptcomp

    FPUMP Handling Process Control Functions   5ESS           PR-5D12010-71
                    AM:INFPUMP  ISSUE 01    PAGE 90
                 SEE PROPRIETARY NOTICE ON COVER PAGE
                                                              11/ 6/91
AM:INFPUMP

[31]        19ea: 2a61 0007                  addh2   &0x1,%r7
            19ee: 1480                       movw    %r8,%r0
            19f0: 29e0 04c7                  cmph    %r7,0x4c(%r0)
            19f4: 9515                       blu     -0x15 <19cc>
[33]        19f6: 1480                       movw    %r8,%r0
            19f8: 4d30 04c0 a100             movzhw  0x4c(%r0),0x10(%fp)
[36]        19fe: 600a                       pushw   %fp
            1a00: 620c 0000 4784 0000        pushw   &0x04784
            1a08: 6248 0004 7080             pushbh  $0x4708
            1a0e: 7600 0003                  call    &0x3,*$0x760000
            1a12: 1a10                       cmpw    %r0,&0x1
            1a14: 8204                       be      +0x4 <1a1e>
[37]        1a16: 620a 4e60                  pushw   &0x4e6
            1a1a: 7600 0001                  call    &0x1,*$0x760000
[43]        1a1e: 1480                       movw    %r8,%r0
            1a20: 5300 0480                  movw    &0x0,0x48(%r0)
[44]        1a24: 1480                       movw    %r8,%r0
            1a26: 4b00 04e0                  movh    &0x0,0x4e(%r0)
[45]        1a2a: 1480                       movw    %r8,%r0
            1a2c: 4b00 04c0                  movh    &0x0,0x4c(%r0)
[48]        1a30: 23f8 0004 7761             cmpb    $0x4776,&0x1
            1a36: 8204                       be      +0x4 <1a40>
            1a38: 23f8 0004 7765             cmpb    $0x4776,&0x5
            1a3e: 810a                       bne     +0xa <1a54>
[49]        1a40: 7600 0000                  call    &0x0,*$0x760000
            1a44: 5178 0000 0060             movtwh  %r0,$0x6
```

```
[52]        1a4a: 7600 0000                 call    &0x0,*$0x760000
            1a4e: 5108 0004 9040            movw    %r0,$0x4904
[54]        1a54: a100 7b20                 ret     &0x2

AP#    91/INptcomp    /INptcomp

    FPUMP Handling Process Control Functions   5ESS           PR-5D12010-71
                  AM:INFPUMP  ISSUE 01    PAGE 90
               SEE PROPRIETARY NOTICE ON COVER PAGE
                                                              11/ 6/91
```

Software Analysis Guide

CONTENTS PAGE

3. C PROGRAMMING LANGUAGE 3-1

3.1 DATA TYPES 3.1-1
3.1.1 Data Types - General 3.1-1
3.1.2 Bitfields 3.1-3
3.1.3 Arithmetic Operators 3.1-4
3.1.4 Relational Operators 3.1-5
3.1.5 Logical Operators 3.1-5
3.1.6 Operator Precedence 3.1-6
3.1.7 Variables (Local and Global) 3.1-10
3.1.8 Structures 3.1-11
3.1.8.1 Struct 3.1-11
3.1.8.2 Unions 3.1-12
3.1.9 Storage Class Specifiers 3.1-12
3.1.10 Preprocessor 3.1-14
3.1.11 Pointers and Arrays 3.1-15
3.1.12 Membership Operator 3.1-19
3.1.13 Casting 3.1-20
3.1.14 Functions 3.1-21
3.1.15 Control Statements 3.1-22

3.2 *5ESS*® SWITCH DIFFERENCES FROM K AND R REFERENCE 3.2-1
3.2.1 Introduction to Section 3.2-1
3.2.2 `Sizeof` 3.2-1
3.2.3 Array Size Limitations 3.2-1
3.2.4 Multiple Structure Assignments 3.2-1
3.2.5 Unsigned Data Types 3.2-1
3.2.6 `Float`/`Double` Floating Point Data Types 3.2-2
3.2.7 Bitfields 3.2-2
3.2.8 Zero-Length Bitfields and Their Alignment 3.2-2
3.2.9 Integer Bitfields and Their Alignment 3.2-2
3.2.10 The Value of An Assignment 3.2-2
3.2.11 Enumeration Comparisons 3.2-2
3.2.12 Function Templates 3.2-2
3.2.13 Flexnames 3.2-3
3.2.14 Integer Size 3.2-3

LIST OF FIGURES

Figure 3.1-1 — Bitfield Layout for the Structure i_bits 3.1-3

Figure 3.1-2 — Array A. **3.1-17**

Figure 3.1-3 — Array B. **3.1-18**

Figure 3.1-4 — Array C. **3.1-18**

Figure 3.1-5 — Array D. **3.1-19**

Figure 3.1-6 — Array E. **3.1-19**

LIST OF TABLES

Table 3.1-1 — Arithmetic Operators. **3.1-4**

Table 3.1-2 — Assignment Operators **3.1-4**

Table 3.1-3 — Relational Operators **3.1-5**

Table 3.1-4 — Logical OR **(||) Operator**. **3.1-5**

Table 3.1-5 — Logical AND **(&&) Operator** **3.1-6**

Table 3.1-6 — Operator Precedence and Direction **3.1-7**

Table 3.1-7 — Preprocessor Commands **3.1-15**

LIST OF EXHIBITS

Exhibit 3.1-1 — Enumeration Example AMMDR_RECS **3.1-2**

Exhibit 3.1-2 — Enumeration Example AMLOSTREASON **3.1-2**

Exhibit 3.1-3 — Structure Example amINPRIV. **3.1-11**

3. C PROGRAMMING LANGUAGE

The purpose of this section is to review aspects of the C language and to enhance the user's overall understanding of how the language is used in the *5ESS*® switch. This section is designed for the experienced C user and is not intended as a teaching tool. The first half of this section is the language review and the second half is an explanation of the language variations that are to be found in the C language of the *5ESS* switch.

3.1 DATA TYPES

3.1.1 Data Types - General

The C language is composed of only a few fundamental data types. Data types are the way the C language represents the storage requirements of a particular type of variable or constant; they are the building blocks of more elaborated data structure definitions. The data type establishes what class of data values can be held by a variable and what variety of operations can be performed on that variable. The C language data types include the following:

`int` a value of type integer

`char` a value of type character

`float` a floating point value (float values are not supported in the *5ESS*® switch)

`double` double-precision floating point value (float values are not supported in the *5ESS* switch).

There are also a number of qualifiers that can be used in conjunction with these data types.

`short` a short integer

`long` long integer

`signed` signed `char` values between –128 and 127, but an `int` is –32,768 to 32767 or –2,148,483,648 to 2,148,483,647 (the signed qualifier is not supported by the *5ESS* switch).

`unsigned` unsigned `char` values between 0 and 255, but an `int` is 0 to 65,585 or (in the AM) 4,294,967,295.

The qualifiers `short` and `long` refer to integer values only. The use of `int` in the declaration of `short` or `long` is optional:

```
short int var1;
long int var2;
```

or

```
short var1;
long var2;
```

Each data type has its own storage requirements and capabilities. In addition, the type of processor used (3B20D or MC68000) will also affect the storage requirements of any particular type. Unsigned numbers are always positive or zero.

The data types and their qualifiers can be grouped under the two general headings of integral types and floating types, based on the way they are stored in the computer.

Integral types:
`int, char, short, long, signed, unsigned`

Floating types:
`float, double`

Since the type `char` is actually an integral type, variables and constants of this type can be manipulated just as you would manipulate an integer because characters have an underlying integer value which is their ASCII representation. For example, the expression `'b' + 1` has the value `'c'`, just as the expression `'A' + 2` would have the value `'C'`.

In addition to these data types, there are also derived data types such as array, function, pointer, structure, and union. Each of these is discussed elsewhere in this document. There is also an enumeration type. An enumeration or `enum` is a data type whose possible values are limited to a list of constant values. These constants are assigned when the type is defined. It also allows names to be used in lieu of integer values when a choice must be made from a collection of items. For example:

```
enum tally {yea, nay, abstain};
     .
     .
     .
enum tally vote= abstain;
```

or

```
enum {yea, nay, abstain} vote = abstain;
```

All `enum` variables and constants perform just like type `int`. The compiler of the *5ESS* switch stores an `enum` in the smallest possible data size that can retain all possible enumeration values, 1 byte, 2 bytes, or 4 bytes. Exhibits 3.1-1 and 3.1-2 are examples of enumerations taken from the code of the *5ESS* switch.

Exhibit 3.1-1 — Enumeration Example `AMMDR_RECS`

```
typedef enum {
    AMNOMDR      = 0,  /* No MDR record */
    AMORIGPRIV   = 1,  /* Originating Private MDR Record */
    AMINPRIV     = 2,  /* Incoming Private MDR Record */
    AMPUBLIC     = 3,  /* Public MDR Record */
    AMSTA_STA    = 4,  /* Station -to- Station MDR Record */
    AMCOUNTS     = 5   /* MDR Counts Record */
} AMMDR_RECS;
```

Exhibit 3.1-2 — Enumeration Example `AMLOSTREASON`

```
typedef enum amLOSTREASON {
   AMSAMEMFULL = 0, /* Stand-alone memory full */
   AMLANSINH = 1,   /* Local Answer recording Inhibited */
   AMNOCR = 2,      /* Could not get Call Record */
   AMTERMINH = 3,   /* Terminating recording inhibited */
   AMBADCTYPE = 4,  /* Invalid Call Type */
   AMBADSCODE = 5,  /* Invalid Structure Code */
   AMBADCOND = 6,   /* Invalid Condition */
   AMNOOPACT = 7,   /* Couldn't format Operator Action */
   AMNOTERMDN = 8,  /* couldn't format Terminating DN */
   AMNOCARTIME = 9, /* Couldn't format Carrier Connect Time */
   AMSDSFULL = 10,  /* SDS Full */
   AMSDSPTR = 11,   /* Bad SDS Pointer */
   AMBADMGLENGTH = 12, /* Bad Message Length */
   AMSMBUFF = 13,   /* Couldn't load SM buffer */
   AMBADSFAM = 14,  /* Bad Structure Code Family */
   AMSP2LREASON = 15, /* Spare 2 */
   AMBADEBAFTYPE = 16, /* Invalid EBAF Type */
   AMODBINH = 17,   /* On Demand B-channel recording inhibited */
   AMSP3LREASON = 18, /* Spare 3 */
   AMSP4LREASON = 19, /* Spare 4 */
   AMSP5LREASON = 20, /* Spare 5 */
   AMSP6LREASON = 21, /* Spare 6 */
   AMSP7LREASON = 22, /* Spare 7 */
   AMSP8LREASON = 23, /* Spare 8 /
   AMSP9LREASON = 24, /* Spare 9 */
   AMSP10LREASON = 25, /* Spare 10 */
   AMSP11LREASON = 26, /* Spare 11 */
   AMSP12LREASON = 27, /* Spare 12 */
   AMSP13LREASON = 28, /* Spare 13 */
   AMSP14LREASON = 29, /* Spare 14 */
```

```
   AMSP15LREASON = 30, /* Spare 15 */
   AMSP16LREASON = 31, /* Spare 16 */
 } AMLOSTREASON;
```

3.1.2 Bitfields

Bitfields are a special form of declaration that allows for the addressing and manipulation of individual bits in a field. The *5ESS* switch compiler supports `char`, `short`, `int` and `long` bitfields. Bitfields in the *5ESS* switch are always unsigned, the `unsigned` qualifier is ignored. These fields are created through a `struct` declaration. Bitfields are packed into machine words, one after the other, in the order in which they were declared.

The maximum size of a bitfield is the number of bits in a machine word. A machine word is usually the size of `int`. Bitfields cannot cross a machine word boundary. If a field would overlap a word boundary, the compiler will move it to the start of the next machine word at compile time. A bitfield of value 0 forces the next bitfield declaration to align itself at the next word boundary. In the *5ESS* switch, the least significant bit is on the right and the most significant on the left. When bitfields are used it is critical to remember that the bits are loaded from left to right.

An example of the declaration for a bitfield is:

```
struct {
    unsigned field1  : 2 ;
    unsigned field2  : 4 ;
    unsigned field3  : 6 ;
} fields ;
```

A detailed knowledge of the hardware specifics used by the compiler is required when bitfields are used. An example of a bitfield declaration in the *5ESS* switch is:

```
struct {
    unsigned short     : 8;
    unsigned short frmtype : 1;
    unsigned short ns   : 3;
    unsigned short p    : 1;
    unsigned short nr   : 3;
} i_bits;
```

The i_bits structure consists of 16 bits. The bitfield construction in the memory would look like the layout in Figure 3.1-1.

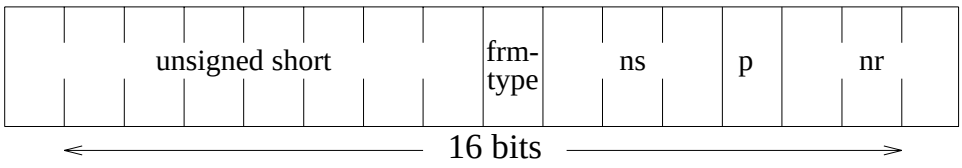


Figure 3.1-1 — Bitfield Layout for the Structure i_bits

3.1.3 Arithmetic Operators

To perform mathematical computations, C uses arithmetic and assignment operators. The arithmetic operators can be classified by the number of operands they require, that is unary (one) and binary (two). The mathematical operations used in the C language are detailed in Table 3.1-1.

Table 3.1-1 — Arithmetic Operators

Symbol	Description
+	Adds value at its right to the value at its left
–	Subtracts value at its right from the value at its left
– +	Unary operators, change the sign of the value at their right (+ not supported)
*	Multiplies value at its right by the value at its left
/	Divides value at its left by the value at its right. Integer values are truncated
%	Returns the remainder when the value at its left is divided by the value to its right (integers only)
++	Increments by 1 the value of the variable to its right or left (prefix mode or postfix mode `++i` or `i++`)
––	Decrements by 1 the value of the variable to its right or left (prefix mode or postfix mode `--i` or `i--`)

In addition to the arithmetic operators, there are also assignment operators that complete the statements in which arithmetic operators are used (see Table 3.1-2).

The assignment operators are used to transfer the value of an expression into a storage location. In the C language the equivalence operator's functions have been divided into two separate operators. There is a difference between the = assignment operator and the == relational operator. The first assigns a value to a variable, and the second only checks for equivalence and does not transform the variables in any way.

Table 3.1-2 — Assignment Operators

Symbol	Description/Example
=	Assigns the value of the expression on the right to the variable on the left.
+=	`x += 2` is equivalent to `x = x + 2`
–=	`x -= 2` is equivalent to `x = x - 2`
*=	`x *= 2` is equivalent to `x = x * 2`
/=	`x /= 2` is equivalent to `x = x / 2`
%=	`x %= 2` is equivalent to `x = x % 2`

3.1.4 Relational Operators

A relational operator compares two operands to determine their association. The relational operators are used most frequently with `if`, `while`, `for` and `?:` conditional statements to determine when an action will start or if it will take place at all. The relational operators are shown in Table 3.1-3.

Table 3.1-3 — Relational Operators

Symbol	Description
==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

3.1.5 Logical Operators

Various relations can be combined by the logical connectives (`&&` and `||`) the logical `AND` and `OR` operators. Expressions related by `&&` or `||` are evaluated from left to right. In the case of the `||` operator, if the first operand is true, then the second operand is not evaluated because the value of the expression must be true since the first operand is true. As shown in Table 3.1-4, it is not necessary for the system to evaluate the second operand when the first is true; the outcome is the same in either case.

Table 3.1-4 — Logical `OR` (`||`) Operator

EVALUATION TABLE (`||`)		
First Operand	Second Operand	Expression Value
T	?	T
F	T	T
F	F	F

In the case of the `&&` operator, however, if the first operand is false, then the second operand is not evaluated. The `&&` operator requires that both elements must be evaluated as true or the entire statement must be evaluated as false. As shown in Table 3.1-5 it is not necessary for the system to evaluate the second operand when the first is false; the outcome is the same in either case.

Table 3.1-5 — Logical AND (&&) Operator

EVALUATION TABLE (&&)		
First Operand	**Second Operand**	**Expression Value**
T	T	T
T	F	F
F	?	F

3.1.6 Operator Precedence

The C language has a built in precedence structure that determines when a particular arithmetic, assignment, logical, or relational operator will be evaluated. Table 3.1-6 explains the direction of association for each type of operator and also explains the order of precedence. The highest precedence is at the top of the table, the lowest at the bottom. The precedence can be altered in statements and expressions by using parentheses. Parentheses force the operator inside of the ( ) to be evaluated before those outside, regardless of precedence.

Table 3.1-6 — Operator Precedence and Direction

Operator Type	Symbol	Direction
Primary	() [] → .	L to R
Unary	! ~ ++ -- + - * & (type) sizeof	R to L
Arithmetic	* / %	L to R
	+ -	
Shift	<< >>	L to R
Relational	< <= > >=	L to R
	== !=	
Bitwise Logical	&	L to R
	|	
	^	
Logical	&&	L to R
	||	
Conditional	? :	R to L
Assignment	= += -= /= %= &= ^= |= <<= >>=	R to L
Comma	,	L to R

Primary Operators

Primary operators are the strongest operators. Of the primary operators, the parentheses are the strongest. The other operators serve to describe the access to data and are stronger than any other operators upon that data.

`()` The function operator. Example: `main( )`. Also used for grouping.

`[]` Array brackets. Example: `arrayname[]`.

→ The indirect membership operator is used with a pointer to a structure or union to identify a member of that structure or union. (See "Membership Operator," Section 3.1.12.)

. The membership operator is used with a structure or union name to specify a member of that structure or union. (See "Membership Operator," Section 3.1.12.)

Unary

Unary operators change the value at their right.

! This is the logical negation operator. Its operand must be an arithmetic type or be a pointer.

~ The one's complement, or bitwise negation, changes each 1 to a 0 and each 0 to a 1. Example: `~(10011011) == (01100100)`.

++ The increment operator increases the value of its operand by one. The operator can be used either before or after the variable, prefix or postfix mode. Example:

```
x=10
y=++x
```

```
a=10
b=a++
```

The value of `y` is 11 because `x` is incremented before the statement is evaluated and the value of `b` is 10 because the statement is evaluated and then the value of `a` is incremented.

`--` The decrement operator decreases the value of its operand by one. The operator can be used either before or after the variable, prefix or postfix mode. Example:

```
x=10
y=--x

a=10
b=a--
```

The value of `y` is 9 because `x` is decremented before the statement is evaluated and the value of `b` is 10 because the statement is evaluated and then the value of `a` is decremented.

`+` As a unary operator, the plus sign means to take the value of its operand. (Not supported by the *5ESS* switch compilers.)

`-` As a unary operator, the minus sign means to take the negative of its operand.

`*` The indirection operator is used with pointers. When followed by the name of the pointer, it gives the value stored at the pointed-to address.

`&` The address operator gives the address of the variable to which it is attached.

`(type)` The cast operator converts the value to its right to the type specified by the enclosed keyword. Example: `(float) 4` changes the integer 4 into a float value.

`sizeof` The sizeof operator gives the size in bytes of the operand to its right.

Arithmetic Operators

Arithmetic operators perform mathematical computations such as addition and multiplication.

`*` The multiplication operator multiplies the two values on either side of it. Example: `printf("%d",3*7)`. The value that prints is 21.

`/` The division operator divides the value on its left by the value on its right. Example: `printf("%d",21/3)`. The value that prints is 7. For integer division, results are truncated.

`%` The modulus operator gives the remainder that results from integer division. Example: `printf("%d",22%3)`. The value that prints is 1.

`+` The addition operator adds the two values on either side of it together. Example: `printf("%d",3+7)`. The value that prints is 10.

–	The subtraction operator subtracts the value on its right from the value on its left. Example: `printf("%d",10-7)`. The value that prints is 3.

Shift Operators

Shift operators are bitwise operations.

<<	The left shift operand shifts the bits of the left operand to the left by the number of positions given by the right operand. The right most positions left vacant by the shift are filled with 0s. Example: `(11110011)<<2==(11001100)`.
>>	The right shift operand shifts the bits of the left operand to the right by the number of positions given by the right operand. The left most positions left vacant by the shift are filled with 0s for unsigned types, for signed types the result is machine dependent. Example: `(11110011)>>2==(00111100)`. The *5ESS* switch does signed shifts by propagating the sign bit (the most significant bit).

Relational Operators

Relational operators compare two operands to determine their association.

<	Less than.
<=	Less than or equal to.
>	Greater than.
>=	Greater than or equal to.
==	Equal to. This does not affect the value of the variables.
!=	Not equal to.

Bitwise Logical Operators

Bitwise logical operators are logical connectives that operate on one or more bits.

&	The bitwise AND (&) makes a bit-by-bit comparison. For each bit, the resulting bit position is 1 only if both corresponding bits in the operands are 1. Example: `(11100011)&(10010010)==(10000010)`.
|	The bitwise OR (|) makes a bit-by-bit comparison. For each bit, the resulting bit is 1 if either of the corresponding bits in the operands is 1. Example: `(10010011)|(00111101)==(10111111)`.
^	The bitwise EXCLUSIVE OR (^) makes a bit-by-bit comparison. For each bit, the resulting bit is 1 if either, but not both, corresponding bits in the operands are 1. Example: `(10010011)^(00111101)==(10101110)`.

Logical Operators

Logical operators are logical connectors such as OR which are used to evaluate the validity of an argument or set of arguments.

&&	The logical AND operator (see Table 3.1-5).
||	The logical OR operator (see Table 3.1-4).

Conditional Operator

`? :` The conditional operator uses a logical value as an operand. It is the only ternary operator and is also considered a statement. See "Control Statements," Section 3.1.15. If the first operand is nonzero, it is interpreted as TRUE and the second operand is evaluated, if it is evaluated as FALSE the third operand is evaluated. Example: `x = 3D (a > 1) ? 4 : 5;` The value of `x` in this example would be 4.

Assignment Operators

Assignment operators store the value computed on their right into the variable on their left. The shorthand used for the operators, which follow the value assignment operator in this list, update the variables at their left by the value at their right.

`=` This is the value assignment operator. It stores the value on its right in the memory location represented by the variable on its left. Example: `abc = 100`.

`+=` Add the right hand quantity to the left hand quantity.

`−=` Subtract the right hand quantity from the left hand quantity.

`/=` Divide the left hand quantity by the right hand quantity.

`%=` Stores the remainder from the division of the left hand quantity by the right hand quantity in the left hand quantity.

`&=` Perform a bitwise `AND` operation storing the result in the left operand.

`|=` Perform a bitwise `OR` operation storing the result in the left operand.

`^=` Perform a bitwise `EXCLUSIVE OR` operation storing the result in the left operand.

`<<=` Perform a bitwise left shift operation storing the result in the left operand.

`>>=` Perform a bitwise right shift operation storing the result in the left operand.

Comma The comma separates expressions that are grouped together. The expressions are evaluated left to right, with the result being the right-most expression.

`,` Example: `f(x,(y=4,y+2),z)` has three arguments, the second argument is equal to 6.

3.1.7 Variables (Local and Global)

A variable is the name for a memory location, that is, a particular byte address. Variables consist of letters and digits and the first character must always be a letter. Variable names are lowercase and symbolic constants are uppercase. Variables are either local or global in scope depending on how they are defined. (See "Storage Class Specifiers," Section 3.1.9, for details on the scope of declarations.)

Local Variables

Variables declared within a function are local to that function and will not be recognized by any others. Local variables come into existence when their function is called and disappear when the function is exited.

Global Variables

Global variables are declared outside any function and can be accessed by all functions subsequent to its declaration. Most global variables are defined at the beginning of the program to make them available to all functions.

3.1.8 Structures

3.1.8.1 Struct

The structure or `struct` is a collection of data types that are combined to form an organized record that can be manipulated as a whole or in part(s). To create a structure, simply declare each of the elements after the structures tag (name):

```
struct example
{
  int  i ;
  char c ;
  float f ;
};
```

This declaration creates a new data type using existing data types. Remember that a declaration of this type does not create storage. To create an actual storage location, you would have to specify the following after creating the structure:

```
struct example first;
```

where `first` is the actual name of the variable (storage location). This is the type of declaration found in the *5ESS* switch. Structures are usually defined in the Global Header Files or Local Header Files but they may also be declared in other places in the source code of the *5ESS* switch.

It is also possible to declare the structure's name at the same time that the data structure is created:

```
struct example
{
  int  i ;
  char c ;
  float f ;
} first, second, third;
```

Exhibit 3.1-3 is an example of a structure used in the *5ESS* switch.

Exhibit 3.1-3 — Structure Example `amINPRIV`

```
struct amINPRIV {              /* start field  */
                               /* byte *  */
  CMAPHDR ap_hdr;              /*  0 * Session ID. */
                               /*  4 * Business Customer ID. */
                               /*  6 * Application ID. */
                               /*  7 * Length */
  AMMDR_RECS msg_type : 8;     /*  8 * Message Type */
  AMMDREVTCD event_code : 8; /*  9 * Call Event Code */
  AMMDRFIC fic : 8;            /* 10 * Feature Interaction Code */
  unsigned char ars_grp;       /* 11 * ARS Pattern Group */
  unsigned char frl;           /* 12 * Facility Restriction Level */
  AMMDRANSW answ_ind : 8;      /* 13 * Answer Indicator */
  AMMDRFTYPE in_type : 8;      /* 14 * Incoming Facility Type */
  AMMDRFTYPE out_type : 8;     /* 15 * Outgoing Facility Type */
  unsigned short in_group;     /* 16 * Incoming Facility Group */
```

```
    unsigned short in_member;  /* 18 * Incoming Facility Member */
    unsigned short out_group;  /* 20 * Outgoing Facility Group */
    unsigned short out_member; /* 22 * Outgoing Facility Member */
    long ans_time;             /* 24 * Answer Time */
    long end_dtime;            /* 28 * End of Dialing Time */
    short date;                /* 32 * Date of Call */
    unsigned char called[16];  /* 34 * Called Number */
    unsigned char fill1[2];    /* 50 * --- FILL --- */
                                /* ---- */
                                /* 52 * Total */
};
```

3.1.8.2 Unions

A `union` is a derived data type that allocates one storage location for various types and sizes of data. Its declaration is similar to that of the `struct`. A union may contain various data types, but unlike the struct, a union can only store one type at any given time. When a union is declared, storage space is allocated for the largest data item and the same storage space is used for all the union variables. During processing, the storage space is continually overwritten whenever a new union member is to be stored in it. Keep in mind that a union must be addressed in the mode of its present contents. That is, an integer data type must be pointed to by an integer pointer. A union declaration follows.

```
union union_label
{
int intval;
float flval;
char *stval;
} union1,union2;
```

3.1.9 Storage Class Specifiers

The storage class specifier in a declaration determines the scope of the declared object. Only one storage class specifier may appear in a declaration. The storage class specifiers are auto, register, static, extern, and typedef.

`auto` The `auto` storage specifier is permitted only in declarations of variables within functions. It indicates that the variable has local (automatic) extent.

- If an automatic is declared having the same name as a global, the automatic is referenced by the function, not the global.
- Automatic variables are known only to the function that declared them.
- Automatic variables are stored on the stack.
- Arguments to functions are automatic variables.

`register` The `register` specifier has the same basic meaning as `auto`, but in addition tells the compiler that the local variable (or parameter) will be heavily used and should be allocated in a way that minimizes access time. (That is, placed in a register if possible.)

`static` The `static` specifier means that the object is permanently allocated storage space for the duration of the program, unlike automatic which is only temporary allocation. These variables may be declared either inside or outside of the function. If declared inside the function, only the function can reference it. If outside the function, only functions in the same file can reference it. Static variables are initialized to zero before

program execution if no initialization is specified. Static storage class is not known to the linker.

`extern` The `extern` storage class specifier may appear in declarations of external functions and variables, either at the top level or at the heads of blocks. It indicates that the object declared has static extent (permanent storage space) and its name is known to the linker. The compiler is informed that this variable does not need stack space because it has been defined elsewhere.

`typedef` The `typedef` storage class specifier does not actually allocate storage space. It is called a storage class for syntactic convenience. The `typedef` allows for the creation of different and perhaps more informative names for data types. In other words, `typedef` does not create a new type, but rather a synonym for an existing type. The word `REAL` can be used instead of `float` to define a floating point number because this term is more familiar. The following statement would tell the compiler that instead of `float`, the term `REAL` will be used.

```
typedef float REAL;
```

From that point on in the program floating point variables can be defined:

```
REAL x, y, z[10], *ptr;
```

The `typedef` is most useful when a number of complicated types will be used. Once the type is defined, it can be used whenever necessary.

An example of a `typedef` in the *5ESS* switch is:

```
/*
 * The EBAF AMA type. This value is put into the call record field
 * "ebaftype" and is the index into the AMebafmod[].
 *
 * Each new EBAF type must have a corresponding row entered in the
 * AMebafmod[] array which defines which modules are valid for that
 * ebaf type.
 */
 typedef enum {
     AMNOTEBAF,
     AMPVN_ETYPE,
     AMICR_ETYPE
 } AMEBAFTYPE;

 /* number of ebaf AMA types in the AMEBAFTYPS enum */
 #define AMNUMEBAFTYPES 3
```

In the *5ESS* switch, `typedef` is also used to ensure data and coding consistency. The *5ESS* switch uses a relational database, ODD, which resides in computer memory and consists of the base relations, domains, and global parameters required to support a specific software release. Relations are rectangular data tables (matrices) with rows that are called tuples and columns that are called attributes. A domain is a specific set of values that an attribute can have.

For example, the domain `CIRCUIT` is defined as:

CIRCUIT	**Internal names for hardware circuits**
TYPE	Unsigned Short
I/O TYPE	HEX

CIRCUIT	Internal names for hardware circuits
DOMAIN LENGTH	n/a
UPPER BOUND	65535
LOWER BOUND	0

The domain CIRCUIT can be used in various subsystems of the switch. In this example from the global header files, CIRCUIT is the domain and determines the construction of the type DMCIRCUIT.

```
GH:GHDR2  5D00600

/*  PTC:
    DM: DMCIRCUIT TRUE TRUE FALSE
    SB: DMCIRCUIT 0 65535
*/

/*****************************************************************
*
*  dom_name  domain description
*  --------  ------------------
*  circuit  Internal names for hardware circuits
*
*****************************************************************/

typedef  unsigned short DMCIRCUIT; /* range = 0, 65535 */
```

The DM that precedes the domain name signifies that this is a domain. The domain CIRCUIT is defined in the 235-600-2xx, *Translations and Dynamic Data Domain Descriptions Manual* and can be used throughout the switch following the guidelines given for the CIRCUIT domain.

3.1.10 Preprocessor

The C preprocessor expands the macro definition and conceptually processes the source text of a C program. It allows for string substitution, conditional compilation, and file substitution. The preprocessor looks at the program before it is sent to the compiler and replaces any symbolic abbreviations in the program with the direction they represent. It accesses called files, and the preprocessor decides how compilation will take place based on the conditional commands given to it.

Table 3.1-7 contains preprocessor commands and their definitions. The *5ESS* switch macros use all capitals in the macro names to make them more distinguishable in the code. Macro definitions are created with the #define statement and are used extensively throughout the *5ESS* switch system. The #define statement is a preprocessor directive that instructs the preprocess to do global substitutions. The #include statement, which is also used extensively in the switch, is used to make local and global header files available for use during processing.

The preprocessor is designated by a special preprocessor command # at the beginning of the line that distinguishes that line from the other code. The preprocessor examines each line independently, it does not consider relationships between lines and it does not perform any action on the C program itself.

Table 3.1-7 — Preprocessor Commands

Command	Definition
`#include`	include a source file
`#define`	define a macro
`#undef`	undefine a macro
`#if`	conditionally include text based on the value of a constant expression
`#ifdef`	conditionally include text based on whether a macro is defined
`#ifndef`	conditionally include text based on whether a macro is undefined
`#else`	conditionally include text if the `if` or `ifndef` test fails
`#elif`	like an `else-if`
`#endif`	terminate the conditional text
`#line`	reset the line number
`#error`	force an error message
`#pragma`	used for implementation-dependent control
`#`	null directive; no effect; used to bracket comments

3.1.11 Pointers and Arrays

A pointer is an object, the contents of which is the address of another object. A pointer may point to any type, including another pointer. Since pointers are a symbolic way of using an address, they can only point to the addresses of variables declared to be the same type as themselves. It may also point to a function (the pointer contains the starting address of the function) or to nowhere (a null pointer).

Pointers use three operators:

- the asterisk, `*` , the indirection operator
- the ampersand, `&` , the address operator
- the right arrow, →, the indirect membership operator.

Using the pointer `ptr` as an example, `*ptr` refers to the value stored at the pointed to address, `&ptr` refers to the address of the pointer itself, and `ptr` alone refers to the address of the object to which it points. The meaning of `ptr→a` is identical to `(*ptr).a`.

Pointers are declared in the same manner as other variables. When declared, they must include the indirection operator to inform the compiler that this variable is indeed a pointer.

```
Given:
   amount=1;
   pti=&amount;
```

Variable	Value
amount	1
&amount	16776260
pti	16776260
&pti	16776256
*pti	1

Pointers are frequently used when working with array structures. An array name can be used as a substitute for the address of the 0th element of the array. The C language uses arrays to describe a collection of variables with the same characteristics. All of the members are addressed by the same name except for the subscript number which identifies the individual element being considered.

This declaration allocates a storage location for an array of type integer. The values of the members of the array are `digits[0]=10`, `digits[1]=20`, and `digits[2]=30`.

```
int digits[3]={10,20,30};
```

The storage allocations in memory are:

10	20	30

Arrays may be accessed by their subscript number, but this can be a slow and resource-consuming process.

```
a=digits[1];

Store the value 20 in the variable a.
```

When an array must be accessed more than once, a pointer to the array is used since this method of access will use less time and fewer resources. Since arrays are stored in contiguous memory locations this method also allows for the use of pointer arithmetic.

When a pointer is initialized, the value of the pointer becomes the address of the value to which it points. If the pointer is initialized to the first element in an array, then the pointer becomes the address of that first member. Incrementing the pointer will make it point to the next element in the array. It adds the appropriate number of bytes to the pointer. The amount added to the address is dependent upon the defined data type of the pointer and its object.

```
Given:
   int numbers[5] = {10,20,30,40,50};
   int *ptr;
   ptr = numbers;
```

Array Identifier	Variable	Value
A	*ptr = 10	The address pointed to is 1048576.
		ptr++ increment pointer
B	*ptr = 20	The address pointed to is 1048580.
		ptr++ increment pointer
C	*ptr = 30	The address pointed to is 1048584.
		ptr++ increment pointer
D	*ptr = 40	The address pointed to is 1048588.
		ptr++ increment pointer
E	*ptr = 50	The address pointed to is 1048592.

&ptr - the address of the pointer itself

*ptr - the value stored at the pointed to address

ptr - the address of the object to which it points

In arrays A through E (Figures 3.1-2 through 3.1-6), the memory area containing the array numbers and the area containing the pointer ptr are shown graphically. The address 1048576 holds the value 10, 1048580 holds the value 20, etc. The value of ptr is incremented (ptr++) by 4 each time because ptr points to an int assuming an int is 4 bytes. In each example, the new values of *ptr, ptr, and &ptr are represented.

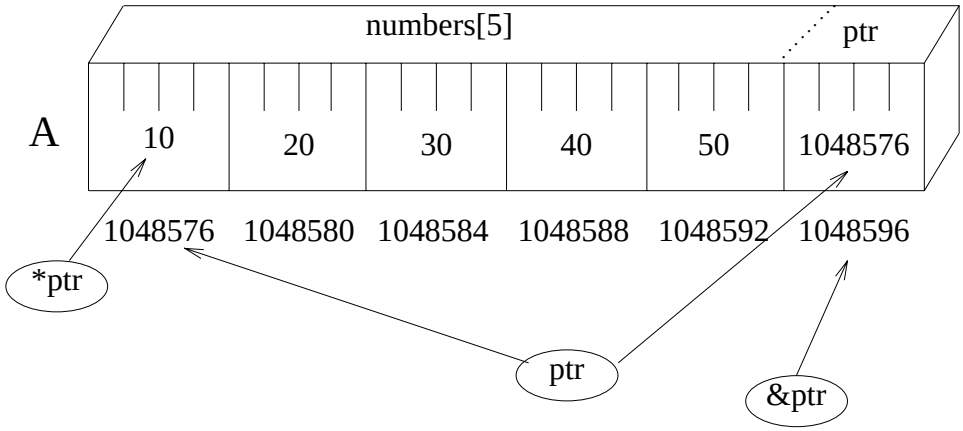


Figure 3.1-2 — Array A

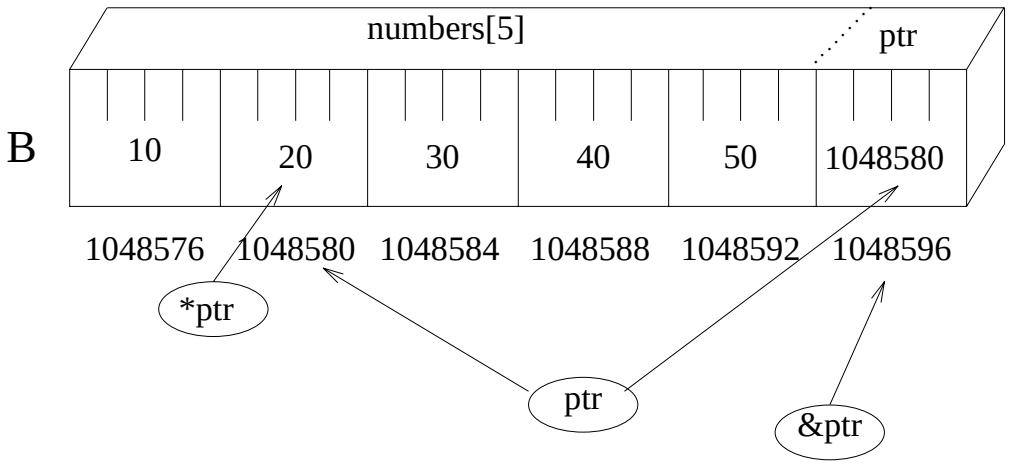


Figure 3.1-3 — Array `B`

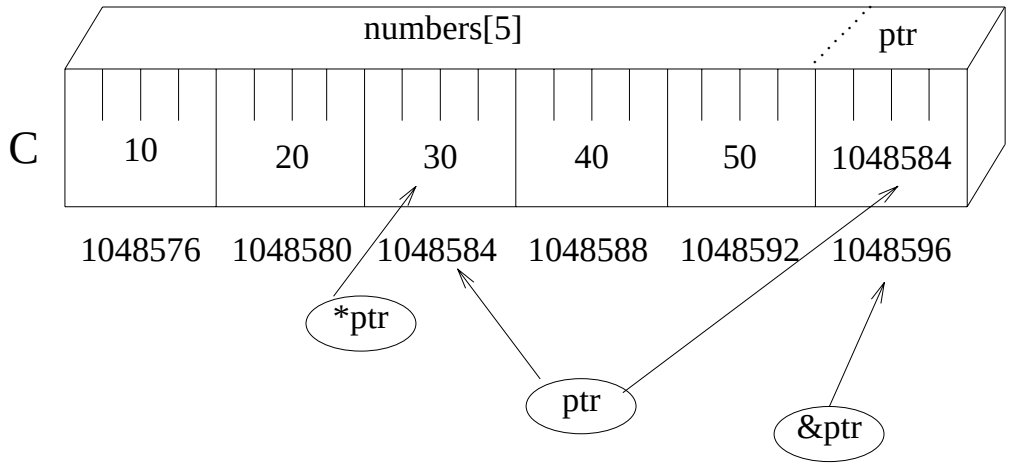


Figure 3.1-4 — Array `C`

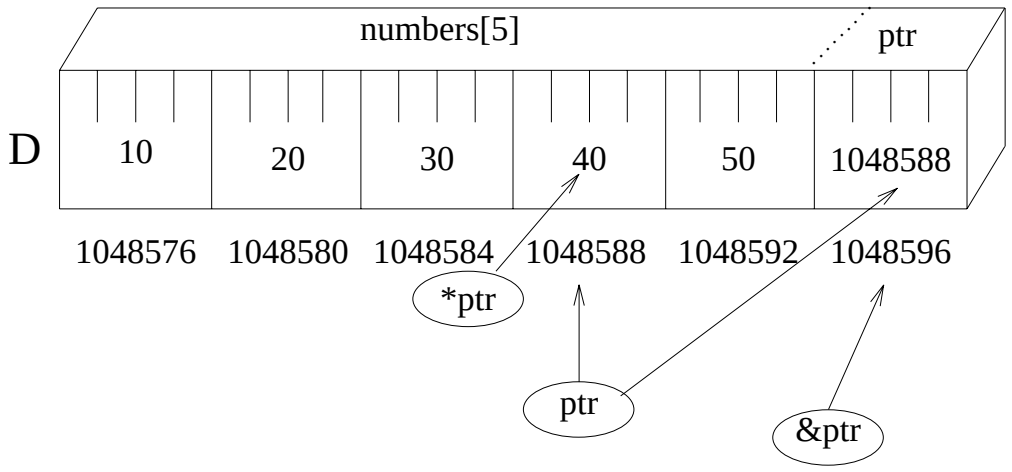


Figure 3.1-5 — Array `D`

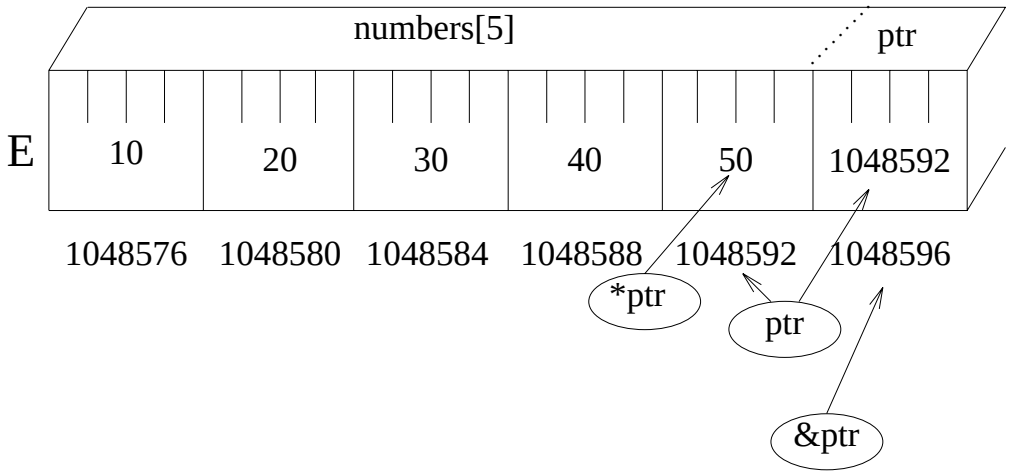


Figure 3.1-6 — Array `E`

3.1.12 Membership Operator

The membership operator, the period (`.`), is used with a structure or a union name to specify one of its members. If `one` is the name of a structure and `i` is a member specified by the structure's template, then `one.i` identifies that member of the structure.

```
struct example
{
 int i;
 char c;
```

```
  float f;
}one, two, three;
```

The indirect membership operator, →, is used with a pointer to a structure or union to identify one of the members. If `ptr` is a pointer to a structure and `i` is a member of that structure, then `ptr→i` identifies that member of the pointed to structure. A 4-byte integer is assumed.

```
Given:
 one.i=10;
 one.c='a';
 one.f=35.1234;
 ptr=one;
```

Variable	Value
one.i	10
ptr→i	10
&ptr→i	16776252
(*ptr).i	10
&ptr	16776248
one.c	a
ptr→c	a
&ptr→c	16776256
(*ptr).c	a
&ptr	16776248
one.f	35.1234
ptr→f	35.1234
&ptr→f	16776260
(*ptr).f	35.1234
&ptr	16776248

3.1.13 Casting

It is sometimes necessary to convert a variable or an expression to another type. Type conversion can be requested in the C language by using the cast operator. The operator consists of the type name surrounded by parentheses before the object of conversion. This conversion is only temporary and affects the variable or expression only when specifically referenced. For example:

```
a = 3.9 ;
b = 4.7 ;
answer = a + b ;
cast_apart = (int)a + (int)b ;
cast_together = (int)(a + b) ;
```

The value of `answer` is 8.6, the value of `cast_apart` is 7 and the value of `cast_together` is 8. Since the cast operator `int` changes the value of 3.9 to 3 and the value of 4.7 to 4, the value of `cast_apart` is 7. When the expression is computed and then the casting is performed, the value of the entire expression is cast. Therefore, the value of `cast_together` is 8. The values of the variables `a` and `b` have not changed. They remain the same numbers and type of numbers that they were before the casting.

3.1.14 Functions

The C language is built on the use of the function. A function is an independent block of program code organized to achieve a particular task. Functions can be equated to the subroutines or procedures of other languages. Every C program must contain at least one function. A function name consists of the word-identifier followed by parentheses, such as `function()`. Any function can call another function, and a function may even call itself.

The arguments of a function can be of any type (excluding arrays), no matter what the type of the function itself. The data type of a function is determined by the data type of the value that the function returns and not by the data type of the arguments given, if any. Functions are assumed by the compiler to be of type `int` unless declared otherwise. Functions have the following characteristics:

- Functions can be passed any number of arguments, and the arguments are passed by value.
- Functions can only return one value.
- Functions cannot be defined inside of another function. The function definition must be separate.

A function can be compiled and stored in a library or an archive from which it can be called by the linker. This enables all programmers to use the same standard functions without having to rewrite them every time. An example is `printf()`, the function used for printing in C.

When a function is called, any arguments that are provided by the caller are simply treated as expressions. The value of each expression is used to initialize the corresponding formal parameters in the called function, which then behave in the same way as any other local variables in the function. If a value is to be returned to the calling function the value is sent via the return statement. This concept is illustrated here.

```
main( )
{
 int i = 0;
 while (++i < 10)
   printf("%d squared is %d \n", i, square(i));
}
square(n)
int n;
{
 return(n*n);
}
```

The function `main()` begins, as all functions do, with the left hand curly bracket. It is followed by the initialization of the variable `i` to 0, which is a local automatic variable. The `while` statement that follows is executed until the value of `i` is greater than or equal to the number 10. The iterations that are performed print the squared values of the integers 1 through 9.

Inside the `printf` statement is the call to the function `square()`. It is called as `square(i)`. This function call sends the *value* of the variable `i` to the function `square()`.

```
square(n)
int n;
{
```

```
    return(n*n);
}
```

The `square()` function receives the value sent to it by the calling function. That value becomes the argument to that function, in this instance the variable `n`. The function then squares the value sent to it and returns one value to the calling function, the square of the function's argument.

The output from this program is:

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
```

Notice that the first number to be squared is `1` even though `i` was initialized to zero. The statement `++i` causes `i` to become 1 before the loop is executed the first time. The statement `++i` means increment the value of `i` and then evaluate the expression `i<10`. If the statement had read `i++`, that would have meant evaluate the expression `i<10` and then increment the value of `i`.

3.1.15 Control Statements

`if-else` The `if` section of this statement conditionally executes a statement or group of statements. The `else` section of this statement is optional and is only executed when the `if` portion of the statement is evaluated as false. The syntax of the `if` statement is:

```
if (x < a + b)
  statement;
else
  statement;
```

Both the `if` and the `else` sections may consist of more than one actual statement. For example:

```
if (x < a + b)
  {
   statement1;
   statement2;
   statement3;
  }
else
  {
   statement4;
   statement5;
   statement6;
  }
```

`while` The `while` statement is a looping structure. It performs an action or group of actions while a condition remains true. As with the `if` statement, the condition may be followed by one or more actions to be taken when the condition is satisfied.

```
while (x < a + b)
 {
  statement1;
  statement2;
```

```
    statement3;
  }
```

`for` The `for` conditional statement takes three arguments. The first is the initialization, the second is the condition under which execution will continue, and the third is the modification — the change to be made that eventually causes the loop to discontinue execution. For example:

```
for(x=1, x<100, x++)
 {
  statement1;
  statement2;
 }
```

This statement initializes the value of `x` to 1, states that the loop will continue until `x` reaches 100, and states that `x` will be incremented at the end of every loop.

`do while` The `do` conditional loop, unlike the `while` and the `for`, evaluates a condition after the loop is executed. This guarantees that the loop will execute at least once. An example is:

```
do
{
 x++
 y++
} while x<10;
```

`switch` The `switch` conditional statement is used when there are a number of possible options available. The `switch` statement is followed by a constant, a variable or an expression that evaluates to a numeric ASCII value of type `int`, which is then used to determine from a list of cases which case number (1, 2, 3 or a, b ,c) should be evaluated. The `case` component of the `switch` statement lists the actual alternatives from which to choose. It is also possible to have a default case that will activate if no other case is valid. For example:

```
main( )
{
  int i = 2;

  switch(i)
  {
   case 1:
      printf("This is case number 1.\n");
      break;
   case 2:
      printf("This is case number 2.\n");
      break;
   case 3:
      printf("This is case number 3.\n");
      break;
   default:
      printf("This is the default case.\n");
      break;
  }
 }
```

`break` and `continue`

The `break` and `continue` statements are used to redirect the flow of control inside loops and (for `break` only) in `switch` statements. The `break` statement consists of just the word `break` followed by a semicolon.

Execution of a `break` statement causes execution of the closest enclosing `while`, `do`, `for`, or `switch` statement to be terminated.

The `continue` statement consists of just the word `continue` followed by a semicolon. Execution of a `continue` statement causes execution of the body of the closest enclosing `while`, `do`, or `for` to be transferred to the end of the body, and execution of the affected iterative statement continues from that point with a re-evaluation of the loop test. In addition, any required incrementing will also be performed.

```
while (x < a + b)
 {
  if x == 20
    continue;
  statement;
  statement;
  statement;
 }
```

The `continue` statement will cause the compiler to skip the three statements following the `if` and re-evaluate the `while` condition at the beginning of the loop. This will occur any time the value of `x` is 20.

`goto` The `goto` statement forces a program's execution to move to the point marked by the stated label name. The `goto` statement cannot send control beyond the function boundaries.

3.2 *5ESS*® SWITCH DIFFERENCES FROM K AND R REFERENCE

3.2.1 Introduction to Section

The C version used by the *5ESS* switch is based on the first edition of *The C Programming Language*, 1978 by Kernighan and Ritchie, it is not based on the *ANSI*[1] C standard. This section will outline the distinctions between the text and the actual *5ESS* switch compilers.

3.2.2 `Sizeof`

The first edition of Kernighan and Ritchie states that the `sizeof` operator yields an integer. In the *5ESS* switch, the MC68, 3B20D, 3B21D, and B16 compilers actually return an unsigned integer. (The other compilers, IAPX and 3B2, return an `int` from `sizeof`.) By definition, an unsigned integer can only be positive or zero, and since the `sizeof` operator must return a positive or zero value (an object cannot have a negative size), many compilers have been changed to reflect this fact.

3.2.3 Array Size Limitations

The Kernighan and Ritchie text states that the subscript type of an array is `int`. The *5ESS* switch also uses `long` for subscripts. If type `int` is used, the size of the array is limited to 32K on a 2-byte integer compiler. An array of 80,000 bytes could not be accessed by an integer subscript, but the array could be fully accessed by a subscript that is type `long`.

It should also be noted that using a subscript of type `long` requires long arithmetic, so it takes more time and uses more space than a subscript of type `int`. Therefore, whenever possible, type `int` is the preferred data type for subscription.

3.2.4 Multiple Structure Assignments

The MC68 short integer compiler does not support cascading structure assignments. An example of a cascading structure assignment is:

```
struct maillist
{
    char fname[11];
    char lname[21 ];
    char addr[31];
    char city[16];
    char state[3];
    char zip[11];
    char committee[31];
    int years;
} bears, wolves, deer;
   .
   .
   .
bears=wolves=deer;
```

Assignments of this type are not allowed in the MC68 compiler.

3.2.5 Unsigned Data Types

The first edition of Kernighan and Ritchie only supports the unsigned type `int`. The MC68 compiler uses `unsigned int`, `char`, `short`, and `long`. The type `signed char` is not supported.

---

1. Registered trademark of American National Standards Institute.

3.2.6 `Float`/`Double` Floating Point Data Types

The MC68 compilers do not support `float` or `double`. If these types are declared, they are mapped to `long` and a warning message is generated.

3.2.7 Bitfields

The MC68 compilers support bitfields of the integral type. The *5ESS* switch contains bitfield definitions for `int`, `char`, `short`, and `long`.

3.2.8 Zero-Length Bitfields and Their Alignment

According to Kernighan and Ritchie, zero-length bitfields align on the next word boundary or `int` boundary. Because the *5ESS* switch allows bitfields to be of any integral type and not just `int`, the MC68 compiler can align on `char`, `short`, or `long` boundaries, as determined by the field's definition. Since characters use one byte of storage, aligning to the next `char` boundary saves space. This gives the programmer additional data control and flexibility.

```
struct {
    char field1     : 2 ;
    char            : 0 ;
    char field2     : 4 ;
    char            : 0 ;
    char field3     : 6 ;
    char            : 0 ;
    char field4     : 4 ;
} fields ;
```

3.2.9 Integer Bitfields and Their Alignment

According to Kernighan and Ritchie, the zero-length integer bitfield is aligned to the next integer or the next word boundary. This seeming contradiction is not a problem unless the size of the integer and the size of the word are different. In the case of the MC68 compiler series, the size of a word is two bytes and the size of the integer is two bytes. The alignment for zero length bitfields is the next integer boundary.

3.2.10 The Value of An Assignment

According to Kernighan and Ritchie, the value of an assignment is the value stored in the left operand after execution. To generate more efficient code, the MC68 compiler uses the value of the right operand after an assignment to a bitfield. This is a problem only if the value being assigned to the bitfield is too large. When the value is assigned, it is masked. The right operand, however, still contains the unmasked value.

3.2.11 Enumeration Comparisons

Enumerations in the MC68 compilers are only allowed to be compared directly by using the == and the `!=` relational operators. To use the remaining relational operators, the enumeration must first be cast to an integral type.

3.2.12 Function Templates

Function templates are function declarations that contain the function return type and parameter types. When templates are included, the compiler verifies that the arguments in any function that follow the definition match in type and number with the template definition.

- A function template can start with a storage class of `extern` or `static` or none at all (the equivalent to `extern`) and a function return type.

- Its parameter list must contain either `void`, a list of declarations, or a list of declarations followed by an ellipsis.
- Declarations do not include variable names.
- The declarations may include any previously-declared `structure`, `union`, or `enumeration` tags or `typedef` names.

A parameter list of `void` indicates that function calls should include no arguments. An empty parameter list cannot be used for this, an empty list is not considered a function template at all.

Ellipses are used to indicate that zero or more arguments may follow. This is how variable length parameter lists are specified. Arguments appearing at or after the position of the ellipsis are not type checked.

3.2.13 Flexnames

For software releases 5E11 and later, the MC68, IAPX, and 3B compilers accept variable names up to 256 characters in length.

The B16 and DSP32 compilers only distinguish the first 8 characters of a name. If the first 8 characters of variable names are identical, the B16 and DSP32 compilers interpret them as the same name. For example; `somename` and `somenametoo` are seen as `somename`.

For software release 5E10, all the compilers are limited to 8 character names. Any larger name is truncated.

3.2.14 Integer Size

Kernighan and Ritchie states that an `int` may be of any size. However, the examples used give the impression that the size of an `int` is the same as the size of a `long`. In the 2-byte integer compilers this is not true. The main problem occurs when a zero value is assigned to a pointer. Since 0 is an integer, it only uses 2 bytes of stack space. When a function looking for a pointer pops from the stack, it anticipates a 4-byte pointer and takes 2 bytes of data and 2 bytes of garbage from the stack. In the *5ESS* switch this problem is avoided. When a null pointer is passed, the value is cast to the type of the pointer and then passed.

Software Analysis Guide

CONTENTS PAGE

4. DISASSEMBLY/ASSEMBLY LANGUAGE 4-1

4.1 3B20D/3B21D PROCESSOR ASSEMBLY LANGUAGE 4.1-1
4.1.1 Values 4.1-1
4.1.2 Constants 4.1-1
4.1.3 Expressions 4.1-2
4.1.4 Machine Instruction Notation 4.1-2
4.1.5 Operands 4.1-4
4.1.6 Instruction Set 4.1-8

4.2 3B20D/3B21D PROCESSOR MACHINE DEPENDENCIES 4.2-1
4.2.1 Data Type Memory Boundaries 4.2-1
4.2.2 Arithmetic Types Supported 4.2-3
4.2.3 Data Conversion Rules 4.2-4
4.2.4 Memory Configuration 4.2-4
4.2.5 Register Notation 4.2-6
4.2.6 Stack Usage 4.2-7

4.3 *MOTOROLA* MC68000 PROCESSOR ASSEMBLY LANGUAGE . . 4.3-1
4.3.1 Values 4.3-1
4.3.2 Constants 4.3-1
4.3.3 Expressions 4.3-2
4.3.4 Machine Instruction Notation 4.3-3
4.3.5 Operands 4.3-4
4.3.6 Instruction Set 4.3-8

4.4 *MOTOROLA* MC68000 PROCESSOR MACHINE DEPENDENCIES 4.4-1
4.4.1 Data Type Memory Boundaries 4.4-1
4.4.2 Arithmetic Types Supported 4.4-2
4.4.3 Data Conversion Rules 4.4-3
4.4.4 Memory Configuration 4.4-3
4.4.4.1 Overview *MOTOROLA* MC68XXX Processor Memory Configuration 4.4-3
4.4.4.2 Optimization 4.4-4
4.4.5 Register Notation 4.4-6
4.4.5.1 Register Classes 4.4-6
4.4.5.2 General Purpose Registers 4.4-6
4.4.5.3 Special Registers. 4.4-7
4.4.6 Stack Usage 4.4-7

4.4.7 Unsupported *MOTOROLA* MC68XXX Processor Instructions 4.4-14

4.5 *INTEL* 80186 PROCESSOR ASSEMBLY LANGUAGE 4.5-1
4.5.1 Values 4.5-1
4.5.2 Constants 4.5-1
4.5.3 Expressions 4.5-2
4.5.4 Machine Instruction Notation 4.5-3
4.5.5 Operands 4.5-4
4.5.6 Instruction Set 4.5-6

4.6 *INTEL* 80186 PROCESSOR MACHINE DEPENDENCIES 4.6-1
4.6.1 Data Type Memory Boundaries 4.6-1
4.6.2 Arithmetic Types Supported 4.6-2
4.6.3 Data Conversion Rules 4.6-3
4.6.4 Memory Configuration 4.6-3
4.6.5 Register Notation 4.6-5
4.6.6 Stack Usage 4.6-7

LIST OF FIGURES

Figure 4.1-1 — 3B20D and 3B21D Processor Instruction — Memory Layout 4.1-8

Figure 4.1-2 — 3B20D and 3B21D Processor Example Instruction Memory Layout 4.1-8

Figure 4.2-1 — 3B20D and 3B21D Processor Stack –– Memory Boundaries 4.2-3

Figure 4.2-2 — *UNIX* RTR Operating System Process Segments 4.2-5

Figure 4.2-3 — 3B20D and 3B21D Processor Stack Frame Organization . . 4.2-8

Figure 4.2-4 — 3B20D and 3B21D Processor Stack Function A Calls Function B 4.2-9

Figure 4.2-5 — 3B20D and 3B21D Processor Stack Function B Takes Control. 4.2-10

Figure 4.3-1 — Effective Address — Memory Layout 4.3-5

Figure 4.3-2 — *Motorola* MC68000 Processor Instruction — Memory Layout 4.3-6

Figure 4.3-3 — *Motorola* MC68000 Processor Example Instruction Memory Layout 4.3-9

Figure 4.4-1 — *Motorola* MC68000 Processor Stack –– Memory Boundaries 4.4-2

Figure 4.4-2 — Stack Frame — One Function 4.4-8

Figure 4.4-3 — Stack Frame — Arguments Pushed Onto Stack 4.4-9

Figure 4.4-4 — Stack Frame — PC Pushed Onto Stack 4.4-10

Figure 4.4-5 — Stack Frame — After `LINK` Statement 4.4-11

Figure 4.4-6 — Stack Frame — Register Variables Pushed Onto Stack . . . 4.4-12

Figure 4.4-7 — Stack Frame — After The `UNLK` Statement 4.4-13

Figure 4.4-8 — Stack Frame — Return To Calling Function 4.4-14

Figure 4.5-1 — *Intel* 80186 Processor Instruction — Memory Layout 4.5-5

Figure 4.5-2 — *Intel* 80186 Processor Example Instruction Memory Layout 4.5-6

Figure 4.6-1 — *Intel* 80186 Processor Stack –– Memory Boundaries 4.6-2

Figure 4.6-2 — Memory Addressing in the *Intel* 80186 Processor 4.6-4

Figure 4.6-3 — Status Word or Flags 4.6-7

Figure 4.6-4 — iAPX-16 bit, Direct Linkage (No TV) 4.6-8

Figure 4.6-5 — iAPX-16 bit, Transfer Vector Linkage 4.6-9

Figure 4.6-6 — iAPX-20 bit, Transfer Vector and Direct Linkage 4.6-9

Figure 4.6-7 — iAPX Stack Frame after Function Call 4.6-11

Figure 4.6-8 — iAPX Stack Frame during Called Function (B) Execution. 4.6-12

LIST OF TABLES

Table 4.1-1 — 3B20D and 3B21D Processor Register Notation 4.1-2

Table 4.1-2 — 3B20D and 3B21D Processor Hardware Registers 4.1-3

Table 4.1-3 — 3B20D and 3B21D Processor Memory Address Modes . . . 4.1-5

Table 4.1-4 — 3B20D and 3B21D Processor Immediate Address Modes . . 4.1-5

Table 4.1-5 — 3B20D and 3B21D Processor Register Address Modes . . . 4.1-5

Table 4.1-6 — 3B20D and 3B21D Processor Operand Encoding — Displacement Mode 4.1-6

Table 4.1-7 — 3B20D and 3B21D Processor Operand Encoding — Displacement Deferred Mode 4.1-6

Table 4.1-8 — 3B20D and 3B21D Processor Operand Encoding — External Address Mode 4.1-6

Table 4.1-9 — 3B20D and 3B21D Processor Operand Encoding — External Address Deferred Mode 4.1-7

Table 4.1-10 — 3B20D and 3B21D Processor Operand Encoding — Absolute Address Mode 4.1-7

Table 4.1-11 — 3B20D and 3B21D Processor Operand Encoding — Absolute Address Deferred Mode 4.1-7

Table 4.1-12 — 3B20D and 3B21D Processor Operand Encoding — Immediate Address Mode 4.1-7

Table 4.1-13 — 3B20D and 3B21D Processor Operand Encoding — Register Address Mode 4.1-7

Table 4.2-1 — 3B20D and 3B21D Processor Data Sizes and Alignment . . . 4.2-2

Table 4.2-2 — 3B20D and 3B21D Processor Data Conversions Rules . . . 4.2-4

Table 4.2-3 — 3B20D and 3B21D Processor Register Notation 4.2-7

Table 4.3-1 — *Motorola* MC68000 Processor General Purpose Registers 4.3-3

Table 4.3-2 — *Motorola* MC68000 Processor Control Registers 4.3-3

Table 4.3-3 — *Motorola* MC68000 Processor Addressing Mode — Effective Address Contents 4.3-4

Table 4.3-4 — *Motorola* MC68000 Processor Operand Address Modes . . . 4.3-6

Table 4.4-1 — *Motorola* MC68000 Processor Data Sizes and Alignment . . . 4.4-1

Table 4.4-2 — *Motorola* MC68000 Processor Data Conversions Rules . . . 4.4-3

Table 4.4-3 — *Motorola* MC68000 Processor General Purpose Registers 4.4-6

Table 4.4-4 — Data Registers 4.4-7

Table 4.4-5 — Unsupported *Motorola* MC68020/MC68030 Processor Instructions 4.4-14

Table 4.5-1 — iAPX Word Registers. 4.5-3

Table 4.5-2 — iAPX 8 Byte Registers 4.5-4

Table 4.5-3 — *Intel* 80186 Processor Effective Address — Mode Field . . . 4.5-5

Table 4.5-4 — *Intel* 80186 Processor Effective Address — Register/Memory Field 4.5-6

Table 4.6-1 — *Intel* 80186 Processor Data Sizes and Alignment 4.6-1

Table 4.6-2 — *Intel* 80186 Processor Data Conversions Rules 4.6-3

Table 4.6-3 — *Intel* 80186 Processor Data Registers 4.6-5

Table 4.6-4 — *Intel* 80186 Processor Pointer and Index Registers 4.6-5

Table 4.6-5 — *Intel* 80186 Processor Segment Registers 4.6-6

Table 4.6-6 — *Intel* 80186 Processor Status and Control Registers 4.6-6

4. DISASSEMBLY/ASSEMBLY LANGUAGE

This section of the *Software Analysis Guide* is a review of the Assembly Language and a guide to understanding the disassembly statements used in the system dumps and assert messages. This section is divided into three parts. The first part is concerned with the 3B20D and 3B21D processors, the second discusses the *Motorola*[1] MC68000 processor family, and the third applies to the *Intel*[2] 80186 processor. The 3B20D and 3B21D processors are used in the administrative module (AM). The MC68000 processor family is used in the switching modules (SM), the communications module processor (CMP) and the protocol handlers (PH) of the *5ESS*® switch. The *Intel* 80186 processor is used in the packet interface (PI).

1. Registered trademark of Motorola Inc.
2. Registered trademark of Intel Corporation.

4.1 3B20D/3B21D PROCESSOR ASSEMBLY LANGUAGE

4.1.1 Values

Values, that is constants, expressions, and labels, are represented in the 3B20D and 3B21D processors by one of the following types:

`UNDEFINED` An `UNDEFINED` value is either a program label that has been encountered or it is a variable that has been defined in another program which will be linked to this program segment.

`ABSOLUTE` An `ABSOLUTE` value never changes. It can be either a constant or an expression which is evaluated by the assembler.

`TEXT` Tells the position of a `TEXT` area variable in memory. When a segment of `TEXT` program is moved, the value of each `TEXT` variable is changed to reflect the new location.

`DATA` Gives the position of a `DATA` variable in memory. When a segment of `DATA` program is moved, the value of each `DATA` variable is changed to reflect the new location.

`BSS` Gives the position of a `BSS` area variable in memory. When a segment of the `BSS` program is moved, the value of each `BSS` variable is changed to reflect the new location.

4.1.2 Constants

Constants are fixed values. The values can be expressed in any of three different number systems: decimal, octal, or hexadecimal.

Decimal The decimal number system is composed of the digits 0 through 9. Each number must begin with a non-zero digit.

```
1234
 325
  43
```

Octal The octal number system is composed of the digits 0 through 7. Each number must begin with a zero digit.

```
 077
0123
0342
```

Hexadecimal
The hexadecimal number system is composed of the digits 0 through f. Each number must begin with a 0x or 0X prefix. The digits a through f can be represented by either upper or lower case letters.

```
  0x3f
0x8ABC
 0xFEA
```

4.1.3 Expressions

An expression is a group of operands separated by operators. Expressions are evaluated from the left to the right, and all operators have equal precedence. If part of the expression is to be evaluated before the rest, it should be enclosed in a set of parentheses. Expressions will be calculated from the innermost parentheses out. After all enclosed elements are evaluated, the remainder of the expression is evaluated from left to right. The following operators are available.

+ Adds two operands. One operand must be absolute (independent of location). The second can be any type. The sum is given the type of the second operand.

– Subtracts the right from the left. If the right operand is absolute, the answer is given the type of the left operand. If the right operand is not absolute, both must be the same type (neither can be undefined) and the answer will be absolute.

× Multiplies two operands. Both must be type absolute, and the answer will also be type absolute.

/ Divides the left operand by the right operand. Both operands must be absolute, and the answer will also be absolute.

4.1.4 Machine Instruction Notation

The machine instructions are mnemonic representations of 3B20D and 3B21D processor machine language. Assembly language for the 3B20D and 3B21D computers allows access to 12 32-bit general purpose registers (0 through 11). Registers 0 through 8 are general, register 9 is the argument pointer, register 10 is the frame pointer and register 11 is the stack pointer. See Table 4.1-1.

Table 4.1-1 — 3B20D and 3B21D Processor Register Notation

Symbolic Name	Description
%rn	the [n] in %rn represents a number 0 through 8
%ap	argument pointer
%fp	frame pointer
%sp	stack pointer

Stack Pointer
The stack pointer is the address of the next available word on the stack.

Frame Pointer
The frame pointer points to the first automatic variable in the stack frame.

Argument Pointer
The argument pointer is the address of the first argument of the function.

There are also hardware registers in the Administrative Module which are also referenced in the *5ESS*® switch. See Table 4.1-2 for a complete listing.

Table 4.1-2 — 3B20D and 3B21D Processor Hardware Registers

Hardware Register	Description	Hardware Register	Description
%BGR	Bidirectional Gating Register	%HM	Halfword Multiplexer
%PSW	Program Status Word	%CDR	Channel Data Register
%PPR	Pulse Point Register	%IS	Interrupt Set Register
%SAR	Store Address Register	%IM	Interrupt Mask Register
%PA	Program Address Register	%SSR	System Status Register
%SCR	Store Control Register	%ER	Error Register
%SDR	Store Data Register	%RTC	Real Time Clock
%SIR	Store Instruction Register	%HSR	Hardware Status Register
%IB	Instruction Buffer	%CAR	Channel Address Register

- The Program Status Word is used by the system software to set and maintain the status of the currently executing program. The Program Status Word register is a 32-bit register used to control program function and record program status. The program status word register is loaded from the destination bus under microprogram control. The program status word register outputs are readable and therefore testable by microcode.
- The Interrupts Set Register is a 32-bit register whose bits may be set by external signals (interrupts) or by microprogram control. The bits are only cleared by the microprogram. When a bit is set in the interrupt set register and recognized by the processor, the section specified for that particular interrupt bit is taken. The Interrupt Set Register logs interrupts from 32 sources. The interrupts in the least significant bit of the IS have the highest priority.
- The Interrupt Mask Register allows the system to ignore specific interrupts. The interrupt mask register is a 32-bit register whose bits are set or cleared by the microprogram. Each bit in the interrupt mask corresponds to the same bit in the interrupt set. Setting of any bit in the interrupt mask prevents the recognition of that corresponding interrupt when it appears in the interrupt set.
- The System Status Register is a 32-bit register that contains processor status information such as system configuration, maintenance and recovery information, and inputs from certain manual switches. Some system status register bits are loaded from the destination bus, while others are read-only. The System Status Register controls the status of the system configuration.
- The Error Register is an error detecting device which is comprised of 32 bits. Bits 0 through 10 are used for stop-and-switch type errors and main store parity errors. The Error Register logs errors from various points in the system. The least significant bits of the ER have the highest priority and the most significant bits the lowest priority.
- The Real Time Clock is a 32-bit synchronous counter normally incremented at 1-ms rate. The 1-ms time interval, which is derived from the real-time counter

prescaler, is loaded from the destination bus and accessed under microprogram control. The real-time clock may be single stepped by forcing the appropriate maintenance state (bits 26 through 31 of the hardware status register) and toggling bit 25 of the pulse point register. A maintenance state bit can be used to inhibit the counting of the real-time clock. The real-time clock is also used as a system clock when time, day, or date is requested.

- Various timers are available to the system. These timers can be read or written by manipulating the proper bits in the timer register.
- The Hardware Status Register is a 32-bit register that contains hardware and control status information. It may be loaded from the destination bus under microprogram control, except for bits 4 through 7 which are read-only. It controls an assortment of hardware throughout the system. Firmware will manipulate different bits in this register, depending on the desired results.

4.1.5 Operands

3B20D and 3B21D processor assembly language statements consist of opcodes and operands. Opcodes define computer actions, operands tell a computer where to act. Each operand can have a location in memory and a value. The value stored in the memory location is used to execute the instruction. An assembly instruction may also have the address of a memory location as an operand and an address may be provided to store results in memory. Occasionally the value is provided by the operand itself. In this case, no memory location is needed. In the 3B20D and 3B21D processors all 12 machine registers can be used as operands.

The 3B20D and 3B21D computers use three categories of address modes:

Memory mode
: Memory mode operands give the memory location of data for the instruction. The value used to execute the command is read from the address given.

Register mode
: Register mode operands specify the register in which the particular value is located.

Immediate mode
: Immediate mode operands provide the value used to execute an instruction.

Both memory mode and register mode operands may specify destinations for finished calculations as well as for sources of data. See Tables 4.1-3 through 4.1-5 for more detail.

Table 4.1-3 — 3B20D and 3B21D Processor Memory Address Modes

Address Mode	Description
`exp(reg)`	**Displacement Mode:** The value of `exp` is added to the contents of the specified register. This gives the location where the operand value is stored.
`*exp(reg)`	**Displacement Deferred Mode:** The value of `exp` is added to the contents of the specified register to determine the location where the address of the required data is stored.
`exp`	**External Address Mode:** The value of `exp` is added to the `pa` (program address) to obtain a new address.
`*exp`	**External Address Deferred Mode:** The value of `(exp+pa)` provides a location in memory that contains the address of the required data. The assembler interprets the operand to determine a physical address.
`$exp`	**Absolute Address Mode:** The value of `exp` is the literal memory location.
`*$exp`	**Absolute Address Deferred Mode:** The value of `exp` provides a location in memory that contains the address of the required data.

Table 4.1-4 — 3B20D and 3B21D Processor Immediate Address Modes

Address Mode	Description
`&exp`	The value of `exp` is the required data. There is no address, so an assembly error will occur if the operand is used as a destination or if an address is requested. Immediate address mode should be used when data is included for the program to manipulate directly.

Table 4.1-5 — 3B20D and 3B21D Processor Register Address Modes

Address Mode	Description
`reg`	The register mode is used when it is necessary to include data from a register for the program to manipulate.

In addition to the address modes, the 3B20D and 3B21D processors also use operand encoding. Each address mode also has a set of possible operand encodings. They are listed in Tables 4.1-6 through 4.1-13.

Table 4.1-6 — 3B20D and 3B21D Processor Operand Encoding — Displacement Mode

Displacement Mode		
`exp(%r)`	0	Indicates the 8-bit positive displacement mode ($R+I_8$). This encoding is an optimization. The 8-bit unsigned positive quantity (9 bits with the sign bit stripped) is added to the contents of the specified register R to determine the address.
`-exp(%r)`	2	Indicates the 8-bit negative displacement mode ($R-I_8$). This encoding is an optimization. The 8-bit unsigned negative number (9 bits with the sign bit stripped) is subtracted from the contents of the specified register R to determine the required address.
`exp(%r)`	4	Indicates the 26-bit displacement mode ($R+I_{26}$). The 26-bit signed quantity is added to the contents of the specified register R to determine the required address.

Table 4.1-7 — 3B20D and 3B21D Processor Operand Encoding — Displacement Deferred Mode

Displacement Deferred Mode		
`*exp(%r)`	1	Indicates the 8-bit positive displacement deferred mode ($R+I_8$). This encoding is an optimization. The 8-bit unsigned positive quantity (9 bits with the sign bit stripped) is added to the contents of the specified register R to determine the location of the required address.
`-*exp(%r)`	3	Indicates the 8-bit negative displacement deferred mode ($R-I_8$). This encoding is an optimization. The 8-bit unsigned negative quantity (9 bits with the sign bit stripped) is subtracted from the contents of the specified register R to determine the location of the required address.
`*exp(%r)`	5	Indicates the 26-bit displacement deferred mode ($R+I_{26}$). The 26-bit signed quantity is added to the contents of the specified register R to determine the location of the required address.

Table 4.1-8 — 3B20D and 3B21D Processor Operand Encoding — External Address Mode

External Address Mode		
`exp`	6	Indicates a 26-bit external address mode. The 26-bit signed quantity is added to or subtracted from the contents of `pa`, the program address register.

Table 4.1-9 — 3B20D and 3B21D Processor Operand Encoding — External Address Deferred Mode

External Address Deferred Mode		
*exp	7	Indicates a 26-bit external deferred address mode. The 26-bit signed quantity is added to or subtracted from the contents of pa, the program address register.

Table 4.1-10 — 3B20D and 3B21D Processor Operand Encoding — Absolute Address Mode

Absolute Address Mode		
$exp	8	Indicates a 26-bit absolute address.

Table 4.1-11 — 3B20D and 3B21D Processor Operand Encoding — Absolute Address Deferred Mode

Absolute Address Deferred Mode		
*$exp	9	Indicates a 26-bit absolute address.

Table 4.1-12 — 3B20D and 3B21D Processor Operand Encoding — Immediate Address Mode

Immediate Address Mode		
&exp	0xA	Indicates an optimized encoding mode. The 12-bit unsigned positive quantity (13 bits with the sign bit stripped) represents the required quantity.
−&exp	0xB	Indicates an optimized encoding mode. The 12-bit unsigned negative quantity (13 bits with the sign bit stripped) represents the required quantity.
&exp	0xC	Indicates either a 16-bit or a 32-bit signed quantity.

Table 4.1-13 — 3B20D and 3B21D Processor Operand Encoding — Register Address Mode

Register Address Mode		
%r	0xE	The register contains the actual data needed.

Opcode The opcode is the component of the instruction which tells the computer what to do with the data or register information.

Opcode Subcode
The opcode subcode identifies a particular member of the opcode family.

Address Mode
This address mode will be a number between 0 and E, the explanation of which can be found in Tables 4.1-6 through 4.1-13.

Destination Operand
The operand that contains the address in which the result of the operation will be located.

Source Operand
The operands (data or address) that will be used as a source for the operation.

The 3B20D and 3B21D processors use a 32-bit instruction. The layout of that instruction is shown in Figure 4.1-1.

Note: Depending on the type of the operand, either operand could be the source or the destination. Instruction `MOVW %r0,*x4(%fp)` would have the opcode 51. Instruction `MOVW *x4(%fp),%r0` would have the opcode 54.

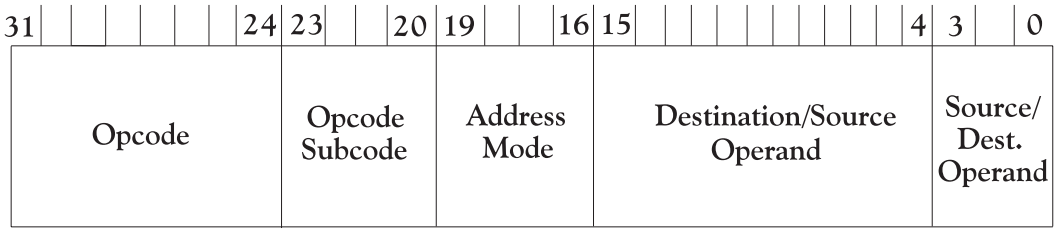
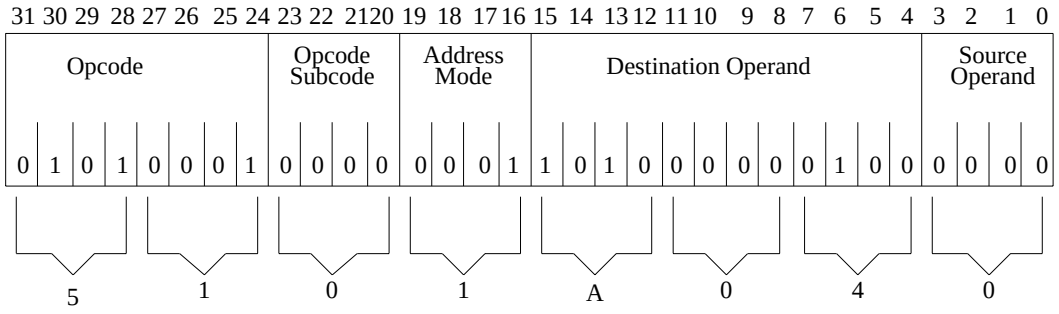


Figure 4.1-1 — 3B20D and 3B21D Processor Instruction — Memory Layout

4.1.6 Instruction Set

The instruction set for the 3B20D and 3B21D processors is based on the IS25 instruction set. The operations are performed on bytes, words, and halfwords. The `add` instruction is either: `addb2` for one byte, `addh2` for a halfword, or `addw2` for a word. All of these addition instructions use two operands.

Instruction: MOVW %r0,*0x4(%fp)



Hexadecimal: 5101 A040

Figure 4.1-2 — 3B20D and 3B21D Processor Example Instruction Memory Layout

The right-most digit of the first halfword shown in Figure 4.1-2, (5101) is the address mode. The move word (movw) instruction in the example places the contents of %r0 into a memory location. To determine where to put the contents of %r0, a 2-step process occurs. An intermediate address is first calculated by adding hexadecimal 4 to the contents of the %fp (frame pointer) register. The value found at the intermediate address is used as the destination address. The contents of %r0 are placed in the memory location at the destination address. When assembled into object code, the 8-bit positive displacement mode (optimization) was used as indicated by M=1, (510**1** A040). This mode was selected by the assembler since hexadecimal 4 is positive and does not exceed 8 bits in length.

For a complete listing of the instruction set for 3B20D and 3B21D computers see Appendix A2 - 3B20 and 3B21 Computer Instruction List.

4.2 3B20D/3B21D PROCESSOR MACHINE DEPENDENCIES

4.2.1 Data Type Memory Boundaries

The processors in the *5ESS*® switch have specific memory and organizational requirements, each processor having variations in size and functionality. The following gives a brief explanation of the byte and word organization of the 3B20D and 3B21D processors.

Byte-length data which contains 8-bits:

- can start in any memory location
- represents an 8-bit string or binary number between 0 and 255.

Halfword-length data which contains 16 bits:

- can start in any even-numbered memory location
- represents a 16-bit string or unsigned binary number between 0 and 65,535 or 2s complement binary number between –32,768 and 32,767 (bit 15 is the sign bit).

Word-length data which contains 32-bits:

- can start at memory locations that are divisible by 4 only
- represents a 32-bit string or unsigned binary number between 0 and 4,294,968,295 or 2s complement binary number between –2,147,483,648 and 2,147,483,647 (bit 31 is the sign bit).

Table 4.2-1 details the data size and alignment of the 3B20D and 3B21D processors. When analyzing output from this processor it is important to reference the data types according to the contents of this table.

Table 4.2-1 — 3B20D and 3B21D Processor Data Sizes and Alignment

Data Type	Size	Memory Alignment
`char`	1 byte	no alignment
`short`	2 bytes	2 byte boundary
`int`	4 bytes	4 byte boundary
`long`	4 bytes	4 byte boundary
`pointer`	4 bytes	4 byte boundary
`structure`	4 byte multiple	4 byte boundary
`union`	4 byte multiple	4 byte boundary
`array`	same as element type	same as element type
`bitfield`	up to the maximum of type declared	same as declared type
data inside `structures` and `unions`:		
`long`	4 bytes	4 byte offset
`pointer`	4 bytes	4 byte offset
`structure`	4 byte multiple	4 byte offset
`union`	4 byte multiple	4 byte boundary
`bitfield`	up to the maximum of type declared (inside structures only)	same as declared type

The visual examples in Figure 4.2-1 may help to explain the memory boundaries that would be established by the 3B20D and 3B21D processors.

```
Given:
  char a
  short b
  int c
  long d
  pointer e
```

The data would be placed on the stack as follows:

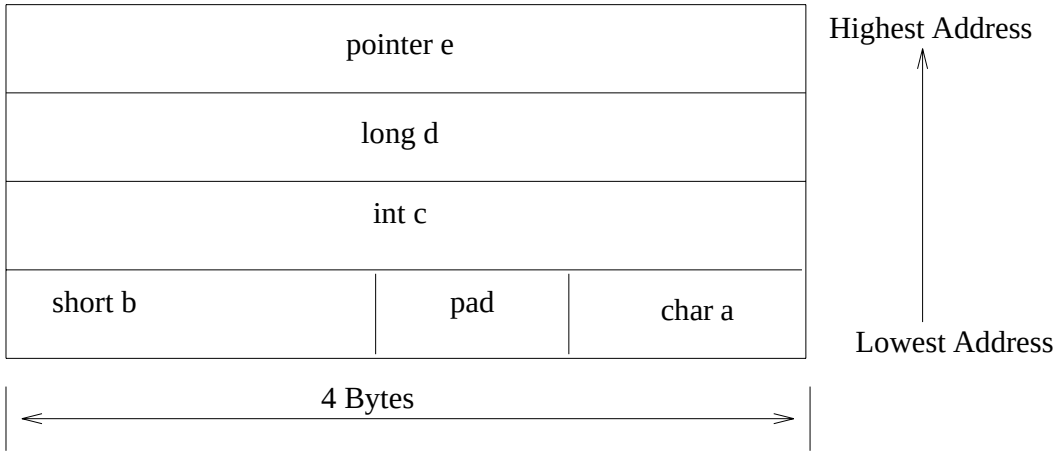


Figure 4.2-1 — 3B20D and 3B21D Processor Stack –– Memory Boundaries

4.2.2 Arithmetic Types Supported

The 3B20D and 3B21D compilers support four arithmetic types. All four types can be `signed` or `unsigned`.

- `char`
- `short`
- `int`
- `long`

The `char` and `unsigned char` are defined as 1 byte. The `unsigned` and `signed char` are the same, since with a character-type conversion, sign extension is not implemented. They can be located on any byte boundary.

The `short int` and `unsigned short int` have an implementation of 2 bytes. These arithmetic types can only be located on a two-byte boundary or even-byte boundary.

The `int`, `unsigned int`, `long`, and `unsigned long` have an implementation of 4 bytes. These arithmetic types can be started at any location that is divisible by four.

Pointers are allocated 4 bytes of memory and must be located on a byte boundary that is divisible by 4.

4.2.3 Data Conversion Rules

Table 4.2-2 — 3B20D and 3B21D Processor Data Conversions Rules

Data Type Conversion	Char	Unsigned Char	Short	Unsigned Short	Int	Unsigned Int	Long	Unsigned Long	Pointer
Char	—	NC	PL	PL	PL	PL	PL	PL	PL
Unsigned Char	NC	—	PL	PL	PL	PL	PL	PL	PL
Short	TL	TL	—	NC	SE	PL	SE	PL	PL
Unsigned Short	TL	TL	NC	—	PL	PL	PL	PL	PL
Int	TL	TL	TL	TL	—	NC	NC	NC	NC
Unsigned Int	TL	TL	TL	TL	NC	—	NC	NC	NC
Long	TL	TL	TL	TL	NC	NC	—	NC	NC
Unsigned Long	TL	TL	TL	TL	NC	NC	NC	—	NC
Pointer	TL	TL	TL	TL	NC	NC	NC	NC	—
NC = Name Change, TL = Truncate on Left, SE = Sign Extend, PL = Pad on Left with zero(s)									

4.2.4 Memory Configuration

Memory management is used for locking or unlocking, growing or shrinking, and swapping the memory segment. Memory management allows several processes to coexist, even though the sum of their memory requirement is larger than the physical memory in the main store. The memory manager which resides in the kernel makes this possible by swapping a process or part of a process between the main store and disk, or swap space. Memory management also protects against misuse such as writing into read-only memory, and unauthorized access by other processes.

Memory management centers around the segment. (See Figure 4.2-2.) A segment is a contiguous piece of virtual memory from 1 to 128K bytes long. Segments are created in main memory by the operating system on demand and disappear when they are no longer needed. A segment is a set of logically related pages. A page is 2048 bytes of contiguous main memory that always begins on an address that is a multiple of 2048. Although the hardware allocates pages in complete units, the software allows a segment to be as small as 1 byte or as large as 128K bytes or 64 pages. The pages that belong to a segment do not have to be physically contiguous. A process is made up of a collection of segments. A process can have up to 512 segments, that is, 512 logically related portions of main memory with consecutive virtual addresses. The operating system supports five types of segments:

- Process control block
- Text
- Data
- Transfer vector
- Stack.

All processes must contain a process control block, text, and stack. The process control block segment contains unique information that identifies the process to the operating system. This information includes the process identification number, type of process,

priority and address space qualifiers that define the virtual address space for a process. Text segments contain the actual executable programs. Data segments contain the information necessary for the text segments to perform their functions. The transfer vector segment supports function replacement by adding a level of indirection to the function call. The stack segment is memory that supports dynamic allocation and freeing of memory for subroutines.

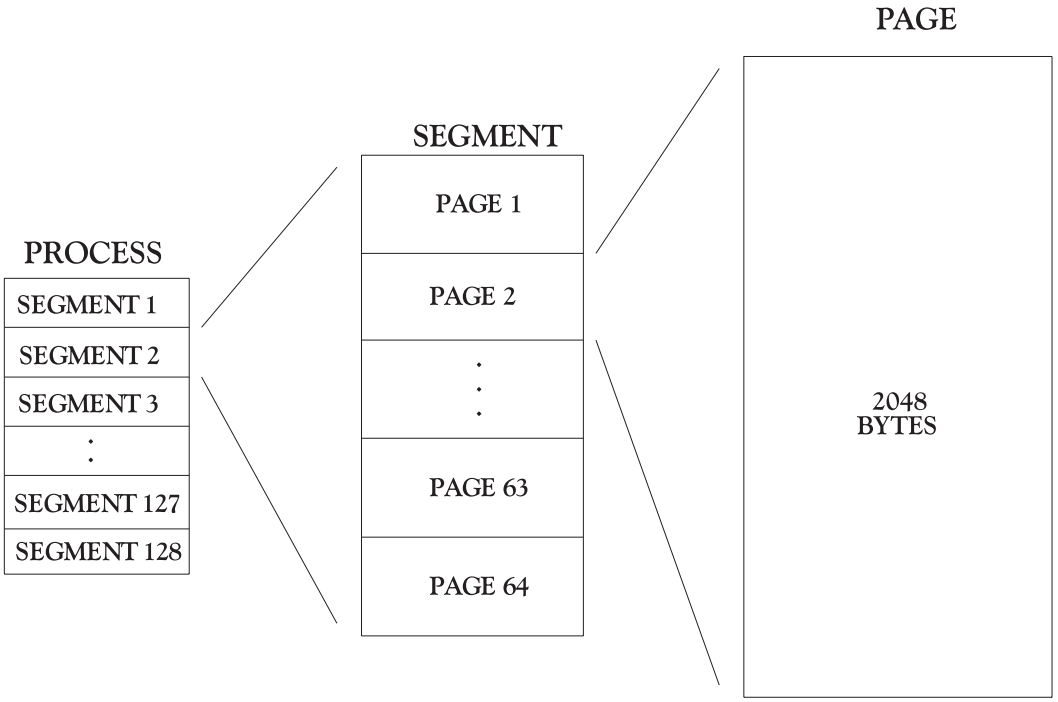


Figure 4.2-2 — *UNIX*[1] RTR Operating System Process Segments

Virtual memory is, for purposes of allocation, partitioned into configured and unconfigured memory. The default condition is to treat all virtual memory as configured. Unconfigured memory is treated as reserved or unusable by the link editor. The link editor combines object files into one, performs relocation, resolves external symbols and supports symbol table information for symbolic debugging. The link editor considers the 3B20D and 3B21D computers to have an address range of one million (hex) bytes, numbered from 0x0000000 to 0x0ffffff (0x3ffffff in VLMM - Very Large Memory Management). This comprises the virtual address space into which all input files are linked. Nothing can be linked into unconfigured memory. Thus, specifying a certain memory range as unconfigured is one way of marking the addresses in that range as illegal or nonexistent with respect to the linking process. Unless otherwise specified, discussion of memory and addresses is, with respect to the configured section of the 3B20D and 3B21D computers, virtual space.

`MEMORY` directives specify the total size of the virtual space of the 3B20D and 3B21D computers and the configured and unconfigured areas of the virtual space. By means

1. Registered trademark of The Open Group.

of MEMORY directive, an arbitrary name of up to eight characters is assigned to a virtual address range. Output sections can be bound to virtual addresses within specific named memory areas.

When MEMORY directives are used, all virtual memory not described in a MEMORY directive is considered to be unconfigured. Unconfigured memory cannot be used in the link editor's allocation process, and hence nothing can be link edited, bonded or assigned to any address within unconfigured memory.

As an option of the MEMORY directive, attributes may be associated with a named memory area. This permits restricting an output section as to where it can be bound.

The attributes that are currently accepted are:

R	Readable memory
W	Writable memory
X	Executable memory
I	Initializable memory

If no attributes are specified in a MEMORY directive, or if no MEMORY directives are supplied, memory areas assume the attributes of W, R, I, and X.

To conserve memory space, the assembler may choose an optimized address mode or an optimized instruction. In the optimized address mode, the assembler chooses an address mode option that best uses memory space. The goals of the optimizer are to:

1. reduce the object's text size
2. reduce the real time used by the object
3. reduce the stack space of the object
4. conserve host CPU time.

The following example contains an optimized address mode.

```
movw %r0,0x10(%fp)
```

The object code assigned by the assembler for this instruction is 5100 a100. The last digit of the first halfword (0) is the address mode. Because hexadecimal 10 does not exceed 8 bits in length, the optimized mode was selected.

A few instructions are optimized without address modes because their operands need less memory space. The following is an optimized instruction.

```
movw %r3,%r7
```

Each register needs just 4 bits to be properly represented. Therefore, the instruction uses 16 bits of memory instead of the usual 32 bits. The object code for this instruction would be 1437.

The key purpose of optimization is to allow memory space to be used more efficiently.

4.2.5 Register Notation

Assembly language for the 3B20D and 3B21D computers allows access to twelve 32-bit general purpose registers. The percent sign (%) is the symbol for a general purpose register. The names of the available registers are given in Table 4.2-3.

Table 4.2-3 — 3B20D and 3B21D Processor Register Notation

Symbolic Name	Machine Register Number	Description
%r0	0	Scratch
%r1	1	Scratch
%r2	2	Scratch
%r3	3	General
%r4	4	General
%r5	5	General
%r6	6	General
%r7	7	General
%r8	8	General
%ap	9	Argument pointer
%fp	10	Frame pointer
%sp	11	Stack pointer

Machine registers 12, 13, 14, and 15 are not available for assembly language programming:

- %r15 — Program Counter
- %r14 — Interrupt Stack Pointer
- %r13 — Process Control Block Pointer
- %r12 — Processor Status Word

4.2.6 Stack Usage

A portion of memory is reserved for temporary data storage so that information may be passed from a calling routine to a called function. This area of memory is called the stack. Much like a stack of books, data is placed upon previously entered data (using the push instruction) and is retrieved (using the pop instruction) starting with the latest data first. In the 3B20D and 3B21D processors, the addresses move from lower to higher.

The stack is manipulated by three registers, the stack pointer (%sp), frame pointer (%fp), and the argument pointer (%ap). The stack also contains a program counter that holds the address of the instruction that is to be executed next.

%sp — The stack pointer designates the top or next available word on the stack.

%fp — The frame pointer points just past the top of the save area. The save area is a region on the stack for saving registers. The layout of the save area is fixed and will contain meaningful data only if registers were actually saved. Just past the save area is a region on the stack where a function can store automatic and temporary variables. The frame pointer points to the first automatic variable.

%ap — Below the save area is an area where all function arguments are stored. The argument pointer points to the first argument of each function.

%pa — Program counter is set to the address of the first executable instruction of the function.

Function call instructions manipulate the environment for functions to save registers and to provide for temporary storage. The environment is implemented as a stack which is pointed to by the stack pointer. (See Figure 4.2-3.)

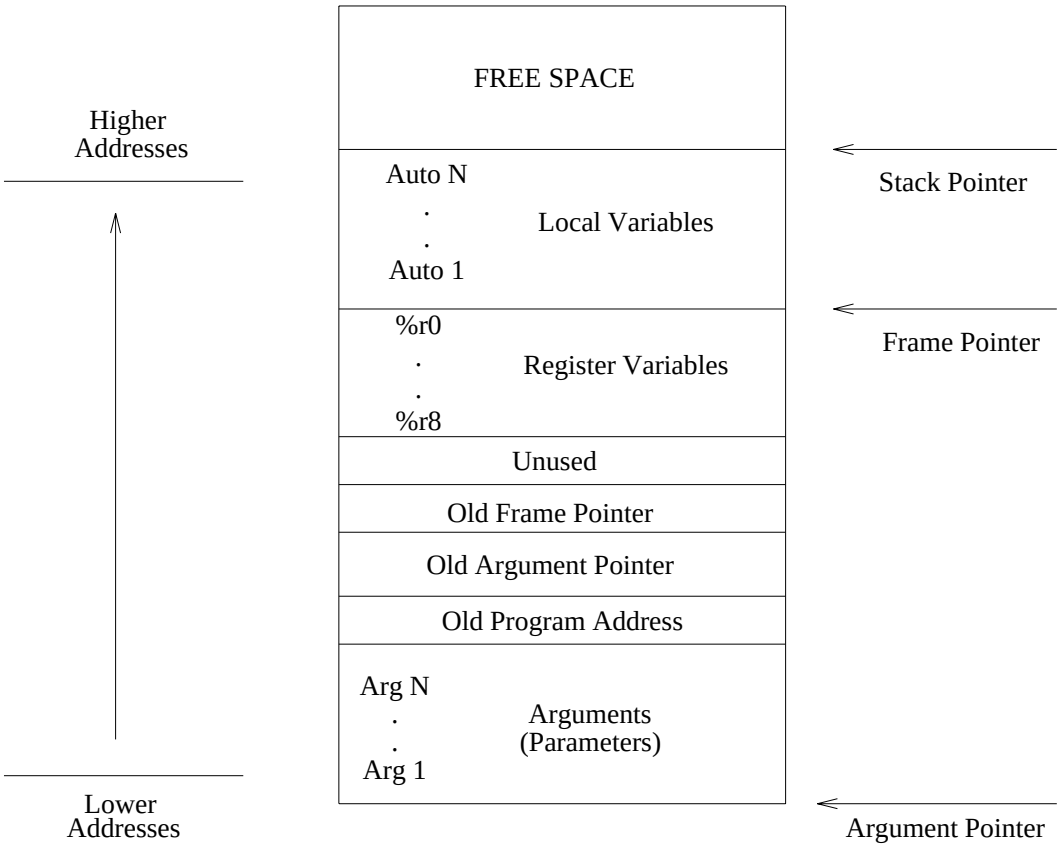


Figure 4.2-3 — 3B20D and 3B21D Processor Stack Frame Organization

A run-time stack is used to provide an environment for C functions. The stack is used for saving registers, passing arguments, and providing local variable storage. The stack is pointed to by a stack pointer and grows in a positive direction, that is, towards higher addresses. The stack pointer (`sp`) always points to the next free word on top of the stack. A stack `push` increments the stack pointer. Stack `pushes` and `pops` should be done in words to keep the stack pointer on an integral word boundary.

The frame pointer (`fp`) always points to the first local variable of the function. The argument pointer (`ap`) always points to the first argument passed to the function. The sequence of events of a function call is as follows:

The actions required of the calling function are to evaluate and `push` the arguments, then execute a `call` to the called function. The `push` and `call` instructions in IS25 are used for these purposes. Since the stack grows in a positive direction, the arguments are `push`ed in the order in which they would appear in a C function call.

Any argument smaller than a word will be converted and `push`ed to occupy a full word on the stack. Also, multiple word arguments, such as structures, require multiple

pushes. By convention, a call to a function that returns a structure also requires that register %r2 be loaded with the address of the area where the returned structure is to be stored by the called function.

In Figure 4.2-4 the call instruction pushed the %pa and %ap registers onto the stack and sets up the new %ap. The %pa is set to the function address specified and control is transferred to the B function. By convention, the register %r0 contains the value, if any, returned by the called function. If the called function is returning a structure, %r0 contains the address of the returned structure.

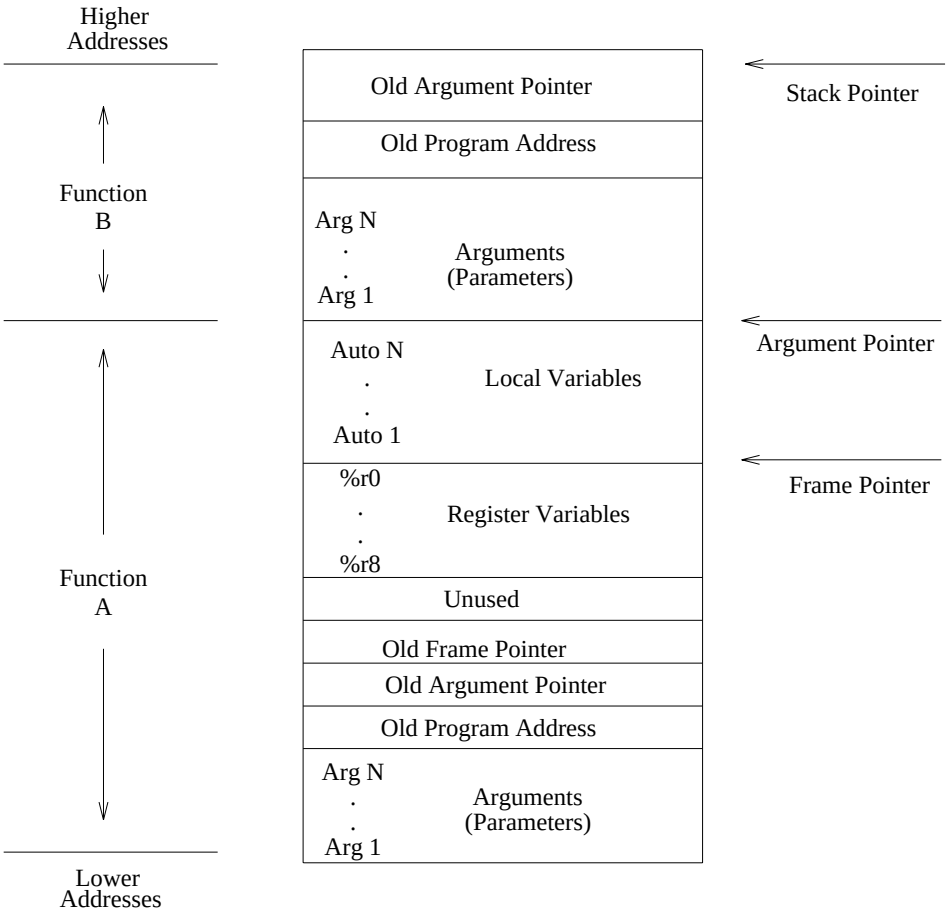


Figure 4.2-4 — 3B20D and 3B21D Processor Stack Function A Calls Function B

The called function, Function B, is responsible for completing the stack frame initialization started by the caller. (See Figure 4.2-5.) This includes saving registers and allocating space on the stack for local variables. The save instruction saves the %fp register and the specified number of registers on the stack and sets the %sp and %fp registers to point to the first word above the save area, which is where the local variables are allocated. After the prologue, local variables are addressable relative to %fp, and arguments are addressable relative to %ap. It should be pointed out that the save area is fixed in size, thus allowing the stack tracing routines for debugging utilities to access the saved %ap, %fp, and %pa registers using the %fp register.

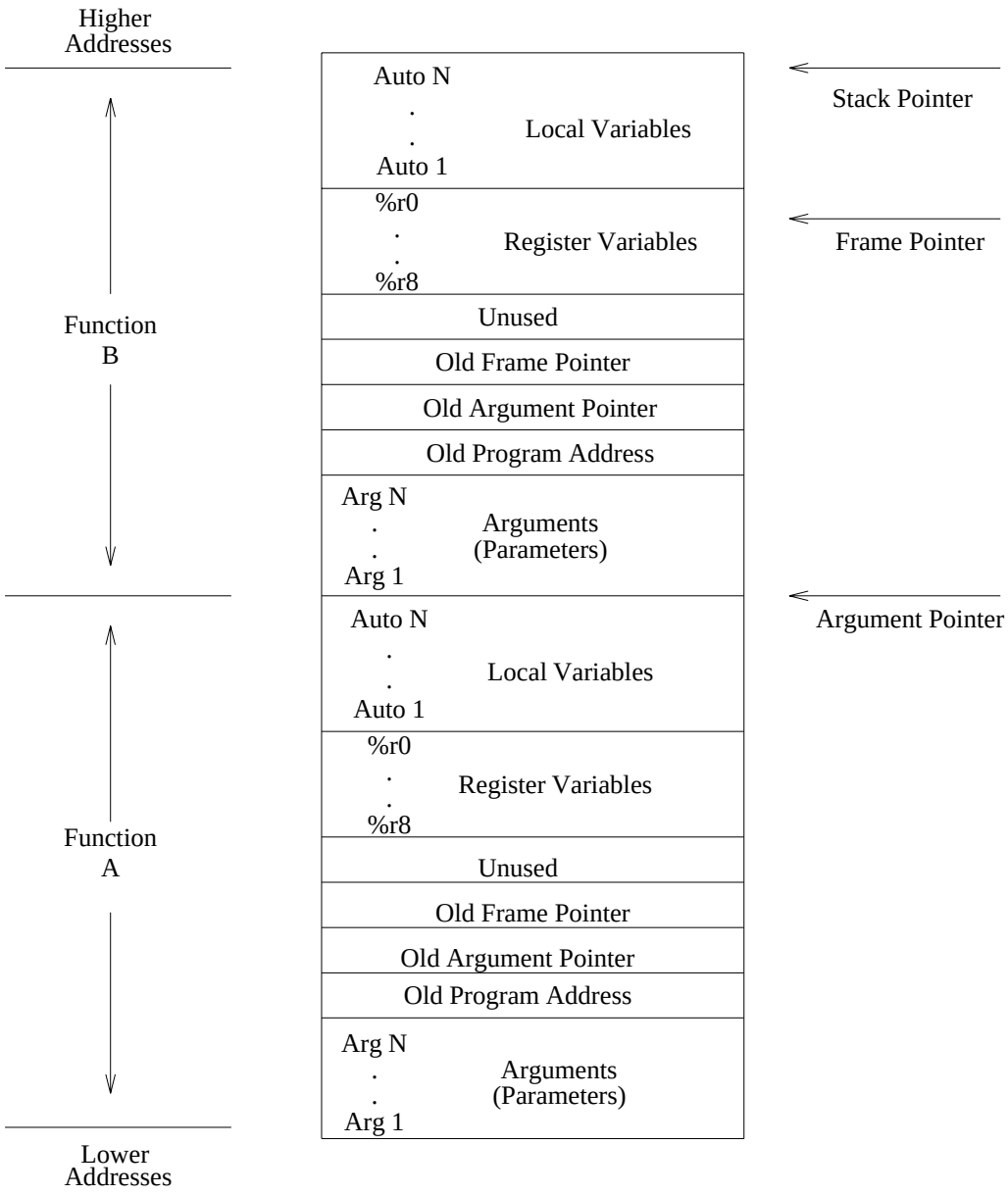


Figure 4.2-5 — 3B20D and 3B21D Processor Stack Function B Takes Control

To return to the calling function, the called function executes `ret &r` where `r` is the same number of registers specified in the `save` instruction of the prologue. This causes the saved registers, including `%ap`, `%fp`, and `%pa`, to be restored and the entire stack frame to be `pop`ped.

If the function is returning a value, the called function must load that value into `%r0` before the return. For a function returning a structure, the called function must load `%r0` with the address of the structure and copy that structure to the address originally passed to it in register `%r2`.

4.3 *MOTOROLA*[1] MC68000 PROCESSOR ASSEMBLY LANGUAGE

4.3.1 Values

In the MC68000 processor, assembler values are represented by 32-bit two's complement numbers. If a constant or the result of evaluating an expression requires more than 32 bits to represent it, the least significant bits of the actual values are used. If an expression generates more than 32 bits of data, the least significant 32 bits are used. Values in the MC68000 processor would be of one of the following types:

`UNDEFINED` An `UNDEFINED` value is a value whose type is undetermined. Examples of `UNDEFINED` values are references to symbols whose definitions have not been encountered, such as forward references, and references to symbols that are defined in programs other than the one currently being assembled, such as external references.

`ABSOLUTE` An `ABSOLUTE` value is a value that never changes, even if sections of the program assembled are relocated. Example of `ABSOLUTE` values are numeric constants and arithmetic expressions with operands that are all numeric constants.

`TEXT` A `TEXT` value is a value that is relative to the beginning of the `.text` section. Whenever the `.text` section is relocated by `N` bytes, `N` should be added to or subtracted from every value of the type `TEXT`.

`DATA` A `DATA` value is a value that is relative to the beginning of the `.data` section. Whenever the `.data` section is relocated by `N` bytes, `N` should be added to or subtracted from every value of type `DATA`.

`BSS` A `BSS` value is a value that is relative to the beginning of the `.bss` section. Whenever the `.bss` section is relocated by `N` bytes, `N` should be added to or subtracted from every value of type `BSS`.

In addition, these types can have the `EXTERNAL` attribute. Symbols with this attribute can be referenced by separately compiled programs.

4.3.2 Constants

A constant is an object of fixed value and `ABSOLUTE` type. The MC68000 processor assembler supports the following types of constants:

Decimal A string of digits (0-9) that begins with a non-zero digit.

```
1234
 325
  43
```

Octal A string of digits (0-7) that begins with a zero digit.

```
 077
0123
0342
```

1. Registered trademark of Motorola Inc.

Hexadecimal

A string of digits (0-f) that is prefixed with 0x or 0X. The digits "a" through "f" can be represented by either upper or lower case letters.

```
   0x3f
0x8ABC
 0xFEA
```

4.3.3 Expressions

An expression is a sequence of operands separated by operators, where an operand can be a constant, symbol, or an expression enclosed in brackets ([ ]). All operators are binary in nature and have equal precedence. However, when an operator is used in a unary context, the operation is performed as though an ABSOLUTE zero prefixed the operator.

Expressions are evaluated from left to right, and if an evaluation other than left to right is desired, square brackets must be used for grouping. The following operators are available:

+	Performs addition. If one operand is of the ABSOLUTE type, the result has the type of the other operand; otherwise, the operation is illegal.
–	Performs subtraction. If the right operand is of the ABSOLUTE type, the result has the type of the left operand. If both operands have the same type, and that type is either TEXT, DATA, or BSS, the result is of the ABSOLUTE type; otherwise the operation is illegal.
*	Performs integer multiplication. This operation requires that both operands be of the ABSOLUTE type and an ABSOLUTE result is produced.
/	Performs integer division. Both operands must be of the ABSOLUTE type and the result will be of the ABSOLUTE type. To avoid confusion with the single slash (/), which starts a comment, this operator must be preceded by a backslash (\).
&	Performs bit-by-bit logical AND comparisons between two 32-bit quantities. Both operands must be of ABSOLUTE type, an ABSOLUTE result is produced.
|	Performs bit-by-bit logical OR comparisons between two 32-bit quantities. Both operands must be of ABSOLUTE type, an ABSOLUTE result is produced.
>>	Shifts the left operand to the right by the number of bits specified by the right operand. This operation requires that both operands be of the ABSOLUTE type, an ABSOLUTE result is produced.
<<	Shifts the left operand to the left by the number of bits specified by the right operand. This operation requires that both operands be of the ABSOLUTE type, an ABSOLUTE result is produced.
%	Returns the remainder of the operation that divided the first operand by the second operand. Both operands must be of the ABSOLUTE type and an ABSOLUTE result is produced. To avoid confusion with register names, this operator must be preceded by a backslash (\).
!	Performs bit-by-bit logical AND comparisons between the first operand

and the bit-by-bit complement of the second operand. This operation requires that both operands are of the ABSOLUTE type and an ABSOLUTE result is produced.

^ Allows you to alter the effects of symbol requirements by allowing symbol types which do not normally interact. The result has the value of the first operand and the type of the second operand. The operands can be of any type.

4.3.4 Machine Instruction Notation

The machine instructions are mnemonic representations of MC68000 processor machine language. For the general purpose register the notation is shown in Table 4.3-1.

Table 4.3-1 — *Motorola* MC68000 Processor General Purpose Registers

General Register	Description
%an	Address Register (0 <= n <= 7)
%dn	Data register (0 <= n <=7)
%sp	Stack pointer (%a7 = %sp)
%fp	Frame pointer (%a6 = %fp)

In addition, a number of control registers are available. These can be used only with specific instructions or addressing modes. See Table 4.3-2.

Table 4.3-2 — *Motorola* MC68000 Processor Control Registers

Special Register	Description
%caar	Cache address register
%cacr	Cache control register
%ccr	Condition code section of status register
%dfc	Destination function code register
%isp	Interrupt stack pointer
%msp	Master stack pointer
%pc	Program counter
%sfc	Source function code register
%sr	Status register
%usp	User stack pointer
%vbr	Vector base register
%zpc	Zero value taken for %pc (pseudo register)

The %zpc register is not a real register. It is a way of differentiating between address register indexed/memory indirect modes and program counter indexed/memory indirect modes.

4.3.5 Operands

The opcode tells the computer what action to perform and the operand designates the location. Instructions for the MC68000 processor family can take zero, one, or two operands. The size of an operand may be a byte (8 bits), a word (16 bits), a long word (32 bits) or a quad word (64 bits) depending on what is specified in the instruction mnemonic. The location of an operand is indicated by an effective address. An effective address may take on one of a number of forms called address modes. See Table 4.3-3 for additional details.

Table 4.3-3 — *Motorola* MC68000 Processor Addressing Mode — Effective Address Contents

Address Mode	Mode Bit No. 5 4 3	Register No. 2 1 0
Data Register Direct	0 0 0	r r r
Address Register Direct	0 0 1	r r r
Address Register Indirect	0 1 0	r r r
Address Register Indirect with postincrement	0 1 1	r r r
Address Register Indirect with Predecrement	1 0 0	r r r
Address Register Indirect with Displacement	1 0 1	r r r
Address Register indirect with Index	1 1 0	r r r
Absolute Short	1 1 1	0 0 0
Absolute Long	1 1 1	0 0 1
Program Counter Relative with Displacement	1 1 1	0 1 0
Program Counter Relative with Index and Displacement	1 1 1	0 1 1
Immediate or Status Register	1 1 1	1 0 0

An effective address (EA) is specified by six bits in the instruction (usually the lowest six bits). The bit values indicate how to find the data for the instruction. Bits 0, 1, and 2 usually signify the register and bits 3, 4, and 5 represent the mode. See Figure 4.3-1.

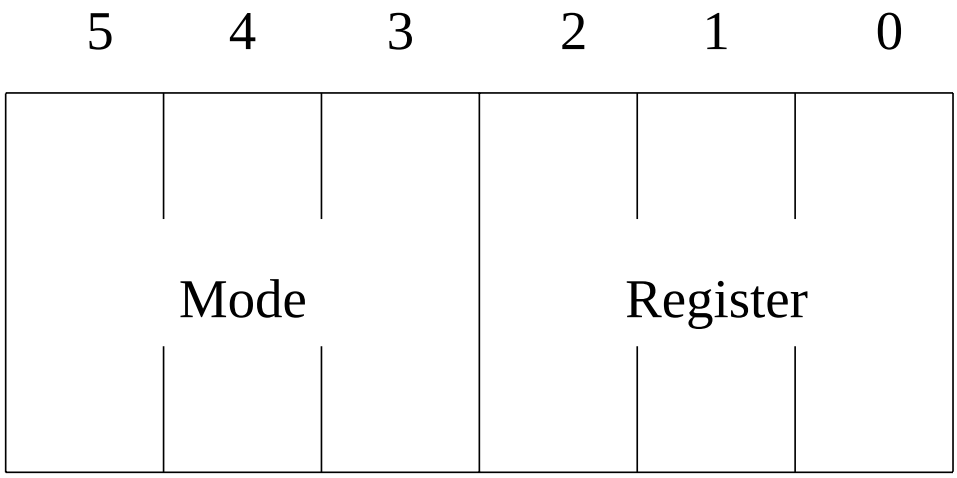


Figure 4.3-1 — Effective Address — Memory Layout

When viewed as a whole, a general rule for the 16-bit instruction consists of the opcode, the data register, the direction bit, the size, the mode, and the register. Not every instruction has this layout, but this does represent the fields that will be found in the layout. See Figure 4.3-2.

Opcode The opcode is the instruction component which tells the computer what to do.

Data Register
The data register is either the source or destination register. It is evaluated based on the DR bit or direction bit or an immediate value.

DR DR or direction bit specifies the instruction direction.

- 0 = The source is the effective address and the destination is the data register.
- 1 = The source is the data register and the destination is the effective address.

Size Specifies the data size used in the instruction.

- 00 = Byte
- 01 = Word
- 10 = Long

Mode The address mode tells the computer how to use the registers and data that have been supplied. See Table 4.3-4.

Register The register is either the source or destination register. It is evaluated based on the DR bit or direction bit.

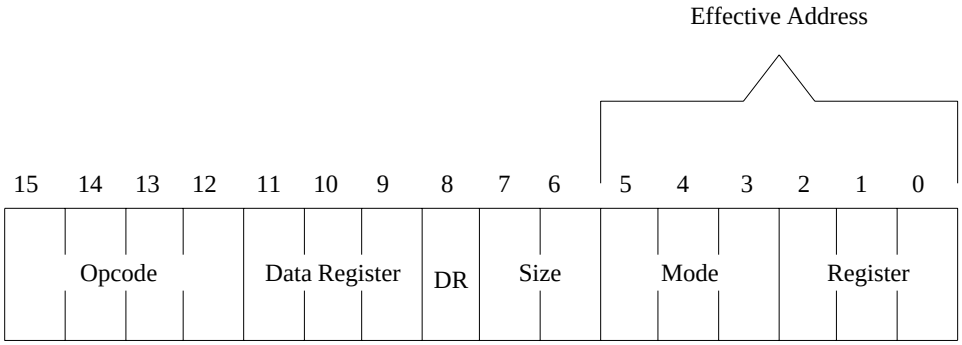


Figure 4.3-2 — *Motorola* MC68000 Processor Instruction — Memory Layout

Table 4.3-4 — *Motorola* MC68000 Processor Operand Address Modes

Address Mode	Description
`%dn`	**Data Register Direct:** The data register contains the operand. Example: `add %D3,%D7` Add the contents of data register 3 to the contents of data register 7 and place the results in data register 7.
`%an`	**Address Register Direct:** The address register indicated by the register field contains the operand. Example: `ADD %A6, %D2` Add the contents of address register 6 to the contents of data register 2 and place the results in data register 2.
`(%an)`	**Address Register Indirect:** The address register indicated by the register field is the address of a memory location that contains the operand. The register is said to point to (contain the address of) the operand. Address register indirection is denoted by enclosing the address register name in parentheses. Example: `ADD (%A6),%D2` Go to the address contained in address register 6 and add the contents found there to the contents of data register 2 and place the result in data register 2.

Table 4.3-4 — *Motorola* MC68000 Processor Operand Address Modes (Contd)

Address Mode	Description
`(%an)+`	**Address Register Indirect with Postincrement:** The address register indicated by the register field contains the address of a memory location that contains the operand. The register is said to point to the operand. The address register is incremented after the data has been obtained from memory. The increment is based on the length of the data item referenced by thinstruction. Example: `ADD (%A6)+,%D2` Go to the address contained in address register 6. Take the contents of that location and add it to the contents of data register 2 and place the results in data register 2 then increment the address in address register 6.
`−(%an)`	**Address Register Indirect with Predecrement:** The address register indicated by the register field contains the address of a memory location that contains the operand. The register is said to point to the operand. The address register is decremented before the data has been obtained from memory. The decrement is based on the length of the data item referenced by the instruction. Example: `ADD −(%A6),%D2` Decrement the address contained in address register A6. Go to the address contained in address register 6. Take value at that location and add it to the contents of data register 2 and place the results in data register 2.
`expr(%an)`	**Address Register Indirect with Displacement:** The address register indicated by the register field is added to the sign-extended 16-bit number (displacement) following the instruction. The result is the address of a memory location that contains the operand. Example: `ADD 0x60(%A6),%D2` Add 60 to the contents of address register 6, then go to that address. Add the contents at that address to the contents of data register 2 and place the result in data register 2.
`expr(%an,%ri[.l])`	**Address Register Indirect with Index:** The address register indicated by the register field is added to the content of another register, plus a sign extended 8-bit displacement. The sum of these three quantities is the address of a memory location that contains the operand. Example: `ADD 0x60(%A2,%A6),%D2` Add the contents of address registers 2 and 6 then add 60 to the result. Go to the address which is the result of that operation, take the contents at that address and add them to the contents of data register 2 and place the result of this operation in data register 2.

Table 4.3-4 — *Motorola* MC68000 Processor Operand Address Modes (Contd)

Address Mode	Description
`expr`	**Absolute Short:** The word following the instruction is an absolute 16-bit address. The 16-bit address is sign-extended before it is used. Example: `add 0x13A,%D2` Go to the address given as the first operand, take the contents of that address add them to the contents in data register 2 and place the result in data register 2.
`expr`	**Absolute Long:** The long word following the instruction is an absolute 32-bit address. Example: `add 0x4A7138,%D2` Go to the address given as the first operand, take the contents of that address add them to the contents in data register 2 and place the result in data register 2.
`expr(%PC)`	**Program Counter Relative with Displacement:** The 16-bit sign-extended displacement following the instruction is a displacement to be added to the program counter in order to obtain a memory address. Example: `ADD 0x4(%pc),%D2` Add 4 to the program counter, go to the address given as a result add the contents of that address to the contents of data register 2 and place the result in data register 2.
`expr(%PC,%ri[.l])`	**Program Counter Relative with Index and Displacement:** The memory address is to be constructed using the value of the program counter, an index register and a sign-extended 8-bit displacement. Example: `ADD 0x4(%pc,%A5),%D2` Add the contents of address register 5 to the program counter and then add 4 to the result. Add the result of that operation to the contents of data register 2 and place the result in data register 2.
`$expr`	**Immediate:** The source data for an instruction is contained in the word or longword that follows the instruction. Example: `ADD $0x60,%D3` Add 60 to the contents of data register 3 and place the result in data register 3.

4.3.6 Instruction Set

The instruction set for the MC68000 processor is based on the *Motorola* MC68XXX processor family instruction set. The operations are performed on bytes, words, and long words. The `add` instruction is `addb` for one byte, `add` for a word, or `addl` for a long word. See Figure 4.3-3 for an example.

For a complete listing of the instruction set, see Appendix A3 - *Motorola* MC680XX Processor Family Instruction Set.

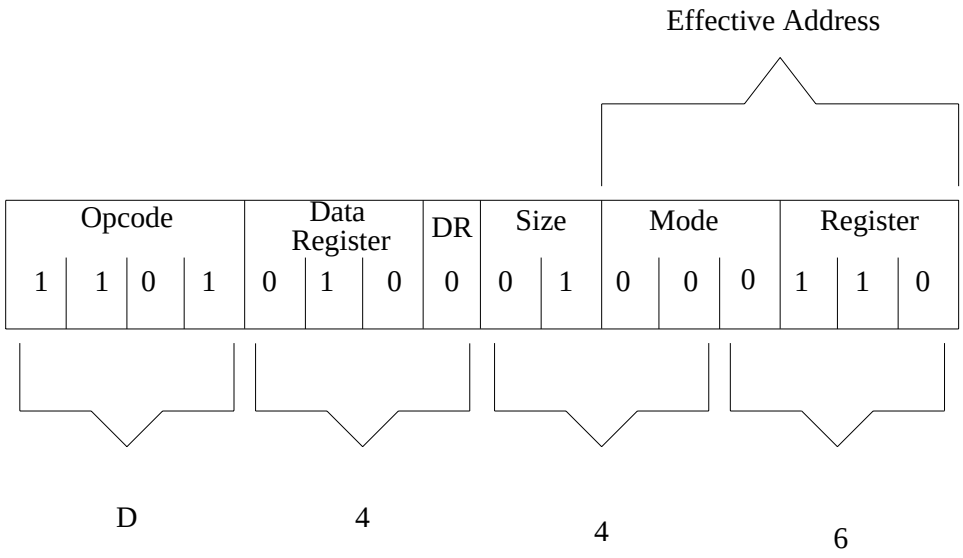


Figure 4.3-3 — *Motorola* MC68000 Processor Example Instruction Memory Layout

4.4 *MOTOROLA*[1] MC68000 PROCESSOR MACHINE DEPENDENCIES

4.4.1 Data Type Memory Boundaries

The processors in the *5ESS*® switch have specific memory and organizational requirements, each processor having variations in size and functionality. The following gives a brief explanation of the byte and word organization of the MC68000 processor.

Byte-length data which contains 8 bits:

- can start in any memory location
- represents an 8-bit string or binary number between 0 and 255.

Word-length data which contains 16 bits:

- can start in any even-numbered memory location
- represents a 16-bit string or unsigned binary number between 0 and 65,535 or 2s complement binary number between –32,768 and 32,767 (bit 15 is the sign bit).

Long Word-length data which contains 32 bits:

- can start in any even-numbered memory location
- represents a 32-bit string or unsigned binary number between 0 and 4,294,968,295 or 2s complement binary number between –2,147,483,648 and 2,147,483,647 (bit 31 is the sign bit).

Table 4.4-1 details the data size and alignment of the MC68000 processor. When analyzing output from this processor it is important to reference the data types according to the contents of this table.

Table 4.4-1 — *Motorola* MC68000 Processor Data Sizes and Alignment

Data Type	Size	Memory Alignment
`char`	1 byte	no alignment
`short`	2 bytes	2 byte boundary
`int`	2 bytes	2 byte boundary
`long`	4 bytes	2 byte boundary
`pointer`	4 bytes	2 byte boundary
`structure`	4 byte multiple	2 byte boundary
`union`	4 byte multiple	2 byte boundary
`array`	same as element type	same as element type
`bitfield`	maximum of type	same as declared type
data inside `structures` and `unions`		
`long`	4 bytes	4 byte offset
`pointer`	4 bytes	4 byte offset
`structure`	4 byte multiple	4 byte offset
`union`	4 byte multiple	4 byte offset
`bitfield`	maximum of type	same as declared type

1. Registered trademark of Motorola Inc.

Figure 4.4-1 will help to explain the memory boundaries that would be established by the MC68000 processor.

```
Given:
 char a
 short b
 int  c
 long d
 pointer e
```

The data would be placed on the data stack as follows:

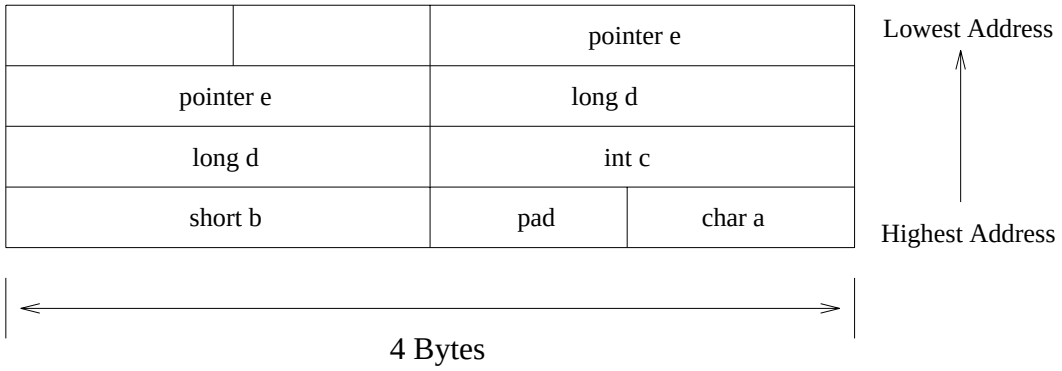


Figure 4.4-1 — *Motorola* MC68000 Processor Stack –– Memory Boundaries

4.4.2 Arithmetic Types Supported

The MC68000 processor compiler supports four arithmetic types. All four types can be `signed` or `unsigned`.

- `char`
- `short`
- `int`
- `long`

The `char` and `unsigned char` are defined to be 1 byte. The `unsigned` and `signed char` are the same, since with a character-type conversion, sign extension is not implemented. They can be located on any byte boundary.

The `int` and `unsigned int` (`short` and `unsigned short`) have an implementation of 2 bytes. These arithmetic types can only be located on an even byte boundary. Two bytes represent one word of memory.

The `long` and `unsigned long` have an implementation of 4 bytes. These two arithmetic types can only be located on an even byte boundary. When these types are members of a `structure` or a `union`, they are located on a 4-byte boundary. Four bytes represents a long word of memory.

Pointers are allocated 4 bytes of memory; 24 bits are required to address the memory spectrum [5E4(2) and earlier]. The highest byte of the four assigned bytes is set to zero. Under VLLM, 25 bits are required to address the memory spectrum [5E5(1) and

later]. A pointer is aligned on an even byte boundary except when it is a member of a structure. In this case, it is aligned on a 4-byte boundary from the beginning of the structure.

Note: A pointer can contain a larger number than 25 bits can represent but the codes will not execute if it tries to access that memory location. The compiler does not restrict bits 26-31 from being set.

4.4.3 Data Conversion Rules

Table 4.4-2 — *Motorola* MC68000 Processor Data Conversions Rules

Data Type Conversion	Char	Unsigned Char	Int	Unsigned Int	Long	Unsigned Long	Pointer
Char	—	NC	PL	PL	PL	PL	PL
Unsigned Char	NC	—	PL	PL	PL	PL	PL
Int	TL	TL	—	NC	SE	PL	PL
Unsigned Int	TL	TL	NC	—	PL	PL	PL
Long	TL	TL	TL	TL	—	NC	NC
Unsigned Long	TL	TL	TL	TL	NC	—	NC
Pointer	TL	TL	TL	TL	NC	NC	—

NC = Name Change, TL = Truncate on Left, SE = Sign Extend, PL = Pad on Left with zero(s),
TL/PL = Truncate on left to 24 bits and pad on left with one byte of zero [5E4(2)]
5E5(1)= pointers use 25 bits and have 7 bits of zero padded on the left.

4.4.4 Memory Configuration

4.4.4.1 Overview *MOTOROLA* MC68XXX Processor Memory Configuration

The MC68000 processor has an address range of 16 megabytes, other members of the MC68XXX processor series have larger address ranges. The physical memory is divided into configured and unconfigured memory and is partitioned with the MEMORY directive. The default condition (no MEMORY directive specified) treats all virtual memory as configured.

MEMORY directives are used to specify:

1. The total size of the physical address space of the target machine.
2. The configured and unconfigured areas of the physical address space.

An arbitrary name consisting of up to eight characters is assigned to a physical address range using MEMORY directives. Output sections can then be bound to physical addresses within specifically named areas. When MEMORY directives are used, all physical memory not described in MEMORY directive is considered to be unconfigured. Unconfigured memory is not used in the link editor's allocation process, and therefore, no data can be link-edited, bound to, or assigned to an address within unconfigured memory.

Attributes can be associated with a named memory area. This is an option of the MEMORY directive, and it restricts which memory areas an output section can be bound to (with specific attributes). When attributes are assigned to output sections in this manner, they are recorded in the appropriate section headers in the output file to provide for error checking.

The currently accepted attributes are:

R	Readable memory
W	Writable memory
X	Executable memory
I	Initializable memory

If no attributes are specified with a MEMORY directive, or if no MEMORY directives are supplied, the memory areas will assume the attributes of W, R, I, and X.

4.4.4.2 Optimization

4.4.4.2.1 Overview of Optimization

Once the compiler has translated the C source code to the MC68000 processor assembly language, the optimizer manipulates the assembly code to improve the text space requirements and execution time of the resulting object code. The goals of the optimizer are to:

- reduce the object's text size
- reduce the real time used by the object
- reduce the stack space of the object
- use as little host CPU time as possible.

Optimization is done on a function-by-function basis. Functions containing assembler escapes are not optimized, but other functions in the same file are. No optimizations are done on global variables.

There are five types of optimization: peep-hole improvements, live-dead analysis, value tracing, code reordering, and variable registerization. These optimizations used together are more effective than the sum of their individual efforts. Improvements made by a previous optimization can facilitate even more improvements by a later optimization pass. Optimizations are done in the following order:

1. Registerization
2. Value Tracing
3. Code-Reordering
4. Live-Dead Analysis
5. Peep-Hole.

The last four optimizations are repeated until no more improvements can be made.

4.4.4.2.2 Registerization

Automatic register variable allocation saves both text space and execution time by assigning local variables to registers. This optimization is actually done in part by the compiler and partly by the optimizer. The compiler collects information to determine which local variables are used most frequently. This data is passed to the optimizer by comments in the assembly language file. The optimizer assigns variables to unused registers based in part on the information collected by the compiler.

The optimizer tries to put local variables and parameters in registers. The optimizer estimates how much time (execution time of the resulting code) and space (text space of the resulting code) will be saved if that variable is put in a register. If gains would be made (and neither quantity would degrade), the variable is registerized.

Not all variables can be placed in registers automatically. Only variables having simple C types declared in the outermost block can be put in registers; that is, only chars, shorts, ints, longs, enumerations, and pointers.

4.4.4.2.3 Value Tracing

Value tracing locates and eliminates redundant and unnecessary instructions. This type of optimization can delete part, and sometimes all, of the assembly code for a particular C statement. The value of each local variable and register is traced. When a value is assigned to a variable that value is substituted whenever that variable is encountered. No optimizations are done on global variables.

4.4.4.2.4 Code Reordering

Code is reordered to reduce the number of jumps and to merge common sequences of code. The merging and resequencing improves the object code. Code can be reordered by:

Common Tail Optimization
: A common tail is a piece of code that is duplicated one or more times in a function at the end of basic blocks.

Loop Rotation
: This optimization rearranges loop constructs.

Branch Merging
: This optimization tries to minimize the number of branches in the code. It does this by merging the branch with its destination whenever possible.

Remove Unreachable Code
: This option removes any piece of code that cannot be reached by an execution path.

4.4.4.2.5 Live-Dead Analysis

Live-dead analysis is done in two parts. The first is an analysis of the code to see which variables are used and the second is a small collection of peep-holes that use the information collected in the analysis.

The analysis examines the flow of control in the code and determines where the local variables and registers are set and where they're used. Then, at each instruction, it determines whether each variable and register is dead or live. A variable is dead if it will be set before it is used again or its value will never be used again. A variable is live if it will not be set before it is used again.

The peep-hole part of the live-dead analysis examines each instruction and the live-dead information associated with it and decides whether to get rid of the instruction or not.

4.4.4.2.6 Peep-Hole Optimizations

The peep-hole optimizations replace small instruction sequences with other, more efficient instruction sequences and removes instructions that have no effect on the results of the assembly language.

4.4.5 Register Notation

4.4.5.1 Register Classes

The MC68000 processor provides two classes of registers: special registers and general registers. The special registers contain or provide system or user information concerning processor execution location and the resulting status of that execution. The general registers come in two types: address registers and data registers.

4.4.5.2 General Purpose Registers

Address Register

There are eight address registers (A0-A7). Each address register is 32 bits. A7 and A6 are defined for addresses that are key to the execution of code. See Table 4.4-3.

Table 4.4-3 — *Motorola* MC68000 Processor General Purpose Registers

Register	Usage
A7	Stack pointer (SP)
A6	Frame pointer (FP)
A5	.
A4	.
A3	.
A2	Scratch
A1	.
A0	.

Stack Pointer

A7 is the stack pointer (two distinct registers, one for the user mode and the other for the supervisor mode). The SP register points to the top of the stack. In the user mode or state, all stack pointer references are to the user stack. In the supervisor state, all stack pointer references are to the supervisor stack.

Frame Pointer

A6 is the frame pointer which is set to point at the base of the current stack frame.

Transfer Vector Pointer

A transfer vector (TV) is an ordered list of entries, similar to an array of pointers. In the *5ESS* switch [for 5E4(2) and earlier], function calls are made through a transfer vector table. The starting address of the table is pointed to by A5, the transfer vector pointer. The slots in the transfer vector table, which are 4 bytes in length, contain the actual function addresses.

Beginning with the 5E5(1) software release, a TV slot in the *5ESS* switch became a 6-byte block of memory that contains a jump instruction to the beginning of a particular function.

The link editor defines the transfer vector as a separate output section known as .tv. Unlike other output sections, the link editor supplies the entire contents for the transfer vector's output section.

Data Register

There are eight data registers (D0-D7). Each data register can be used to hold values that are 8 bits, 16 bits, or 32 bits in size. Data registers cannot be used to address memory in an instruction. Instead these registers are used as temporary locations where data may be stored.

The D0 register is reserved for passing the called function results (scaler results) back to the calling function and is named the function result register. See Table 4.4-4.

Table 4.4-4 — Data Registers

Register	Usage
D7	Scratch
D6	.
D5	.
D4	.
D3	.
D2	.
D1	.
D0	Function result register

4.4.5.3 Special Registers

Program Counter

The program counter register is a 32-bit register used to control execution of the program in memory. The program counter is always pointing to the next instruction to be executed. When an instruction is executed, the program counter is advanced to the next instruction.

Status The status register is a 16-bit register used to store information about the status of the processor. This register is used by the conditional branch instructions to retrieve information about the last instruction.

4.4.6 Stack Usage

Stacks are last-in-first-out (LIFO) data structures for storage. Placing data on the stack is called a push. The push uses the address register indirect with predecrement addressing mode. Removing data from the stack is called a pop. The pop uses the address register indirect with post increment addressing mode. Stacks in the MC68000 processor have the following characteristics:

- move from higher addresses to the lower addresses
- are divided into frames on a per-function basis
- are used to store automatic variables, temporary variables, and function arguments.

Address registers A6 and A7 are used to denote a stack frame. A7 is the stack pointer that points to the last element on the stack (lowest address). A6 is pointing to the location that contains the old A6 of the calling function.

The order for the stack frame is (highest to lowest address):

1. Arguments

2. Return Address — former program counter
3. Old Frame Pointer — former A6
4. Local Variables — temporaries and automatics
5. Register Variables — placed on the stack when required.

Each function is allocated space on the stack using the following format. The addresses move from highest to lowest. When one function calls another, the calling function will appear as shown in Figure 4.4-2. The called function would be started on the stack in the next available space after the stack pointer. The calling function (Function A) will have a stack frame like the one shown below which contains arguments, return address (old program counter), old frame pointer, local variables, and register variables.

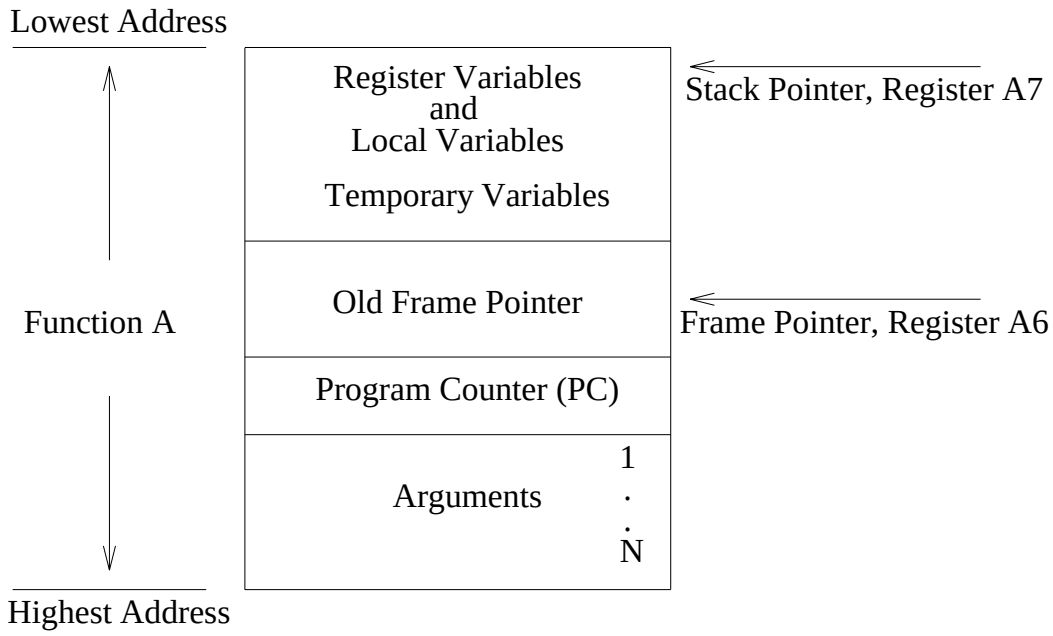


Figure 4.4-2 — Stack Frame — One Function

Function A will prepare to call another function (Function B) by pushing arguments onto the stack. The arguments will be pushed onto the stack using the MOVE (MOVEB, MOVE, and MOVEL) instruction. The move instruction transfers the contents of the source to the destination location. Function B has not been called at this point, the system is only preparing to call the function. At this point the stack will look like Figure 4.4-3.

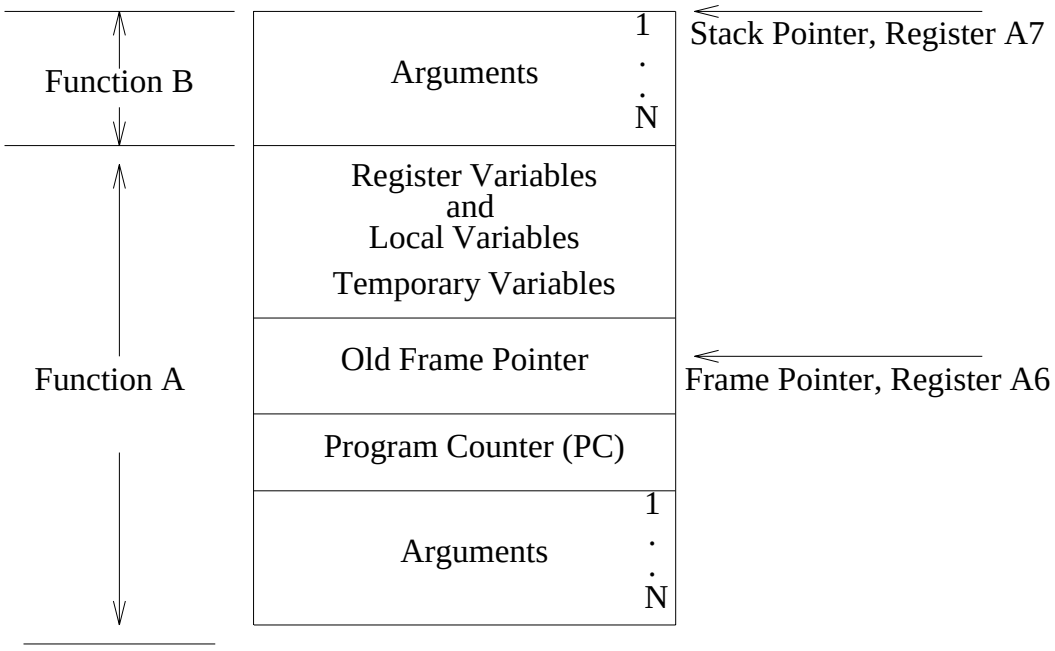


Figure 4.4-3 — Stack Frame — Arguments Pushed Onto Stack

After all the arguments have been moved onto the stack, control will be transferred to Function B by using the `jump to subroutine (JSR)` instruction. The `JSR` instruction pushes the address of the instruction immediately following it onto the system stack. Program execution continues at the address specified in the instruction. The present program counter `(PC)` is pushed onto the stack. At this point the stack will appear as represented in Figure 4.4-4.

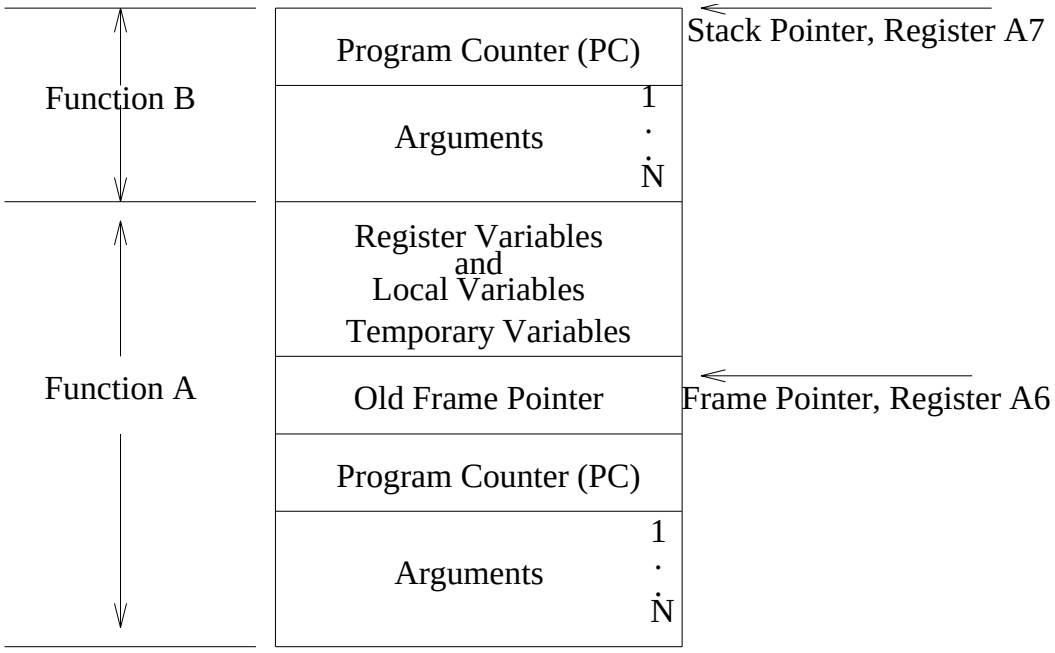


Figure 4.4-4 — Stack Frame — PC Pushed Onto Stack

Each function begins with a `LINK` instruction to allocate space on the stack for the stack frame. The `LINK` instruction pushes the current contents of the frame pointer onto the stack. After the push, the frame pointer is loaded from the updated stack pointer. Finally, the sign-extend displacement is added to the stack pointer to allocate space for the local variables. Once the `LINK` instruction is executed, control is transferred to the called function. After the execution of the `LINK` instruction, the stack will appear as in Figure 4.4-5.

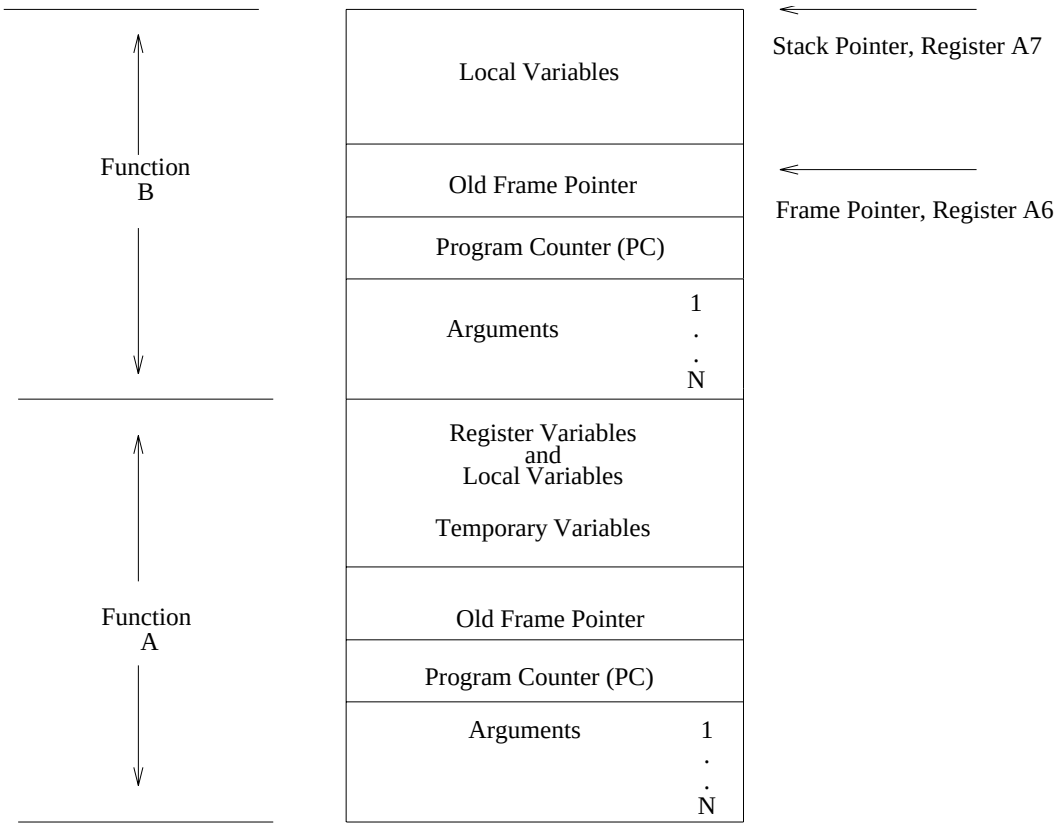


Figure 4.4-5 — Stack Frame — After `LINK` Statement

If any registers are to be used as register variables, then the current content of those registers will be moved onto the stack. This is accomplished by using the `MOVE` instruction. The stack will appear as in Figure 4.4-6.

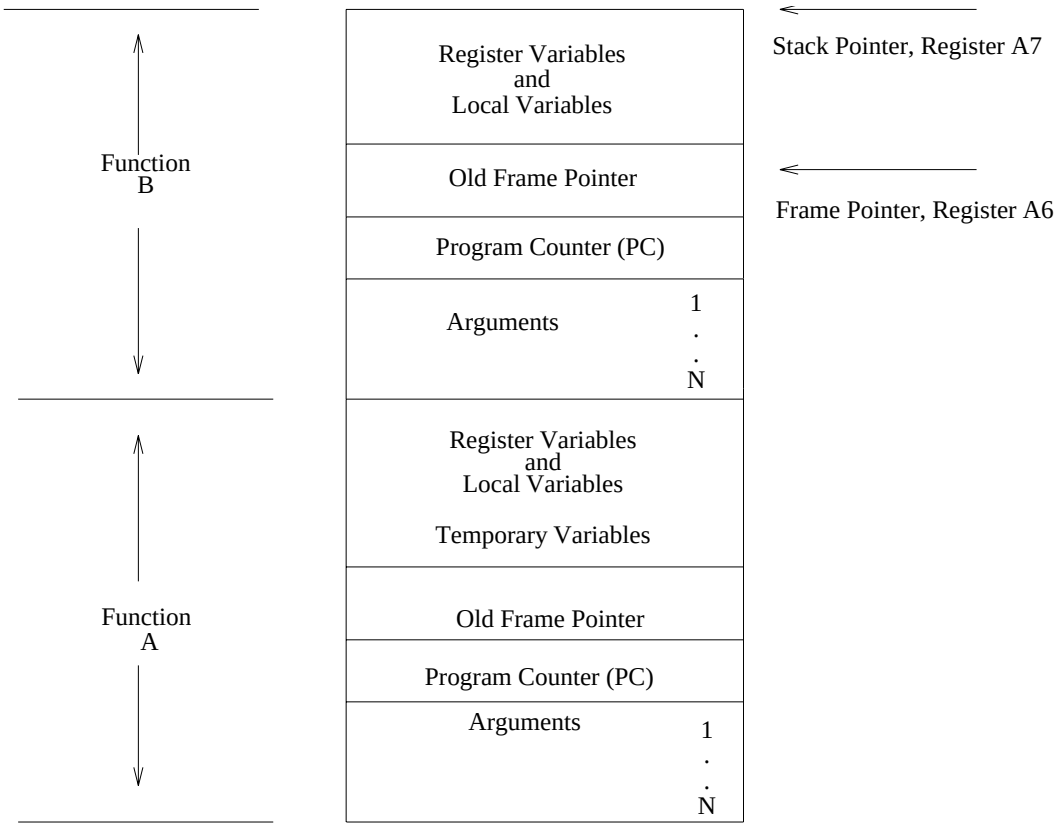


Figure 4.4-6 — Stack Frame — Register Variables Pushed Onto Stack

After the called function (Function B) has completed its task, the program flow must then return to its calling function, Function A. The first action is to reload the registers that were saved on the stack. Next the `unlink (UNLK)` instruction is performed. This instruction will deallocate the stack frame for the called function. The `UNLK` instruction loads the stack pointer from the specified frame pointer. The frame pointer is then loaded with the value from the top of the stack. The stack will look like the Figure 4.4-7.

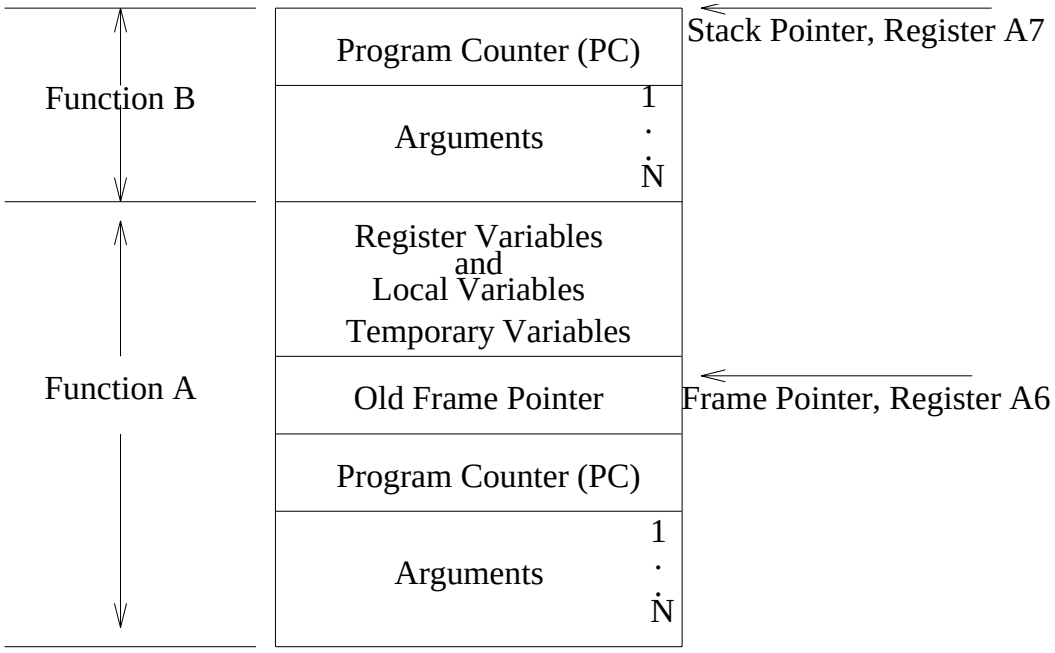


Figure 4.4-7 — Stack Frame — After The `UNLK` Statement

After the `UNLK` instruction, the called function (Function B) will execute the `RTS` (`return`) instruction to return control back to the calling function.

The `RTS` instruction execution will `pop PC` off the stack. This is the program counter for Function A. The stack frame will look like the Figure 4.4-8.

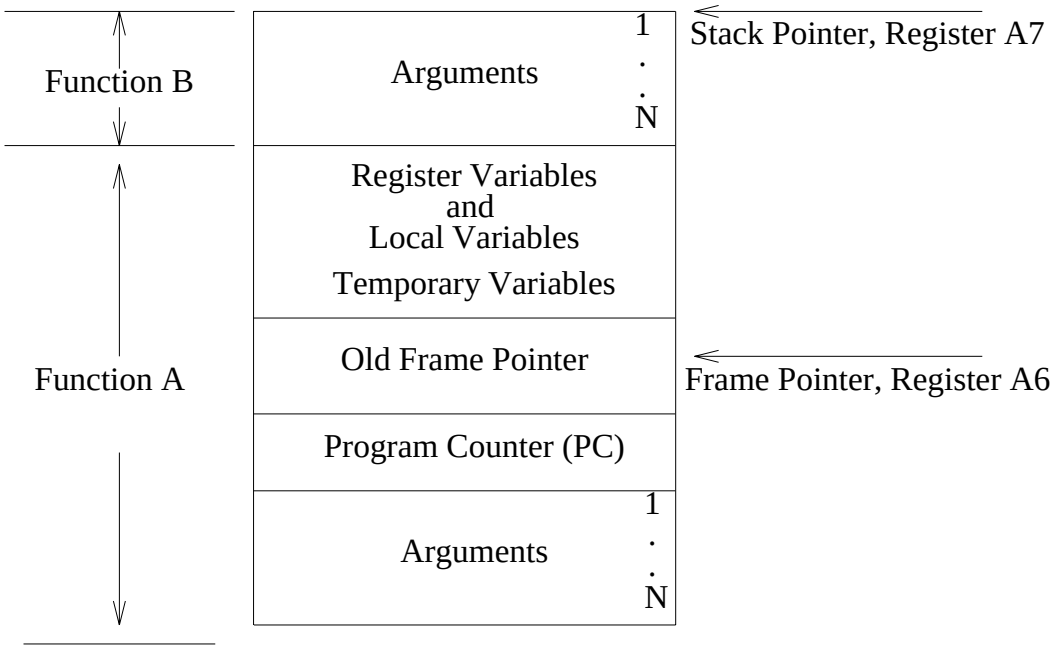


Figure 4.4-8 — Stack Frame — Return To Calling Function

The final operation to complete this calling and returning of a function is to adjust the stack pointer. The stack pointer must be moved below the passed arguments to the called function. This adjustment is accomplished by using the `LEA` or `ADDL` instruction. The `LEA` instruction loads the effective address into the specified address register. This instruction afects only 32-bit address registers.

4.4.7 Unsupported *MOTOROLA* MC68XXX Processor Instructions

The following are not currently supported by the MC68XXX assembler family. All MC68000 and MC68012 processor instructions are supported and, therefore, no table is included for them. Those MC68020 and MC68030 processor specific instructions which are unsupported are given in Table 4.4-5.

Table 4.4-5 — Unsupported *Motorola* MC68020/MC68030 Processor Instructions

Mnemonic	Description
`Bcc`	Branch Conditionally (32-bit version)
`BRA`	Branch Always (32-bit version)
`BSR`	Branch to Subroutine (32-bit version)
`CALLM`	Call Module (MC68020 processor only)
`cpXXX`	Coprocessor Instructions
`RTM`	Return from Module (MC68020 processor only)
`TRAPcc`	Trap on Condition

The MC68XXX processor family consists of several upward compatible *Motorola* microprocessors.

MC68000 Processor

The earliest of the MC68XXX based processors. It has a 24-bit address bus, 16-bit data bus, 8 data address registers each of which is 32 bits wide.

MC68012 Processor

Based on the MC68000 chip, this processor differs in the following ways: it has a 31-bit address bus and loop-mode caching, includes additional instructions, and supports virtual memory through instruction continuation.

MC68020 Processor

Based on the MC68000 chip, this processor differs in the following ways: it has a 32-bit address bus, a 32-bit data bus, a 256-byte instruction cache (direct-mapped), a microcoded coprocessor interface, fewer alignment restrictions (only instructions must be word aligned), additional control registers, additional addressing modes, extensions to the instruction set, two more stack pointers and more status register bits, and supports virtual memory through instruction continuation.

MC68030 Processor

Based on the MC68000 chip, this processor differs from the MC68000 processor in the following ways: it has a 32-bit address bus, a 32-bit data bus, 256-byte instruction and data caches (direct-mapped), an on-chip memory management unit, a microcoded coprocessor interface, fewer alignment restrictions (only instructions must be word aligned), additional control registers and addressing modes, extensions to the instruction set, two more stack pointers and more status register bits and supports virtual memory through continuation.

MC68040 Processor

Based on the MC68030 chip, this processor differs from the MC68030 processor in the following ways: it has 4-way set associative caches that are 4K bytes for both instructions and data. It has on chip floating point support (not used by the *5ESS* switch), a new Memory Management Unit (MMU), and several changes in the instruction set. There is no `%carr` register on the MC68040 processor for accessing cache. There are five new registers: transparent translation registers `%dtt0`, `%dtt1`, `%itt0`, `%itt1`, and user root pointer register `%urp`.

MC68060 Processor

Based on the MC68040 chip, this processor differs from the MC68040 processor in the following ways: it has a deep pipeline, dual issue superscalar execution, a branch cache, and a high floating point unit. It also has a full internal Harvard architecture featuring separate 8-Kbyte instruction and 8-Kbyte data cache. This processor allows simultaneous execution of two integer instructions (or an integer and a float instruction) and one branch instruction during each clock. In addition, the MC68060 processor automatically powers down internal functional blocks that are not needed.

4.5 *INTEL*[1] 80186 PROCESSOR ASSEMBLY LANGUAGE

4.5.1 Values

Values are represented in the *Intel* 80186 processor assembler language by 16-bit 2's complement numbers. All arithmetic is performed using 16 bits of precision. If a constant or the result of evaluating an express requires more than 16 bits to represent it, the least significant 16 bits of the actual value are used.

Every value is an instance of one of the following types:

`UNDEFINED` An `UNDEFINED` value is one whose type has not yet been determined. Examples of `UNDEFINED` values are references to symbols whose definitions are not encountered yet (i.e. forward references) and references to symbols that are defined in programs other than the one currently being assembled (i.e., external references).

`ABSOLUTE` An `ABSOLUTE` value is one that never changes, regardless of relocation of any of the sections of the program being assembled. Examples of `ABSOLUTE` values are numeric constants and arithmetic expressions whose operands are all numeric constants.

`TEXT` A `TEXT` value is a value that is relative to the beginning of the `.text` section. Whenever the `.text` section is relocated by `N` bytes, `N` should be added to or subtracted from every value of the type `TEXT`.

`DATA` A `DATA` value is a value that is relative to the beginning of the `.data` section. Whenever the `.data` section is relocated by `N` bytes, `N` should be added to or subtracted from every value of type `DATA`.

`BSS` A `BSS` value is a value that is relative to the beginning of the `.bss` section. Whenever the `.bss` section is relocated by `N` bytes, `N` should be added to or subtracted from every value of type `BSS`.

4.5.2 Constants

A constant is an element of `ABSOLUTE` type and fixed value. The values can be expressed in any of three different number systems: decimal, octal, or hexadecimal.

Decimal A decimal constant is represented by a string of digits 0 through 9. Each number must begin with a non-zero digit.

```
1234
 325
  43
```

Octal The octal is represented by a string of digits 0 through 7. Each number must begin with a zero digit.

```
 077
0123
0342
```

1. Registered trademark of Intel Corporation.

Hexadecimal

The hexadecimal constant is represend by a string of digits 0 through f. Each number must begin with 0x or 0X prefix. The digits a through f can be represented by either upper or lower case letters.

```
    0x3f
 0x8ABC
  0xFEA
```

4.5.3 Expressions

An expression is a sequence of operands separated by operators. An operand is either a constant, a symbol, or an expression enclosed in square brackets ([ ]). Operands can be omitted; an omitted operand is assumed to have ABSOLUTE type and a value of zero.

The following operators are available:

+ Performs 2's complement addition. If one operand is of the ABSOLUTE type, the result has the type of the other operand; otherwise, the operation is illegal.

– Performs 2's complement subtraction. If the right operand is of the ABSOLUTE type, the result has the type of the left operand. If both operands have the same type, and that type is either TEXT, DATA, or BSS, the result is of the ABSOLUTE type; otherwise the operation is illegal.

* Performs 2's complement integer multiplication. This operation requires that both operands be of the ABSOLUTE type and an ABSOLUTE result is produced. The * must be preceded by a backslash (\) to avoid confusion with the notation for indirection in jumps and calls.

/ Performs 2's complement integer division. Both operands must be of the ABSOLUTE type and the result will be of the ABSOLUTE type. To avoid confusion with the single slash (/), which starts a comment, this operator must be preceded by a backslash (\).

& Performs bit-by-bit logical AND between two 16-bit quantities. Both operands must be of ABSOLUTE type, an ABSOLUTE result is produced.

| Performs bit-by-bit logical OR between two 16-bit quantities. Both operands must be of ABSOLUTE type, an ABSOLUTE result is produced.

>> Shifts the left operand to the right by the number of bits specified by the right operand. This operation requires that both operands be of the ABSOLUTE type, an ABSOLUTE result is produced.

<< Shifts the left operand to the left by the number of bits specified by the right operand. This operation requires that both operands be of the ABSOLUTE type, an ABSOLUTE result is produced.

% Returns the remainder of the operation that divided the first operand by the second operand. Both operands must be of the ABSOLUTE type and an ABSOLUTE result is produced. To avoid confusion with register names, this operator must be preceded by a backslash (\).

! Performs bit-by-bit logical AND between the first operand and the

bit-by-bit complement of the second operand. This operation requires that both operands are of the ABSOLUTE type and an ABSOLUTE result is produced.

^ This is the only operator that allows you to alter the usage requirements for the types of symbols. It returns a quantity that has the value of the first operand the type of the second. The operands can be of any type.

4.5.4 Machine Instruction Notation

The iAPX assembly language is not the same as *Intel* processor assembly language. The order of operands is changed to reflect the overall usage in the *5ESS*® switching system. The iAPX assembler recognizes a large number of pseudo-ops for specifying symbol attributes for symbolic testing.

There are several differences between iAPX assembly language and *Intel* processor assembly language:

- All register names use percent sign (%) as a prefix to distinguish them from symbol names. (See Tables 4.5-1 and 4.5-2.)
- Instructions with two operands use the left as the source and the right as the destination. This follows the *UNIX*[2] system's assembler convention, and it is reversed from *Intel* processor notation.
- Every w at the end of a word instruction has been dropped, and b has been added as a suffix to byte operations.
- Two additional operators <o> and <s> have been added to handle 20-bit pointers. They stand for offset and segment, respectively.

Table 4.5-1 — iAPX Word Registers

Symbolic Name	Description
%ax	Register A (Accumulator)
%bx	Register B (Base)
%cx	Register C (Count)
%dx	Register D (Data)
%sp	Stack pointer
%bp	Base pointer
%si	Source (or stack) index
%di	Destination (or data) index
%cs	Code segment register
%ds	Data segment register
%es	Extra segment register
%ss	Stack segment register

2. Registered trademark of The Open Group.

Table 4.5-2 — iAPX 8 Byte Registers

Symbolic Name	Description
%ah	High byte of %ax
%al	Low byte of %ax
%bh	High byte of %bx
%bl	Low byte of %bx
%ch	High byte of %cx
%cl	Low byte of %cx
%dh	High byte of %dx
%dl	Low byte of %dx

4.5.5 Operands

Three kinds of operands are generally available to instructions: register operands, memory operands, and immediate operands. Indirect operands are available to `jump` and `call` instructions; but no other instructions can use indirect operands.

Register operands
Register operands are encoded in the instruction in just a few bits and there operations are performed entirely within the central processor. They may serve as source operands, destination operands, or both.

Immediate operands
The immediate operands are constant data contained in an instruction. They can be either 8 or 16 bits in length, but they can only serve as source operands.

Memory operands
The memory operands, unlike the register, must be transferred to and from the processor.

Almost all *Intel* 80186 processor instructions can operate on either 8-bit (byte) or 16-bit (word) operands, though some references will require a double word or 32 bits. Memory can be accessed as bytes or words, and words do not have to begin on an even address. Each of the three kinds of operands can be either a byte or a word.

Operands in the current data segment of memory are addressable directly with a 16-bit offset address, or indirectly with base (`bx` or `bp`) and/or index (`si` or `di`) registers added to a displacement constant of 8 or 16 bits, the displacement constant is optional. Operands which reside in the memory are addressable in four ways:

Direct 16-bit offset address	The effective address is taken directly from the displacement field of the instruction.
Indirect base register	The effective address is the sum of a displacement and the contents of the `BX` register or the `BP` register.
Indirect index register	The effective address is determined by adding the displacement and the contents of an index register.
Indirect base register and index register	The effective address is determined by adding a base register `BP` or `BX`, and an index register, `SI` or `DI`, and a displacement from the segment register.

The operands location in a register or in the memory is denoted by three fields in the instruction. These fields are the mode field, the register field, and the register/memory field. (See Figure 4.5-1). The second byte of the instruction sequence will contain the operands when they are used.

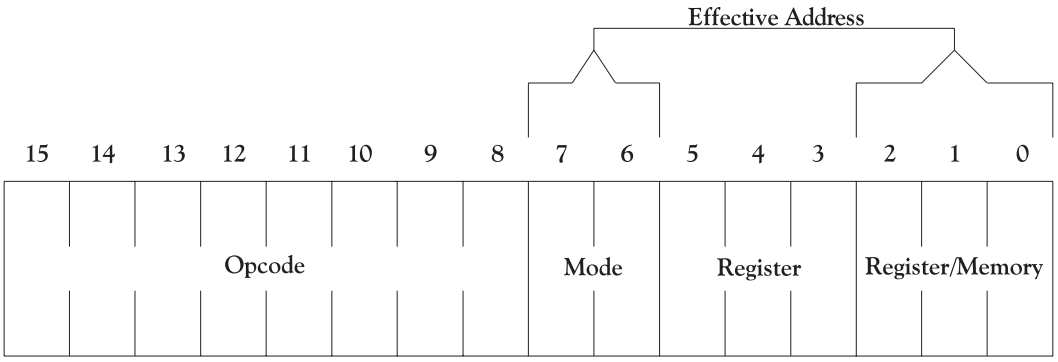


Figure 4.5-1 — *Intel* 80186 Processor Instruction — Memory Layout

The two most significant bits of the byte contain the mode field and determine how the register/memory field (bits 2,1,0) is used in locating the operand. The register/memory field can name a register that holds the operand or can specify an addressing mode that points to the location of the operand in memory. The register field occupies bits 5, 4, and 3 following the mode field, and can specify that one operand is either an 8-bit register or a 16-bit register.

The effective address (EA) is the offset that is calculated for a memory operand. The effective address (EA) of the memory operand is determined according to the mode and register/memory fields by summing a displacement, the contents of a base register and the contents of an index register. See Tables 4.5-3 and 4.5-4.

Table 4.5-3 — *Intel* 80186 Processor Effective Address — Mode Field

Mode	Displacement
00	0, disp-low and disp-high are absent
01	disp-low sign-extended to 16 bits, disp-high absent
10	disp-high

Table 4.5-4 — *Intel* 80186 Processor Effective Address — Register/Memory Field

Register/Memory	Effective Address
000	bx + si + disp
001	bx + di + disp
010	bp + si + disp
011	bp + di + disp
100	si + disp
101	di + disp
110	bp + disp
111	bx + disp

4.5.6 Instruction Set

The instructions set for the iAPX is based on the *Intel* 80186 processor family instruction set. The operations are performed on bytes, words, and long words. See Figure 4.5-2.

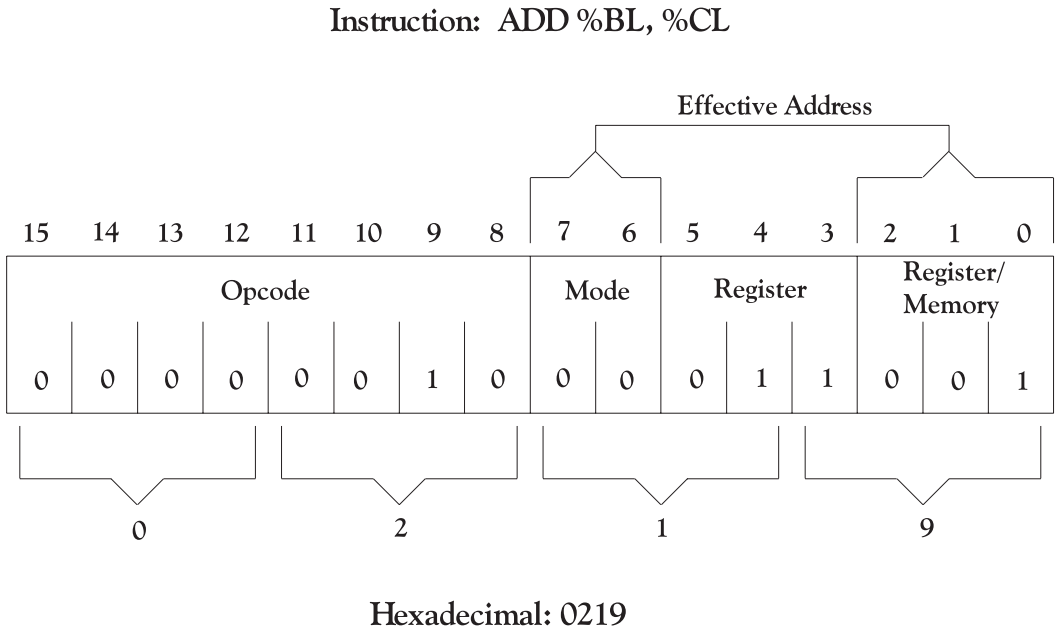


Figure 4.5-2 — *Intel* 80186 Processor Example Instruction Memory Layout

For a complete listing of the instruction set for iAPX see Appendix A4 - *Intel* 8086 and 80186 Processor Instruction Set.

4.6 *INTEL*[1] 80186 PROCESSOR MACHINE DEPENDENCIES

4.6.1 Data Type Memory Boundaries

The processors in the *5ESS*® switch have specific memory organizational requirements, each processor having variations in size and functionality. The following gives a brief explanation of the byte and word organization of the *Intel* 80186 processor.

Byte-length data which contains 8-bits:

- can start in any memory location
- represents an 8-bit string or binary number between 0 and 255.

Word-length data which contains 16 bits:

- can start in any even-numbered memory location
- represents a 16-bit string or unsigned binary number between 0 and 65,535 or 2s complement binary number between –32,768 and 32,767 (bit 15 is the sign bit).

Long Word-length data which contains 32-bits:

- can start in any even-numbered memory location
- represents a 32-bit string or unsigned binary number between 0 and 4,294,968,295 or 2s complement binary number between –2,147,483,648 and 2,147,483,647 (bit 31 is the sign bit).

Table 4.6-1 details the data size and alignment of the *Intel* 80186 processor. When analyzing output from this processor it is important to reference the data types according to the contents of this table.

Table 4.6-1 — *Intel* 80186 Processor Data Sizes and Alignment

Data type	Size	Memory Alignment
`char`	1 byte	no alignment
`short`	2 bytes	2 byte boundary
`int`	2 bytes	2 byte boundary
`long`	4 bytes	2 byte boundary
`pointer`	4 bytes	2 byte boundary
`structure`	4 byte multiple	2 byte boundary
`union`	4 byte multiple	2 byte boundary
`array`	same as element type	same as element type
`bitfield`	2 bytes	2 byte boundary
Data inside `structures` and `unions`		
`long`	4 bytes	4 byte offset
`pointer`	4 bytes	4 byte offset
`structure`	4 byte multiple	4 byte offset
`union`	4 byte multiple	4 byte offset

1. Registered trademark of Intel Corporation.

Figure 4.6-1 outlines the memory boundaries that would be established by the *Intel* 80186 processor.

```
Given:
 char a
 short b
 int  c
 long d
 pointer e
```

The data would be placed on the data stack as follows:

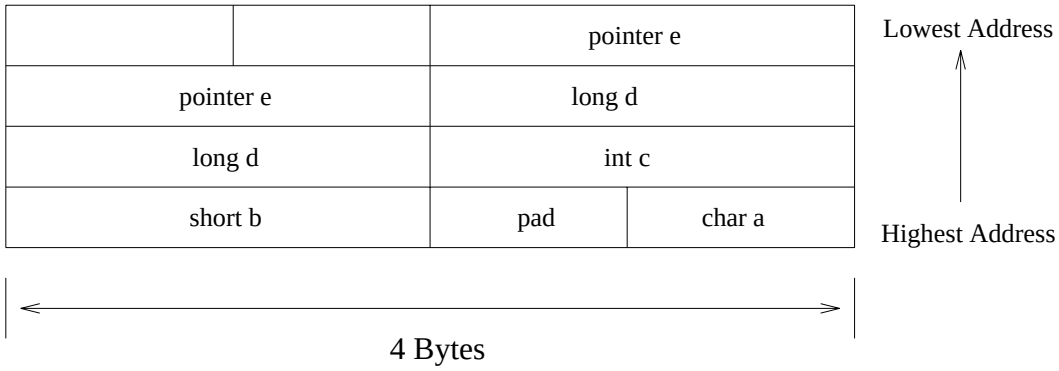


Figure 4.6-1 — *Intel* 80186 Processor Stack –– Memory Boundaries

4.6.2 Arithmetic Types Supported

The *Intel* 80186 compiler in the *5ESS* switch supports four arithmetic types. All four types can be `signed` or `unsigned`.

- `char`
- `short`
- `int`
- `long`

4.6.3 Data Conversion Rules

Table 4.6-2 — *Intel* 80186 Processor Data Conversions Rules

Data Type Conversion	Char	Unsigned Char	Int	Unsigned Int	Long	Unsigned Long	Pointer
Char	—	NC	PL	PL	PL	PL	PL
Unsigned Char	NC	—	PL	PL	PL	PL	PL
Int	TL	TL	—	NC	SE	PL	PL
Unsigned Int	TL	TL	NC	—	PL	PL	PL
Long	TL	TL	TL	TL	—	NC	TL/PL
Unsigned Long	TL	TL	TL	TL	NC	—	TL/PL
Pointer	TL	TL	TL	TL	NC	NC	—

NC = Name Change, TL = Truncate on Left, SE = Sign Extend, PL = Pad on Left with zero(s), TL/PL = Truncate on left to 24 bits and pad on left with one byte of zero

4.6.4 Memory Configuration

On the *Intel* 80186 processor an address has two components: segment number (a segment register), and a 16-bit offset. For any memory reference, the content of the four segment registers is added to the offset to form an address. Segment registers are set to the base address of a region, and all references in the text are resolved with respect to the virtual address. A region refers to a range of memory that begins with a virtual address of zero.

There are three restrictions on regions:

- The first is that they must begin at an address aligned to a 16-byte boundary; that is, the low four bits of the address must be zero. (For example, 0, 16, 32, etc. are valid region origins, in hexadecimal the values would be 0, 10, 20.) This is because the 20-bit physical address is stored in a segment register as a 16-bit quantity and shifted left four bits before adding the offset.
- The second restriction is that regions may not exceed 64K in size, or else there can be no direct references to addresses beyond 64K into the region.
- The third is that the physical memory assigned to user-specified regions may not overlap.

A segment register will contain a 16-bit value that is used to point to the start or base of a physical segment. The contents of that segment register will determine the upper 16 bits or a 20-bit address. The hexadecimal address 12340 is given as 1234. When a segment register points to this location, it is loaded with the value 1234, which defines a 64K byte segment starting at absolute address 12340. (See Figure 4.6-2.)

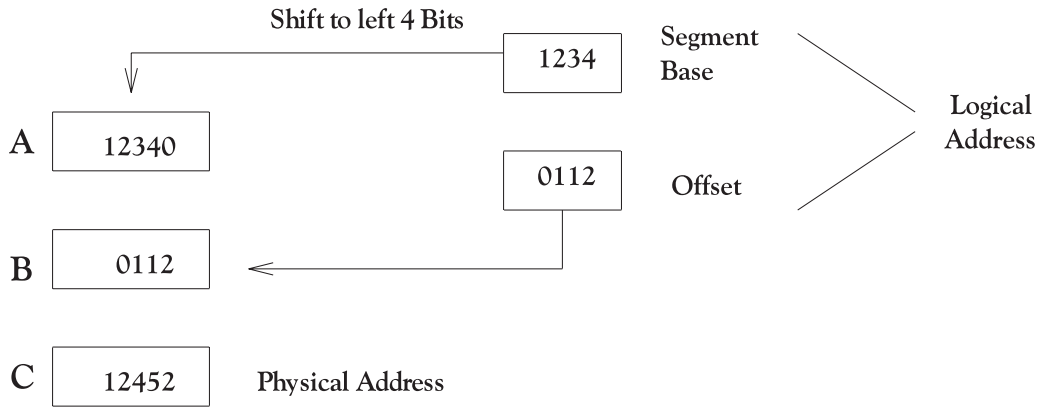


Figure 4.6-2 — Memory Addressing in the *Intel* 80186 Processor

A section of an object file is the smallest unit of relocation and must be a contiguous block of memory. A section is identified by a starting address and a size. Information describing all the sections in a file is stored in "section headers" at the start of the file.

The physical address of a section or symbol is the relative offset from address zero of the address space. The virtual address of a section or symbol refers to the 16-bit relocatable address with respect to the beginning of its region.

The virtual memory for the *Intel* 80186 processor is partitioned into configured and unconfigured memory. The default condition is to treat all memory as configured. Unconfigured memory is treated as reserved or unusable by the link editor. Nothing can be linked into unconfigured memory.

When `MEMORY` directives are used, all virtual memory not described in a `MEMORY` directive is considered to be unconfigured. Unconfigured memory is not used in the link editor's allocation process, and hence nothing can be link edited, bound, or assigned to any address within unconfigured memory.

As an option on the `MEMORY` directive, attributes may be associated with a named memory area. This restricts the area to which an output section can be sent. The attributes assigned to output sections in this manner are recorded in the appropriate section headers in the output file, to allow for possible error checking in the future.

The attributes that are currently accepted are:

`R`	Readable memory
`W`	Writable memory
`X`	Executable memory
`I`	Initializable memory.

By specifying `MEMORY` directives, the link editor can be told that memory is configured in some manner other than the default.

4.6.5 Register Notation

The *Intel* 80186 processor consists of a set of eight 16-bit general registers. These general registers are divided into two sets: the first is the data registers and the second set consists of the pointer and the index registers. In the data registers each 16-bit register is divided into two 8-bit registers which allow its upper (high) and lower halves to be separately addressed. Therefore, each data register can be used interchangeably as a 16 bit-register, or as two 8-bit registers, Table 4.6-3.

Table 4.6-3 — *Intel* 80186 Processor Data Registers

16-Bit Register	16-Bit Description	8-Bit Register	8-Bit Description
%ax	Register A (Accumulator)	%ah %al	The high byte of %ax. The low byte of %ax.
%bx	Register B (Base)	%bh %bl	The high byte of %bx. The low byte of %bx.
%cx	Register C (Count)	%ch %cl	The high byte of %cx. The low byte of %cx.
%dx	Register D (Data)	%dh %dl	The high byte of %dx. The low byte of %dx.

Pointer and index registers are usually for addressing data in memory (see Table 4.6-4).

Table 4.6-4 — *Intel* 80186 Processor Pointer and Index Registers

Register	Description
%sp	Stack pointer
%bp	Base pointer
%si	Source index
%di	Destination index

In addition, four special word registers called segment registers are used in addressing. They are not generally available as operands, but some instructions do allow them. The *Intel* 80186 processor is divided into segments of 64K bytes. The four segment registers give the base locations for these segments, Table 4.6-5.

Table 4.6-5 — *Intel* 80186 Processor Segment Registers

Register	Description
%cs	Code segment register — all references to the instruction space use this register.
%ds	Data segment register — the default register for most references to memory operands.
%es	Extra segment register — alternative segment register for data.
%ss	Stack segment register — the default segment register for references to the function linkage stack.

The *Intel* 80186 processor also contains status and control registers, as shown in Table 4.6-6 and Figure 4.6-3.

Table 4.6-6 — *Intel* 80186 Processor Status and Control Registers

Register	Description
%ip	Instruction pointer register. This register is equivalent to the program counter.
Status Word/Flags	The Status word or flags consists of three control flags and six status flags. The status flags record the results of logical and arithmetic instructions and the control flags direct the operation of the central processor within a particular operating mode.

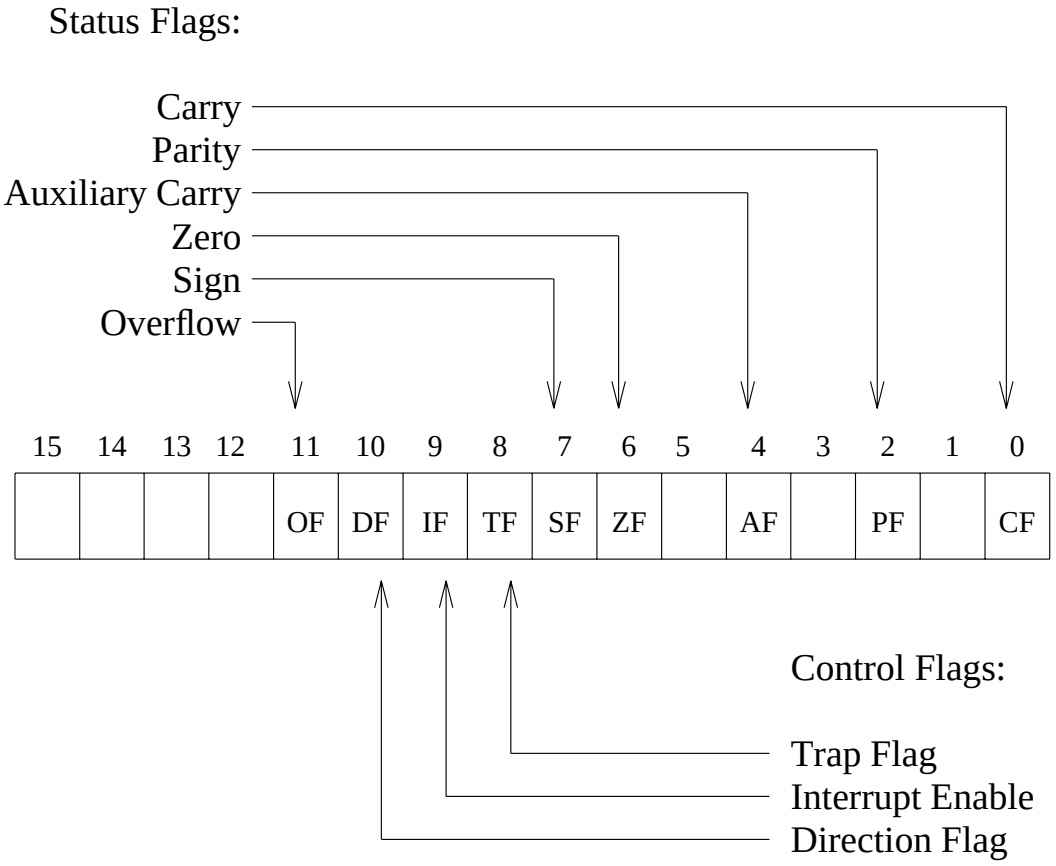


Figure 4.6-3 — Status Word or Flags

4.6.6 Stack Usage

The *Intel* 80186 processor stack frames are accessed by the stack segment register, `%ss`, the stack pointer register, `%sp`, or the base pointer register, `%bp`. A stack can be up to 64K bytes long, which is the maximum length of a segment. The `%ss` register contains the base address of the stack and the `%sp` register points to the top of the stack. Stacks are 16 bits wide, meaning that an instruction that operates on stacks removes stack items one word at a time.

Two function linkages are supported (transfer vector and direct) and two addressing options (16 and 20 bit). As a result, there are three stack frames which are similar, yet not identical. The three distinct stack frames are for:

1. 16 bit direct linkage,
2. 16 bit transfer vector linkage, and
3. 20 bit, both transfer vector and direct linkage.

A transfer vector (TV) is an ordered list of entries, similar to an array of pointers or jump table. Each entry contains the physical address of an external or static identifier. The address is one word long, and the address is right-justified within the word. The

entire transfer vector is a zero-origined, one-dimensional array of long integers. The first slot in the transfer vector cannot be used because of a conflict with the null pointer, whose value is zero.

The stack frame grows from high addresses in the stack segment that is pointed to in memory by the segment register `%ss`. A stack frame is created for each instance of a function call at runtime. It is destroyed at the time the function makes a return to the calling function. Each stack frame contains the information needed to restore the calling function to its state before it made the function call in the save area. It also contains the arguments passed to it by the caller, space for its automatic variables, and space for any temporaries needed during execution. The iAPX 20 bit stack also contains the normalized frame pointer, i.e., the address `%ss:%bp` in normalized form. Diagrams of samples of each of the three stack frames are shown in Figures 4.6-4, 4.6-5, and 4.6-6.

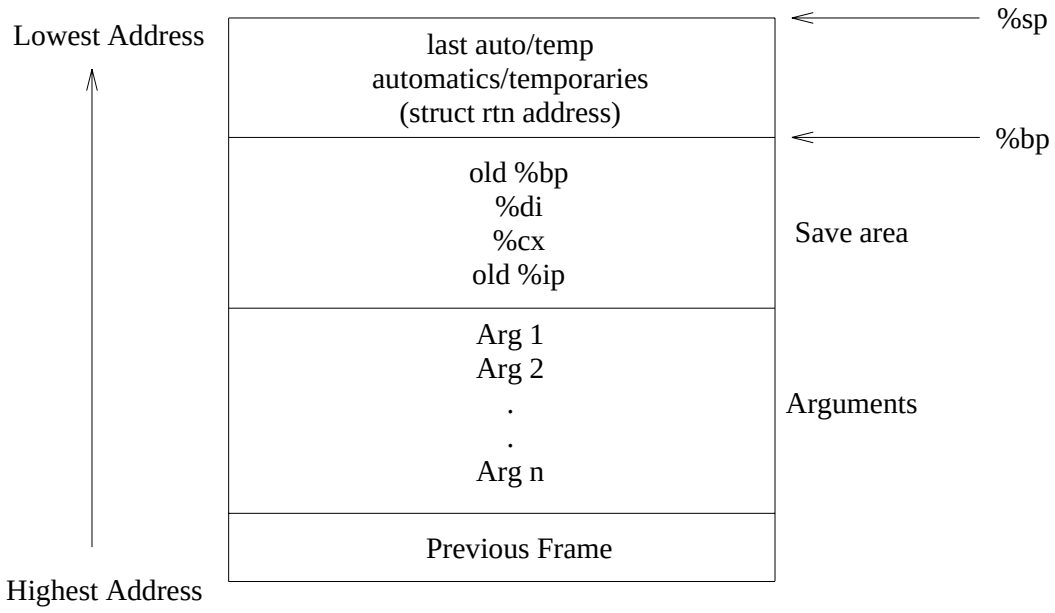


Figure 4.6-4 — iAPX-16 bit, Direct Linkage (No TV)

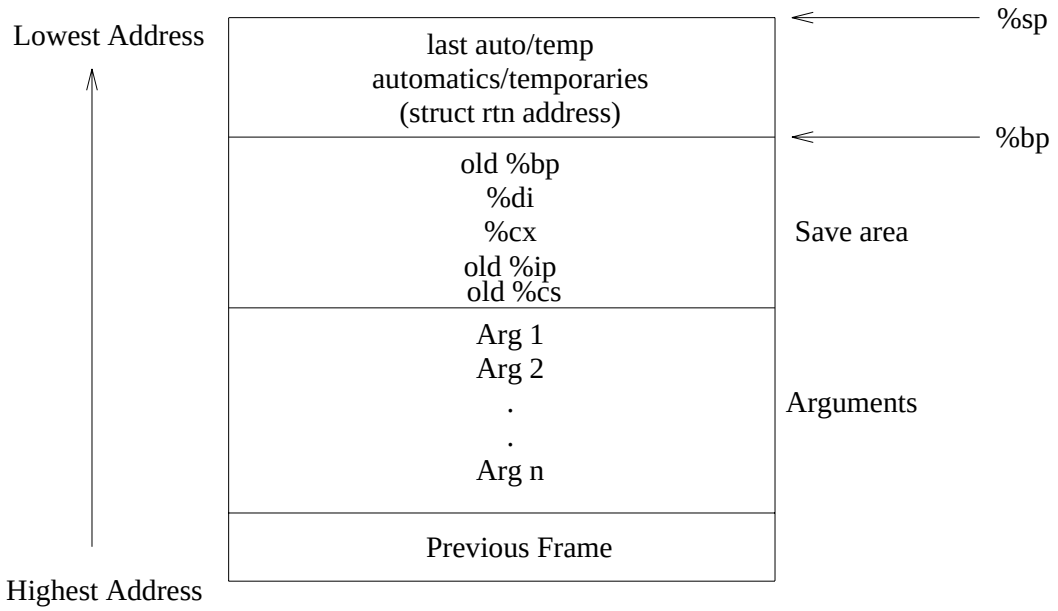


Figure 4.6-5 — iAPX-16 bit, Transfer Vector Linkage

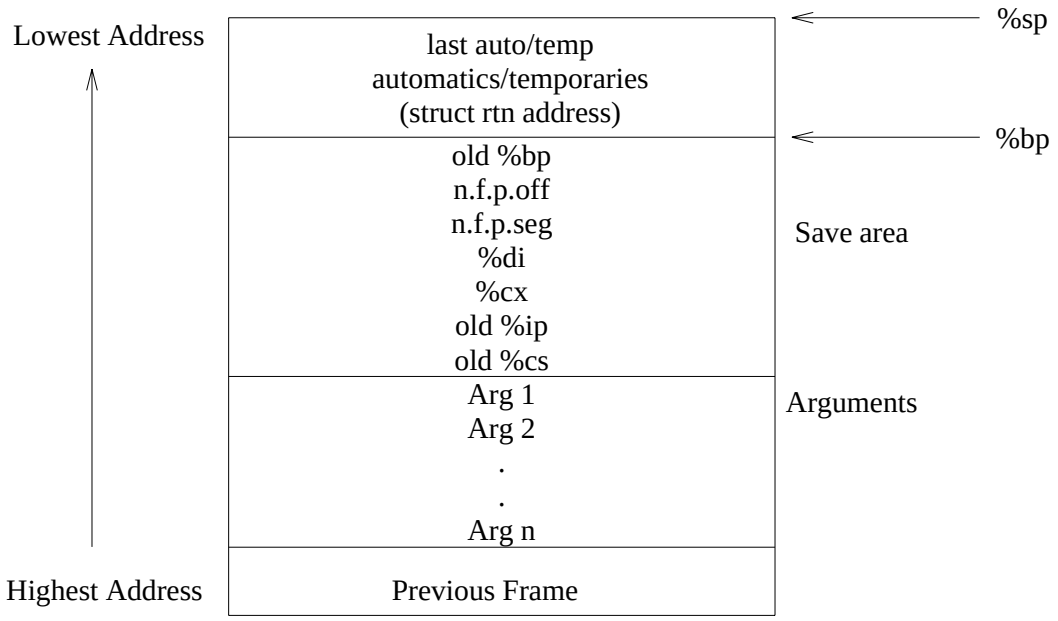


Figure 4.6-6 — iAPX-20 bit, Transfer Vector and Direct Linkage

Two register variables are saved along with the old frame pointer `%bp` and the old instruction pointer `%ip`. In iAPX 16 bit with TV and in iAPX 20 bit with and without

TV, the old code segment register (%cs) is also saved. Thus 4, 5, and 7 words are required for the respective save areas. The frame pointer (%bp) always points to the old frame pointer. The stack pointer (%sp) always points to the top word of the stack. All accessing of data in the stack frame by a routine is done with an offset from the frame pointer.

Automatic storage begins with the first byte following the old frame pointer in the stack. The area for automatics contains an even number of bytes as will the area for temporaries that follows it to ensure that the next stack frame begins on a word boundary. For efficiency, integers and larger entities are aligned on word boundaries; characters alone can have odd offsets from %bp. The first word of automatic storage is address c(%bp)−2. Since push and pop operations always work with word operands, the argument space is also an even number of bytes, giving all arguments an even offset from %bp. Argument 1 is the first at location c(%bp)+8, c(%bp)+10, or c(%bp)+14 for each of the respective cases. Again, beware that if the optimizer is shortening the stack frame, the first argument may be at a lower offset.

The stack frame, as shown previously, is typical during the execution of a function. When a function proceeds to make a subroutine call it must first place the arguments on the stack in the reverse order that they appear in the function call. Every argument must be pushed on the stack as an even number of bytes. Characters are cast into integers, and structures of uneven length are filled out to a word boundary, though the last byte is meaningless. If the call is to a function that returns a structure, the address of the structure return area is placed in the register %si (and %dx for 20 bit addressing) prior to the call. The calling function then executes a call to the desired function. After the call has been executed, the stack frame appears as shown in Figure 4.6-7.

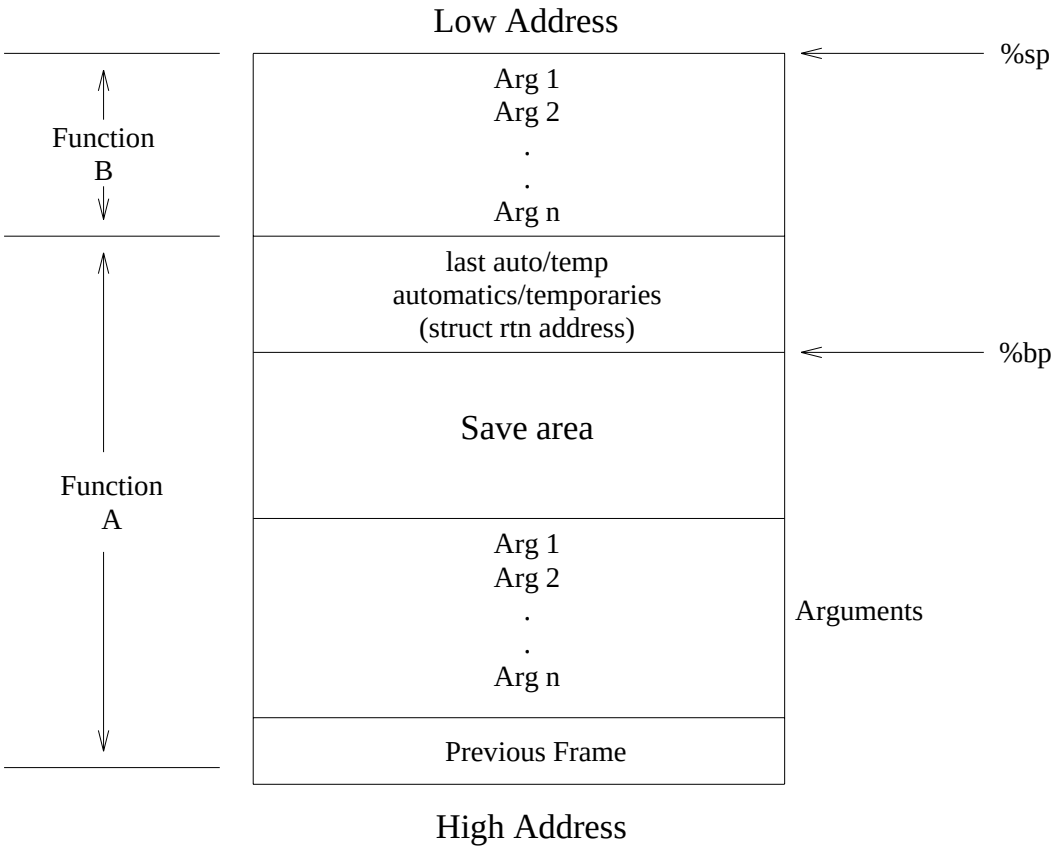


Figure 4.6-7 — iAPX Stack Frame after Function Call

The first responsibilities of the called function are to perform the context save, the saving of the system conditions at the time of the call, for the caller, to set up the new frame, and to allocate space on the stack for the automatics and temporaries of the called function. If the function is one that returns a structure, the return address of the structure must be moved from the register `%si` (and `%dx`) to the position on the stack of the first automatic.

The first thing that the called function does is to call the appropriate system routine to perform the context save for the caller. There are eight versions of the routine for each of the combinations of 16 or 20 bit addressing, transfer vector, or direct linkage for iAPX186. The call to this routine pushes another return address on the stack that is immediately popped and placed into the `%bx` register [and `%dx` (16 TV) or `%ds` (20 bit)]. With the stack pointer now pointing at the original return address, the routine pushes the two register variables `%cx` and `%di` into the save area.

The final responsibility of the called function, if it is to return a structure, is to move the structure return address from the scratch registers, `%si` and (`%dx`) to the stack. The result of the operation leaves the stack in the condition shown in Figure 4.6-8. Only after this has been accomplished will the called function begin to execute.

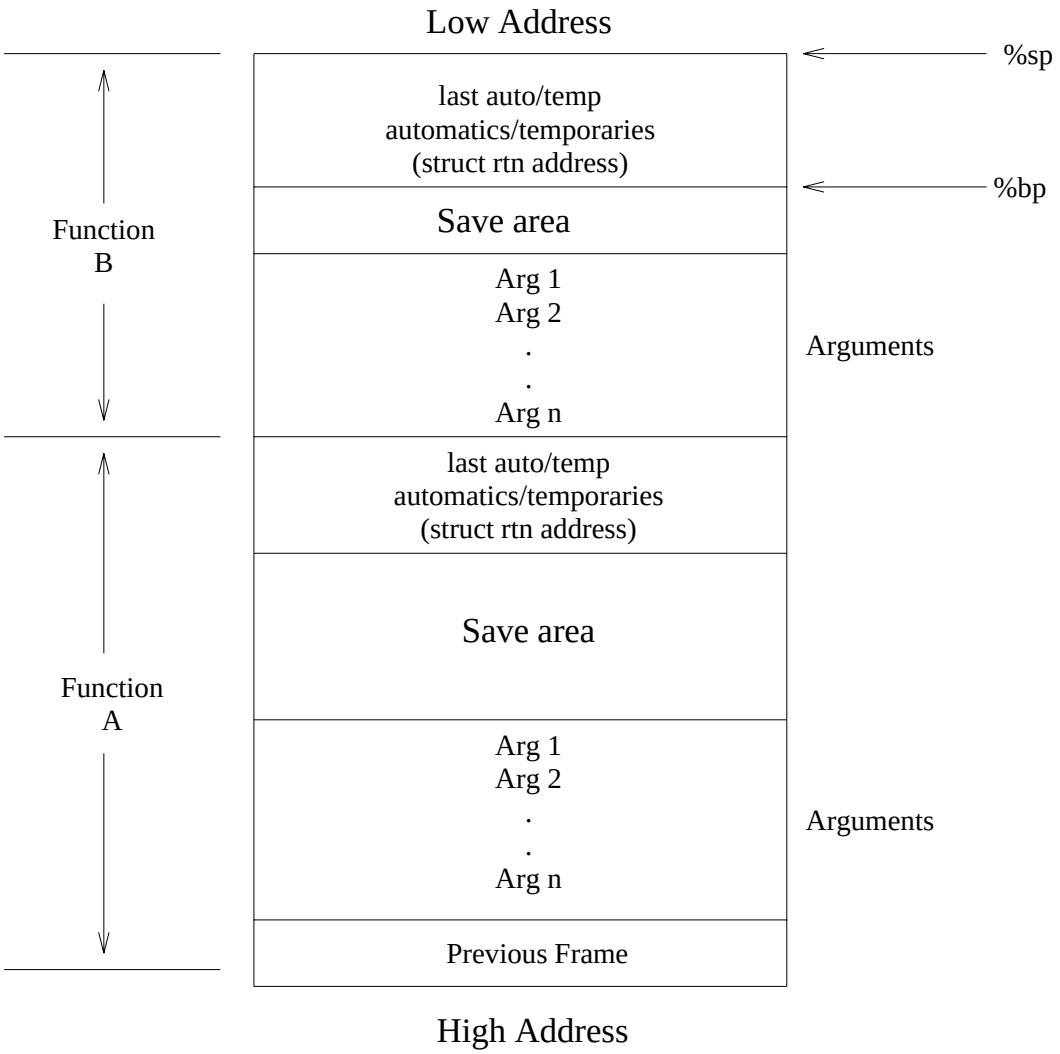


Figure 4.6-8 — iAPX Stack Frame during Called Function (B) Execution

To return to the caller, a function executes a `jump` to the proper system routine which first loads `%sp` with the contents of the frame pointer, destroying the space for all automatics and `pops %bp` restoring it to its old value. Next for 20 bit addressing code, 4 is added to `%sp` destroying the space in the stack frame for the normalized frame pointer. Then the next two words of the stack are popped into `%di` and `%cs`, respectively, restoring the state of the calling function. Finally, a `return` is executed popping the old `%ip` (and the old `%cs`) from the stack. The calling function has one last responsibility; it must deallocate the space used by the arguments by adding an appropriate constant to the stack pointer. After this, the stack frame for the called function is completely destroyed.

A stack trace is very easy to perform since `%sp` always points to the last word in the save area (the old frame pointer). The chain of frame pointers can be followed

backward until a desired previous invocation of a function is found. By examing the old `%ip` and comparing this address with the known beginning and ending addresses of a function, the previous invocation of a function can be found.

Software Analysis Guide

CONTENTS PAGE

5. ASSERT ANALYSIS 5-1

5.1 ASSERT ANALYSIS DESCRIPTION 5.1-1
5.1.1 ASSERT DEFINITION 5.1-1
5.1.1.1 Assert Messages 5.1-1
5.1.1.2 Assert Handler 5.1-1
5.1.1.3 Assert Macros 5.1-2
5.1.2 RECEIVE ONLY PRINTER OUTPUT (ROP) 5.1-5
5.1.2.1 ROP Messages 5.1-5
5.1.2.2 Defensive Check Failure (DCF) Asserts . . . 5.1-5
5.1.2.3 MANUAL ACTION Asserts 5.1-6
5.1.2.4 RTA Asserts 5.1-6
5.1.3 STACK TRACE DEBUGGING 5.1-7
5.1.3.1 Stack Formatting Overview 5.1-7
5.1.3.2 AM Stack Layout (3B2XD Processor) 5.1-8
5.1.3.3 SM/CMP Stack Layout (MC68XXX Processor Family) 5.1-11
5.1.4 FAILING FUNCTION 5.1-15

5.2 ASSERT ANALYSIS EXAMPLES 5.2-1
5.2.1 ASSERT ANALYSIS EXAMPLE — AM 5.2-1
5.2.2 ASSERT ANALYSIS EXAMPLE — SM 5.2-1
5.2.3 ASSERT ANALYSIS EXAMPLE — CMP 5.2-9
5.2.4 ASSERT ANALYSIS EXAMPLE — RTA DCF 5.2-10
5.2.4.1 RTA DCF ROUTING Error 5.2-10
5.2.4.2 RTA DCF: RTDUMPDATA() 5.2-12

LIST OF FIGURES

Figure 5.1-1 — AM Assert Formatting 5.1-11

Figure 5.1-2 — SM Stack Frame Format 5.1-13

Figure 5.1-3 — SM Local Data Stack Frame 5.1-15

Figure 5.2-1 — Stack Grid 5.2-8

LIST OF TABLES

Table 5.1-1 — 3B2xD Processor vs. MC68XXX Processor Stack Variations 5.1-8

Table 5.1-2 — 3B2XD and MC68XXX Processor Family Data Sizes and Alignments 5.1-10

LIST OF EXHIBITS

Exhibit 5.1-1 — Example of CALLPLOG1 Output 5.1-7

Exhibit 5.2-1 — SM Assert Printout 5.2-1

Exhibit 5.2-2 — Disassembly Code for the Function SMiclvco 5.2-3

Exhibit 5.2-3 — C code for the Function SMiclvco 5.2-4

Exhibit 5.2-4 — CMP Assert Printout 5.2-10

Exhibit 5.2-5 — RTA RTRTGERR 5.2-10

Exhibit 5.2-6 — RTnw_conn.c RTRTGERR() Error Example 5.2-11

Exhibit 5.2-7 — Routing States 5.2-11

Exhibit 5.2-8 — Systems Integrity RTA DCF: Data Dump Report. 5.2-12

Exhibit 5.2-9 — CALLPLOG1 File Output Example 5.2-13

5. ASSERT ANALYSIS

This section discusses the analysis procedures to be used on assert components such as stimulus message, stack trace, and stack frame for the AM, CMP, and the SM processors. It presents methods that can be used to determine assert classifications such as processor, environment, and recovery strategies. It also presents ways of locating the asserting function and tracing the calling functions. The objective of this section is to impart the necessary skills to analyze an assert.

5.1 ASSERT ANALYSIS DESCRIPTION

5.1.1 ASSERT DEFINITION

5.1.1.1 Assert Messages

An assert is a report of a software error or inconsistency detected during program execution. Asserts may simply report these events or initiate a range of corrective actions. Three types of assert messages are generated by the *5ESS*® switch:

1. Defensive Check Failure (DCF)
2. MANUAL ACTION (a.k.a. Craft asserts)
3. RTA ROUTING

DCF and MANUAL ACTION asserts report software-detected errors in all subsystems of the switch. RTA ROUTING asserts report errors in the RTA subsystem.

DCF assert messages are the most common and are triggered for a variety of reasons, including invalid, missing, or inconsistent data detected by data verification statements. MANUAL ACTION asserts report problems that require documented intervention by switch technicians. RTA ROUTING asserts report software errors detected during the routing states of call processing.

The types of checks used with asserts include:

- Range checks on pointers, global data, and function arguments to ensure that reads and writes occur only within restricted ranges
- Redundancy checks made on duplicate copies of data stored at different locations to detect memory mutilation
- Point–to/point–back linkage checks, key checks
- Consistency checks between logically related blocks of data; for example, in one data block a resource may be marked as idle while in another data block it may be marked as in use
- Validity checks on return values from function calls.

Asserts may cause a number of messages to be printed. See "Receive Only Printer Output," Section 5.1.2 for more detail.

5.1.1.2 Assert Handler

5.1.1.2.1 Assert Handler Definition

The assert handler is the software called to process an assert. The assert handler is responsible for the reporting and the recovery of an assert. The assert handler does not correct errors directly. Each of the three types of asserts has a different assert handler capable of performing different actions.

5.1.1.2.2 DCF Assert Handler

The DCF assert handler can do one or more of the following:

- Dump data relevant to the error. This includes process-related data and data specified by the assert macro, including stack traces and stack frames.
- Invoke audits
- Initiate single process purges

- Initiate selective initialization
- Escalate/de–escalate the requested recovery action.

5.1.1.2.3 MANUAL ACTION Assert Handler

The MANUAL ACTION assert handler outputs a concise message describing the error, dumping relevant data and calling for manual action. See "Receive Only Printer Output," Section 5.1.2 for more detail.

5.1.1.2.4 RTA Assert Handler

The RTA assert handler analyzes the assert and dumps data relevant to the reported error. See "Receive Only Printer Output," Section 5.1.2 for more detail.

5.1.1.3 Assert Macros

5.1.1.3.1 Assert Macros Description

Asserts are invoked via macros which interface with the assert handlers. The different assert types are invoked with different macros as described here.

5.1.1.3.2 DCF Assert Macros

The assert macros used to invoke DCF asserts are:

AUASRTA — This macro is used to format the basic assert information. The assert output includes a stimulus message, a stack trace, two stack frames (one for each of the two functions that precede the point of assertion), and a register dump message. This is the most common type of assert. This macro can:

- Schedule an audit
- Purge the running processes
- Cause a selective init.

This macro is given two pieces of information:

1. The first is a condition that should be true (i.e,. an assertion). The assert will fire only when the condition is false.
2. The second is the actual assert mnemonic used to identify the specific error that has been detected and specifies what action should be taken.

The macro call would be of the form:

```
AUASRTA(CONDITION,MNEMONIC);
```

ASSERTB1
ASSERTB2
ASSERTB3

These macros provide the same information as the AUASRTA macro but also provide additional data specified by a list of pointers. ASSERTB[123] will be found within a failure leg of code and will contain up to three pointers to data. This data will print out in Report Data messages. See "Receive Only Printer Output," Section 5.1.2 for more detail. The ASSERT B macros are available only in the OSDS environment.

These macros are given two, three, or four pieces of information respectively:

1. The first is the assert mnemonic that identifies a specific error.
2. The second, third, and fourth are pointers to pertinent data. The data pointed to by each pointer is output by a separate REPT DATA message.

The macro calls would be of the form:

```
ASSERTB1(MNEMONIC,ptr1);

ASSERTB2(MNEMONIC,ptr1,ptr2);

ASSERTB3(MNEMONIC,ptr1,ptr2,ptr3);
```

AUASRTC

This macro is identical to the AUASRTA macro except that it will also purge a specified non–running process.

This macro is given three pieces of information:

1. The first is a condition that should be true. The assert will only fire when the condition is false.
2. The second is the assert mnemonic used to identify the specific error that was detected.
3. The third is the process number of the process to be purged.

The macro call would be of the form:

```
AUASRTC(CONDITION,MNEMONIC,PROCESS_NUMBER);
```

ASSERTD1
ASSERTD2
ASSERTD3

These macros are a combination of an ASSERT B and an ASSERT C. They provide the ability to dump additional data specified by a set of pointers plus the ability to purge a specified non-running process.

These macros are given three, four, or five pieces of information:

1. The first is the assert mnemonic which identifies a specific error.
2. The second is the process number of the process to be purged.
3. The third, fourth, and fifth are pointers to pertinent data.

The macro calls would be of the form:

```
ASSERTD1(MNEMONIC,PROCESS_NUMBER,ptr1);

ASSERTD2(MNEMONIC,PROCESS_NUMBER,ptr1,ptr2);

ASSERTD3(MNEMONIC, PROCESS_NUMBER,ptr1,ptr2,ptr3);
```

5.1.1.3.3 MANUAL ACTION Assert Macros

Two macros are used for manual action asserts:

AUCFTASRT This macro will format one assert output message containing pertinent data only. See "Receive Only Printer Output," Section 5.1.2 for more detail. The source of the assert is indicated via a source file name and line number within that source file.

This macro is given up to eleven pieces of information:

1. The first is the assert mnemonic.
2. The second is the format string of the desired output message written in double quotes or a pointer to a character string.
3. The remaining nine define values or character strings to be printed according to the corresponding formatting instruction in the format string. If the information pieces are unused, they are specified as 0.

   The macro call would be of the form:

   ```
   AUCFTASRT(ASRT_NO,FORMAT_STRING,DATA1,...,DATA9);
   ```

AUCFTREFASRT

This macro will format one assert output message containing pertinent data only. See "Receive Only Printer Output," Section 5.1.2 for more detail. The source of the assert is indicated via a source file name and a reference number.

This macro is given up to twelve pieces of information:

1. The first is a reference number to identify the assert location within the source file.
2. The second is the assert mnemonic.
3. The third is the format string of the desired output message written in double quotes or a pointer to a character string.

4. The remaining nine define values or character strings to be printed according to the corresponding formatting instruction in the format string. If the information pieces are unused, they are specified as 0.

The macro call would be of the form:

```
AUCFTREFASRT(REF_NO,ASRT_NO,FORMAT_STRING,DATA1,...,DATA9);
```

5.1.1.3.4 RTA Assert Macros

The following two macros are used with RTA asserts:

RTRTGERR This macro is used by the RTA subsystem. It reports software errors detected during the routing states of call processing.

The RTRTGERR macro is given three pieces of information:

1. The first is an error type (RTERRTYPE).
2. The second is an error cause.
3. The third is an enumeration of the type of data to be printed via the RTDUMPDATA macro.

The macro call would be of the form:

```
RTRTGERR(ERROR_TYPE,ERROR_CAUSE,DATA_TYPE);
```

RTDUMPDATA

This macro will print data to the /log/log/CALLPLOG1 file on the AM. The data may reside on the asserting processor or may be retrieved from a remote processor.

The RTDUMPDATA macro is given four pieces of information:

1. The first is an enumeration indicating the type of the data (a list of possible values can be found in the RTdumpdata.h header file).
2. The second is a pointer to the data, if the data resides on the originating processor (otherwise this parameter must be 0).
3. The third is the process ID of the owner of the data to be dumped.
4. The fourth is the processor number of the originating processor.

The macro call would be of the form:

```
RTDUMPDATA(DATA_TYPE,DATA_PTR,PROCESS_ID,ORIGINATING_PROCESSOR);
```

5.1.2 RECEIVE ONLY PRINTER OUTPUT (ROP)

5.1.2.1 ROP Messages

Asserts may print a number of messages on the ROP. Refer to the 235-600-500, *Asserts Manual* and the 235-600-750, *Output Messages Manual* for more information on these output messages.

5.1.2.2 Defensive Check Failure (DCF) Asserts

For DCF asserts, some (or all) of the following output messages may be included:

Stimulus Message —

```
INIT:AM-LVL,

INIT:SM-LVL-EVENT,

INIT:CMP-LVL
```

Stack Trace —
REPT:STACK-TRACE

Stack Frame —
REPT:STACK-FRAME

Data Dump —
REPT:DATA

Register Dump —
REPT:REGISTER

PMDB Dump —
REPT:SM-ENV-SRC

5.1.2.3 MANUAL ACTION Asserts

MANUAL ACTION asserts print one message to the ROP that contains all information related to the assert (REPT:MANUAL).

5.1.2.4 RTA Asserts

RTA asserts will print a REPT:RTA-DCF-RE message to indicate the source of the assert and data related to the assert. In addition, one or more REPT:DATA-DUMP messages may be printed indicating that additional information has been printed in the /log/log/CALLPLOG1 file on the AM.

The information printed in the CALLPLOG1 file will contain a header and a hexadecimal dump of the data structure listed in the header. You will need to refer to a header template to more easily identify the exact mapping of values to the data structure. Refer to Exhibit 5.1-1 for the following description:

EVENT — This is the event number of the data dump report that notified the technical personnel that this dump would occur. Note that multiple dumps can occur from the same event (and processor).

PCR — This is the processor on which the data dump report was generated. This field and the event number are used to map this dump to the information printed on the ROP from the data dump report.

TYPE — This is the type of dump and identifies the hexadecimal dump that follows. Note that some of the dump types need more than 1 segment since RTDUMPDATA can print only 228 bytes of data per segment. In this example, the RT_MSG dump takes 2 segments. Multiple segment dumps will always be contiguous. The following list shows a few examples of the type of data that may be dumped. Check the RTdatadump.h header file for a complete list.

- PCBLA (Process Control Block Linkage Area)
- PORTLA (Port Linkage Area)
- PCB (Process Control Block)
- CCBCOM (Channel Control Block)
- COUPLER (Port to Process Coupler)
- RDBLK (RTA Routing Data Block)
- CFBLK (RTA Call Flow Data Block)
- GPBLK (RTA Group Data Block)

- RT_MSG (MGRT_GEN message)
- LGMSG (any large message)
- MESSAGE (any message or data)

Each hexadecimal dump segment will always contain 228 bytes, even when the data structure being dumped is not that large. In these cases, the remaining bytes do not contain any useful data.

Exhibit 5.1-1 — Example of CALLPLOG1 Output

```
S570-68 93-08-28 13:53:22 000665 CAPR_LOG
RTDUMPDATA GENERATED
       CALL PROCESSING ERROR DUMP EVENT=323 PCR=2 TYPE=RT_MSG
       00181011 04070000 00214000 00480000 04000000 00000000 00000000 30393131
       00000000 00000000 00000000 78010602 B5F30200 2775220E 00000000 00000000
       00000000 00000000 00000000 03A50251 03FF03FF 00000000 02000000 00080000
       00008800 8A000002 03A50251 00D00002 8D010000 20144102 4000000A 001F0012
       00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
       0B5F0033 30308000 00000000 00000060 00000007 FE000000 00000000 00000000
       00000000 00000000 0EE00000 42100124 040000A0 42100124 040000A0 00000000
       00000000

S570-68 93-08-28 13:53:29 000665 CAPR_LOG
RTDUMPDATA GENERATED
       CALL PROCESSING ERROR DUMP EVENT=323 PCR=2 TYPE=RT_MSG
       00000000 00000000 00000000 00000000 00000000 00000004 00000000 02000003
       0005326B 01570000 00000000 07000000 00000000 00000000 4C000000 00000000
       00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
       01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
       00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
       00000000 00000000 00000000 00000000 00000000 00000000 00000000 03A50251
       010C0700 8A130000 00000000 00000000 00000000 00000000 00000000 00000101
       000C0A39

S570-68 93-08-28 13:54:10 000665 CAPR_LOG
RTDUMPDATA GENERATED
       CALL PROCESSING ERROR DUMP EVENT=323 PCR=2 TYPE=PCBLA
       03A51800 01687804 00000000 00000000 00600000 00010000 00000000 00110000
       00000000 FFFF0000 FFFF0000 8D013102 00D00002 FFFF0000 B4C5518A 00001068
       01080010 008401F3 06000000 00000426 02AE0002 FFFF0000 FFFC8026 00020000
       00000000 00000000 00000000 00000000 00000000 01EC0000 00620000 00000000
       00FF00FF 00790079 00FF00FF 8D010000 00000000 00000000 00000000 00000000
       00000000 00000000 00000000 00000000 00000000 00000000 00000000 03A50251
       010C0700 8A130000 00000000 00000000 00000000 00000000 00000000 00000101
       000C0A39
```

5.1.3 STACK TRACE DEBUGGING

5.1.3.1 Stack Formatting Overview

To analyze a stack trace (e.g., an assert), the relationship between the printed stack frame and the actual stack frame in the processor's memory must be understood. Each function that is running on the AM, SM, or CMP is allotted a section of memory to use for the operation at hand. This allotted portion of memory is called a stack frame. The stack frame messages that are printed on the ROP are formatted printouts of these memory allocations.

The AM stores stack frames differently than the SM and CMP. (The CMP uses the MC68030 processor, which is a member of the same processor family as the SM.) Because of these distinctions, the data printed in a stack frame message must be interpreted differently. The variations between the AM and SM/CMP stacks are shown in Table 5.1-1.

Table 5.1-1 — 3B2xD Processor vs. MC68XXX Processor Stack Variations

AM Memory	SM & CMP Memory
Stacks grow from lower to higher addresses	Stacks grows from higher to lower addresses
Registers are saved in the stack frame between the arguments and the local variables	Registers are saved at the "top" of the stack frame (toward the lower addresses) if they are needed
Processor uses an argument pointer indicating the first argument in the stack frame	Processor does not use an argument pointer

Since the AM/SM/CMP processors/compilers have built in minimum sizes of data that may be pushed on to the stack, some arguments are "type converted" to meet this requirement when they are pushed. Specifically, arguments that are smaller in size than an integer become the size of an integer (e.g., arguments that are declared to be of type `char` are padded to take the space of an integer). In the AM an integer is 4 bytes and in the SM/CMP it is 2.

5.1.3.2 AM Stack Layout (3B2XD Processor)

The stack frame output message has three sections. The function address section contains the address of the function. The parameters and local data sections contain the arguments passed to the function and the function's local variables. The contents of these two sections are discussed here in more detail.

The data in the parameters section is mapped onto the stack in 4-byte segments starting with the first argument. Since the 3B2XD processor memory stack, Figure 5.1-1, is populated from lower to higher addresses, the first argument is at a lower address than the last argument. If the C program statement is

```
FUNCTION(ARG1,ARG2,ARG3,ARGn)
```

then `ARG1` would be mapped onto the 3B2XD processor memory stack followed sequentially by `ARG2`, `ARG3`, and `ARGn`.

The 3B2XD processor architecture has a register dedicated for a pointer to the first argument in the stack frame. This is known as the argument pointer.

When the assert handler prints the information for the parameters section of the stack frame printout, it uses the argument pointer as a reference and prints toward higher addresses for 40 bytes. Since the 3B2XD processor stack is populated from lower to higher address, the first 4 bytes in the parameters section are the first argument passed to the function. The next 4 bytes is the second argument and so on for 40 bytes. The stack frame may of course have more than 40 bytes worth of arguments, but the stack frame printout is limited to 40 bytes by the assert handler.

The local data section of the stack frame printout contains the function's local variables. This data can be any allowable C declaration type (`array`, `structure`, `enumeration`, `long`, `short`, etc.). Because the 3B2XD processor stack is populated from lower to higher addresses, the local variables in the stack frame message printout will appear in the same order as they appear at the beginning of each source function listing. The 3B2XD processor architecture places a few requirements on the population of the 3B2XD processor memory stack; these are that `structures` and `unions` are aligned on 4-byte boundaries, and `shorts` are aligned on even-byte

boundaries. Longs, pointers, and integers are aligned on 4-byte boundaries. For a complete listing of data types and sizes, see Table 5.1-2.

For example, the source function listing is:

```
FUNCTIONB (ARG1,ARG2,ARG3)
char ARG1;
short ARG2;
long ARG3;
{
  short a;
  long b;
  struct ctag {
     char d;
     long e;
  }c;
}
```

The assert handler prints the data from the 3B2XD processor stack by using another register called the frame pointer as a reference. The frame pointer points to the first local variable. The assert handler will print for 160 bytes, even though there may be more than that amount of data. Since the assert handler uses the frame pointer as a reference, and knowing that the 3B2XD processor memory stack is populated from low to high, the first 2 bytes of the local data printed will contain the variable a from the previous example. Following the variable a will be 2 bytes of zero followed by the variable b using 4 bytes and starting on a 4-byte alignment since it is defined as a long. The structure ctag follows and will contain 1 byte for the variable d a character followed by 3 bytes of zero, to force alignment, followed by 4 bytes containing the variable e. For a comprehensive view of the stack, see Figure 5.1-1.

Table 5.1-2 — 3B2XD and MC68XXX Processor Family Data Sizes and Alignments

3B2XD Processor Family		
Data Type	**Size**	**Memory Alignment**
char	1 byte	no alignment
short	2 bytes	2 byte boundary
int	4 bytes	4 byte boundary
long	4 bytes	4 byte boundary
pointer	4 bytes	4 byte boundary
structure	4 byte multiple	4 byte boundary
union	4 byte multiple	4 byte boundary
array	same as element type	same as element type
bitfield	up to the maximum of type declared	same as declared type
Data inside structures and unions:		
long	4 bytes	4 byte offset
pointer	4 bytes	4 byte offset
structure	4 byte multiple	4 byte offset
union	4 byte multiple	4 byte boundary
bitfield	up to the maximum of type declared	same as declared type

MC68XXX Processor Family		
Data type	**Size**	**Memory Alignment**
char	1 byte	no alignment
short	2 bytes	2 byte boundary
int	2 bytes	2 byte boundary
long	4 bytes	2 byte boundary
pointer	4 bytes	2 byte boundary
structure	4 byte multiple	2 byte boundary
union	4 byte multiple	2 byte boundary
array	same as element type	same as element type
bitfield	2 bytes	2 byte boundary
Data inside structures and unions:		
long	4 bytes	4 byte offset
pointer	4 bytes	4 byte offset
structure	4 byte multiple	4 byte offset
union	4 byte multiple	4 byte offset
bitfield	2 bytes	2 byte boundary

Fill or padding is generated by the compiler to align the variables on these boundaries.

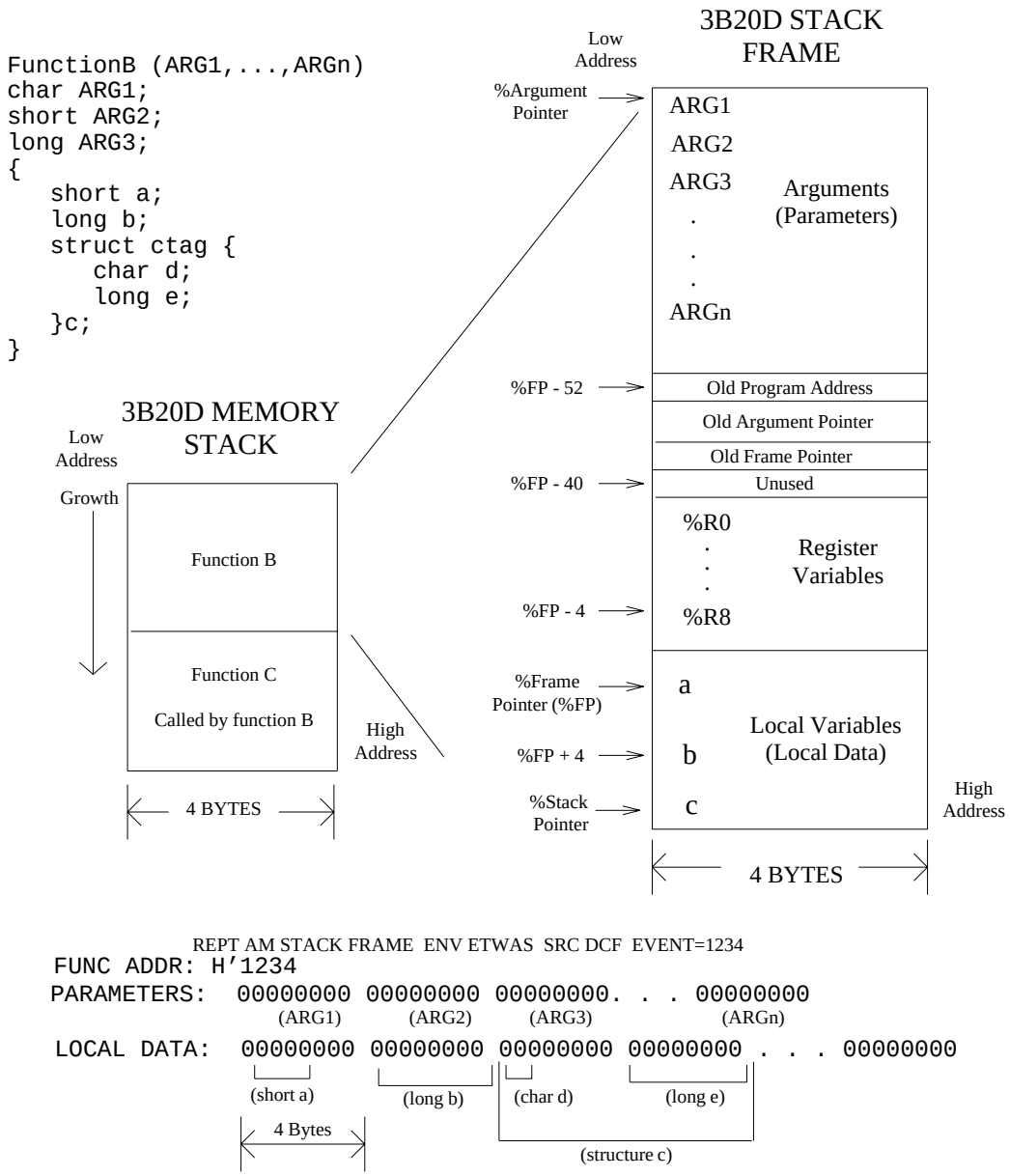


Figure 5.1-1 — AM Assert Formatting

5.1.3.3 SM/CMP Stack Layout (MC68XXX Processor Family)

The SM and CMP stacks (MC68XXX processor family) grow from the higher addresses toward the lower addresses. This difference between the AM and SM/CMP memory stacks can make understanding the SM stacks more difficult. The SM assert stack frame message printout will still contain a parameter and local data section, but the data therein must be interpreted differently.

The MC68XXX processor family frame pointer, Figure 5.1-2, is used to access both the local data (negative offset) and the parameters (positive offset). To support functions

with a variable number of arguments, the arguments are pushed onto the stack from right to left. This ensures that the first argument is always at a fixed offset (%FP+8) from the frame pointer.

Unlike the 3B2XD processor stack frame, arguments are mapped onto the SM stack frame in reverse order. For example:

```
FUNCTION (ARG1,ARG2,ARG3,ARGn)
```

ARGn would be placed in the SM stack frame at the highest address followed by ARG3 and so on towards the lower addresses. This difference does not affect how the arguments will appear in the parameters section of the stack frame printout because of the way they are extracted from the SM stack frame. (See Figure 5.1-2.) The assert handler uses the frame pointer plus 8 bytes and takes the data out of SM stack frame towards higher addresses for 40 bytes. This results in the arguments being printed in the same order that they appear in the source function listing, (e.g., ARG1, ARG2, ARG3, ARG4).

The SM stack, like the 3B2XD processor memory stack, has certain rules for the alignment of the local data variables.

- All structures and unions must be aligned on 2-byte boundaries and be a multiple of 4 bytes in size
- All long (4-byte) variables must be aligned on even-byte boundaries
- All longs, pointers, unions, and structures within structures or unions must be aligned on 4-byte offsets.

Enumerations usually have less than 256 values, which could be stored in 1 byte. So in the *5ESS* switch, the size of an enumeration is controlled by the range of internal values. A range of 0-255 is stored in a char, values from –32768 to +32768 are stored in a short, and larger values are stored in a long. For a complete listing of data types and memory alignment, consult Table 5.1-2.

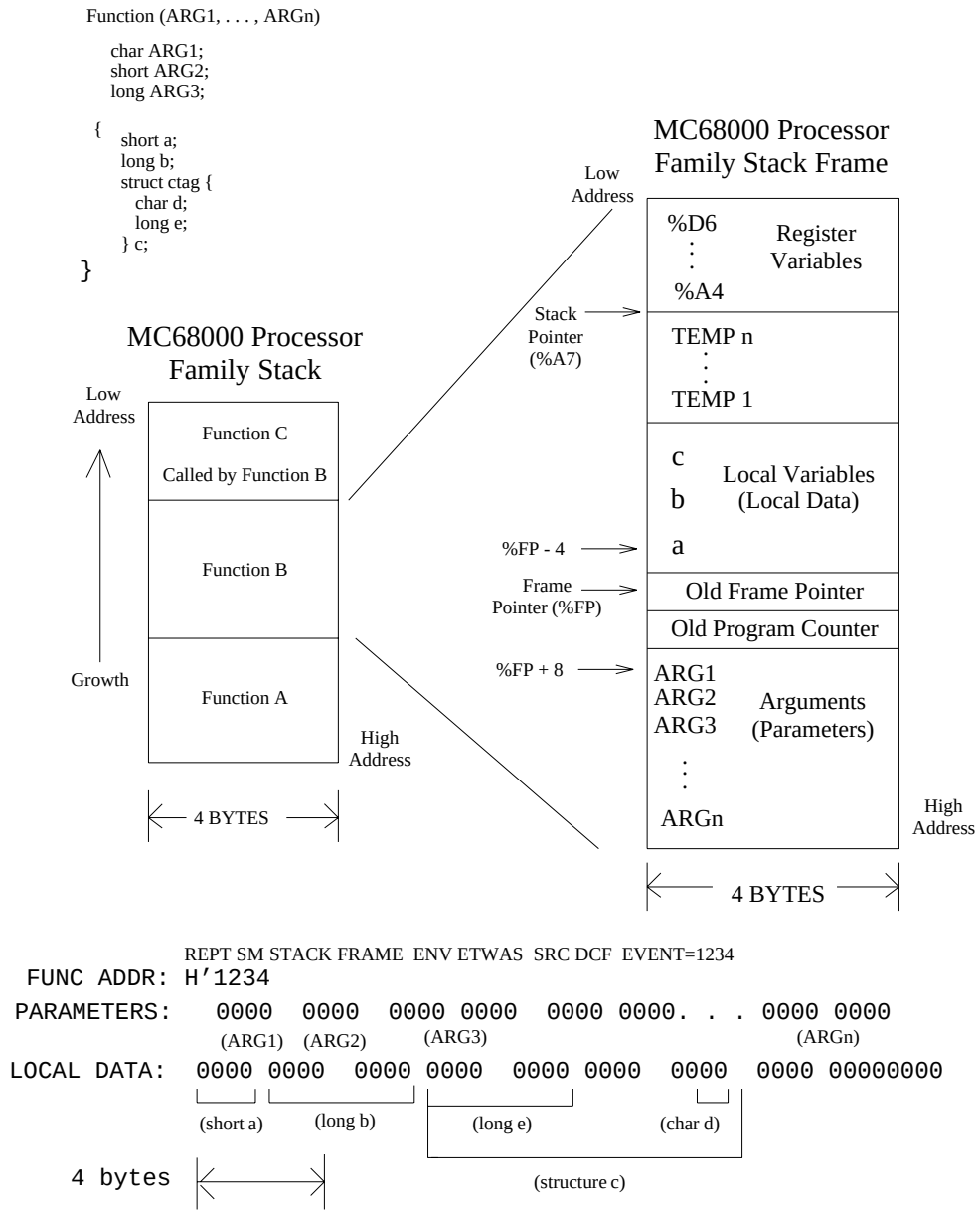


Figure 5.1-2 — SM Stack Frame Format

When the assert handler extracts the data from the SM stack frame to be printed, it starts with the frame pointer and extracts toward lower addresses for a total of 180 bytes, one byte at a time. This causes the data in the printed stack frame message to be formatted in a way that must be deciphered differently from the local data in the AM stack frame printout. This situation can be best understood by reviewing the following example. (See Figure 5.1-3.)

In Figure 5.1-3 the local data section of the example function is represented as it would appear on the stack and as it would appear on the ROP printout of the stack

frame. The stack is 4 bytes across and the data section is filled by using negative displacement from the frame pointer.

For the sake of discussion the value of the frame pointer will be 1000.

- The first variable, `short a` (which is 2 bytes in length), begins at the frame pointer minus 2 bytes or at memory address 998. The size of the variable is calculated by the switch and then the data are placed in the space allowed.
- The second variable, `long b` (which is 4 bytes in length), begins at the frame pointer minus 6 bytes or at memory address 994.
- The third variable is a `structure`. The size of the `structure` is calculated and then the values are placed on the stack from lower to the higher addresses. It is important to keep in mind that data inside a `structure` will actually be placed in memory from lowest to highest address though variables are allocated space from higher to lower.

  The `structure` requires a total of 8 bytes though the actual contents of the `structure` contains 5 bytes. One byte for the `char d` and 4 bytes for the `long e`. (Every `structure` must be a multiple of 4 bytes.) The beginning address of the `structure c` from our example would be 986. The `char d` would be at 986. The `long e` would begin at 990, which is a 4-byte offset from the beginning of the `structure`. The addresses 987, 988, and 989 would be padded with zeros.

The stack frame is built one byte at a time starting at one byte from the frame pointer. The first byte from the frame pointer would be 999, the least significant bits of the variable `short a`. The contents of that first byte according to our example would be the numbers `34`. The second byte contains the numbers `12`. The third byte contains the fourth byte of the variable `long b`. The fourth byte is placed onto the stack first, then the third byte, then the second, and finally the first.

The output at the bottom of Figure 5.1-3 prints as `3412`. To properly analyze this data you must "swab" the bytes so that the proper content of the stack is ascertained. The actual value would be `1234` when swabbing is completed.

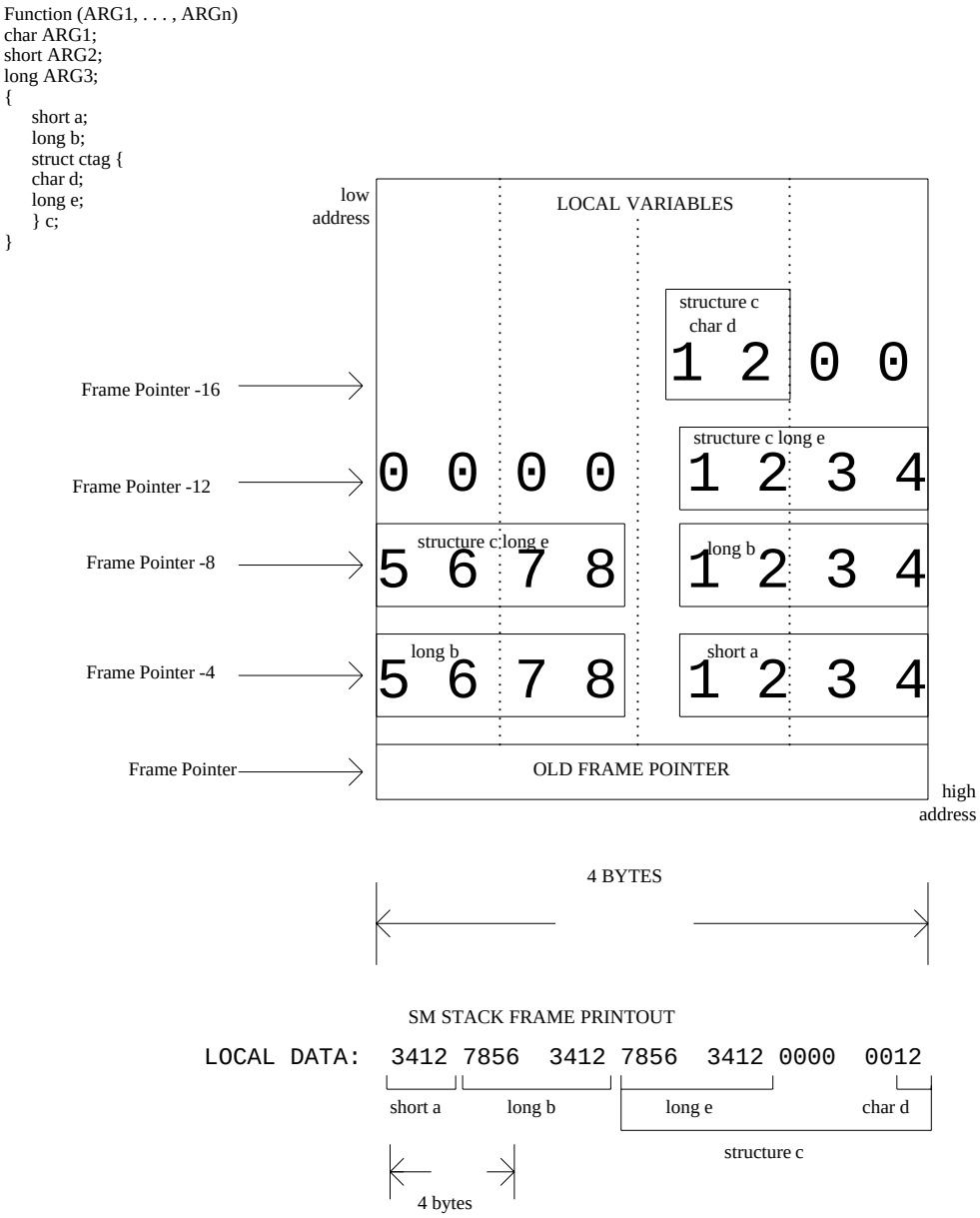


Figure 5.1-3 — SM Local Data Stack Frame

5.1.4 FAILING FUNCTION

To determine the function where the failure occurred, look at some of the information that the stimulus message provides.

Search through the ROP printout until you find all the information that is related to this assert. The `EVENT` number is the key to this search and is the same for all messages associated with this assert.

To get a rough idea of the failure, find the `DEF-CHK-FAIL` number in the 235-600-500, *Asserts Manual*.

To delve deeper into the problem, analyze the C code and the disassembly code for this failure. Look up the failing function using the `FAILING-ADDR` and the on-line listings.

5.2 ASSERT ANALYSIS EXAMPLES

5.2.1 ASSERT ANALYSIS EXAMPLE — AM

For an example of a AM assert analysis see "EXAMPLE - Using the Program Listings," Section 2.6.

5.2.2 ASSERT ANALYSIS EXAMPLE — SM

This example describes the method used to analyze an SM assert. It is not intended to reflect the current *5ESS*® switch code, assert descriptions or data layout.

1. From the stimulus message we need to get the failing address, the processor (SM), and the defensive check failure. The assert printout, Exhibit 5.2-1, shows this information as follows:

   - `INIT SM=96,1`: An SM assert, the SM number is 96 and the active side is side 1.
   - Environment: This is a loaded SM. To find the SM type, see `OP:SYSSTAT` in the 235-600-700, *Input Messages Manual* and 235-600-750, *Output Messages Manual*.
   - `FAIL-ADDR`: This is used to find the failing function. The value from the sample assert is `h'4e46d8`. Note that this information is also found in the `STACK TRACE` message as the first address in the stack trace.
   - Defensive check fail `(DCF)` number is `26204`. Use the 235-600-500, *Asserts Manual* to find the associated assert mnemonic and description. In this example the mnemonic is `SMC_DBCDFI_MAP` and the assert description says `SMrsmcdfh process unable to read a tuple from the relation RLCDFI_MAP`.

Exhibit 5.2-1 — SM Assert Printout

```
S570-66 90-01-11 10:54:16 000630 LSWRPIN E U.BWM89-0063
INIT SM=96,1 LVL=RPI EVENT=25 COMPLETED
  DEF-CHK-FAIL=26204 NO-AUD-SCHED
  SW-ERR FAILING-ADDR=H'4e46d8 SM-MODE=NORMAL TIME=53:34:9
  PROCESS: BG=80,0,H'4e3a14,RPI CM=NONE, FG=NONE,,

S570-66 90-01-11 10:54:40 000646 LSWRPIN E U.BWM89-0063
REPT SM=96 STACK TRACE ENV=OSDSM SRC=DCF EVENT=25
  USER: 004E46D8 004E3E38 004E3CA6 004E3A2E

S570-66 90-01-11 10:54:48 000650 LSWRPIN E U.BWM89-0063
REPT SM=96 STACK FRAME ENV=OSDSM SRC=DCF EVENT=25
  FUNC ADDR: H'4e46d8
  PARAMETERS: 0010A026 00000001 000A1FC8 00020000 00000005
              0001CB01 000A004E 60000094 0610A026 001EEAC2
  LOCAL DATA: C0006000 18003000 00303030 80210033 30204554
              4953202D 20454241 4C030000 00007E1F 0A004502
              BE3E4E00 AA1F0A00 38BC5A00 00000000 26050000
              60A030A0 30000000 000000C0 00030000 00000000
              0000002F 02000800 00000000 0000531F 0A001400
              2F00D6F2 5100761F 0A000000 00000000 00000000
              00000000 000080A0 00000000 0000005C 26051CFB
              5A007E1F 0A0084E5 61000C37 29006C1F 0A00D0CA
              00000100 010124E0 7500FFFF 03000000 0000DA24

S570-66 90-01-11 10:54:57 000654 LSWRPIN E U.BWM89-0063
REPT SM=96 STACK FRAME ENV=OSDSM SRC=DCF EVENT=25
  FUNC ADDR: H'4e3e38
  PARAMETERS: 000A1FEE 004E3A2E 00000000 000A1FFC 001E89C2
              FFFD0000 0000000A 1FFC0000 00000000 00000000
```

```
  LOCAL DATA: 00014544 FFFF8352 6F021100 2F00C2EA 1E0026A0
              10069400 00604E00 0A0001CB 01000500 00000000
              0200C81F 0A000100 000026A0 1000383E 4E00DA1F
              0A00C000 60001800 30000030 30308021 00333020
              45544953 202D2045 42414C03 00000000 7E1F0A00
              4502BE3E 4E00AA1F 0A0038BC 5A000000 00002605
              000060A0 30A03000 00000000 00C00003 00000000
              00000000 002F0200 08000000 00000000 531F0A00
              14002F00 D6F25100 761F0A00 00000000 00000000

S570-66 90-01-11 10:55:05 000658 LSWRPIN E U.BWM89-0063
REPT SM=96 REGISTER DUMP ENV=OSDSM SRC=DCF EVENT=25
  REGISTER DATA= 000A1E96 000A1DE2 00000000 00000000
                 00000000 00000000 000A1E74 000A1E2A
                 0000FF9B 00000002 00000001 0002BF20
                 00000000 00000001 0000A080 00000000

S570-66 90-01-11 10:55:13 000662 LSWRPIN E U.BWM89-0063
REPT SM=96 DATA=DCF-DATA,0 ADDR=H'a1f1c ENV=OSDSM SRC=DCF EVENT=25
  A0800000 00000000 00000000 00000000 0000000A 1F760051 F2D6002F 0014000A
  1F530000 00000000 00080002 2F000000 00000000 00000300 C0000000 00000030
  A030A060 00000526 00000000 005ABC38 000A1FAA 004E3EBE 0245000A 1F7E0000
  0000034C 41424520 2D205349 54452030 33002180 30303000 00300018 006000C0
  000A1FDA 004E3E38 0010A026 00000001 000A1FC8 00020000 00000005 0001CB01
```

2. With this information we should be able to find the failing function using the `UPD:FTRC` method.

   See the 235-600-700, *Input Messages Manual* and 235-600-750, *Output Messages Manual* for a description of `UPD:FTRC`. For this example, `upd:ftrc:sm=96,addr=h'4e46d8` was used to obtain the following output:

```
UPD FTRC REPT

SM=96, CONFIG=LOADED
OBJECT_FILE=/no5text/im/smtxt/IM.out

ADDRESS FUNCTION  START    SIZE   OFFSET   TV       FILE      SYMINDEX
------- -------- ------- -------- ------ ----- -------------- --------
 4e46d8 SMiclvco  4e4630      11a     54  8862  SMrsmcdfh_c.c [ 24038]
```

   The offset added to the starting address of the disassembly code (Exhibit 5.2-2) will provide the location of the failure.

```
  h'34e6   Starting address of disassembly code.
  + h'54   Offset.
  -------
  h'353a   Location in disassembly code.
```

3. Use the on-line listings as described in "EXAMPLE - Using the Program Listings," Section 2.6 to get the disassembly and C listings for the function.

   (See Exhibit 5.2-2 and Exhibit 5.2-3 for breakpoint line numbers, the disassembly and the program code for this example.)

   The failing address is normally the address of the instruction following the assert which fired. The failing address under these circumstances will be one instruction less than the failing address. This rule of thumb may not hold true when register variables are present or when optimization has occurred. See "Memory Configuration," Section 4.2.4, for more detail on optimization. In this case, when we look into the disassembly code we find that line `[32]` is the line number of the failing address. The C code shows that the error detected was caused by a data base read error. Since it is a type `ASSERTB1` assert, the second element is a pointer to the address where the data dump will begin.

Exhibit 5.2-2 — Disassembly Code for the Function `SMiclvco`

```
       34e6: 4e56 ffae       link         %fp,$-0x52
       34ea: 48e7 1020       moveml       <%d3,%a2>,-(%sp)
 [18]  34ee: 202e 0008       movel        0x8(%fp),%d0
       34f2: 5980            subl         $0x4,%d0
       34f4: 2440            moveal       %d0,%a2
 [21]  34f6: 302e 000c       move         0xc(%fp),%d0
       34fa: 0240 ff01       and          $-0xff,%d0
       34fe: 0040 0080       or           $0x80,%d0
       3502: 3d40 ffb2       move         %d0,-0x4e(%fp)
 [26]  3506: 3d6e ffb2 ffe0  move         -0x4e(%fp),-0x20(%fp)
 [27]  350c: 41ee ffd8       lea          -0x28(%fp),%a0
       3510: 2f08            movel        %a0,-(%sp)
       3512: 3f3c 0041       move         $0x41,-(%sp)
       3516: 4eb9 0000 0000  jsr          0x0
       351c: 5c8f            addl         $0x6,%sp
       351e: 4a40            tst          %d0
       3520: 6738            beq          0x38 <355a>
 [32]  3522: 41ee ffd8       lea          -0x28(%fp),%a0
       3526: 2d48 ffae       movel        %a0,-0x52(%fp)
       352a: 41ee ffae       lea          -0x52(%fp),%a0
       352e: 2f08            movel        %a0,-(%sp)
       3530: 3f3c 665d       move         $0x665d,-(%sp)
       3534: 4eb9 0000 0000  jsr          0x0
       353a: 5c8f            addl         $0x6,%sp
       353c: 6078            bra          0x78 <35b6>
 [67]  353e: 3f2e 000c       move         0xc(%fp),-(%sp)
       3542: 2f2e 0008       movel        0x8(%fp),-(%sp)
       3546: 4eb9 0000 0000  jsr          0x0
       354c: 5c8f            addl         $0x6,%sp
 [68]  354e: 422e fffa       clrb         -0x6(%fp)
 [69]  3552: 3d7c 00ca fff8  move         $0xca,-0x8(%fp)
       3558: 606c            bra          0x6c <35c6>
 [36]  355a: 4240            clr          %d0
       355c: 102e ffe6       moveb        -0x1a(%fp),%d0
       3560: ec48            lsr          $0x6,%d0
       3562: 4a40            tst          %d0
       3564: 6712            beq          0x12 <3578>
 [41]  3566: 4243            clr          %d3
       3568: 0c43 001e       cmp          $0x1e,%d3
       356c: 6c2e            bge          0x2e <359c>
 [43]  356e: 1d92 30b7       moveb        (%a2),-0x49(%fp,%d3)
 [44]  3572: 528a            addl         $0x1,%a2
 [45]  3574: 5243            add          $0x1,%d3
       3576: 60f0            bra          -0x10 <3568>
 [53]  3578: 206e 0008       moveal       0x8(%fp),%a0
       357c: 4240            clr          %d0
       357e: 1028 000d       moveb        0xd(%a0),%d0
       3582: 4a40            tst          %d0
       3584: 6738            beq          0x38 <35be>
       3586: 0c40 0001       cmp          $0x1,%d0
       358a: 67b2            beq          -0x4e <353e>
 [77]  358c: 4243            clr          %d3
       358e: 6006            bra          0x6 <3596>
 [79]  3590: 1d92 30b7       moveb        (%a2),-0x49(%fp,%d3)
 [80]  3594: 5243            add          $0x1,%d3
       3596: 0c43 001e       cmp          $0x1e,%d3
       359a: 6df4            blt          -0xc <3590>
       359c: 3d6e ffb2 ffb4  move         -0x4e(%fp),-0x4c(%fp)
       35a2: 1d7c 001e ffb6  moveb        $0x1e,-0x4a(%fp)
       35a8: 41ee ffb4       lea          -0x4c(%fp),%a0
       35ac: 2f08            movel        %a0,-(%sp)
       35ae: 4eb9 0000 0000  jsr          0x0
       35b4: 588f            addl         $0x4,%sp
 [97]  35b6: 4cdf 0408       moveml       (%sp)+,<%d3,%a2>
       35ba: 4e5e            unlk         %fp
       35bc: 4e75            rts
 [58]  35be: 422e fffa       clrb         -0x6(%fp)
```

```
[59]  35c2: 426e fff8       clr          -0x8(%fp)
[87]  35c6: 3d6e ffb2 fff4  move         -0x4e(%fp),-0xc(%fp)
[88]  35cc: 1d7c 000b fff6  moveb        $0xb,-0xa(%fp)
[89]  35d2: 206e 0008       moveal       0x8(%fp),%a0
      35d6: 1d68 000f fffc  moveb        0xf(%a0),-0x4(%fp)
[91]  35dc: 3d7c 0801 fff0  move         $0x801,-0x10(%fp)
[92]  35e2: 422e fff2       clrb         -0xe(%fp)
[93]  35e6: 1d7c 000c fff3  moveb        $0xc,-0xd(%fp)
[95]  35ec: 41ee ffec       lea          -0x14(%fp),%a0
      35f0: 2f08            movel        %a0,-(%sp)
      35f2: 41ee ffdc       lea          -0x24(%fp),%a0
      35f6: 2f10            movel        (%a0),-(%sp)
      35f8: 4eb9 0000 0000  jsr          0x0
      35fe: 508f            addl         $0x8,%sp
      3600: 60b4            bra          -0x4c <35b6>
```

Exhibit 5.2-3 — C code for the Function `SMiclvco`

```
@FUNCTION: SMiclvco
        /*
         * Name:         SMiclvconex()
         *
         * Abstract:     Handles verification of the connectivity exercise
         *               acknowledgement message received over an RCL.
         *
         * Usage:        void
         *               SMiclvconex( msg_ptr, rclname )
         *               SMCDFHMSG  *msg_ptr;
         *               DMCIRCUIT  rclname;
         *
         * Parameters:   *msg_ptr - pointer to message buffer
         *               rclname - RCL name that message was received on
         *
         * Externals:    none
         *
         * Returns:      void
         *
         * Description:  Handles the reception of the connectivity exercise ack
         *               message over an RCL. It verifies that the test is valid
         *               and forwards the results to the MRA terminal process
         *               currently working on the RCL. If errors are detected,
         *               appropriate error routines are called.
         *
         * Calls:        SMllaerrchk() - print ROP message concerning
         *                               connectivity failure
         *               DBfrdtup() - read a data base tuple
         *               SMbad_ddl() - sends a "bad ddl" msg to PERFR
         *
         * Macros:       ASSERTB1() - assert
         *               OSSENDMSG() - send an osds message to another process
         *               SMRCL2FAC() - translate RCL name to CFAC name.
         */
        void
        SMiclvconex( msg_ptr, rclname )
        SMCDFHMSG       *msg_ptr;
        DMCIRCUIT       rclname;
        {
                struct {
                        OSMSGHEAD               msghead;
                        struct mgCDFH2MRA       text;
                } mramsg;    /* outgoing message buffer */
                struct rlSMEST          smest;     /* local SMEST tuple buffer */
                struct mgDFIHMSG        outmsg;    /* buffer for PERFR msg*/
                DMCIRCUIT               cfac_name; /* CFAC internal name */

                register char           *ddlptr; /* DDL message pointer*/
                register short          i;
                /* msg_ptr is a pointer to x bytes (x = CMSZDDLDATA) of */
```

```
                /* text sent by CMdfimsg. There are 4 bytes of header info */
                /* prior to this, thus we set the pointer back 4 bytes. */

[18]            ddlptr = (char *)msg_ptr - 4;

                /* derive the internal CFAC circuit name */
[21]            cfac_name = SMRCL2FAC( rclname );

                /* verify that MRA has an active terminal  */
                /* process waiting for this message   */

[26]            smest.unit_id = cfac_name ;

[27]            if ( DBfrdtup( RLSMEST, &smest ) != GLSUCCESS ) {

                        /* data base read error - assert */
                        /* can't do anything else here */

[32]                    ASSERTB1( SMC_DBSMEST, &smest );
                        return;
              }

[36]             if ( smest.stat_act != DBBUSY ) {

                         /* if circuit isn't busy, no TP active on */
                         /* it - format and send bad ddl msg to PERFR */

[41]                     for ( i = 0; i < DDL_MSG_SIZE; i++ ) {

[43]                             outmsg.ddl_msg[ i ] = *ddlptr;
[44]                             ddlptr++;
[45]                     }
                         outmsg.unit_id = cfac_name;
                         outmsg.num_bytes = sizeof( outmsg.ddl_msg );
                         SMbad_ddl( &outmsg );
                         return;
                 }
                 /* so far so good - look at message to see if test passed */
[53]             switch ( msg_ptr->errcode ) {
                 case SMLLANOERR:

                         /* no errors found */

[58]                     mramsg.text.result.resp = SMCMPL;
[59]                     mramsg.text.result.info = SMNUL;
                         break;

                 case SMLLACONERR:
                         /* we have a connection failure -  */
                         /* call function to print ROP message */

[67]                     (void) SMllaerrchk( msg_ptr, rclname );
[68]                     mramsg.text.result.resp = SMCMPL;  /* COMPLETED */
[69]                     mramsg.text.result.info = SMRSM23; /* TST OF FAC FAILED */
                         break;
                         default:

                                 /* should be no other results coming here */
                                 /* - format and send bad ddl msg to PERFR */
[77]                             for ( i = 0; i < DDL_MSG_SIZE; i++ ) {

[79]                                     outmsg.ddl_msg[ i ] = *ddlptr;
[80]                             }
                                 outmsg.unit_id = cfac_name;
                                 outmsg.num_bytes = sizeof( outmsg.ddl_msg );
                                 SMbad_ddl( &outmsg );
                                 return;
                         }
```

```
[87]                    mramsg.text.cfac_name = cfac_name;
[88]                    mramsg.text.verb = SMTEST;
[89]                    mramsg.text.serial = msg_ptr->serial;
[91]                    mramsg.msghead.type = MGCDFH2MRA;
[92]                    mramsg.msghead.priority = 0;
[93]                    mramsg.msghead.length = sizeof( mramsg.text );

[95]                    OSSENDMSG( smest.tp_pid, &mramsg );
                        return;
[97]        }
```

4. Line [32] in the source code is an assert macro that is passed two arguments. Line [32] in the source code is as follows:

```
 [27]           if ( DBfrdtup( RLSMEST, &smest ) != GLSUCCESS ) {

                        /* data base read error - assert */
                        /* can't do anything else here */

 [32]           ASSERTB1( SMC_DBSMEST, &smest );
                return;
  }
```

   The first argument is SMC_DBSMEST and the second is &smest (the address of the structure smest).

   The assert code with mnemonic SMC_DBCDFI_MAP, in this example, says SMrsmcdfh process unable to read a tuple from the relation RLCDFI_MAP. Given the assert message and the second element of the assert itself, it can be determined that the structure smest contains erroneous data (i.e. the key, smest.unit_id or cfac_name, used to read the desired tuple is not valid). The data can be analyzed by determining how the data is laid out on the stack.

5. The stack builds from higher to lower addresses unless it is a structure. In a structure the size of the structure is determined first, next the amount of space needed for the structure is allocated, and then the contents of the structure are pushed onto the stack from lower to higher addresses.

   This list shows an example of the construction of the first two local variables, the structure mramsg and the structure smest. This is not necessarily how these structures are currently laid out.

```
mramsg:     msghead:    from:      procno     short    2
          [OSMSGHEAD]  [OSPID]     pcrid      uchar    1
                                   uniq       uchar    1

                       type        ushort     2
                       priority    uchar      1
                       length      uchar      1

            text:      cfac_name    ushort     2
          [MGCDFH2MRA] verb         enum       1
                       padding                 1

                       result:     info            enum    2
                      [DMMRARTVAL] resp            enum    1
                       serial      uchar      1

              mramsg =  15 bytes + 1 byte of padding
```

Note: To follow the header trail used in this example to construct the structure layout, see Appendix A6.

```
smest:      orig_pid:     procno      short    2
            [OSPID]       pcrid       uchar    1
                          uniq        uchar    1

            tp_pid:       procno      short    2
            [OSPID]       pcrid       uchar    1
                          uniq        uchar    1

            unit_id                   ushort   2
            msgtype                   ushort   2
            atpcount                  uchar    1
            errorblk                  uchar    1
            stat_act                  enum     2 bits
            bas_stat                  enum     6 bits
            qual                      enum     2 bits
            qual_2                    enum     6 bits
            transient                 enum     1 bit
            dgstat                    enum     3 bits
            qual_1                    enum     4 bits
            mccupd                    enum     1 bit
            rexinh                    enum     1 bit
            progflag                  enum     1 bit
            state                     enum     2 bits
            quarstate                 enum     3 bits
            pinhtype                  enum     2 bits
            ais_alm                   enum     1 bit
            ylw_alm                   enum     1 bit
            red_alm                   enum     1 bit
            iatouch                   enum     1 bit

          smest = 18 bytes and 6 bits + 10 bits of padding
```

Note: To follow the header trail used in this example to construct the structure layout, see Appendix A6.

The variable `rlSMEST` is a relation. To determine the data structure of the relation, look at the definition for this relation in the 235-600-2xx, *Dynamic Data Manual*. The types `DMMRAVERB`, `DMMRARTVAL`, and `DMMRAINFO` are `enumerations`. The number of bits used for these types can be located in the 235-600-2xx, *Translations & Dynamic Data Domain Descriptions*.

Given this data construction, the stack would be organized as follows. Remember that in the MC68XXX processor, data is placed onto the stack from higher to lower address. (See Figure 5.2-1.)

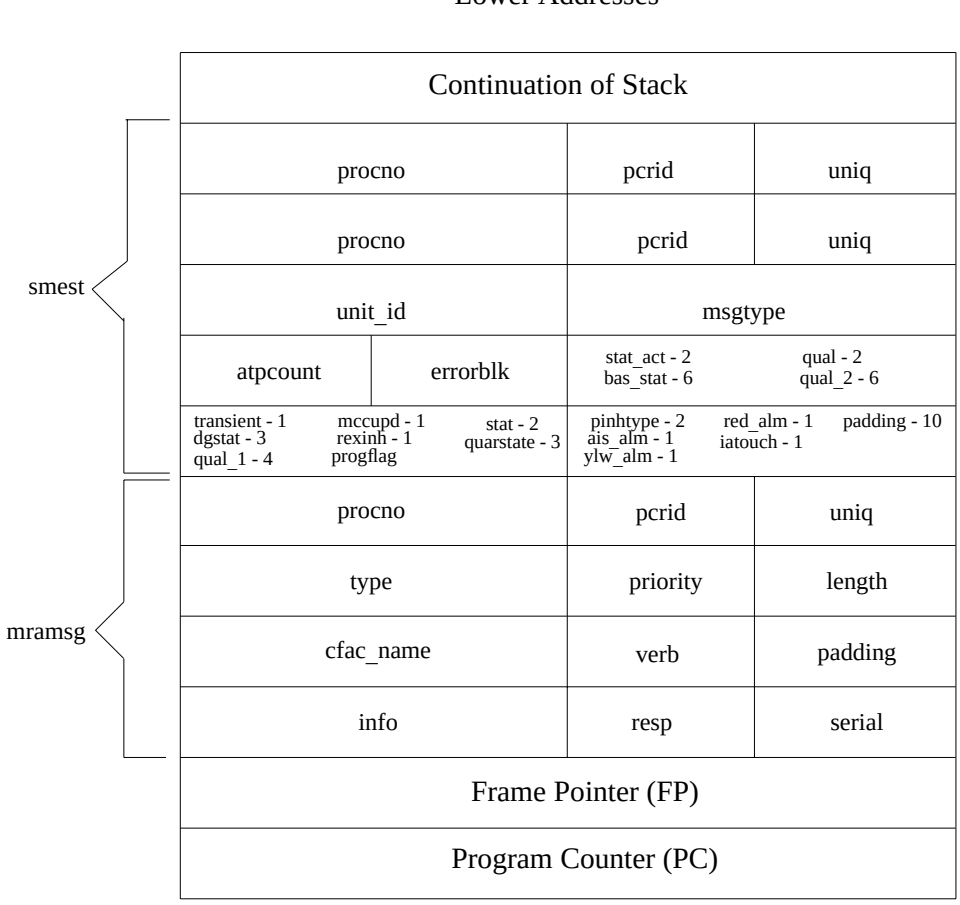


Figure 5.2-1 — Stack Grid

When the data is taken from the stack and dumped onto the stack frame, it is taken one byte at a time. At this point, you must swab the bytes to make the data useful. Given this stack frame:

```
S570-66 90-01-11 10:54:48 000650 LSWRPIN E  U.BWM89-0063
REPT  SM=96  STACK FRAME ENV=OSDSM SRC=DCF EVENT=25
 FUNC ADDR:  H'4e46d8
 PARAMETERS: 0010A026  00000001  000A1FC8  00020000  00000005
             0001CB01  000A004E  60000094  0610A026  001EEAC2
 LOCAL DATA: C0006000  18003000  00303030  80210033  30204554
             4953202D  20454241  4C030000  00007E1F  0A004502
             BE3E4E00  AA1F0A00  38BC5A00  00000000  26050000
             60A030A0  30000000  000000C0  00030000  00000000
             0000002F  02000800  00000000  0000531F  0A001400
             2F00D6F2  5100761F  0A000000  00000000  00000000
             00000000  000080A0  00000000  0000005C  26051CFB
```

```
5A007E1F  0A0084E5  61000C37  29006C1F  0A00D0CA
00000100  010124E0  7500FFFF  03000000  0000DA24
```

This is an example of how the data would be analyzed:

Variable	Size of variable	Hexadecimal	Swabbed Value
serial	1 byte	C0	C0
resp	1 byte	00	00
info	2 bytes	6000	0060
padding	1 byte	18	18
verb	1 byte	00	00
cfac_name	2 bytes	3000	0030
length	1 byte	00	00
priority	1 byte	30	30
type	2 bytes	3030	3030
uniq	1 byte	80	80
pcrid	1 byte	21	21
procno	2 bytes	0033	3300

The assert analysis for the switching module is more complex due to the swabbing that is necessary and therefore will require more time and patience to perform than an administrative module assert analysis.

Apparently a `smest` tuple with key `h'0030` does not exist. At this point, the switch database should be checked to verify that the indicated tuple really does not exist and then an investigation into whether the tuple should exist would follow. It is also possible that the `SMiclvconex()` function passed erroneous data, or included wrong options. Further study of the database and related functions using the above method should lead to a root cause.

5.2.3 ASSERT ANALYSIS EXAMPLE — CMP

This analysis example is for a CMP assert. It does not necessarily reflect the current CMP code, assert meaning, or data layout.

1. From the stimulus message we need to get the processor, the failing address, and the defensive check failure (`DCF`). From the example, Exhibit 5.2-4, this information is:
   - `INIT CMP=1-0`: A CMP assert on PRIMARY processor.
   - `FAILING-ADDR=H'bceba7c`: This value is used to find the failing function. Note that this information is also found in the stack trace message as the first address in the stack trace.
   - `DEF-CHK-FAIL=21412`: The assert code with the mnemonic `DBAC0012` which says `The Database Manager ran out of open relation blocks (dbopndict). The opndc audit is scheduled to run.`
2. With this information, we should be able to find the function using the `UPD:FTRC` method. From this point on, the analysis of a CMP and SM assert are the same (since both processors are from the MC68XXX processor family). See the "Asserts Analysis Example — SM," Section 5.2.2.

Exhibit 5.2-4 — CMP Assert Printout

```
S57O-262273 9O-1O-18 1O:23:27 O2O328 ASRT FIVE1
INIT CMP=1-O PRIM LVL=RPI EVENT=52 COMPLETED
  DEF-CHK-FAIL=21412 AUD-SCHED=OPNDC ELEV-MODE
  FAILING-ADDR=H'bceba7c  CMP-MODE=NORMAL  TIME=22:57.8
  PROCESS: BG=7O,27,H'bcbc6c6,RPI  INTJ=NONE  FG=NONE

S57O-262273 9O-1O-18 1O:23:35 O2O329 ASRTMON FIVE1
REPT  CMP=1-O PRIM  STACK TRACE ENV=CMP-AP SRC=DCF EVENT=52
  USER:  OBCEBA7C  343BD7OE  OBCBC9B4  OBC6C5DA

S57O-262273 9O-1O-18 1O:23:43 O2O33O ASRTMON FIVE1
REPT  CMP=1-O PRIM  STACK FRAME ENV=CMP-AP SRC=DCF EVENT=52
  FUNC ADDR:   H'bceba7c
  PARAMETERS:  OOOOOO82  OBEDFCF4  OBEDFED6  OO129DAO  OO129DAO
               OO11O082  O3BOO466  O4CFO469  O2F9O1EE  O4C8O382
  LOCAL DATA:  2O5AE1OB  6O15E1OB  27OO66OO  4AFF2BO1  F8O53C34
               OOOOC1OO  OOOOOOOO  O5OOOOOB  4O5BE1OB  O6OOEDOB
               8O5AE1OB  A4537CBA  CEOBD4FC  EDOB9EFC  EDOBO1OO
               OOOOO2OO  OOOO8OOO  OOOOOOOO  FFFF11OO  OOOO1109
               OOOO1109  OOOOOOOO  OOOO5AFC  EDOB9EFC  EDOBOOOO
               OOOO2A76  E2OB2A76  E2OB6O15  E1OB3O3C  OO1C3CD2
               E2OB9EFC  EDOB9EFC  EDOBOOOO  OOOOA453  54F9C7OB
               9EFCEDOB  O6OOOCOO  2A76O7OB  4EO6D7OB  O5OOA453
               9EFCEDOB  9EFCEDOB  OOOOOOOO  E4OB5EOO  CFOB4EFC

S57O-262273 9O-1O-18 1O:23:51 O2O331 ASRTMON FIVE1
REPT  CMP=1-O PRIM  STACK FRAME ENV=CMP-AP SRC=DCF EVENT=52
  FUNC ADDR:   H'343bd7Oe
  PARAMETERS:  OOO5OOO1  OBEDFEFA  OBEDFED6  OOOOC35O  OOOOOOOO
               OBEDFFEE  OOOOOOO1  BC95OOOO  FF9BOBE1  5B4OOBEE
  LOCAL DATA:  O1OOB645  D6OBFE87  12OO6OAA  E6OBO1FF  94OOEDOB
               2A76E2OB  8O5AE1OB  OOOO13OO  7CDECEOB  92FEEDOB
               OOOOOOOO  OO59E1OB  OOC211OO  727EE6OB  1OOOOOC2
               11OOF6E1  CEOBCEOB  62FEEDOB  7COO1COO  B7OO1301
               3A5AEOOB  4O5BE1OB  E1OBC653  EOOB9O5B  465AEOOB
               9O5BE1OB  13O116OO  13O1286C  1COO1301  DO69C4OB
               42FEEDOB  1COOOOOO  8OOA11OO  OOOOOOOO  OOOO7601
               O2CFCDOB  4EFE42O3  6603A8O3  61OO6503  6CO46703
               1CO51BO5  34051A05  19053303  C7OO93O2  38OO8CO3

S57O-262273 9O-1O-18 1O:23:59 O2O332 ASRTMON FIVE1
REPT CMP=1-O PRIM REGISTER DUMP ENV=CMP-AP SRC=DCF EVENT=52
  REGISTER DATA= OBE2D23C  1COO3C3O  OBE1156O  OBE2762A
                 OBE2762A  OOOOOOOO  OBEDFC9E  OBEDFC5A
                 OOOOOOOO  OOOOO911  OOOOO911  OOOOOO11
                 FFFFOOOO  OOOOOO8O  OOOOOOO2  OOOOOOO1
```

5.2.4 ASSERT ANALYSIS EXAMPLE — RTA DCF

5.2.4.1 RTA DCF ROUTING Error

The following examples illustrate the method used to investigate an RTA assert, but do not necessarily reflect the current state of the *5ESS* switch code. The RTA DCF ROUTING error message contains the file name which allows the technical personnel to use the program listings to analyze problems that occur during call processing.

Exhibit 5.2-5 — RTA RTRTGERR

```
SO-26504 93-03-13 11:43:01 22
REPT RTA DCF ROUTING ERROR EVENT=24 HSM 1 FILE RTnw_conn.c LINE 429
  ERRTYPE INTERNAL CAUSE FAIL STATE NW_CONN SEQ RT_GENREQ
  MSGTYPE RT_GEN REQTYPE O RETURN 3 RIC O RETRY 1
  DN 22OO57O NOC 1 SI 1 DI 1O PI 1 ROUTETYPE NULL RI O
  ORIGPORT H'bOOO 1 OPARTY O TERMPORT H'O 194 TPARTY O
  GRP O GRPTYPE LINE SIZE O HUNT NONE LPCR 1 LPID 167 LUNQ 1
```

```
   RTGSTS   1  24   5  18   5   7   0
```

The routing error in Exhibit 5.2-5 is an internal error caused by a failure. The event number is 24 and the error occurred in the host switching module (HSM) number 1. The source file from which the RTRTGERR() macro was called is RTnw_conn.c at line 429. Line 429 of the RTnw_conn.c file is given in Exhibit 5.2-6. Note that the first argument of the function is RT_INTERNAL, which matches the ERRTYPE given in Exhibit 5.2-5 and the second argument is RTG_FAIL, which matches the CAUSE field.

Exhibit 5.2-6 — RTnw_conn.c RTRTGERR() Error Example

```
/*
 *      Name:           RTnw_conn
 *
 *      Module:         RTrtgsts
 *
 *      Package:        RTnw_conn will be part of the base.
 *
 *      Function:       RTnw_conn checks the terminating path type
 *                      to determine if this required path is digital
 *                      or metallic in nature. When valid route types and
 *                      terminating classes have been found, RTtmmsg
 *                      is invoked, and then the routing state is updated
 *                      to RT_FSMDONE.
            .                       .                   .
            .                       .                   .
            .                       .                   .
        default:
                RTRTGERR( RT_INTERNAL, RTG_FAIL, DUDANULL );
                break ;
            .                       .                   .
            .                       .                   .
```

The LPCR field (next to last line in Exhibit 5.2-5) gives the number of the processor of the sending message which is 1, the LPID gives the process ID of the sender of the message, 167, and the LUNQ specifies the uniqueness value of the process of the sender of this message. The RTGSTS field (last line in Exhibit 5.2-5) represents the decimal values of the routing states that were hit during this routing attempt through the RTA finite state machine (FSM). State numbers are printed in the order that the states were entered, reading from left to right. In Exhibit 5.2-5 the first state was 1, which maps to RT_SCRNING in the RTA local header, RTstates.h. See Exhibit 5.2-7. The second state was 24, which maps to RT_PRE_DNTRAN; the third state is 5, which maps to RT_INTEGRITY; and so on. The state that the FSM was in when the exception occurred is the state before the 0 (used as a delimiter) which is 7 and maps to RTNW_CONN, which is (and should be) the same state as listed under the STATE field (third line in Exhibit 5.2-5).

Exhibit 5.2-7 — Routing States

```
typedef enum rtRTGSTATES {
        RT_NULLSTATE = 0,
        RT_SCRNING,
        RT_RTING,
        RT_DN_TRAN,
        RT_ERROR,
        RT_INTEGRITY,
        RT_FSMDONE,
        RT_NW_CONN,
        RT_NW_DISC,
        RT_TMMSU,
        RT_MLGSPREHUNT,
```

```
        RT_MLGHUNT,
        RT_MLGBUSY,
        RT_TRKSPREHUNT,
        RT_TRKHUNT,
        RT_TRKBUSY,
        RT_COALTRTE,
        RT_FIXEDRT,
        RT_SWITCH,
        RT_STANDALONE,
        RT_POSTSCRN,
        RT_TRKDPREHUNT,
        RT_DEUQUE,
        RT_ADDQUE,
        RT_PRE_DNTRAN,
        RT_MCRTING,
        RT_SFG,
        RT_SFGDN,
        RT_TKESSX,
        RT_LNESSX,
        RT_PPSCRN,
        RT_LNQHUNT,
        RT_LN_DQUE,
        RT_POST_DNTRAN,
        RT_MLGDPREHUNT,
        RTTERMSFG,
        RT_CCS_REST,
        RT_POLYGRID,
        RT_TKQMS,
        RT_TKDEQUE,
        RT_TKPOSTDQ,
        RT_TKQUPSTAT,
        RT_TKENQUE,
        RT_DTRTING,
        RT_RMSD3,
        RT_ICIDET,
        RT_TKMEM,
        RT_APTLRT,
        RT_HNTICI,
        RT_WBRTING
} RTRTGSTATES;
```

5.2.4.2 RTA DCF: RTDUMPDATA()

The RTDUMPDATA() capability logs additional information for call processing errors. The Data Dump Report Event=25057 messages given in Exhibit 5.2-8 inform the technical personnel that pertinent data has been dump for these asserts in the CALLPLOG1 log file. These messages indicates that an exception has occurred in call processing software and that in order to provide additional information to debug this problem, data is being directed to the /log/log/CALLPLOG1 file accessible on the 3B *UNIX*[1] terminal in the office. Note that the event number of this data dump report will map to the same event number (for the given SM) that will be part of the relevant data being dumped in the CALLPLOG1 file.

Exhibit 5.2-8 — Systems Integrity RTA DCF: Data Dump Report

```
S570-131130 93-08-28 13:53:02 000660 ASRTMON
REPT DATA DUMP REPORT PCR=2 EVENT=123
     CALL PROCESSING ERROR OCCURRED AT FUNCTION MCbad_msg.c  LINE 113
     A DATA DUMP WAS SENT TO /log/log/CALLPLOG1

S570-131130 93-08-28 13:53:04 000661 ASRTMON
REPT DATA DUMP REPORT PCR=2 EVENT=124
```

1. Registered trademark of The Open Group.

```
     CALL PROCESSING ERROR OCCURRED AT FUNCTION MCbad_msg.c  LINE 113
     A DATA DUMP WAS SENT TO /log/log/CALLPLOG1

S570-131130 93-08-28 13:53:06 000662 ASRTMON
REPT DATA DUMP REPORT PCR=2 EVENT=125
     CALL PROCESSING ERROR OCCURRED AT FUNCTION MCbad_msg.c  LINE 113
     A DATA DUMP WAS SENT TO /log/log/CALLPLOG1
```

The RTDUMPDATA() output for these RTA DCF assert messages can be found in Exhibit 5.2-9. The data comes from the file MCbad_msg.c line 113. The structure of the dump can be determined by looking at the construction of the message in the file that caused the assert to fire.

Exhibit 5.2-9 — CALLPLOG1 File Output Example

```
S570-68 93-08-28 13:53:22 000665 CAPR_LOG
RTDUMPDATA GENERATED
       CALL PROCESSING ERROR DUMP EVENT=123 PCR=2 TYPE=MESSAGE
       77770039 00020037 77770039 0003001F 7777215C 00010006 7777215C 00030006
       7777215C 00080002 77770096 00020023 77770096 00080002 77770096 00030006
       77772135 00020022 77772146 00030002 7777214E 00080000 00000000 00000000
       00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
       00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
       00088518 010BD2AE 0165CC40 0163F940 00390000 00390165 CC400163 F9400008
       8672010D 68DA0039 0165CC40 0163F940 00000039 00000001 0165CC40 0163F940
       00010000

S570-68 93-08-28 13:53:29 000667 CAPR_LOG
RTDUMPDATA GENERATED
       CALL PROCESSING ERROR DUMP EVENT=124 PCR=2 TYPE=MESSAGE
       77770039 00020037 77770039 0003001F 7777215C 00010006 7777215C 00030006
       7777215C 00080002 77770096 00020023 77770096 00080002 77770096 00030006
       77772135 00020022 77772146 00030002 7777214E 00080000 7777213E 00030028
       7777213E 0002002F 7777213E 00010026 7777217D 0003001F 77770039 00020037
       77770039 0003001F 7777215C 00010006 7777215C 00030006 7777215C 00080002
       00086518 010BD2AE 0165D880 0163FB40 00390000 00390165 D8800163 FB400008
       6672010D 68DA0039 0165D880 0163FB40 00000039 00000001 0165D880 0163FB40
       00010000

S570-68 93-08-28 13:54:10 000668 CAPR_LOG
RTDUMPDATA GENERATED
       CALL PROCESSING ERROR DUMP EVENT=125 PCR=2 TYPE=MESSAGE
       77770039 00020037 77770039 0003001F 7777215C 00010006 7777215C 00030006
       7777215C 00080002 77770096 00020023 77770096 00080002 77770096 00030006
       77772135 00020022 77772146 00030002 7777214E 00080000 7777213E 00030028
       7777213E 0002002F 7777213E 00010026 7777217D 0003001F 77770039 00020037
       77770039 0003001F 7777215C 00010006 7777215C 00030006 7777215C 00080002
       00081518 010BD2AE 0165EB00 01640000 00390000 00390165 EB000164 00000008
       1672010D 68DA0039 0165EB00 01640000 00000039 00000001 0165EB00 01640000
       00010000
```

Software Analysis Guide

CONTENTS PAGE

6. GENERIC ACCESS PACKAGE (GRASP)/ENHANCED GRASP 6-1
6.1 PURPOSE 6-1
6.2 GRASP/EGRASP OVERVIEW 6-1
6.3 GRASP/EGRASP CAPABILITIES 6-1
6.3.1 Data Transfer Functions 6-1
6.3.2 Breakpoints 6-6
6.3.3 Overriding A Default Time Limit 6-10
6.3.4 Transfer Trace Function 6-10
6.3.5 Trace And Matching Messages 6-11
6.4 LAYOUTS 6-14
6.4.1 Input Message Acknowledgements 6-14
6.4.2 Register Mnemonics 6-14
6.4.3 GRASP/EGRASP Output Message Layout. 6-15
6.5 GRASP/EGRASP EXAMPLE. 6-16

6. GENERIC ACCESS PACKAGE (GRASP)/ENHANCED GRASP

6.1 PURPOSE

This section on the Generic Access Package (GRASP)/Enhanced Generic Access Package provides the user with a sense of GRASP/EGRASP capabilities as used in the *5ESS*® switch application, and explains how GRASP/EGRASP may effectively support overall trouble analysis procedures.

6.2 GRASP/EGRASP OVERVIEW

The Generic Access Package (GRASP)/Enhanced Generic Access Package (EGRASP) is a subsystem of the software release of *UNIX*[1] RTR operating system software. It is designed to be a single-user utility system that facilitates the analysis of software faults and the investigation of the flow of program execution. GRASP/EGRASP allows the behavior of *UNIX* RTR operating system software to be observed in an operational environment. It is intended to be used to gather information on a known problem.

Caution: Although GRASP can be used by maintenance personnel, it should be used in consultation with your next level of technical support. Improper use of GRASP can result in program mutilation or excessive utilization of system resources.

6.3 GRASP/EGRASP CAPABILITIES

6.3.1 Data Transfer Functions

6.3.1.1 Data Transfer, Memory-to-Memory (COPY)

A COPY function is provided to transfer data from one storage medium or memory location to another. The data at the specified origin remains unaltered after the function is complete.

Main Memory Immediate

This message causes data resident in main memory to be transferred to a utility variable. Execution of this function is immediate. It copies data from virtual addresses in main memory to utility variables as an immediate action. Indirect addressing may be specified. The first listed offset is added to the value of the source address and the result is used as a virtual address of a location in main memory. The number of offsets specified defines the length of the chain of virtual addresses to be accessed in this way before accessing the desired range of source locations.

```
COPY:UID=a,ADDR=b[,OFF=c][,L=d|NL=d]:UVAR=e[:WORD];
COPY:PID=a,ADDR=b[,OFF=c][,L=d|NL=d]:UVAR=e[:WORD];
```

1. Registered trademark of The Open Group.

Main Memory On Breakpoint

This message causes data resident in main memory to be transferred to a utility variable, register, or elsewhere in main memory when conditions in an associated WHEN statement are satisfied. Execution of this function occurs on breakpoint. It copies data from virtual addresses in main memory to other virtual addresses, registers, or utility variables as a response to a breakpoint. Indirect addressing may be specified. The first listed offset is added to the value of the source address and the result is used as a virtual address of a location in main memory. The number of offsets specified defines the length of the chain of virtual addresses to be accessed in this way before accessing the desired range of source locations.

```
COPY:ADDR=a[,OFF=b][,L=c|,NL=c]{:ADDR=d|:UVAR=e|:REG=f}[;WORD]!
```

Utility Variable Immediate

This message causes data resident in a utility variable to be transferred to another utility variable. Execution of this function is immediate. It copies data from another utility variable as an immediate action or as an action triggered by a breakpoint.

```
COPY:UVAR=a[,OFF=b][,L=c|,NL=c]:UVAR=e[:WORD]{|;}
```

Utility Variable On Breakpoint

This message causes data resident in a utility variable to be transferred to main memory or a register when conditions specified in an associated WHEN statement are satisfied. Execution of this function occurs on breakpoint. It copies data from a utility variable to virtual addresses in main memory and in registers as an action associated with a breakpoint.

```
COPY:UVAR=a[,OFF=b][,L=c|,NL=c]{:ADDR=d|:REG=f}[:WORD]!
```

Register Immediate

This message causes data contained in a register to be transferred to a utility variable. Execution of this function is immediate.

```
COPY:REG=a[,OFF=b][,L=c|,NL=c]:UVAR=r[:WORD]{!|;}
```

Register On Breakpoint

This message causes data contained in a register to be transferred to main memory or another register when conditions specified in an associated WHEN statement are satisfied. Execution of this function occurs on breakpoint.

```
COPY:REG=a[,OFF=b][,L=c|,NL=c]{:ADDR=d|:REG=f}[:WORD]!
```

6.3.1.2 Data Transfer, Memory-to-ROP (DUMP)

A DUMP function is provided to send data to the maintenance terminal and to print system data on the ROP. In general, up to 128 bytes may be printed.

Main Memory Immediate

This message causes data resident in main memory to be displayed at the maintenance terminal and printed at the ROP. Execution of this function is immediate. It dumps the contents of a specified range of virtual addresses in main memory in the address space of the process with the specified utility identifier. If only one address is given, indirect addressing may be specified. In this case, the first offset listed is added to the content of the given address and the result is interpreted as a virtual address.

```
DUMP:UID=a,ADDR={b&&c|b[,OFF=d][,{L|NL}=e]}[:WORD];
```

Dumps the contents of the specified range of virtual addresses in main memory in the address space of the process with the specified process identifier. If only one address is given, indirect addressing may be specified. In this case, the first offset listed is added to the content of the given address and the result is interpreted as a virtual address.

```
DUMP:PID=a,ADDR={b&&c|b[,OFF=d][,{L|NL}=e]}[:WORD];
```

Main Memory On Breakpoint

This message causes memory within the process identified by the WHEN command to be displayed at the maintenance terminal and printed at the ROP. Execution of this function occurs on breakpoint. It dumps the contents of a specified range of virtual addresses in main memory as an action associated with a breakpoint.

```
DUMP:ADDR=a&&b|a[,OFF=c][,{L|NL}=d][:WORD]!
```

Kernel Memory Immediate

Dumps the contents of a specified range of virtual addresses in the administrative module kernel as an immediate action. The range is specified by two addresses or an address and a length. The length defaults to a value of 1. If only one address is given, indirect addressing may be specified. In this case, the first offset listed is added to the content of the given address and the result is interpreted as a virtual address.

```
DUMP:KERN={a&&b|a[,OFF=c][,{L|NL}=d]};
```

Physical Memory Immediate

Dumps the contents of a specified range of physical addresses in main memory. The range is specified by two addresses or by one address and an optional byte length count. The default length is 4 bytes.

```
DUMP:PMEM={a&&b|a{L|NL}=c]}{!|;}
```

Physical Memory On Breakpoint

Dumps the contents of a specified range of physical addresses in main memory. The range is specified by two addresses or by one address and an optional byte length count. The default length is 4 bytes.

```
DUMP:PMEM={a&&b|a{L|NL}=c]}{!|;}
```

Utility Variable Immediate

This message causes data resident in a utility variable to be displayed at the maintenance terminal and printed at the ROP. Execution of this function is immediate. It dumps the contents of one or more utility variables as an immediate message.

```
DUMP:UVAR=a[,OFF=b][,L|NL}=c][:WORD]!
```

Utility Variable On Breakpoint

This message causes data resident in a utility variable to be displayed at the maintenance terminal and printed at the ROP when conditions specified in an associated `WHEN` statement are satisfied. Execution of this function occurs on breakpoint.

```
DUMP:UVAR=a[,OFF=b][,L|NL}=c][:WORD]!
```

Register Immediate

This message causes data resident in a readable register(s) to be displayed at the maintenance terminal and printed at the ROP.

```
DUMP:REG=a!
```

Register On Breakpoint

This message causes data resident in a readable register(s) to be displayed at the maintenance terminal and printed at the ROP when conditions specified in an associated `WHEN` statement are satisfied. In addition to dumping the registers themselves on breakpoint, the `DUMP:REG` command can be used with `INDIR` to dump the breakpoint process's memory.

```
DUMP:REG=a[,OFF=b][,{L|NL}=c][:WORD][!|;}
```

6.3.1.3 Data Transfer, MCC-to-Memory (LOAD)

The LOAD command alters the content of a register, utility variable, or location in main memory. The new value for the location is supplied as part of the command. If the destination is a register or utility variable, the data is treated as an unsigned number (sign extension is not done) and the least significant (rightmost) bytes are changed. The number of bytes changed depends on the length specified. For a main memory destination, the data is located byte by byte starting at the address and continuing for the given length. Only a utility variable load may be executed immediately.

Caution: *LOAD* ***can cause switch downtime if used improperly.***

Main Memory On Breakpoint

This message only affects the process that fired at the breakpoint. It loads a virtual address with specified data as an action associated with a breakpoint. For a software breakpoint that fires on execution of an instruction, exactly one process is effected. However, every time the breakpoint fires the load is performed. For hardware breakpoints that fire on data access, the load is made into the address space of the process performing the access each time the breakpoint fires.

```
LOAD:ADDR=a[,OFF=b][,L=c][:WORD]DATA,D=d!
```

Utility Variable Immediate

This message causes data to be loaded into the utility variable specified on the command line. If a length smaller than four is specified, the data is moved into the least significant bytes and the most significant bytes are zeroed. Execution of this function is immediate. Loads AM utility variables with specified data an immediate operation.

```
LOAD:UVAR=a[,L=b]{:WORD}:DATA,D=c{!|:}
```

Utility Variable On Breakpoint

This message causes data to be loaded into the utility variable specified on the command line when conditions specified in an associated WHEN statement are satisfied. Execution of this function occurs on breakpoint.

```
LOAD:UVAR=a[,L=b]{:WORD}:DATA,D=c{!|:}
```

Register On Breakpoint

This message causes data to be loaded into a writable register specified on the command line when conditions specified in an associated WHEN statement are satisfied. Execution of this function occurs on breakpoint.

```
LOAD:REG=a[,OFF=b][,L=c][:WORD]:DATA,D=d!
```

6.3.2 Breakpoints

6.3.2.1 Breakpoint Definition

Breakpoints detect the existence of some set of conditions on the machine. The definition of a breakpoint has two parts.

1. First, the conditions that are to be matched have to be described.
2. Second, the actions that are to occur when there is a match are listed.

The described conditions might include the execution of a particular leg of code or the reading of a particular data address by a particular process. Other actions might be changing the value of some register or dumping the stack.

The WHEN command starts a list of GRASP commands that are performed when a specified breakpoint condition exists. After a WHEN command with its conditions and action list is entered successfully, the breakpoint is assigned a number by GRASP. The breakpoint is then referred to exclusively by its number. Up to twenty different breakpoints can be defined in the system at any point. The numbers assigned to breakpoints during a debugging session will not be reused.

```
WHEN:UID=h'120,ADDR=h'22034;W!
DUMP:REG=PSW!
DUMP:ADDR=h'20160!
END:WHEN
```

GRASP prints two output messages in response to a breakpoint after the PF is given. The first message assigns a number to the breakpoint. This message should appear soon after the PF. The second message confirms that the breakpoint was set up successfully, or, that the breakpoint was aborted, and gives the reason why.

Breakpoints that fire on the execution of a specific instruction are called software breakpoints because of the way they are implemented. The breakpoint itself is a special instruction that transfers control to GRASP when it is executed.

Software breakpoints are set up at the location specified by the UID or PID and ADDR keywords of the WHEN commands as soon as possible after the breakpoint is defined. The opcode itself is not changed until the breakpoint is allowed. Processes are described by the UID or PID and, in some cases, a user process name. However, more than one process can be active with the same UID and process name. When this happens, GRASP sets up the first breakpoint in one of the matching processes at random. If another breakpoint is defined for the same UID or process name, GRASP sets up the breakpoint in the same process as the first.

Hardware breakpoints, because of their distinctive implementation, have some very different characteristics from software breakpoints. Breakpoints which fire on accesses of data are implemented with the hardware of the utility circuit. The hardware on the utility circuit has "matchers" for utility IDs, addresses, access modes, etc. To set up a hardware breakpoint, GRASP configures the matchers that are needed and supplies the values that are to be matched. The circuitry continually compares the values that GRASP told it to match with what is taking place on the machine. The breakpoint will fire when all the matchers specified during set up match, but only if the breakpoint is enabled. If a hardware breakpoint is disabled, the hardware still passively tries to match; but it will not interrupt the processing on the machine. Disabled hardware breakpoints do not use any resources of the machine. If the condition matcher is already being used for a trace, a trigger allocation error results if an attempt is made to define a condition breakpoint.

6.3.2.2 Breakpoint Conditional Command

The WHEN command is the conditional statement and can be used to break on a condition or break on a specific process or utility.

Break on Condition

A WHEN statement starts a list of commands that are to be performed when an external event breakpoint condition exists. An external event condition exists when the external event backplane signal becomes active.

```
WHEN:COND=E;
```

Break On Specific Process or Utility

Requests that a list of messages be executed when a specified breakpoint condition exists. Two types of breakpoint conditions are recognized:

1. instruction-execution, and
2. conditions and data-access conditions.

Instruction execution is detected by specifying a virtual address within a process. When the instruction at that address is executed, the action listed is executed. The first byte of the instruction opcode expected to be found at that address must be specified in the message. The breakpoint definition is rejected if this opcode does not agree with the opcode found at the virtual address.

For process identification:

```
WHEN:PID=a,ADDR=b,OPC=e:EXC[,WORD]!
```

For utility identification:

```
WHEN:UID=a,ADDR=b,OPC=e:EXC[,WORD]!
```

Data access is indicated by specifying both the virtual address within the process and the type of access to be detected: read, write, and read/write. A data access breakpoint will fire if the location specified contains addresses with the correct access type. A data access breakpoint that covers a range of data locations will fire if any location within the range is addressed by the correct access type.

For process identification:

```
WHEN:PID=a,ADDR{b[&&c][,L=d]}:{R|W|RW}[,WORD]!
```

For utility identification:

```
WHEN:UID=a,ADDR{b[&&c][,L=d]}:{R|W|RW}[,WORD]!
```

6.3.2.3 Breakpoint Manipulation Commands

Breakpoints can be allowed or inhibited from firing, their definition can be cleared, and a summary of all breakpoints can be printed. The commands to manipulate breakpoints are given in the following sections.

Enabling Individual Breakpoints

A breakpoint will not fire when it is first defined. It has to be enabled. This is done with the ALW command:

```
ALW:UTILFLAG a {!|/}
```

where UTILFLAG a is the breakpoint number assigned at the time the breakpoint was defined. This command can be used either in the action list of a WHEN or an immediate command. The message

```
ALW UTILFLAG a COMPLETED
```

is printed confirming that the breakpoint was enabled.

Enabling All Breakpoints

For convenience, the following command is provided to enable all breakpoints at once:

```
ALW:UTIL {!|/}
```

This can be used either as an immediate command or as an action for a breakpoint. After all the breakpoints have been enabled, the confirmation message

```
ALW UTIL COMPLETED
```

is printed.

Disabling Individual Breakpoints

To disable a breakpoint, the INH command is used

```
INH:UTILFLAG a {!|/}
```

where UTILFLAG a is the breakpoint number assigned at the time the breakpoint was defined. This can be used either in the action list for a breakpoint or as an immediate action. The output message

```
INH UTILFLAG a COMPLETED
```

confirms the completion of the command.

Disabling All Breakpoints

All breakpoints can be disabled at once with the message:

```
INH: UTIL {!|/}
```

which can be used either as an immediate action or in an action list. The confirmation message

```
INH UTIL COMPLETED
```

will be printed after all the breakpoints have been inhibited.

Disabling Breakpoints (Self-referential)

The following special command can only be used in an action list of a breakpoint to disable the breakpoint itself:

```
INH: UTILFLAG ME {!|/}
```

The special command is needed because a number has not yet been assigned to the breakpoint at the time its action list is being defined. The use of this command in an action list will cause the breakpoint to fire exactly once. Any time it is enabled again, it will fire once more and disable itself. The `INH: UTILFLAG ME` command can occur anywhere in the work list and will not affect the rest of the work list processing for that firing of the breakpoint.

Clearing Individual Breakpoints

To clear an individual breakpoint the following message is used:

```
CLR: UTILFLAG a!
```

where the `UTILFLAG a` is the breakpoint number assigned at the time the breakpoint was defined. This command can never be used in the action list of a breakpoint. The message

```
CLR UTILFLAG a COMPLETED
```

will be printed after the breakpoint has been successfully cleared. The message

```
CLR UTILFLAG a NGINST
```

indicates that the input command was unsuccessful in removing the planted breakpoint.

Clearing All Breakpoints

All breakpoints can be cleared at one time with the

```
CLR: UTIL!
```

command. This is also only allowed as an immediate action. The confirmation message

```
CLR UTIL COMPLETED
```

will be printed after all the breakpoints have been successfully cleared.

Displaying Breakpoint Status

To get a summary of all currently defined breakpoints,

```
OP: UTIL!
```

is used. For every breakpoint currently in the system, the breakpoint number, utility ID (specified in octal), process ID (specified in decimal), address (specified in hexadecimal), length, mode (`R`, `W`, `RW`, or `EXC`), and state (`ENABLED` or `DISABLED`) will be printed. The mode will be annotated with `(H)` for hardware items.

6.3.3 Overriding A Default Time Limit

The `IN:DTIME` input message overrides the GRASP/EGRASP default dynamic real-time limit with a specified limit until the specified clock time. The real-time limit is a rough approximation of the percentage of processor time.

In GRASP, the limit is 100. With this limit, GRASP sometimes does not get 100 percent of processor time over long enough periods to perform all breakpoint actions. In EGRASP, the parameter limit is increased to 10000 to allow EGRASP enough time to complete breakpoint processing.

```
IN:DTIME=10000:UNTIL=2359;
```

If GRASP uses all of the time it is allowed according to the value of the dynamic real-time limit, an output message is printed indicating that all GRASP breakpoints were inhibited. The breakpoints must be selectively re-allowed.

6.3.4 Transfer Trace Function

The primary purpose of the transfer trace is to indicate the flow of execution around a target event. It is particularly useful in debugging programs. Like breakpoints, the trace must be set up in advance, describing exactly what is to be traced. The trace must then be started in a separate step. It then can be stopped, restarted, restopped, etc., until the decision to clear it. Each trace must be cleared before another can be defined. Dumping the utility circuit's trace memory is a relatively slow operation, as is setting up the circuit for the trace. However, while the trace is running, it does not use the resources of the main processor.

The trace can be used in either of two basic ways: to record the events leading to some target event or to record the flow following some event. This is done by using a breakpoint to detect the occurrence of the event and, in its action list, to either start or stop the trace as desired.

- To record the flow of execution leading to an event, the trace is set up and started as an immediate command. It then begins recording and maintains the most recent 2048 or 8192 entries in memory. It does this by always overwriting the oldest information with the new. The utility circuit, which has its own small memory in which the flow of execution is recorded, is used for the trace feature. It has room for 2048 entries for UN61 or 8192 for UN615. A breakpoint is also defined with a description of the target event upon which the trace will be turned off, and its action list must include the command to stop the trace (along with other actions if desired). A message is printed when the breakpoint fires and the trace is turned off. The trace memory can then be examined using the appropriate message. Because the trace runs without using any time from the

main memory processor, a trace such as this can be allowed to run for long periods without interfering with the machine's tasks.

- To record the flow that follows a target event, the trace is set up, and a breakpoint is defined for the target event with the command to start the trace in the action list. With this type of trace, the tracing stops when its memory is filled up resulting in a message being printed. The trace memory can then be examined.

The trace records 2048 or 8192 entries of three types — UIDs, "from" addresses, and "to" addresses. The UID of the new process is recorded whenever control passes from one process to another. Whenever a branch is taken within a process, the address of the branch is recorded as a "from" address. The address that is branched to is recorded as a "to" address.

Normally, the pattern of entries in the trace memory is alternating "from-to" addresses, with an occasional UID. All addresses following a UID should be interpreted as virtual addresses within a process identified by that UID. The "from-to-from-to" pattern is sometimes altered without loss of any real information. Sometimes, in code generated by a compiler, there are branch instructions that jump to other branch instructions. A trace of this would have a "to-from" pair with the same values; the address of the branch "jumped to." The utility circuit in such a case saves room in its memory by collapsing the pair into a single "to" address. The sequence that actually gets recorded is "from-to-to." Whenever "to" addresses are adjacent in the trace, it is because a branch was taken to another branch.

Several options are available on the trace that effect the information recorded. These are useful for getting a longer history of the flow. Due to the size of the trace, memory is limited to get a longer picture, some information must be sacrificed.

The transfer trace function is controlled with the trace operations defined in "Trace and Matching Messages," Section 6.3.5. Starting and stopping the trace is performed separately from the definition giving more flexibility in control. The definition of the trace consists of:

1. choosing the wrap or no-wrap mode,
2. choosing the information to be recorded,
3. specifying any additional conditions restricting the recording of information.

The trace can be started and/or stopped from breakpoints or by immediate commands. Such a breakpoint must be implemented with hardware, therefore, knowledge of hardware availability is required when setting up a trace of this type.

6.3.5 Trace And Matching Messages

Trace Initialize Monitoring

This message specifies the definition of an AM trace set up. The five operations available to the trace program flow and the commands to implement these operations are given here. The trace goes into the `NEW` state with successful completion of this command.

```
INIT:UMEM[,{UID=a|PID=b}][,ADDR={c&&d|c[,L=e|NL=f]}]}][,STORE=g]
   [,STKADD=h[,STKSZ=i]][,STOP=FULL][[,COND=j][,j][,j]][:WORD];
```

Trace Start Monitoring

This message causes the 3B20D computer GRASP/EGRASP transfer trace to start monitoring the flow of execution as previously set up with an INIT:UMEM input command. This command can be used either as an immediate action, or it can be used in the action list of a WHEN command. The trace goes into the RUNNING state with successful completion of this command.

```
ALW:UMEM{!|;}
```

Trace Stop Monitoring

This message causes the 3B20D computer GRASP/EGRASP transfer trace to stop monitoring the flow of execution. The transfer trace goes into the STOPPED state with successful completion of the command. This command can be used either as an immediate action, or it can be used in the action list of a WHEN.

```
INH:UMEM{!|;}
```

Trace Definition Clear

This message causes the definition of a 3B20D computer GRASP/EGRASP transfer trace to be removed. The trace goes in the UNDEF state with successful completion of the command.

```
CLR:UMEM[:UCL];
```

Trace Dump To File

This message causes the contents of the 3B20D computer GRASP/EGRASP trace to be dumped to a file. The trace goes into the DUMPED state with the successful completion of the command.

```
OP:UMEM[:UCL];
OP:UMEM[:UCL][MCH];
```

Any of these operations can be done as immediate operations. Only the commands to start and stop the trace are allowed in breakpoint action lists. Each trace will be "undefined," "new," "running," "stopped," or "dumped."

Before any trace command is executed, the trace state is checked and the command is rejected if it is logically incorrect for the trace state. There are seven tracing types available to gather the information that is recorded in the trace memory. These options are described here.

UID Trace

Because the trace memory is limited, the duration of the trace is inversely proportional to the amount of detail recorded. One way to get a long history of activity is to restrict the trace to store only the UIDs of the processes that run. This gives a good, long picture but little resolution. With this type of trace, the output indicates every process switch including those to the kernel and the special processes.

Transfer Trace

An alternate method (which is the default) is to store the addresses involved in every nonsequential change in execution flow. That is, for every branch, jump, call, and return instruction, the address of the instruction (or "from" address) and the destination (or "to" address) are recorded. In addition, whenever a change in process occurs, the new process UID is recorded so the "to" and "from" address can be interpreted in context. This gives more detail than the UID-only option.

Data History Trace

The data history mode allows recording of program data accesses. Each time a data access occurs, the trace memory records the data, the data address, and the current program address. All four trigger functions are capable of controlling the recording activity. When an address range is specified on the `INIT` input message, the block matcher is used and the trace only records data when a memory location within the range is accessed. By using a UID comparison, the trace can be limited to a unique process.

Simultaneous Transfer and Data History Trace

A simultaneous transfer and data history trace records all data associated with a data history trace and a transfer trace with the exception of a load or store flag indicating data accesses.

Function Trace

The function trace memory mode records software function changes. The 3B20D computer native instructions `SAVE` and `RETURN` are set up using opcode matchers and any other conditions established by the `INIT` input message. When the `SAVE` instruction is executed, the `CALL` address (the previous program address), the `SAVE` address, and the current `UID` value are recorded. Execution of `RETURN` instruction allows trace memory to record the `RETURN` address (the current program address), the following program address, and the current `UID` value.

Function with Parameters Trace

The trace of functions with parameters records the call instruction address, the save instruction address, and the parameters pushed on the stack. The stack address and stack size may be specified with the `INIT` input message. If these values are not supplied, default values will be used. Unlike the function trace, return instructions will not be recorded.

Simultaneous Data History and Function with Parameter Trace	A simultaneous trace of data functions with parameter trace records data history trace information with the exception of a load/store flag and function with parameter trace information.

6.4 LAYOUTS

6.4.1 Input Message Acknowledgements

The following is a list of the input message acknowledgements and definitions:

OK	The message was received, the appropriate process was initiated, and work was completed.
NG	No Good. The message was received, the appropriate process was initiated, but the process was unable to complete the work satisfactorily. Before being re-entered, the message should be checked in the 235-600-700, *Input Messages Manual* to verify that it was typed correctly. An incorrect number in the information field can cause this acknowledgment, or equipment trouble may also cause this result.
NA	No Acknowledgement. Normally, the system acknowledges an input within 5 to 10 seconds. If control of the message processing has been lost, NA is output to indicate an acknowledgement failure.
PF	Printout Follows. A printout will follow sometime later to explain in detail the results of the work initiated by the input message.
RL	Request Later. Request cannot be executed now due to unavailable system resources; for example, action was requested for a unit that is being diagnosed.
?I	Identification field (to the right of the first colon) contains an error. The message should be checked in the 235-600-700, *Input Messages Manual*.
?T	Time-out has occurred on the channel. Input has not been received in the allotted time and processing of the message has been aborted.
IP	In Progress. Request received and initiated. Further output will follow.
?A	Action field (to the left of the first colon) contains an error.
?E	Error exists in the message but cannot be resolved to the proper field.

6.4.2 Register Mnemonics

The registers that are readable in the administrative module are as follows:

R0	R1	R2	R3
R4	R5	R6	R7
R8	R9	R10	R11
RTC	TIMERS	SDR	PSW
HSR	T0	T1	T2
T3	T4	T5	T6
ATBSDR	ATBSCR	ATBQ	ATBPSW
ATBBGR	SBR0	SBR1	SBR2
SBR3	SBR4	SBR5	SBR6
SBR7	SYSBASE	TOPIS	UINTER
UINT0	UINT1	UINT2	UINT3
HG	PPR	BGR	CDR
SP	SAR	SCR	SIR
HM	DSTBUS	SRCBUS	ER
ONES	IB	SCRATCH0	SCRATCH1
ATBSAR	ATBQ		

The registers to be written as the destination of the GRASP commands are as follows:

R0	R1	R2	R3
R4	R5	R6	R7
R8	R9	R10	R11
RTC	TIMERS	SDR	PSW
HSR	T0	T1	T2
T3	T4	T5	T6
ATBSDR	ATBSCR	ATBQ	ATBPSW
ATBBGR	SBR0	SBR1	SBR2
SBR3	SBR4	SBR5	SBR6
SBR7	SYSBASE	TOPIS	UINTER
UINT0	UINT1	UINT2	UINT3
HG	PPR	BGR	CDR
IM	ISC	CAR	SSR
ERC	ISS	SSRS	HSRBGC
RNULL	PA	FP	AP
SP			

6.4.3 GRASP/EGRASP Output Message Layout

The purpose of the REPT GRASP message is to report on GRASP conditions of a general or emergency nature. The REPT GRASP output message can indicate that a GRASP real-time limit has expired, and has been reset to the normal installation default value. The message that indicates the static real-time limit has expired is:

```
REPT GRASP STATIC RESET
```

and to indicate that the dynamic real-time limit has expired the message is:

```
REPT GRASP DYNAMIC RESET
```

The REPT GRASP message can also indicate that there are processes with which GRASP cannot coexist and processing is being terminated:

```
REPT GRASP FLDOP ABORT
```

In addition, this message can also warn when data is being lost:

```
REPT GRASP DATA BUF OVFL
```

For complete detail on the GRASP output messages see the 235-600-750, *Output Messages Manual*.

6.5 GRASP/EGRASP EXAMPLE

Scripting is a method that allows the user to group a number of input messages or breakpoint definitions as a file and save the file in the user's directory. An execute statement reads this file one line at a time. The instruction in the file causes the GRASP Package to respond according to the coding as it would to entering the same commands at the MCC terminal.

Caution: Do not use the ALW statement in script files.

After a WHEN command, with its conditions and action list, is entered successfully, the breakpoint is assigned a number by GRASP. The breakpoint is then referred to exclusively by its number. Up to 20 different breakpoints can be defined in the system at any time. The numbers assigned to breakpoints during a debugging session are not reused.

There are three types of breakpoints:

Breakpoint on Execution of an Instruction

Breakpoints that fire on execution of an instruction are called software breakpoints because of the way they are implemented. The breakpoint itself is an instruction that transfers control to GRASP when it is executed.

```
WHEN:PID=2468,ADDR=h'98b2,OPC=70:EXC!
DUMP:REG=PA!
END:WHEN!
```

Breakpoint on Access of Data

Breakpoints that fire on accesses of data are implemented with hardware using matchers on either the UN21 UC, UN61 DUC, or UN615 DUC.

To set up a hardware breakpoint, GRASP configures the matchers that are needed and supplies the values that are to be matched. The circuitry continually compares the values with what is taking place on the machine. If the breakpoint is enabled, the breakpoint fires when all the matchers specified during set up match. If hardware breakpoint is disabled, the hardware passively tries to match but does not interrupt processing on the machine.

Because the amount of hardware on the utility circuit is limited, only four hardware breakpoints can be defined at one time.

```
WHEN:PID=2468,ADDR=h'98b2;RW!
DUMP:REG=PA!
END:WHEN!
```

Breakpoint on External Condition

A breakpoint can be defined to fire upon receiving an active external event backplane signal. This is implemented using a hardware trigger and the DUC matcher.

The condition breakpoint fires immediately upon receipt of the external event regardless of an executing process. The processor can in fact be idle when this occurs.

```
WHEN:COND=E!
DUMP:REG=PA!
END:WHEN!
```

Software Analysis Guide

CONTENTS PAGE

7. GENERIC UTILITIES 7-1
7.1 GENERIC UTILITIES OVERVIEW 7-1
7.2 CREATE AND SAVE A BREAKPOINT FOR FUTURE USE - GENERAL 7-4

LIST OF EXHIBITS

Exhibit 7-1 — Breakpoint Example To Inhibit OSDS Monitor 7-2

Exhibit 7-2 — Breakpoint Example: Function `RTsc_nscm` 7-2

Exhibit 7-3 — Breakpoint Example: Structure `mgLNT_REQ` 7-3

Exhibit 7-4 — Breakpoint Example: Function `SImoninh`. 7-4

7. GENERIC UTILITIES

7.1 GENERIC UTILITIES OVERVIEW

Generic Utilities refers to a set of switch resident, real-time software debugging tools available to maintenance personnel to localize and make temporary corrections to application programs in various *5ESS*® switch processors. They can also be used to analyze program flow. Generic utilities allow a user to do any of the following:

- Dynamic display of memory
- Temporary memory overwrites
- DUMP, LOAD, and COPY memory
- Nested IF/ELSE commands
- Limited symbolic debugging
- Multiple line entry capability
- Utility variables for temporary storage of data and constants
- Function call and GOTO capabilities
- Disassembly capability.

Utility commands allow the user to read, move, and (with certain restrictions) overwrite data contained in virtually any addressable location in the system. Conditionals permit the performance of utility commands when specified events occur in the designated processor. WHEN clauses allow for the temporary interruption of program flow, the collecting of system or process information, and the resumption of program flow. More detailed information about the generic utilities is found in 235-105-110, *Maintenance Requirements and Tools*.

The use of Generic Utilities requires detailed technical knowledge and sound judgement. UT code will, for example, read or write (almost) any valid memory address. This provides UT with its considerable power - along with its potential for risk. Powerful and versatile though it may be, Generic Utilities cannot "look ahead" to prevent degradation in system integrity or performance. It is the user's responsibility to decide where to place a breakpoint, define the commands that should be associated with that breakpoint, and to validate its structural integrity. The *5ESS* switch *Input Messages Manual*, 235-600-700 cautions that these messages can cause problems if they are misused. Some things to consider before placing a breakpoint:

- A breakpoint places special instructions in memory (also called a "trap"). Breakpoints must be cleared from memory before performing any program update activity or hashsum errors will occur.
- Dumping memory-mapped I/O in the SM may cause interrupts.
- If a UT breakpoint is hit too often, it may impact certain timed operations. If a utility is placed in a branch of code which is heavily executed, it may cause an overload or possibly even an initialization.
- Automatic variables declared within a block of code are only accessible while that block is being executed. This applies to automatic variables within a function, and automatics within blocks within a function.

- Planting a breakpoint on a link (or unlink) instruction will probably not have the desired results because these instructions allocate (or deallocate) stack space for use by a function.
- Altering the contents of dynamic data structures by use of the UT `LOAD` or `COPY` commands can disrupt call processing.
- Verify that a breakpoint is placed at the correct address. Computation and typographical errors are a common source of problems and aggravation.
- Use care to pass valid parameters to functions that require them.
- Verify calculations of memory location to be dumped, loaded, copied, etc. to protect system integrity. (The `COPY` command has a limit of 4 bytes per iteration.)
- Verify that the command to be used is supported by the processor in question. For example, a local stack can only be accessed if the breakpoint or match `WHEN` clause has been implemented for that processor type.

The breakpoint used in the following example shuts off the OSDS monitor in SM 2 when a specified event occurs in SM 8. The breakpoint for the 5E9(2) software release is given in Exhibit 7-1.

Exhibit 7-1 — Breakpoint Example To Inhibit OSDS Monitor

```
WHEN:UT:SM=8,FUNC="RTsc_nscm",OFF=h'1a,OPC=h'3028,FOREVER,NOPRINT!
IF:UT:SM=8,REG1=A6,INDIR1=2,OFF1=8-64,L1=2,EQ,VAL2=H'771,L2=2!
EXC:UT:SM=8,CALL,FUNC="SImoninh",ARG=1,PARM=2!
END:UT:SM=8,WHEN;
```

The breakpoint uses a pointer passed to function `RTsc_nscm()` to access the `mgLNT_REQ` structure. Function `RTsc_nscm()` determines how a call will be rerouted to the next series completion member.[1]

Structure `mgLNT_REQ` contains a port value, `DMGPORT (port)`, which is compared to `VAL2=H'771`. If there is a match, a call is made to function `SImoninh()` to shut off the OSDS monitor in SM 2. See Exhibit 7-2 for the structure of pointer `lnterm_ptr` in function `RTsc_nscm`.

Exhibit 7-2 — Breakpoint Example: Function `RTsc_nscm`

```
@FUNCTION: RTsc_nscm
             .                    .                    .
             .                    .                    .
             .                    .                    .
   /*
    * NAME:      RTsc_nscm()
    *
    * ABSTRACT:
    *            This function determines whether a series completion(SC)
    *            call will be rerouted to the next SC member(NSCM) directly
    *            or by DN. If the SC call is routed directly, this function
    *            appropriately sets up the needed information in the RTRERTE
    *            message and sends it to the LRSP where the NSCM resides.
    *
```

1. Series Completion (SC) is a feature that directs calls to another, customer specified Directory Number (DN) when the originally called DN is busy. The SC feature is a form of hunting that starts with the called DN [i.e., the Originally Dialed Member of the Series Completion group (ODMSC)], and searches a linked list of DNs until one of the following conditions is met: (1) an idle line is found, (2) the end of the linked list of DNs is reached (regular SC group), (3) the call is rerouted back to the ODMSC (circular SC group), or (4) the call has been rerouted 16 times (exhausted SC attempts).

```
     * ENVIRONMENT:
     *            processor:  SM
     *            operating system: OSDS
          .                   .                   .
          .                   .                   .
          .                   .                   .
    RET_VAL
    RTsc_nscm(lnterm_ptr, servclass)
    struct{
            OSMSGHEAD msghead;
            struct mgLNT_REQ text;
    }*lnterm_ptr;
    {
          .                   .                   .
          .                   .                   .
          .                   .                   .
```

Pointer `lnterm_ptr` points to a structure which contains the message header `msghead` and structure `text`. The header (`msghead`) consists of 8 bytes. Structure `text` is much larger, but only the first 60 bytes are of any interest. A partial layout of structure `mgLNT_REQ` is given in Exhibit 7-3. In structure `mgLNT_REQ`, domain `DMPHPTHDSC (path_desc)` is 24 bytes, domain `DMRTG_DATA (rtg_data)` is 28 bytes, domain `DMOSPID (far_pid)` is 4 bytes, and domain `DMGPORT (port)` is 4 bytes. The first two bytes of domain `DMGPORT` are the domain `DMPORT`. It is the value of `DMPORT (port)` that is being compared for a possible match.

Exhibit 7-3 — Breakpoint Example: Structure `mgLNT_REQ`

```
struct  mgLNT_REQ {
            DMPHPTHDSC      path_desc;
            DMRTG_DATA      rtg_data;
            DMOSPID         far_pid;
            DMGPORT         port;
            DMGPORT         pikport; /* Port that initiated the Pickup */
            DMASCID         cid;
              .                   .                   .
              .                   .                   .
              .                   .                   .
};
```

The `IF` statement in the breakpoint uses indirection and offset to determine the address where the value of the port is stored:

```
IF:UT:SM=8,REG1=A6,INDIR1=2,OFF1=8-64,L1=2,EQ,VAL2=H'771,L2=2!
```

An offset of 8 is added to the address contained in the frame pointer (A6). 64 bytes is then added to the contents of this address to determine the address of the port itself. In this example, the frame pointer (A6) contains the address `h'138e0e`.

1. Go to the address (`h'138e0e`) contained in the frame pointer.
2. Offset that address by 8 bytes (i.e., `h'138e0e` + 8 = `h'138e16`).
3. Go to the address found at address `h'138e16`, (in this case, `h'9a2552`). This is the start address of structure `mgLNT_REQ`.
4. Offset this address by 64 bytes (i.e., `h'9a2552` + `h'40` = `h'9a2592`). The next 2 bytes of data is the value of the port.

When this breakpoint fires and the above condition is true, function SImoninh() is called. The parameter passed by UT to the function is the switching module number in which the OSDS monitor is to be inhibited. Exhibit 7-4 shows a partial listing of function SImoninh().

Exhibit 7-4 — Breakpoint Example: Function SImoninh

```
/*
 * Function:    SImoninh( smnum )
 *
 * Parameters:  smnum - SM Number where Monitor is to be inhibited
 *              255 - use SImonainh array to inhibit far processors
 *
 * Description: This function is called by a breakpoint and will
 *              inhibit the OSDS Monitor in a selected SM or in a
 *              predefined set of SMs.
     .                  .                   .
     .                  .                   .
     .                  .                   .

RET_VAL
SImoninh( smnum )
register char   smnum;
{
        struct {
                OSMSGHEAD               msghead;
                struct mgMON2MICO       micomsg; /* msg to send MICO */
        } msg;
     .                  .                   .
     .                  .                   .
     .                  .                   .
```

7.2 CREATE AND SAVE A BREAKPOINT FOR FUTURE USE - GENERAL

The following is an example on how to create and save a breakpoint for future use.

1. Enter *UNIX*[2] Shell:

   ```
   rcv:menu:sh;
   ```

2. Change to a directory with write permissions and create a file:

   ```
   cd /tmp
   ed scriptA
   ```

3. Append the file:

   ```
   a
   cd /cft/shl
   /cft/bin/pdshl.app "WHEN:UT:SM=8,FUNC=\"RTsc_nscm\",OFF=h'1a,OPC=h'3028,FOREVER,NOPRINT!" 2> /dev/null
   /cft/bin/pdshl.app "IF:UT:SM=8,REG1=A6,INDIR1=2,OFF1=8-64,L1=2,EQ,VAL2=H'771,L2=2!" 2> /dev/null
   /cft/bin/pdshl.app "EXC:UT:SM=8,CALL,FUNC=\"SImoninh\",ARG=1,PARM=2!" 2> /dev/null
   /cft/bin/pdshl.app "END:UT:SM=8,WHEN;" 2> /dev/null
   ```

   Note: Each command line in the file cannot be split into more than one line. The parameters shown are an example, all parameters are not valid for all SM types.

4. Quit the append mode by entering a period:

   ```
   .
   ```

5. Write the file:

   ```
   w
   ```

2. Registered trademark of The Open Group.

6. Quit the editor:

   ```
   q
   ```

7. Change the mode of the file to executable:

   ```
   chmod 777 scriptA
   ```

8. Exit *UNIX* Shell:

   ```
   <Control D>
   ```

9. The utility commands stored in the file `scriptA` can now be executed with the command:

   ```
   EXC:ENVIR:UPROC,FN="/tmp/scriptA";
   ```

   This will read the generic utility statements in file `scriptA` one line at a time, as if they were being input from a keyboard.

10. To allow the utility, input the message:

    ```
    ALW:UT:SM=8,UTILFLAG=x;
    ```

    where `x` is the identification number of a specific `WHEN` clause that is to be enabled to an active state.

Software Analysis Guide

CONTENTS PAGE

8. INTERRUPT ANALYSIS 8-1

8.1 3B20D PROCESSOR INTERRUPTS 8.1-1
8.1.1 Introduction to 3B20D Processor Interrupts 8.1-1
8.1.2 Interrupt Stack 8.1-1
8.1.3 Error Interrupt Handler 8.1-2
8.1.4 Execution Levels 8.1-3
8.1.5 Non-Operational Interrupt: Error 8.1-3
8.1.5.1 Introduction to Non-Operational Interrupt Errors 8.1-3
8.1.5.2 Software Error Handling 8.1-4
8.1.5.3 Hardware Error Handling. 8.1-5
8.1.6 Non-Operational Interrupt: Maintenance 8.1-6
8.1.7 Error Register 8.1-6
8.1.8 Interrupt Source Register 8.1-11
8.1.9 Interrupt Masking 8.1-12

8.2 *MOTOROLA* MC68000 PROCESSOR FAMILY INTERRUPTS . . . 8.2-1
8.2.1 Module Controller/Time Slot Interchange 8.2-1
8.2.1.1 MCTSI Functions 8.2-1
8.2.1.2 Network Control and Timing Links 8.2-2
8.2.1.3 Service Groups 8.2-2
8.2.1.4 Scan and Distribute Operations 8.2-2
8.2.1.5 MCTSI Subunits 8.2-3
8.2.1.6 MCTSI Interfaces 8.2-9
8.2.1.7 ISDN Peripheral Units 8.2-13
8.2.2 Categories Of Interrupts 8.2-14
8.2.2.1 Interrupt Categories. 8.2-14
8.2.2.2 Service Requests. 8.2-14
8.2.2.3 PICB Circuitry Related Errors 8.2-14
8.2.2.4 Control Interface Errors 8.2-14
8.2.2.5 Time Slot Interchange Errors 8.2-15
8.2.2.6 Signal Processor Errors 8.2-15
8.2.2.7 Dual Link Interface Errors 8.2-16
8.2.2.8 Switching Module Processor Errors 8.2-16
8.2.3 Interrupt Levels 8.2-17
8.2.3.1 Interrupt Priority Levels 8.2-17
8.2.3.2 SM Interrupt Levels 8.2-17
8.2.3.3 Vectored and Autovectored Interrupts . . . 8.2-17
8.2.4 MCTU Interrupt Registers 8.2-18

8.2.5 Module Controller Interrupts (Level 7) 8.2-18
8.2.5.1 Module Controller Interrupt Registers 8.2-18
8.2.5.2 Reset Source Register 8.2-19
8.2.5.3 Software Error Source Register 8.2-21
8.2.5.4 Hardware Error Source Register 8.2-22
8.2.5.5 Memory Error Source Register 8.2-23
8.2.5.6 DLI Error Source Register 8.2-23
8.2.6 Subunit and Peripheral Hardware Interrupts (Level 4) 8.2-24
8.2.6.1 Subunit and Peripheral Hardware Interrupt Registers 8.2-24
8.2.6.2 PIC B Register 8.2-25
8.2.6.3 PIC C Register 8.2-26
8.2.6.4 PI Error Source Register 8.2-26
8.2.6.5 CI Error Source Register 8.2-27
8.2.6.6 SP Error Source Register 8.2-28
8.2.6.7 TSI Error Source Register 1 . . . . . 8.2-29
8.2.6.8 TSI Error Source Register 2 . . . . . 8.2-30
8.2.6.9 TSI Error Source Register 3 . . . . . 8.2-32
8.2.6.10 DLI Error Source Register 1 . . . . . 8.2-32
8.2.6.11 DLI Error Source Register 2 . . . . . 8.2-33
8.2.7 *Motorola* MC68XXX Processor Family Distinctions . . . 8.2-33
8.2.7.1 Family of *Motorola* MC68XXX Processors 8.2-33
8.2.7.2 *Motorola* MC68000 Processor 8.2-34
8.2.7.3 *Motorola* MC68012 Processor 8.2-34
8.2.7.4 *Motorola* MC68020 Processor 8.2-34
8.2.7.5 *Motorola* MC68040 Processor 8.2-35
8.2.7.6 *Motorola* MC68060 Processor 8.2-47
8.2.8 Interrupt Masking 8.2-50
8.2.8.1 Interrupt Masking 8.2-50
8.2.8.2 Status Register 8.2-51

8.3 INTERRUPT RECEIVE ONLY PRINTER (ROP) OUTPUT 8.3-1
8.3.1 AM Interrupt ROP Output 8.3-1
8.3.2 SM Interrupt ROP Output 8.3-1

8.4 INTERRUPT ANALYSIS EXAMPLE — HARDWARE 8.4-1

8.5 INTERRUPT ANALYSIS EXAMPLE — SOFTWARE 8.5-1

LIST OF FIGURES

Figure 8.1-1 — 3B20D Processor Interrupt Stack 8.1-2

Figure 8.2-1 — MCTSI Subunits 8.2-4

Figure 8.2-2 — MCTSI Subunits and Interfaces 8.2-10

Figure 8.2-3 — PIDB Data Time Slot Configuration 8.2-11

Figure 8.2-4 — PICB Bit Configuration and All Seems Well (ASW) Code Definitions 8.2-12

Figure 8.2-5 — NCT Link — Control Time Slot Content 8.2-13

Figure 8.2-6 — MCTU Interrupt Registers 8.2-18

Figure 8.2-7 — Module Controller (Level 7) Interrupt Registers 8.2-19

Figure 8.2-8 — Subunit and Peripheral Hardware (Level 4) Interrupt Registers 8.2-24

Figure 8.2-9 — Interrupt Hierarchy 8.2-35

Figure 8.2-10 — SMP40 Non-maskable Interrupt Hierarchy. 8.2-37

Figure 8.2-11 — PIC A Register 8.2-38

Figure 8.2-12 — PIC B Register 8.2-41

Figure 8.2-13 — PIC C Register 8.2-43

Figure 8.2-14 — Interrupt Enable Register 8.2-45

Figure 8.2-15 — Read Interrupt Status Register 8.2-46

Figure 8.2-16 — Auxiliary Status Register 8.2-47

Figure 8.2-17 — Test Utility Bus Status Register. 8.2-47

Figure 8.2-18 — SMP60 Non-maskable Interrupt Hierarchy. 8.2-49

Figure 8.2-19 — CORE60 Status and Control Register 8.2-50

Figure 8.2-20 — Status Register and Interrupt Masking Structure 8.2-51

Figure 8.4-1 — PERAD Register 8.4-2

LIST OF TABLES

Table 8.1-1 — Error Register 8.1-7

Table 8.1-2 — Interrupt Source (IS) Register 8.1-12

Table 8.2-1 — Reset Source Register 8.2-20

Table 8.2-2 — Software Error Source Register 8.2-21

Table 8.2-3 — Hardware Error Source Register 8.2-22

Table 8.2-4 — Memory Error Source Register 8.2-23

Table 8.2-5 — DLI Error Source Register 8.2-24

Table 8.2-6 — `PIC B` Register 8.2-25

Table 8.2-7 — `PIC C` Register 8.2-26

Table 8.2-8 — PI Error Source Register 8.2-27

Table 8.2-9 — CI Error Source Register 8.2-27

Table 8.2-10 — SP Error Source Register 8.2-29

Table 8.2-11 — TSI Error Source Register 1 . . . . . 8.2-29

Table 8.2-12 — TSI Error Source Register 2 . . . . . 8.2-31

Table 8.2-13 — TSI Error Source Register 3 . . . . . 8.2-32

Table 8.2-14 — DLI Error Source Register 1 . . . . . 8.2-32

Table 8.2-15 — DLI Error Source Register 2 . . . . . 8.2-33

LIST OF EXHIBITS

Exhibit 8.2-1 — `PIC A` Register: File `SMmp_icldr.h` 8.2-38

Exhibit 8.2-2 — `PIC B` Register: File `SMmp_icmdr.h` 8.2-41

Exhibit 8.2-3 — `PIC C` Register: File `SMmp_ichdr.h` 8.2-43

Exhibit 8.4-1 — Hardware Interrupt Example 8.4-2

Exhibit 8.5-1 — `FCoo_ans()` Function Example 8.5-2

8. INTERRUPT ANALYSIS

Multitasking allows peripheral devices to perform much of their own processing, independently of the main processor. Whenever a peripheral device requires the attention of the main processor, the peripheral device issues a signal called an interrupt.

In response to an interrupt, the main processor sets aside its current activity and tends to the device. The main processor then returns to whatever it was doing when the interrupt occurred. In this way, the main processor can carry out several simultaneous, or concurrent, tasks. Program interrupts permit quick response to events that occur at unpredictable times.

Two types of interrupt occur in the *5ESS*® switch:

- Operational interrupt — normal to the system and implemented to trigger planned operations. This section does not discuss operational interrupts.
- Maintenance interrupt — response by the switching system to a particular trouble condition. The maintenance interrupt performs service protection, localization, and fault isolation. Maintenance interrupts can be classified into a hierarchy of interrupt levels related to the extent that service is affected by the interrupt.

8.1 3B20D PROCESSOR INTERRUPTS

This 3B20D Processor Interrupts section includes the following topics:

- Interrupt stack
- Error interrupt handler (EIH)
- Execution levels
- Non-operational error interrupts
- Non-operational maintenance interrupts
- Error register
- Interrupt source (IS) register
- Interrupt masking.

8.1.1 Introduction to 3B20D Processor Interrupts

When a process causes an error in the administrative module (AM), an interrupt is generated and the process data is placed on the interrupt stack; this is usually referred to as the saved state.

The interrupt causes the suspension of running processes in favor of higher priority jobs. Priority is the rank that one process has in relation to another. Priority determines whether a process keeps running when an interrupt occurs, or whether it is suspended while another process is allowed to run.

Interrupt masks, plus program interrupt requests (PIRs), control the interrupt hierarchy.

8.1.2 Interrupt Stack

The machine state of a suspended process is saved on the interrupt stack. The interrupt stack is currently in a separate segment of the operating system. The saved state of a process is a 20-word structure, as shown in Figure 8.1-1. As a series of interrupts occur, the interrupt state of running processes is pushed on and popped off the interrupt stack.

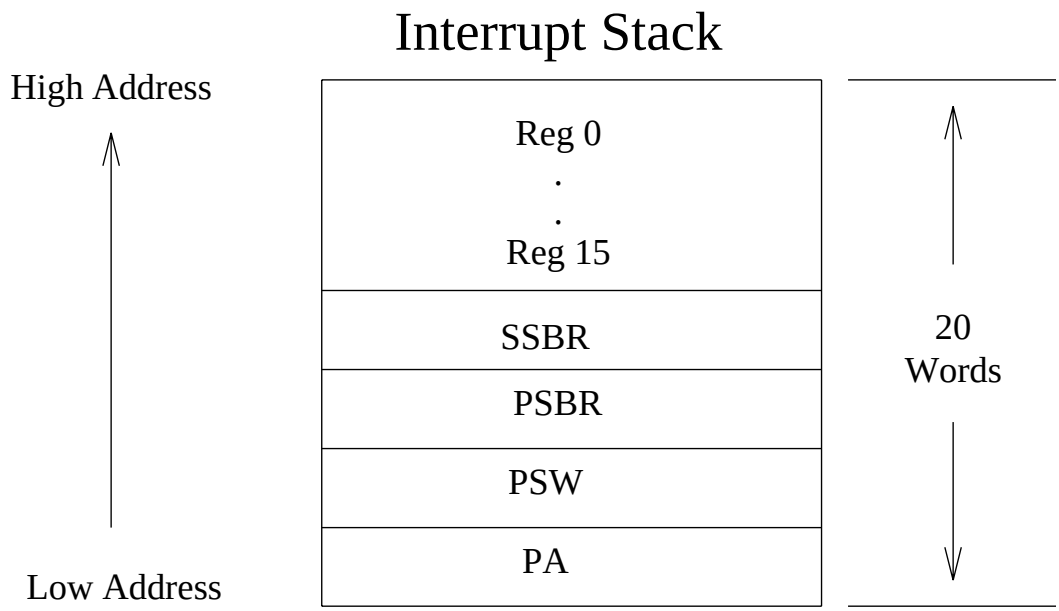


Figure 8.1-1 — 3B20D Processor Interrupt Stack

The interrupt stack saved-state information is useful in identifying which process was running when the maintenance reset function (MRF) occurred. The program address (PA), processor status word (PSW), and primary segment base register (PSBR) are the first entries on the interrupt stack, along with the utility ID of the suspended process. The utility ID and program address provide enough information to positively identify a process.

PA — The program address counter is the address where the process encountered the error.

PSW — The processor status word register describes the level, privileges, and math result flags.

PSBR — The primary segment base register is the base register used by the process to allow it to employ virtual memory. The value in this register is decoded by the hardware to the actual physical location in memory where the process is located.

SSBR — The secondary segment base register is used when a process needs to access data which is not within its addressing space. To access another address space, the secondary base register is loaded with the second address space and assembler instructions are used for data transfer between the two address spaces.

8.1.3 Error Interrupt Handler

The next process allowed to run is the error interrupt handler (EIH), which is the principle hardware fault recovery controller. It receives all hardware error interrupts and controls the recovery sequences that follow.

The EIH is responsible for collecting the saved-state data, determining the severity of the error, and taking the appropriate recovery actions. The EIH places the information in low physical memory, where it will later be found and output to the receive only printer (ROP) and the error log file.

8.1.4 Execution Levels

The program interrupt request logic of *UNIX*[1] RTR allows 16 individual execution levels.

- Supervisor processes use levels 0 and 1.
- System processes, which are kernel processes, use level 2.
- All other kernel processes use levels 3 through 15.

  Application kernel processes run in these 13 levels. The EIH uses level 15 exclusively to assure proper fielding of error conditions.

  EXAMPLE:

  This example shows how interrupts control the execution level of the computer.

  Suppose a kernel process at execution level 5 is running. The 10-millisecond hardware timer that runs at level 15 fires. The timer interrupt, which is at a higher execution level than the currently running process, interrupts the kernel process. The machine state of the kernel process is saved on the interrupt stack, and the timer routine gains control of the computer. Because the timer routine is at level 15, it cannot be interrupted.

  If the timer routine sends timeout events to two other kernel processes at levels 3 and 6, the level 6 process gains control and runs. While this level 6 process is running, the timer interrupt fires again, interrupts the level 6 process, and regains control of the computer. Levels 3, 5, and 6 kernel processes are now on the interrupt stack.

  The timer finishes and the level 6 process is popped off the stack and continues to execute. When the level 6 process terminates, the level 5 kernel process regains control. Finally, when this process finishes, the level 3 process will pop off the stack and start running.

  The same priority strategy applies when one process sends an event or message to another process. If a kernel process that is currently running at execution level 4 sends a message to a kernel process at level 3, the current process continues to run because it has a higher priority. However, if the level 4 process sends a message or event to a level 8 kernel process, control is immediately given to the level 8 process. Only when this new level 8 process is finished, and if no higher priority jobs are pending, will the original level 4 process be popped off the interrupt stack and allowed to run again.

8.1.5 Non-Operational Interrupt: Error

8.1.5.1 Introduction to Non-Operational Interrupt Errors

Two types of non-operational interrupts can occur in the 3B20D computer.

- error interrupts

1. Registered trademark of The Open Group.

- maintenance interrupts

Error interrupts are non-fatal errors detected by the machine, occurring in either the online or standby processor.

When an error interrupt occurs, the EIH takes the following steps:

1. The error is classified by analyzing the error register in the processor or interrupting channel.
2. An error counter is incremented by calling a configuration management error reporting function. This function determines if the threshold has been exceeded.

When error interrupts occur, the associated information is written in either the memory history log file (MEMLOG) or the error interrupt handler log file (ERLOG).

MEMLOG The MEMLOG file contains additional information about memory errors. At the time the error took place, a report message (REPT) associated with the entries is produced.

The additional information consists of error registers collected from memory controller hardware, which depict the state of the hardware when the error occurred. The contents of these registers may provide an indication of what happened.

ERLOG The ERLOG contains a history of all error interrupts handled by the error interrupt handler (EIH) that are not associated with the main memory. Among the errors handled are input/output errors, direct memory access errors, other processor hardware errors, and software errors.

The first three bits in the interrupt source (IS) register are used to determine fault classification.

IS **Bit**	**Fault Classification**
0	Online hardware
1	Offline hardware
2	Online software

The errors may implicate either the active processor or its mate. Errors for the mate are only handled if the mate is in the standby state. An out-of-service processor will not have its error and interrupt leads enabled in the active mate. This means that errors in an out-of-service processor are not detected by the active processor.

The EIH gains control of the computer at a pre-determined error handling routine, determined by which bit is set in the IS register. The locations of the error handling routines are placed in the interrupt vector table during system initialization or on completion of a processor restore.

8.1.5.2 Software Error Handling

In the 3B20D computer, software errors are of two basic types:

- Errors detected by the hardware that cause the EIH process to be dispatched.
- Errors detected by the microcode during the execution of a processor instruction.

The software fault entry of the EIH process is attached to IS register bit 2. A software error affects this bit and causes the software fault routine to gain control of the computer. The microcode also sets IS register bit 2 when it detects an error.

When a process performs some action that causes a software error, its execution is suspended and the EIH gains control. The state of the suspended process is placed on the interrupt stack. When the error handler determines the error type, it also clears out the error indications in the error register and temporary registers used to pass data by microcode.

Error interrupts can escalate to a processor switch or an initialization if preset thresholds are exceeded.

8.1.5.3 Hardware Error Handling

The online and offline hardware error handling capabilities of the EIH process can be divided into two classes:

Online	Offline
Memory errors	Memory-related errors
All remaining online hardware-related errors	Violation of maintenance channel protocol

The EIH process attached to offline interrupts and offline memory is placed in the update mode. When the update circuit to the offline processor is enabled, every memory read and write in the online processor's memory is also performed via the update circuit to the offline memory. This allows the hardware to check both the offline and online memories for parity errors and read/write completion while at the same time keeping both memories identical in content.

Memory update ensures that the offline memory controller and memory are functional. Any remaining hardware not associated with memory update is not checked by the self-check hardware, nor is it handled by the EIH process. The remaining hardware is routinely tested by the audit process in the active processor, on processor softswitch, and by automatic diagnostics.

Standby Processor Errors

The types of errors which can occur in the standby processor are:

- Invalid maintenance channel order
- Single-bit parity error
- Multiple-bit parity error
- Memory controller timeout.

These errors occur in the offline processor, but are detected by the online processor.

The other-store-out-of-range error signals a problem in the offline memory, but is detected as an online error. This error may occur if the offline processor is equipped with less memory than the amount the online processor is actively using. This error is detected on a system access in the online processor, and the error is classified as a software error.

8.1.6 Non-Operational Interrupt: Maintenance

Maintenance interrupts are a higher level of interrupt than the error interrupt. They are caused by hardware or software sources and involve initialization of the online processor or a switch to the secondary processor.

The information associated with this type of interrupt is written to the postmortem logfile (PMLOG). The AM postmortem dump is divided into the following sections:

- Heading

  The PMLOG reports are either labeled MAINTENANCE INTERRUPTS or POSTMORTEM DUMP. POSTMORTEM DUMP appears for a manually requested initialization and sometimes when the initialization was started due to the application software.

- Initialization message

  This section indicates the source of initialization, the online processor at the time of initialization, the processor actually involved in the initialization, and the recovery action that took place. Also, the source of the request is indicated by the SOURCE field (hardware, software, or manual request).

- Requested initialization parameters
- Emergency action interface (EIA) buffer
- Timer
- General registers
- Faulty processor registers
- Interrupt stack saved state
- Offline registers
- Real-time clock.

When the initialization is a hardware request, the faulty control unit registers, timers, and the main store registers are of primary interest.

When the initialization is a software request, the requested initialization parameters and the general registers are of interest. The initialization message should be analyzed for both types of initialization.

An analysis of postmortem dump information requires the ability to interpret several hardware registers. The most important of these are the error register and the IS register.

8.1.7 Error Register

The error register is an error detecting device which is comprised of 32 bits.

- Bits 1 through 10 are used for stop-and-switch (SAS) type errors and main store parity errors.
- Bits 11 through 26 are used for various classes of interrupt errors. These errors are classified into four categories.
  - — Less serious hardware errors (I/O errors and main store refresh parity)
  - — Errors related to the offline processor

— Software related errors (privileged instruction error, address translation buffer [ATB] protection violation, and accessing unequipped memory)

— Memory management related errors (ATB page fault)

- Bits 27 through 31 are used to capture the destination bus parity when the bidirectional gating register is specified as the destination.

Table 8.1-1 describes the bit layout of the error register. Each bit is discussed in greater detail following the table.

Table 8.1-1 — Error Register

Bit	Error	Class	Active
0	Source bit parity	Stop-and-switch	0
1	Microcontrol parity	Stop-and-switch	0
2	Clock match error	Stop-and-switch	0
3	IB parity error	Stop-and-switch	0
4	ATB parity	Stop-and-switch	0
5	Cache error	Stop-and-switch	0
6	MYSERA	Stop-and-switch	0
7	My store time-out	Stop-and-switch	0
8	MYSERC	Online error interrupt	
9	Data manipulation unit (DMU) error	Stop-and-switch	0
10	Store address controller (SAC) error	Stop-and-switch	0
11	Invalid maintenance channel (MCH)	Offline error interrupt	0
12	Other store error A	Offline error interrupt	0
13	Other store refresh parity	Offline error interrupt	0
14	Other store data parity	Offline error interrupt	0
15	Other store time-out	Offline error interrupt	0
16	Channel error	Online error interrupt	0
17	I/O response error	Online error interrupt	0
18	I/O addressing error	Online error interrupt	0
19	Parity divert error	Online error interrupt	0
20	MYSERD	Online error interrupt	0
21	Protection violation	Software error interrupt	0
22	Virtual address out-of-range (VORA)	Software error interrupt	0

Table 8.1-1 — Error Register (Contd)

Bit	Error	Class	Active
23	Out-of-range address (MYSERB)	Software error interrupt	0
24	Out-of-range reference (other store)	Software error interrupt	0
25	Privileged instruction	Software error interrupt	0
26	Bad alignment on memory reference	Software error interrupt	0
27	Unused		
28	Source bus parity bits	—	1
29	Source bus parity bits	—	1
30	Source bus parity bits	—	1
31	Source bus parity bits	—	1

This section describes each bit in the error register.

Bit 0 **Source Bus Parity:** When this bit is set, there is a source bus bit rotate parity error. The source bus is an internal bus of the 3B20D computer. It is one of the primary internal buses used in data manipulation and computation.

This error causes an immediate stop-and-switch (SAS) of the currently processing processor to the mate processor. The error is detected by the hardware that initiates the SAS sequence. Due to certain recovery actions taken during a system initialization, this bit should be ignored if other bits of the error register (bits 4 through 31) are set.

Bit 1 **Microcontrol Parity:** This bit indicates bad parity in the micro-instruction register (MIR). The MIR controls the sequencing circuitry of the microprogram unit (MPU). It is also the buffer register for instruction read from the microstore.

This error results in an immediate SAS of the currently processing processor to the mate processor. The error is detected by the hardware that initiates the SAS sequence. Due to certain recovery actions taken during a system initialization, this bit should be ignored if other bits of the error register (bits 4 through 31) are set.

Bit 2 **Clock:** This bit indicates mismatches of the auxiliary and primary result clocks of the MPU. As with bits 0 and 1, this bit monitors hardware internal to the MPU and results in an immediate SAS with no microcode involvement. Due to certain recovery actions taken during a system initialization, this bit should be ignored if other bits of the error register (bits 4 through 31) are set.

Bit 3 **Instruction Buffer (IB) Parity:** This error occurs when the main IB register contains bad parity, and results in an immediate SAS sequence.

Bit 4 **Address Translation Buffer (ATB):** This bit checks parity over the address leads of the ATB. The ATB is a small memory used by the memory

management logic in the translation of a virtual address. The error is detected by the microcode which eventually initiates a processor SAS.

Bit 5 **Cache Parity:** When this bit is set, a parity error in the cache is present. The cache is a high-speed memory used to store frequently accessed main memory data. This error generates an SAS without microcode involvement.

Bit 6 **My Store Error A (MYSERA):** This bit indicates that a hardware error is present in the memory controller. A MYSERA can occur on either a refresh cycle or a system access. It is a logical OR of several hardware check circuits of the memory controller. The MYSERA is an SAS error that does not involve the microcode.

Bit 7 **My Store Time-Out (MS Time-Out):** This bit indicates whether or not the memory controller responded to a memory request within a period of time determined by the microcode. When the microcode detects an MS time-out, it initiates an SAS of the currently running processor. The contents of the error register are also copied to the microinterrupt error register (UER).

Bit 8 **My Store Error C (MYSERC):** When this bit is set, a non-correctable parity error occurred as the currently running processor attempted to read a location in memory.

Bit 9 **Data Manipulation Unit (DMU):** This error results in an SAS. It is detected by low-level hardware circuits and does not involve the microcode in the decision to SAS. The only recourse is to run diagnostics on the faulty processor once the system has recovered from the error.

Bit 10 **My Store Address Controller (SAC):** This error results in an SAS. It is detected by low-level hardware circuits and does not involve the microcode in the decision to SAS. The only recourse is to run diagnostics on the faulty processor once the system has recovered from the error.

Bit 11 **Maintenance Channel Order (MCH Order):** This error can occur under two circumstances:

1. The mate processor tries to communicate across the maintenance channel to the active processor. Such communication is not allowed. A processor that has its central control (CC) bit set will reject most commands across the maintenance channel. The CC bit is used to determine which processor has the right to execute as the active processor. A processor that is not active, in most instances, will be stopped and will not send MCH orders to the active processor.
2. The mate processor attempts to SAS to the active processor. When the active processor's maintenance channel receives the SAS request, it will not honor it but will instead set bit 11 of the error register. An SAS can be initiated by either hardware circuits or 3B computer microcode. In most instances, however, the hardware errors are inhibited in the mate processor and the mate is stopped.

Bit 12 **Other Store Error A (OTHSERA):** This error is in principle the same error as the MYSERA (bit 6), except that an OTHSERA occurs in a standby processor. For the system to detect the error, the EIH process must be attached to IS register bit 1 (offline hardware error).

Bit 13 **Other Store Error D (OTHSERD):** This error is similar to the MYSERD (bit 20). The major difference is that an OTHSERD occurs in the standby

processor while a MYSERD occurs in the active processor. Assuming that interrupts are not blocked and the currently executing process is below level 15, the EIH will be dispatched to clear up the error. If this is not the case, the EIH will be dispatched once interrupts are enabled and the process level drops below 15.

Bit 14 **Other Store Error C (OTHSERC):** This error is similar to the MYSERC (bit 8) error. The major difference is that an OTHSERC occurs in the standby processor while a MYSERC occurs in the active processor. Unlike a MYSERC, an OTHSERC does not present the possibility of a process getting bad data. The error is in the mate processor's memory; the online processor's memory is not effected.

Bit 15 **Other Store Time-Out (OS Time-Out):** This error is similar to an MS time-out (bit 7) in that the memory controller does not respond within a specified time interval on a memory access. Because an OS time-out is in the standby processor, the error is transparent to all system processes except those directly involved in clearing and reporting the error.

Bit 16 **Channel Error:** This bit is a logical OR of the error leads from all channels. When this bit is set, IS register bit 0 (online hardware error) is set as well.

Bit 17 **I/O Response:** This error is detected on the leading edge of a pulse point and indicates that a 3-out-of-6 (3/6) acknowledgment of a pulse point is active. At this point there should be no activity and the response is deemed to be invalid. The only error information collected is the hardware status register.

Bit 18 **I/O Address/Pulse Point (I/O addr/pp):** This error is detected on the trailing edge of the pulse point and indicates either an illegal 3-out-of-6 (3/6) code on the central control input output (CCIO) bus or that multiple pulse points are active. The only data that is collected on this error is the hardware status register.

Bit 19 **Parity Divert (pardiv):** The normal mode of channel operation assumes the channels respond with good parity. Where bad parity is expected, parity checks can be inhibited. When a channel response returns bad parity and good parity was expected, the result is a parity divert error. This error usually indicates a device problem since the channel does not regenerate parity over the data it receives.

Bit 20 **My Store Error D (MYSERD):** This bit indicates that a correctable (single-bit) error occurred on a memory read or refresh cycle.

Bit 21 **Protection Violation:** This error indicates that a process attempted to access a segment in which access was not allowed; for example, an attempt to write into a text segment. The error is detected by microcode which then dispatches the EIH by setting IS register bit 2 (online software error).

Bit 22 **Virtual Address Out-of-Range (VORA):** This error is detected by microcode and it can occur only under very large main memory (VLMM). The VLMM system can address up to 64 megabytes of memory. Therefore, the valid address values are limited to bits 25-0. If any of the remaining high order bits (31-26) are set during a memory fetch, the VLMM microcode will generate an interrupt and the EIH will be dispatched.

Bit 23 **My Store Error B (MYSERB):** This error indicates that a process attempted to access a location in memory that is not physically equipped. The most likely reason for an error of this type is mutilation of system segment and

page tables. However, incorrect memory equipage or bad hardware can also cause this error. In most cases, incorrect memory equipage problems are detected before the system ever achieves kernel processing capability on initialization.

The assumption is that the error is associated with a virtual memory access. It is also possible, however, for this error to occur on a physical memory access. This usually is due to a process using a bad address or pointer.

Bit 24 **Other Store Error B (OTHSERB):** This bit indicates that there is a problem with the offline memory equipment, the memory hardware, or a process accessing offline physical memory using a bad address.

Bit 25 **Instruction Privilege Violation:** This bit indicates that a process attempted to execute an instruction and did not have appropriate processor status word (PSW) privileges. This error is detected by microcode which attempts to dispatch the EIH by setting IS register bit 2 (online software error).

A system error counter is incremented on each instance of this error. If a sufficient number of errors occur within a specified time interval, recovery will be attempted. The recovery may include a processor switch and various levels of initialization.

Bit 26 **Address Violation:** This bit indicates that a process attempted to access a location in memory using a byte address. All 3B20D computer instructions that access memory are restricted to a full-word boundary. When this error occurs, the EIH is dispatched via IS register bit 2 (online software error).

Bits 27-31 **Source Bus Parity Bits:** These bits are used to trap the destination bus parity when the bidirectional gating register is specified as the destination. These bits are not error bits, but provide a means for examining the parity bits in the processor.

8.1.8 Interrupt Source Register

The interrupt source (IS) register is a 32-bit register whose bits may be set by external signals (interrupts) or by microprogram control. The bits are only cleared by microprogram control. When a bit is set in the IS register and recognized by the processor, the action specified for that particular interrupt bit is taken.

Table 8.1-2 describes each bit in the IS register.

Table 8.1-2 — Interrupt Source (IS) Register

Bit	Meaning
0	Online hardware error interrupt
1	Other CC error interrupt
2	Software error interrupt
3	Not connected
4	5-millisecond timer
5	10-millisecond timer
6	1A processor simulation
7	Flash interrupt (R6.4 and later, not connected)
8	Utility circuit interrupt
9	Stop the world interrupt
10	DMA channel interrupts
11	DMA channel interrupts
12	Not connected
13	Not connected
14	Not connected
15	Not connected
16	Not connected
17-31	Programmed Interrupt Request (PIR) 15 through 1

8.1.9 Interrupt Masking

The priority of a given process specifies which interrupt mask is to be associated with the process. The mask allows or inhibits interrupts, allowing the process to continue when an asynchronous interrupt occurs or is suspended. Currently, 32 interrupts can be masked or inhibited.

The interrupt mask register is a 32-bit register whose bits are set or cleared by the microprogram. Each bit in the interrupt mask corresponds to the same bit in the interrupt source (IS) register. Setting of any bit in the interrupt mask prevents the recognition of that corresponding interrupt when it appears in the IS register.

Of these 32 interrupt sources, 15 are software interrupts called program interrupt requests (PIRs), PIR 1 through PIR 15; they correspond to execution levels 1 through 15. The remaining interrupts are generated by hardware such as the 3B20D computer itself or external devices. Because these hardware interrupts are maskable, they allow the execution flow to be controlled.

8.2 *MOTOROLA*[1] MC68000 PROCESSOR FAMILY INTERRUPTS

The *Motorola* MC68000 Processor Family Interrupts section includes the following topics:

- Introduction to maintenance interrupts in the SM
- The module controller/time slot interchange (MCTSI), its subunits, and its interfaces
- Categories of interrupts and related error sources
- Interrupt levels
- Description of each interrupt register
- *Motorola* MC68000 processor family distinctions.

Maintenance interrupts in the switching module (SM) are used to report errors that could have an effect on system performance and subscriber service. An interrupt in the SM forces the switching module processor (SMP) to execute the instruction at a memory location specific to that interrupt.

Peripheral units employ interrupts to inform the processor of bad parity or other error conditions. The processor can also be interrupted by any of the following subunits: control interface (CI), time slot interchange (TSI), signal processor (SP), and dual link interface (DLI). If the active processor receives a reset signal, it will interrupt the mate (standby) processor.

There are three key information items in an SM interrupt.

- the MCTSI reporting the error
- the hardware subunit implicated by the error
- the type of interrupt

This information can be used to determine the cause of the interrupt. The specific error itself may suggest a course of action to resolve the problem.

8.2.1 Module Controller/Time Slot Interchange

8.2.1.1 MCTSI Functions

The module controller/time slot interchange (MCTSI) performs the following functions:

- Interfaces with other units within the SM to transmit control information from the SMP to its peripherals.
- Interfaces with units within the SM for pulse code modulation (PCM) data.
- Provides call processing, call supervision, and maintenance functions.
- Provides time division switching under control of the SMP.
- Pre-processes the signaling and control bits of time slot data, and provides the SMP with access to these bits.

1. Registered trademark of Motorola Inc.

8.2.1.2 Network Control and Timing Links

Four network control and timing (NCT) link pairs provide the transmission medium from the MCTSI to the time multiplexed switch (TMS).

The NCT link is the source of the office timing information for the SM. The dual link interface (DLI) extracts this timing information to keep the SM synchronized with the rest of the system. The MCTSI selects one of the NCT link pairs as the "master" reference timing source. The TMS, on the other hand, acts as a circuit switch for passing voice and data between the SMs.

8.2.1.3 Service Groups

In general, a peripheral unit is made up of duplicated service groups (SGs) controlling a periphery. A service group is a complete set of resources which can be affected by a single hardware fault.

Most peripheral units are comprised of two service groups (0 and 1); each receives orders from the active SMP but sends its response to both the active and standby processors. Each service group has its own common control (CC) board which distributes control and data information to the circuits within the service group.

The SMP has access to each service group by means of a peripheral interface control bus (PICB). This two-way bus carries scan and distribute orders from the SMP to each peripheral unit (via the control interface [CI]), and response and interrupt requests from the peripheral to the SMP.

Within the CI, each peripheral unit is identified by its associated PICB number. This PICB number is also part of a circuit's internal name, which is its identification within the system application software.

8.2.1.4 Scan and Distribute Operations

In preparation for a scan or distribute operation, the address register is loaded with the desired peripheral address, link address (CI board number and PICB number), and read/write bit. In the SMP, a scan order appears as an assignment from the data register to some memory location.

When the READY bit is set in the CI, it activates the READY lead. This causes the SMP to wait until the CI has completed its operations. Thus in the synchronous mode, a scan or distribute order will appear as one instruction in the SMP.

Serial Control Messages

Two types of serial control messages may be sent over a PICB.

Distribute A control message to cause a hardware action to take place within the SM.

Scan A control message to a scanner within the SM to return the status of specific scan point(s).

The format of these orders and replies depends on the type of unit with which the CI is communicating. Characteristics of orders and replies are:

- Each order has a start code that is examined by the peripheral unit service group (PUSG) when it is received. A bad start code is reported by a unique configuration of the "all seems well" bits in the reply. See Figure 8.2-4.
- All orders have a parity bit. Bad parity on an order received by the PUSG is reported via the "all seems well" bits of the reply.

- A portion of the address is used to enable a subunit (such as a circuit pack) within the peripheral unit. The remaining address bits are used to enable a register or circuit on the subunit in which data is to be read (scan order).

  If no register or circuit is enabled (or if more than one register or circuit is enabled) the state of the "all seems well" bits will indicate this error condition.

- The fourth configuration of the "all seems well" bits is used to indicate a successful system operation.

8.2.1.5 MCTSI Subunits

The MCTSI contains the following subunits:

- Time slot interchange (TSI)
- Data interface (DI)
- Dual link interface (DLI)
- Control interface (CI)
- Signal processor (SP)
- Switching module processor (SMP)
- Bootstrapper (BTSR)
- Packet interface (PI)

Figure 8.2-1 shows the relationships between the MCTSI subunits.

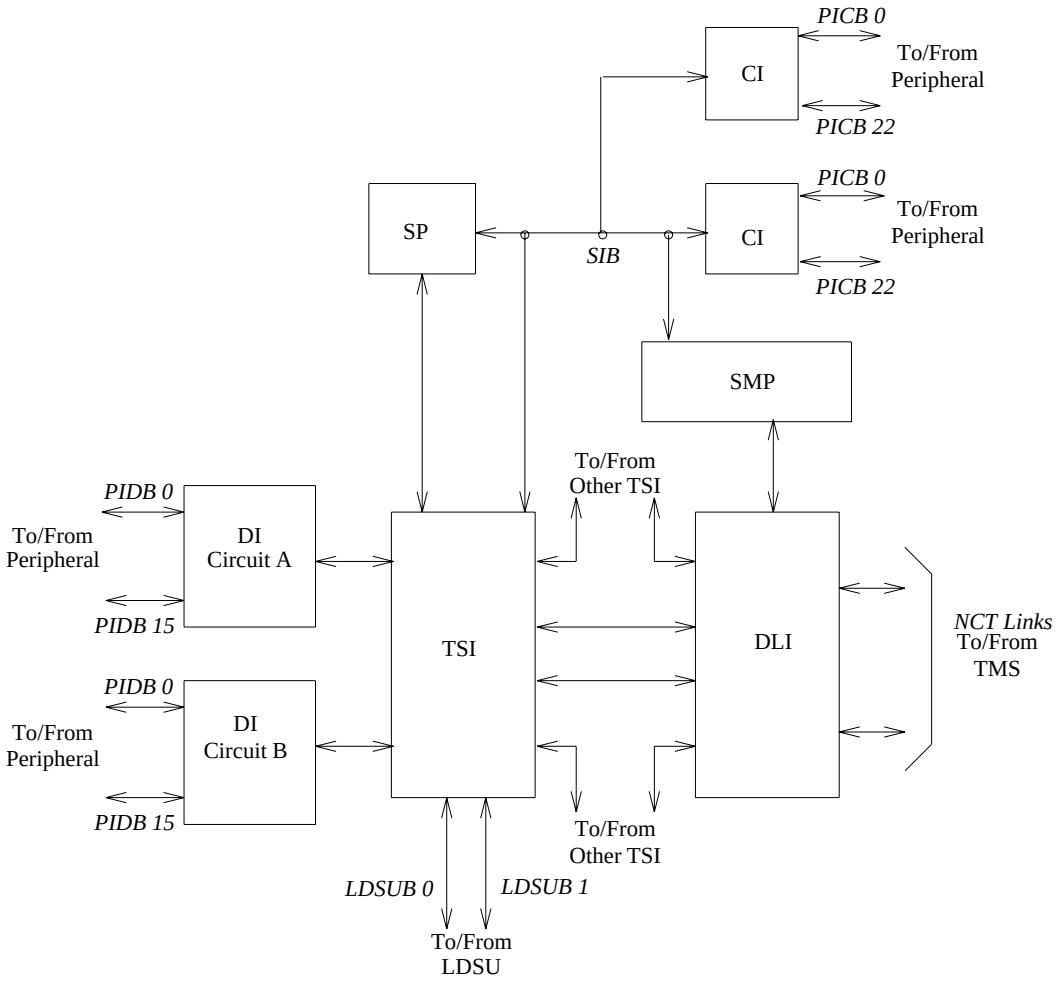


Figure 8.2-1 — MCTSI Subunits

The following describes each subunit.

TSI

The time slot interchange (TSI) performs the time division switching between the peripheral units of the SM and the time multiplexed switch (TMS).

In each SM, the outputs from lines and trunks are converted into 16-bit channels (time slots). These bits are used to encode voice or data, and for signaling, control, and parity. The time slots are switched through the TSI and time multiplexed onto the NCT links connecting each SM with the TMS.

The TMS is a time multiplexed space-division switch that provides the physical path for the digital signals carrying data and control information between SMs, and between SMs and the administrative module (AM). The TMS interconnects the modules through the NCT links.

Multiplexing

Multiplexing is the process that enables a digital transmission system to transmit multiple telephone conversations over a single strand of fiber or copper wire.

The multiplexer connects coded messages to a transmission line during one time slot period. After connecting the first coder to the transmission line, the multiplexer connects the second coder output to the transmission line for one sample period, then the third, and so on. The result of this process is that samples from each coder are placed on the transmission line one after the other.

If trains A, B, and C were used as an analogy, a car from each of the original trains would be interleaved in the multiplexer to create a longer and faster train. One complete pass of the multiplexer places one car of each train on the track. One pass of the multiplexer constitutes a frame. The first frame would contain the first car from train A, the first car from train B, and the first car from train C.

TSI Functions

The TSI provides time-division switching by delivering any of the 512 time slots received from the data interfaces (DIs) to both dual link interfaces (DLIs) on any of the 512 network time slots.

The TSI connects

- any peripheral time slot from the DI to any DLI time slot
- any peripheral time slot to any other peripheral time slot, or to any local digital service unit (LDSU) time slot
- any DLI time slot to any peripheral time slot, or to any other DLI time slot, or to any LDSU time slot

Each SM contains duplicated TSIs. The active TSI is associated with the active SMP, and the standby TSI is associated with the standby SMP. Each has a copy of the same data, but only the active TSI is selected for data transfer. The interface units contain an input switch that is used to select data, clock, and synchronization from the active TSI only.

DI

The data interface (DI) terminates the peripheral interface data buses (PIDBs). These bidirectional buses carry data time slots between the peripheral units and the MCTSI via the DI. The DI provides the interface for the PCM data, the signaling bits, and the clock and time slot synchronizations between the TSI and the peripheral units.

The interfaces to the periphery are provided in two groups of 256 time slots each. One group interfaces even TSI time slots, the other interfaces odd TSI time slots; a DI is required for each. Each DI has 16 PIDBs connected to the periphery. A PIDB is a 16-bit serial bus with 32 time slots for each direction of transmission.

DLI

The dual link interface (DLI) contains a common clock and control circuit, and two link interface (LI) circuits. Each LI receives time slot data from the TMS by way of the NCT links. The DLI delivers this time slot data to the TSI.

For transmission in the opposite direction, the DLI selects time slot data from the active TSI and delivers them to the TMS via the NCT links.

Control Time Slot

One time slot on each NCT link is permanently connected to the message switch (MSGS) through the TMS. This dedicated time slot is referred to as the control time slot (CTS). The DLI delivers CTS data to the SMP as a 48K-bps serial bit stream. Similarly, the DLI receives a serial bit stream from the SMP and inserts it into the CTS for transmission to the MSGS.

Timing Information

The SM clock is derived from the NCT link data stream by the DLI. This timing information is then delivered by the DLI to the TSI for distribution to the rest of the subunits in the SM.

The active TSI receives timing from the active DLI. To ensure that the TSIs can switch the timing source from one DLI to the other without introducing data errors, the DLIs are cross-coupled so that one DLI is dedicated to this timing source (referred to as the "master"). The phase-locked loop (PLL) circuitry in the DLIs ensures that no rapid shifts in clock phase occur as the master source is moved from one DLI to the other.

CI

The control interface (CI) is located between the SMP and the peripheral units and is used to communicate control information. The CI is connected to the SMP via the subunit interface bus (SIB).

The CI reads and writes registers in the peripheral units by means of control orders (scan and distribute orders) that are sent over the peripheral interface control buses (PICBs). The CI reports a failing control order to the SMP by setting the appropriate bits(s) in the error source register (ESR). A control order may fail due to problems in the peripheral unit or within the CI.

CI Functions

The CI performs the following functions.

- Terminates the PICBs which carry scan and distribute orders from the SMP to peripheral units, and returns their response to the SMP. The "all seems well" bits are used to monitor the result of scan and distribute orders to the periphery. Failures, as indicated by these bits, will result in a maintenance interrupt request to the SMP.
- Monitors communication between the SMP and peripheral units, and reports any errors detected.
- Receives, latches, and reports service requests from the peripheral units to the SMP.

PICBs

A CI provides up to 23 PICBs to the various interface units of the SM. Two CIs can be equipped: CI0 and CI1.

- On CI0, two PICBs are reserved for the LDSU.
- CI1 is optional in a stand-alone office, and has no LDSU PICBs. Six PICBs on CI1 are reserved when a module is configured as a remote switching module (RSM).

Each PICB connects to a service group of a peripheral unit. A PICB consists of five twisted wire pairs which carry clock, data, and controller select information to the interface units and returns data and service requests to the CI. Each PICB can access up to 256 source or destination registers in each interface unit.

SP

The signal processor (SP) checks the time slot signaling bits for changes in their state and reports the changes to the SMP. The SP provides the SMP with access to the signaling and control bits of the time slots transmitted to and received from the interface units. The SP enhances the processing capacity of the SMP by performing the real-time repetitive task of checking the signaling and control bits for changes in their state.

The SP scans for changes in supervision on time slots received from the TSI and notifies the SMP of "hit-timing" supervision changes. The SP stores the signaling bits in an immediate access RAM that the SMP can read at any time.

The SP also provides a means of transmitting to the TSI the state of the seven signaling bits associated with each time slot.

The SPs of the active and standby MCTSI perform co-ordinated, simultaneous three-millisecond hit scan timing. This synchronization ensures that state changes are detected at the same time by both SPs. This is necessary to ensure that a switch to the standby controller will occur without the loss of transient calls.

SMP

The switching module processor (SMP) performs call processing and maintenance functions, controls peripheral units, and communicates with other SMs, the CMP, and the AM.

Each SMP uses a *Motorola* MC68000-family microprocessor. The processing power of this microprocessor is augmented by an SP that performs the real-time iterative task of detecting changes in signaling states.

BTSR

The bootstrapper (BTSR) provides a means of initializing the RAM in the SMP at a 192K/second rate. The BTSR is only one part of the "fast pump" data link. The other part is the pump peripheral controller (PPC), located in the MSGS.

In earlier versions of the MCTSI, the BTSR is a simplex board which interfaces to the update bus, thereby, providing access to either SMP memory.

In later versions of the MCTSI, the BTSR is an integrated part of the SMP and interfaces directly to the internal processor system bus used to access the SMP memory.

The BTSR supplies address, data, and parity bits, and the necessary control signals for a direct memory access (DMA) transfer. The selection of the SMP side (0 or 1) that will be pumped is under software control.

The BTSR receives data over a PIDB in 2K blocks. Each block has a 16-byte header containing 2 bytes of start code, 2 bytes of word count, 1 byte of hashsum check, and 4 bytes of starting address for that block. The remaining 7 bytes in the header contain a fixed data pattern inserted between each byte of valid initialization data.

PI

The packet interface (PI) is only present in SMs that offer integrated services digital network (ISDN) service (i.e., "loaded" SMs). This optional subunit routes packet information between the SMP and the protocol handlers (PHs) of the packet switch unit (PSU). The PI terminates the packet bus (PB) that carries signaling information between the SMP and each service group of the PSU.

The PI provides the following functions:

- Provides the means by which the SMP communicates with the PHs.
- Monitors communication between the SMP and PHs, and reports any errors detected.

The PI has an internal environment similar to that of the PH, capable of distributed processing and local error recovery.

8.2.1.6 MCTSI Interfaces

The MCTSI has the following interfaces:

- Peripheral interface data buses (PIDBs)
- Peripheral interface control buses (PICBs)
- Local digital service unit buses (LDSUBs)
- Packet buses (PBs)
- Network Control and Timing links (NCT).

Figure 8.2-2 shows the MCTSI subunits and interfaces.

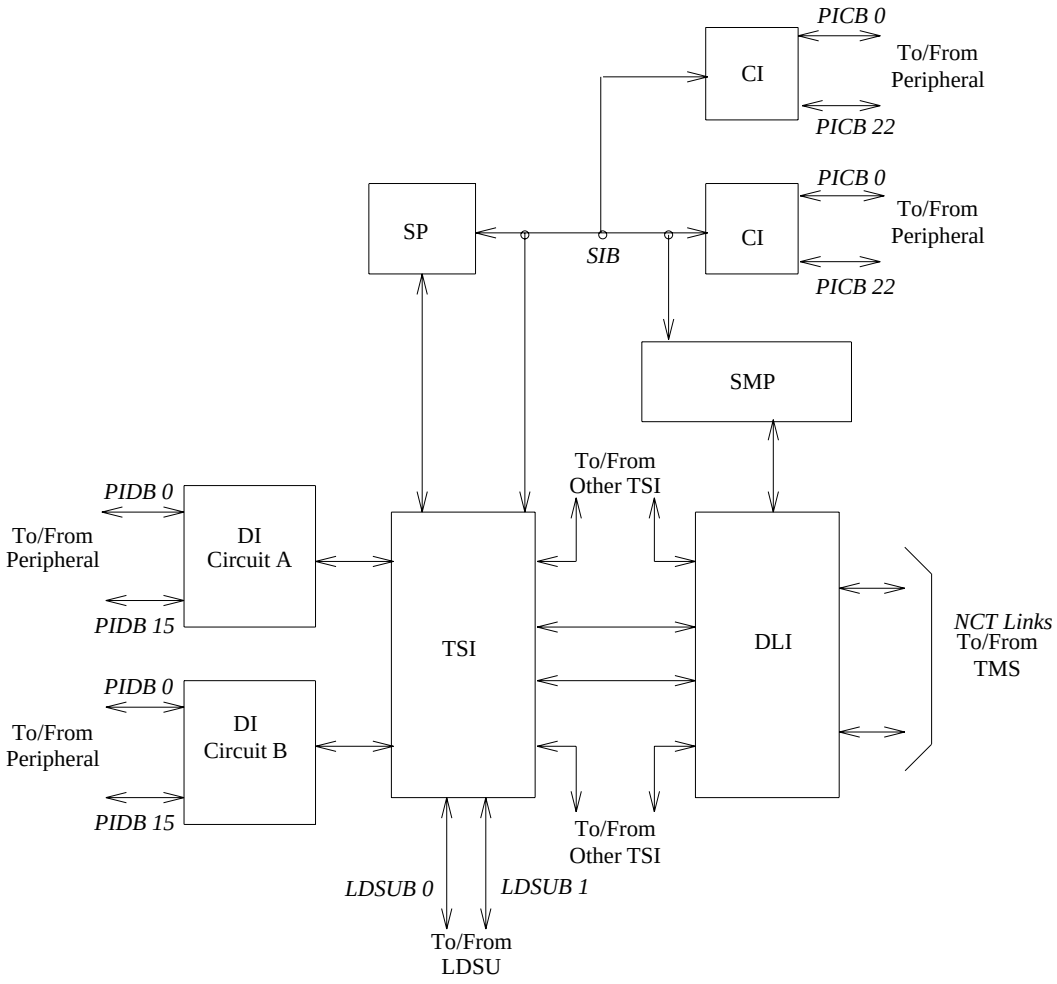


Figure 8.2-2 — MCTSI Subunits and Interfaces

The following describes each interface.

Peripheral Interface Data Buses

A peripheral interface data bus (PIDB) is a duplex bus providing 32 time slots of voice data between the data interface (DI) and a peripheral unit in the SM. (See Figure 8.2-3).

PIDB DATA TIME SLOT

D7	D6	D5	D4	D3	D2	D1	D0	A	B	C	D	E	F	G	P
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bit No.

15 - 8 = PCM Bits
7 - 4 = Signaling Bits
3 = Busy/Idle Bit
2 = TMS Buffer Test Bit
1 = Framing Bit
0 = Parity Bit

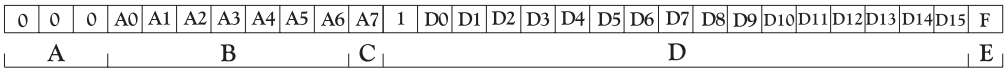
Figure 8.2-3 — PIDB Data Time Slot Configuration

Peripheral Interface Control Buses

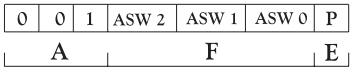
A peripheral interface control bus (PICB) is a duplex bus that carries control messages between the control interface (CI) and an SM's peripheral units. (See Figure 8.2-4).

PICB FORMAT
CONTROL MESSAGES AND REPLIES

DISTRIBUTE ORDER: MODULE CONTROL/TIME SLOT INTERCHANGER TO PERIPHERAL UNIT

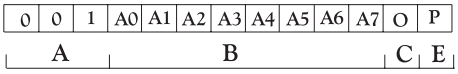


DISTRIBUTE REPLY: FROM MCTSI TO PU

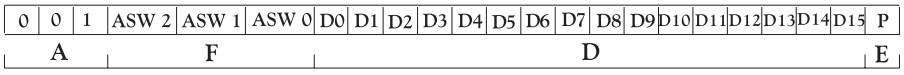


A) 3 Start Bits
B) 8 Address Bits
C) R/W Bit
D) 16 Data Bits
E) Parity Bit
F) 3 All Seems Well Bits

SCAN ORDER: FROM MCTSI TO PU



SCAN REPLY: FROM PU TO MCTSI



ASW CODE DEFINITIONS			
ASW 2	**ASW 1**	**ASW 0**	**Description**
0	0	0	Peripheral Detected Bad Address
0	0	1	Parity Error on Message from Controller
0	1	0	All Seems Well
0	1	1	Bad Start Code on Message from Controller

Figure 8.2-4 — PICB Bit Configuration and All Seems Well (ASW) Code Definitions

Local Digital Service Unit Buses

A local digital service unit bus (LDSUB) carries data channels from the local digital service unit (LDSU) to the time slot interchange (TSI).

Packet Bus

The packet bus (PB) carries signaling information between the SMP and the packet switch unit (PSU).

Network Control and Timing Links

The NCT links are internal fiber optic links that connect the SMs with the communications module (CM) to provide time slot paths for network connections, carry control messages (time slots) to the modules, and distribute timing to the modules. (See Figure 8.2-5.)

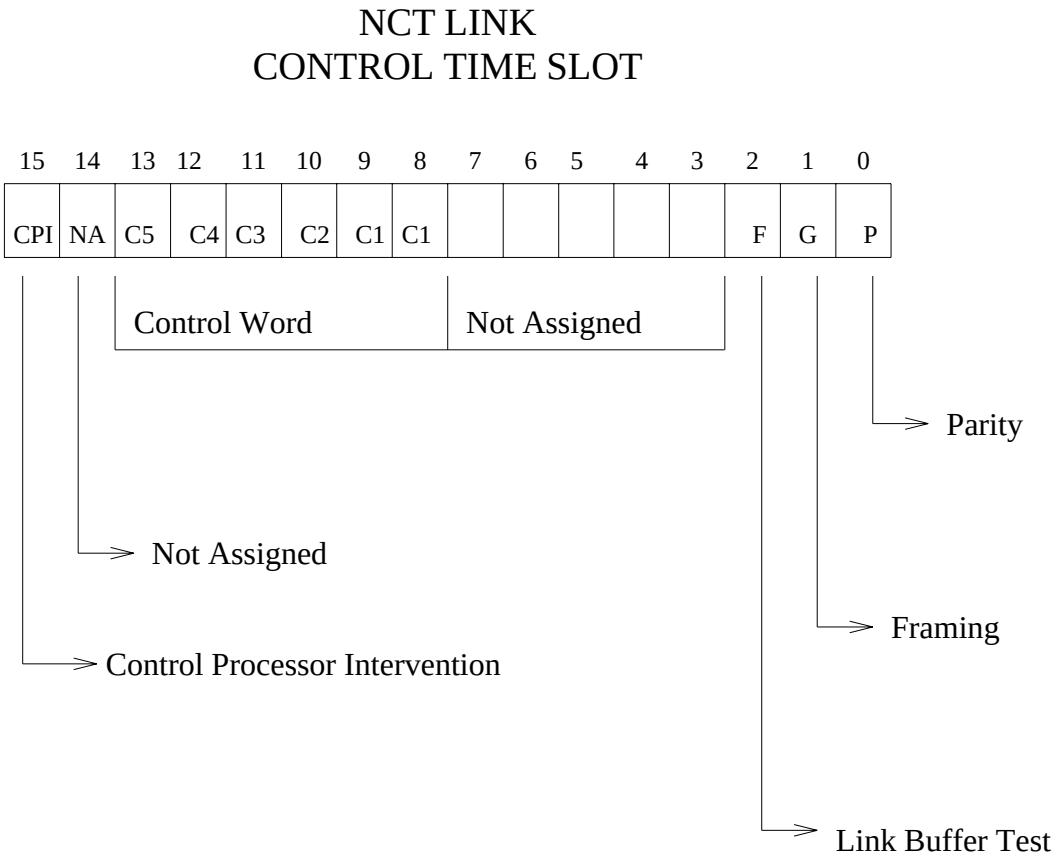


Figure 8.2-5 — NCT Link — Control Time Slot Content

8.2.1.7 ISDN Peripheral Units

There are two types of switching within the ISDN environment circuit switching and packet switching.

- With circuit switching, each call uses its own path. Each path is dedicated to one user.
- With packet switching, one path may be used simultaneously by many users. The information on the path is split up into small packets which are then switched independently.

Two different types of peripheral units are used to provide ISDN service:

- Integrated services line unit (ISLU)

  The ISLU provides a physical entrance into the *5ESS*® switch, connecting the basic rate interface (BRI) to the switch.

- Packet switch unit (PSU)

  The PSU switches packet data and signaling information.

8.2.2 Categories Of Interrupts

8.2.2.1 Interrupt Categories

The following interrupt categories are reviewed in this section:

- Service requests
- Peripheral interface control bus (PICB) circuitry related errors
- Control interface (CI) errors
- Time slot interchange (TSI) errors
- Signal processor (SP) errors
- Dual link interface (DLI) errors
- Switching module processor (SMP) errors.

8.2.2.2 Service Requests

Interrupts from a peripheral unit to the CI are referred to as service requests. A peripheral unit requests "service" by causing the service request lead to be activated when any bit in its interrupt register is set.

Every 40 milliseconds, the peripheral control (PC) foreground function `CIscan()` scans the CI's interrupt source register (`ISR`) for service requests from the peripherals.

Each PICB has an interrupt lead that connects the peripheral unit to the CI remote `ISR`. These sources are summarized in an error source register (`ESR`). They are normally masked out and not reported to the SMP.

A service request can be either an operational or a maintenance request. An operational service request occurs when a peripheral unit needs service for call processing. A maintenance service request occurs when a peripheral unit detects an internal abnormal condition.

8.2.2.3 PICB Circuitry Related Errors

PICB circuitry-related interrupts are due to faulty PICB circuitry between the CI and a peripheral unit. PICB-related errors could be attributed to:

- Timeout (no reply to an order)
- Bad start code on reply
- Bad parity on reply
- Peripheral unit detected a bad address
- Peripheral unit detected a bad start code
- Peripheral unit detected bad parity.

8.2.2.4 Control Interface Errors

Control interface (CI) errors are related to the operations of the CI itself. They indicate a CI or SMP fault.

Peripheral Unit Error Sources

There are six possible error sources for peripheral unit problems:

- Peripheral unit detected a bad start code

- Peripheral unit detected bad parity
- Peripheral unit detected a bad address
- CI detected a bad start code
- CI detected bad parity
- CI timed out waiting for a reply.

CI Error Sources

There are five possible CI error sources.

- Multiplexer selection error

  Multiplexers are used to communicate with the peripheral units. Up to 23 PICBs can be used to connect the CI with the peripherals; for each PICB, a clock and data out (or data in) multiplexer must be selected.

  "1-out-of-N" checkers reported an error indicating a problem in the multiplexer circuitry. This error occurs if no PICB or more than one PICB becomes active during a scan or distribute operation.

- Improper address selection

  An attempt was made to read or write a nonexistent register, or to read or write a register that does not have that capability.

- Address parity error

  During a read or write operation to the CI, the register address presented to the CI by the SMP had bad parity.

- Data parity error

  During a write operation to the CI, the data presented to the CI by the SMP had bad parity.

- PICB address error

  An attempt was made by the SMP to write the address register with an invalid PICB address.

8.2.2.5 Time Slot Interchange Errors

To perform the time slot interchanging function in the TSI, several control RAMs (including the alternate data RAM) are implemented. Byte parities are checked and are potential error sources.

Address and data bus parity errors, as well as DLI interface parity errors, are also error sources. Control and data errors within the TSI are additional sources for errors.

8.2.2.6 Signal Processor Errors

The signal processor (SP) receives the control and signaling bits for 512 time slots from the periphery. The SP monitors these bits for a change of state. If a change of state occurs, the control and signaling bits are loaded into an SMP-readable first-in first-out (FIFO) memory. The SP also sends control and signaling bits for 512 time slots towards the periphery.

There are two possible sources of errors in the SP:

- FIFO full

The FIFO memory is monitored for possible overflow and a bit is set in the error source register (ESR) if the memory becomes full.

- Parity errors

  Parity is checked at various points throughout the SP.

8.2.2.7 Dual Link Interface Errors

The DLI provides the interface between the SM and the CM by means of the NCT links. The DLI also receives timing from the CM and distributes it throughout the SM.

The following are possible DLI error sources:

- Link errors
  - — Parity errors on time slot information (both transmit and receive)
  - — Framing errors (G bit)
- Phase lock loop slip (checks for synchronization to the CM)
- Module controller interface errors - parity errors in the SMP interface
- Buffer errors (buffer test bit - F bit)
- Clock errors
- TSI interface errors - parity errors on data received and transmitted to/from the TSI

8.2.2.8 Switching Module Processor Errors

The SMP error source registers are organized into two distinct groups those that report hardware errors and those that report software errors. Of the numerous interrupt sources, there are only two that are unmaskable sanity timer reset and central processor intervention (CPI).

- Software error sources
  - — Write protect violations
  - — Invalid I/O operations (write of a read only point, read of a write only point, accessing unequipped points, etc.)
  - — I/O timer and sequence violations
  - — Stack protection violations
- Hardware error sources
  - — Parity errors at the interfaces
  - — Subunit enable mismatches (any subunit on the subunit interface bus [SIB])
  - — I/O data bus parity errors
  - — Ready timeouts
  - — DLI interface errors
- Memory type errors - double bit error, refresh failure, multiple board select, cache errors, correctable bit errors
- Sanity timer reset

An interrupt will be triggered if the sanity timer is allowed to time out. Similarly, an interrupt will be triggered if the sanity timer is being restarted too frequently.

- Central processor intervention (CPI)

  The CPI circuitry provides a communication path from the AM to each module controller in the SM. If an initialization of the SM is required, the CPI circuitry furnishes this reset as an interrupt source prior to initialization.

- Mate request

  A mate request does not indicate a system error. It is a built-in mechanism in which the active processor wakes up the standby processor to perform some function.

- Microprocessor errors - parity in the SMP
  - — Addressing errors
  - — Divide by zero
  - — Illegal instruction
  - — Other traps

8.2.3 Interrupt Levels

8.2.3.1 Interrupt Priority Levels

Interrupts in the microprocessor can occur at one of seven levels of priority, level 1 represents the lowest priority and level 7 the highest.

The microprocessor compares the level of the interrupt request with an interrupt mask in the status register to determine if the interrupt should be processed. An interrupt mask of 0 permits all interrupts and an interrupt mask of 7 allows no interrupts (except level 7). The interrupt will be processed only if it has a higher priority than the mask level set in the status register.

8.2.3.2 SM Interrupt Levels

There are three levels of SM maintenance interrupts.

Level 7 All hardware and software errors detected within the module controller.

Level 5 Generated by the central processor intervention (CPI) circuitry and causes a switch of the DLIs (if directed by the AM).

Level 4 Generated by the three programmable interrupt controllers (`PIC A`, `PIC B`, `PIC C`).

The `PIC B` and `PIC C` registers summarize subunit and peripheral hardware error sources (maintenance interrupts). The `PIC A` register reflects operational interrupts only and will not be addressed here.

8.2.3.3 Vectored and Autovectored Interrupts

The *Motorola* MC68XXX processor can employ both vectored and autovectored interrupts.

- With vectored interrupts, the interrupting device sends a vector number which is used to index a table that contains the address of the interrupt routine.

- When autovectoring is used, the device which generated the interrupt initiates an interrupt request. The SMP examines the priority status of the interrupt to determine which vector number will be used.

Level 5 and level 7 interrupts implement the autovector mode to obtain the vectors required to process these interrupts. A level 4 interrupt does not use the autovector mode. The interrupt is generated by one of the programmable interrupt controllers (PICs), and the exception vector is read directly from the controller.

8.2.4 MCTU Interrupt Registers

Three pins on the *Motorola* MC68XXX microprocessor chip are used by external devices to cause an interrupt. These pins are called IPL0, IPL1, and IPL2 (IPL stands for interrupt priority level). These three inputs form a 3-bit code used to request interrupts.

The setting of the IPLs for interrupt levels 4, 5, and 7 are shown here:

Interrupt Level	IPL2	IPL1	IPL0
7	1	1	1
5	1	0	1
4	1	0	0

Figure 8.2-6 shows the hierarchy of error registers that are used for interrupts.

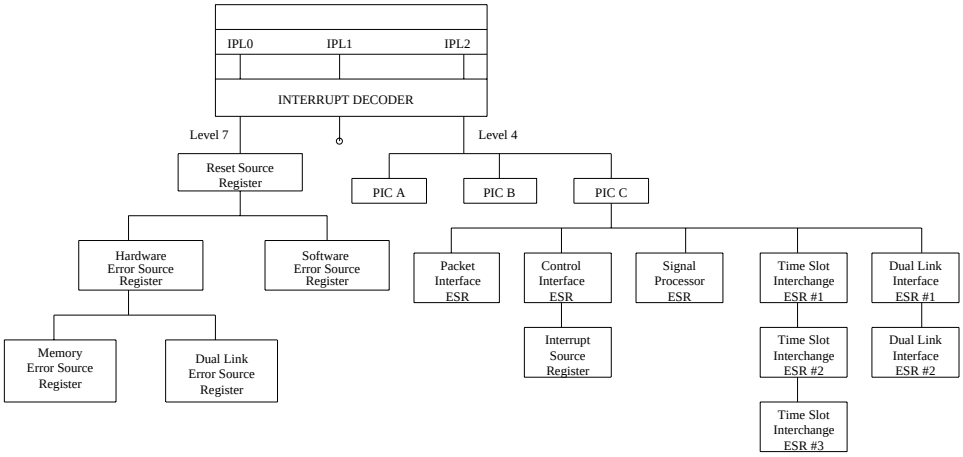


Figure 8.2-6 — MCTU Interrupt Registers

8.2.5 Module Controller Interrupts (Level 7)

8.2.5.1 Module Controller Interrupt Registers

The module controller interrupt registers record all hardware and software errors that can occur within the module controller (i.e., level 7 interrupts). See Figure 8.2-7.

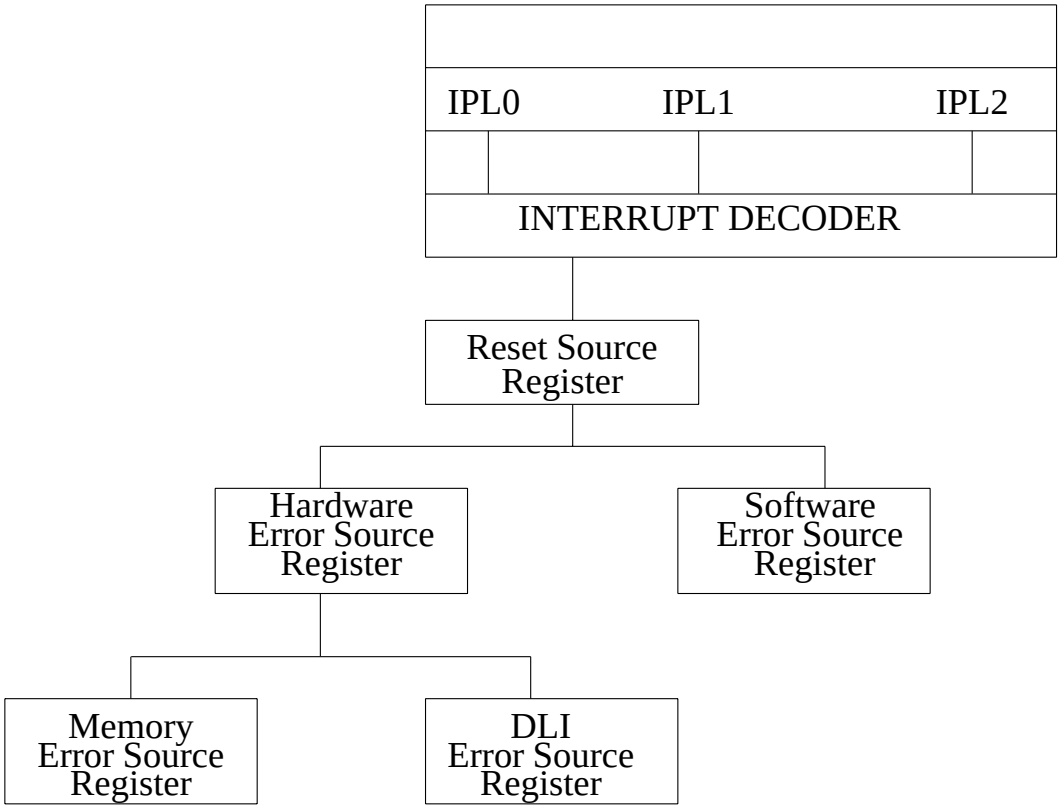


Figure 8.2-7 — Module Controller (Level 7) Interrupt Registers

The following list indicates which table describes each of these registers.

Interrupt Register	Table
Reset Source Register	Table 8.2-1
Software Error Source Register	Table 8.2-2
Hardware Error Source Register	Table 8.2-3
Memory Error Source Register	Table 8.2-4
DLI Error Source Register	Table 8.2-5

8.2.5.2 Reset Source Register

Table 8.2-1 describes the bit layout of the reset source register.

Table 8.2-1 — Reset Source Register

Bit	Function
7	Sanity timer error
6	
5	Reset from CPI circuit
4	
3	Reset request from the mate controller (maskable only if the receiving processor is active)
2	Hardware error
1	Correctable memory error
0	Software error

This section describes the function of each bit in the reset source register.

Bit 7 Sanity timer error. The sanity timer, which can run for a maximum of 699 ms, must be periodically cleared by the SAN_CLR strobe point. If 699 ms expire since the last time the timer was cleared, bit 7 is set in the reset source register. This error is not maskable.

A check is also made to ensure that the sanity timer is not cleared until at least 233 ms have elapsed since the last reset. If this check fails, bit 7 of the reset source register is set.

Bit 6

Bit 5 CPI force reset. Stops the processor and forces it to a base address for restarting. A CPI reset may be requested manually with the ORD:CPI input message. This interrupt source is not maskable.

The CPI circuitry provides a communication path from the AM to each controller in the SM. The gate array in the CPI circuitry recognizes five types of messages, each of which can be requested manually:

- **Diagnostic message** verifies the performance of the CPI gate array and associated circuitry.
- **Force reset message** resets certain hardware in the SM and causes a full initialization.
- **Force active message** forces one controller active and the other inactive.
- **Inhibit/allow sanity timer** inhibits or allows the timer from causing an interrupt due to a timeout.
- **CP interrupt** can take one of three actions.
  - — FRC: Forces the specified MCTSI to the active state. This will cause a single process purge (SPP) if a processor switch is involved.
  - — GRSW: Generic retrofit switch. Forces the specified MCTSIs to the active state and causes a full initialization. Used only during a retrofit.

— SW: Switches the MCTSI to a specified side and forces that side active.

Bit 4

Bit 3 Set by a "reset mate controller" request from the mate controller. If this controller is not active, the setting of this bit will generate an interrupt and the processor will reset this controller.

If this controller is active, no interrupt is generated. However, bit 3 is set to indicate that the attempt was made.

Bit 2 Set when a hardware related interrupt is generated from the hardware error source register. See Table 8.2-3.

Bit 1 A correctable memory error was detected by the memory controller circuitry. This error is masked and should never be seen.

Bit 0 Set when a software related interrupt is generated from the software error source register. See Table 8.2-2.

8.2.5.3 Software Error Source Register

Table 8.2-2 defines the bit layout of the software error source register. The errors recorded in this register may be caused by hardware or software errors, though they are predominantly software related.

When an interrupt is generated from this register, bit 0 (software error) is set in the reset source register (see Table 8.2-1.)

Table 8.2-2 — Software Error Source Register

Bit	Function
7,6	Unused
5	Stack protection violation
4	I/O double UNLOCK error
3	I/O double LOCK error
2	I/O timer timeout
	Length of I/O timer is 114 µsec
1	Illegal I/O operation
0	Write protection violation

This section describes the software error source register bits.

Bit 5 User or system stack write protect violation. The stack currently in use is specified by the value written in the stack window register.

Any attempt to write to a stack space not specified by the stack window register will be aborted, and a stack protect error will be generated.

Bits 4, 3, 2 These three bits are used together. The I/O timer provides a degree of protection to the I/O registers. Many of the I/O registers require the I/O timer to be unlocked (started) to access them. This is accomplished by a write to the UNLOCK strobe point. After the I/O timer has been unlocked, the requested operation may be performed.

The I/O timer must be locked (stopped) again after the operation by a write to the I/O timer LOCK strobe point. If the timer is not locked within 114 µsec of its unlocking, the timer will time out and an I/O timer timeout error will be generated. If the correct sequence is not followed, a double lock or double unlock error will be generated.

Bit 1 One of the following illegal I/O operations was attempted:

- Reading a write only point
- Writing a read only point
- Accessing an unequipped point
- Accessing a point without I/O timer
- Accessing an 8-bit I/O as a 16-bit address (I/O bus is 8 bits, no parity)
- Accessing an 8-bit I/O on an odd address (all I/O addresses are even).

Bit 0 Write protection violation. Text in memory is protected in 4K blocks. Data space (including EPROM, I/O, and static RAM) is protected in 1K blocks.

Whenever a write is attempted to an address that has the write protect bit associated with that block set, the write is aborted and a write protect violation error is generated.

8.2.5.4 Hardware Error Source Register

Table 8.2-3 defines the bit layout of the hardware error source register. This register summarizes hardware related errors detected in the module controller, though software errors can lead to some of these errors (such as data parity errors or memory system errors caused by an out-of-range read).

When an interrupt is generated from this register, bit 2 (hardware error) is set in the reset source register. (See Table 8.2-1.)

Table 8.2-3 — Hardware Error Source Register

Bit	Function
7	Data parity error (read or write)
6	Address parity error
5	Ready timeout caused by the local controller
4	Memory system error
3	Ready timeout caused by an access of the mate controller
2	I/O data bus parity error
1	Subunit enable mismatch
0	DLI interface error (SDLC or DLI parity, or invalid DLI access)

This section describes the hardware error source register bits.

Bit 7 System data bus detected error. The system bus refers to the control, address, and data buses from the module processor.

Bit 6 Address parity error.

Bit 5 Ready timeout, local controller.

Access time to the system bus may be extended for such things as I/O access, mate read operations, refresh clashes, correct-on-the-fly mode, etc. The ready timeout circuitry protects the processor from indefinite time extensions for these types of functions. The ready timeout circuitry will terminate a bus cycle after 56 µsec.

Bit 4 Reports on faults detected by the memory control board. These faults are recorded in the memory error source register. See Table 8.2-4.

Bit 3 Ready timeout caused by an access of the mate controller.

Bit 2 Parity error on the I/O data bus (interface to subunits). Parity is calculated over 16 data bits and 6 address bits.

Bit 1 Each subunit receives its own subunit select signal. This bit is set when no subunit was selected, or when more than one subunit was selected.

Bit 0 Module processor DLI interface error. This bit summarizes the dual link interface (DLI) error source register. See Table 8.2-5.

8.2.5.5 Memory Error Source Register

Table 8.2-4 defines the bit layout of the memory error source register. This register records faults detected by the memory control board.

When an interrupt is generated from this register, bit 4 (memory system error) is set in the hardware error source register. (See Table 8.2-3.)

Table 8.2-4 — Memory Error Source Register

Bit	Function
7	Cache error
6	Correctable bit error
5	More than one board was selected, or no board was selected
4	Double bit error detected (noncorrectable)
3	Bad parity on system address bus
2	Bad address parity returned from memory boards
1	Unused
0	Dynamic refresh failure

8.2.5.6 DLI Error Source Register

Table 8.2-5 defines the bit layout of the DLI error source register. This register records peripheral errors related to a DLI or a synchronous data link controller (SDLC).

When an interrupt is generated from this register, bit 0 (DLI interface error) is set in the hardware error source register. (See Table 8.2-3.)

Table 8.2-5 — DLI Error Source Register

Bit	Function
7	Unused
6	Invalid DLI switch setting
5	Invalid access to DLI 1
4	Invalid access to DLI 0 (Either the processor was not active, or the DLI SW register was not set correctly, or both.)
3	Data parity error on reading DLI 1
2	Data parity error on reading DLI 0
1	Parity error on SDLC B
0	Parity error on SDLC A

8.2.6 Subunit and Peripheral Hardware Interrupts (Level 4)

8.2.6.1 Subunit and Peripheral Hardware Interrupt Registers

The subunit and peripheral hardware interrupt registers record errors that can occur in the subunits or the peripheral hardware (Level 4 interrupts). See Figure 8.2-8.

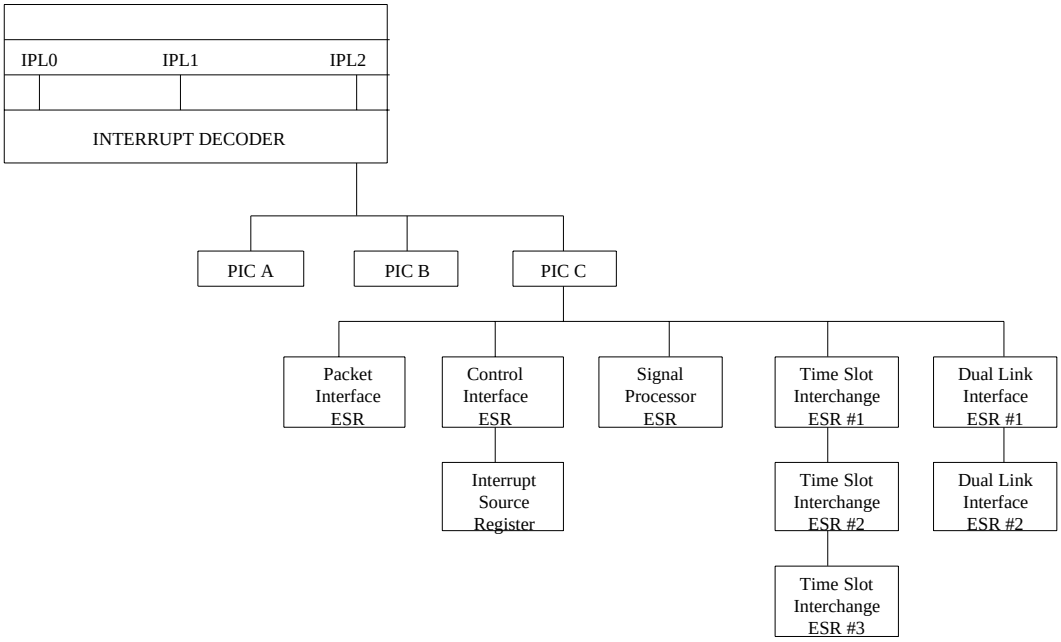


Figure 8.2-8 — Subunit and Peripheral Hardware (Level 4) Interrupt Registers

The CI, TSI, SP, and DLI have interrupt registers that reflect the state of interrupts generated in each unit. They record errors in an internal error source register (ESR). A bit set in the ESR results in an interrupt from the unit to the SMP. The SMP will then

read the interrupting unit's ESR for more detail. This list indicates which table describes each of these registers.

Interrupt Register	Table Number
PIC A (operational interrupts)	not discussed
PIC B	8.2-6
PIC C	8.2-7
PI Error Source	8.2-8
CI Error Source	8.2-9
Interrupt Source	not discussed
SP Error Source	8.2-10
TSI Error Source 1	8.2-11
TSI Error Source 2	8.2-12
TSI Error Source 3	8.2-13
DLI Error Source 1	8.2-14
DLI Error Source 2	8.2-15

The PIC A, PIC B, and PIC C registers are shown in the ROP output as LOW, MED, and HIGH, respectively.

8.2.6.2 PIC B Register

Table 8.2-6 describes the bit layout of the PIC B register.

Table 8.2-6 — PIC B Register

Bit	Function
7	Mate maintenance interrupt
6	Mate processor error
5,4	Unused
3	CPI error
2	Correctable bit error
1,0	Unused

This section describes the PIC B register bits.

Bit 7 Any unmasked maintenance interrupt occurring on a standby processor will set this bit in the active processor. (Level 4 interrupt in mate.)

Bit 6 The mate reset source register latched a processor error. (Level 7 interrupt in mate.)

Bits 5, 4 Unused

Bit 3 Diagnostics detected a fault with the CPI gate array.

Bit 2 Correctable bit error was detected by the memory controller.

Bits 1, 0 Unused

8.2.6.3 PIC C Register

The PIC C register summarizes faults originating from the PI, CI, SP, TSI, and DLI subunits.

Table 8.2-7 defines the bit layout of the PIC C register.

Table 8.2-7 — PIC C Register

Bit	Function
7	Unused
6	PI errors
5	CI 1 errors
4	CI 0 errors
3	SP errors
2	TSI errors
1	DLI 1 errors
0	DLI 0 errors

This section describes the PIC C register bits.

Bit 7 Unused

Bit 6 The packet switch unit (PSU) interfaces with the module processor via the packet interface (PI) and the subunit interface bus (SIB). The port packet processor (an *Intel*[2] 80186 processor) controls the interface to the module processor and to the PSU over the packet buses (PBs).

Bit 6 summarizes errors detected in the PI. See Table 8.2-8.

Bits 5, 4 The CI connects to the module processor via the subunit interface bus (SIB). Bit 4 summarizes errors detected in CI 0, and bit 5 summarizes errors detected in CI 1. See Table 8.2-9.

Bit 3 Bit 3 summarizes errors detected in the SP. See Table 8.2-10.

Bit 2 Bit 2 summarizes errors detected in the TSI. The TSI has three error source registers. See Table 8.2-11, Table 8.2-12, and Table 8.2-13.

Bits 1, 0 Bit 0 summarizes errors detected in DLI 0, and bit 1 summarizes errors in DLI 1. Each DLI has two error source registers. See Tables 8.2-14 and 8.2-15.

8.2.6.4 PI Error Source Register

Table 8.2-8 defines the bit layout of the PI error source register. This register records errors detected in the PI.

When an interrupt is generated from this register, bit 6 (PI errors) is set in the PIC C register. (See Table 8.2-7.)

2. Registered trademark of Intel Corporation.

Table 8.2-8 — PI Error Source Register

Bit	Function
4	Hamming bit check inhibit
3	Write protect error
2	T1 timer output error
1	Non-correctable memory error
0	SIB parity error

This section describes the PI error source register bits.

Bit 4 The hamming code generator/checker is part of the dual access RAM (DARAM) and is inhibited only during diagnostics.

Bit 3 A write attempt was made to a write protected address spectrum.

Bit 2 Timeout of this software controller timer indicates loss of sanity.

Bit 1 Hamming detected uncorrectable double error in memory.

Bit 0 Parity error detected on the SIB.

8.2.6.5 CI Error Source Register

Table 8.2-9 defines the bit layout of the CI error source register. This register records errors related to the CI.

When an interrupt is generated from this register in CI 0, bit 4 (CI 0 errors) is set in the PIC C register; when an interrupt is generated from this register in CI 1, bit 5 (CI 1 errors) is set in the PIC C register. (See Table 8.2-7.)

Table 8.2-9 — CI Error Source Register

Bit	Function
15	Remote source interrupt
14-11	Unused
10	Peripheral detected bad start code
9	Peripheral detected bad address
8	Peripheral detected bad parity
7	Bad parity error
6	Bad start error
5	Timeout error
4	PICB address error
3	Data parity error
2	Address parity error
1	Improper address selection (invalid CI register)
0	Multiplexer selection error

This section describes the CI error source register bits.

Bit 15 An active service request was received from a peripheral unit. To

determine which peripheral is requesting service, a read of the remote interrupt source register is necessary because the remote source interrupt is normally masked. The CI interrupt source register is checked by the CI every 40 ms.

Service requests are maskable on a per PICB basis, accomplished through the remote interrupt inhibit register. If any bit corresponding to a PICB address is a logical 1 in this register, service requests from the peripheral at that address will not be reported to the error source register.

Bits 14–11 Unused

Bit 10 A peripheral detected a bad start code in a message from the CI.

Bit 9 A peripheral detected an addressing error in its own circuitry while attempting to do a scan or distribute requested by the CI.

Bit 8 A peripheral detected bad parity in a message from the CI.

Bit 7 A peripheral reply message had bad parity.

Bit 6 A peripheral reply message had a bad start code.

Bit 5 A peripheral failed to reply to a CI message before the timeout counter in the receive sequencer circuitry reached its terminal count.

Bit 4 An attempt was made by the SMP to write the address register with an invalid PICB address.

Bit 3 During a write operation to a peripheral unit, the data presented to the CI by the SMP had bad parity.

Bit 2 During a read or write operation to the CI, the register address presented to the CI by the SMP had bad parity.

Bit 1 An attempt was made to read or write to a nonexistent register, or to read or write to a register that does not have that capability. This includes an attempted read of the reset register.

Bit 0 The "1-out-of-N" checkers reported an error in the selection of output clock or output data links. This indicates a problem in the output clock or output data multiplexer circuitry. This error occurs if no link or more than one link becomes active during a scan or distribute operation.

8.2.6.6 SP Error Source Register

Table 8.2-10 defines the bit layout of the SP error source register. This register records errors related to the SP.

When an interrupt is generated from this register, bit 3 (SP errors) is set in the `PIC C` register. (See Table 8.2-7.)

Table 8.2-10 — SP Error Source Register

Bit	Function
7	FIFO full
6	Parity error input data buffer - ignore RAM
5	Parity error change/hit timing circuit
4	Parity error received TSI data
3	Parity error internal data bus
2	Parity error module processor data bus
1	Parity error module processor address bus
0	Parity error M RAM

This section describes the SP error source register bits.

Bit 7 The SP performs status change calculations on all time slot control and signaling bits from the TSI. Changes are stored in the FIFO memory where the SMP has access to them.

By design, the FIFO memory should never fill up. If it does, the memory could be faulty. However, it is more likely that a peripheral unit is causing a massive number of originations or disconnects, etc. In such cases, the integrity of the periphery is suspect.

Bits 6-0 These bits report bad parity detected at certain points within the SP.

8.2.6.7 TSI Error Source Register 1

Time slot interchange (TSI) error source register 1 is a read-only 16-bit register. Any write to this register will clear it. A bit set via error reporting circuitry will cause the appropriate TSI interrupt level to be activated until the error condition is resolved and the register is cleared.

Table 8.2-11 defines the bit layout of TSI error source register 1. This register records errors related to the TSI.

When an interrupt is generated from this register, bit 2 (TSI errors) is set in the PIC C register. (See Table 8.2-7.)

Table 8.2-11 — TSI Error Source Register 1

Bit	Function	Bit	Function
15	SMP data bus write high	7	Time slot counter
14	SMP data bus write low	6	I/O address bus
13	SMP data bus read high	5	Read modify write high
12	SMP data bus read low	4	Read modify write low
11	DLI B output	3	Control RAM B high
10	DLI A output	2	Control RAM B low
9	DLI B input	1	Control RAM A high
8	DLI A input	0	Control RAM A low

This section describes the TSI error source register 1 bits.

Bit 15 Parity error over the high byte of the subunit bus while the SMP was attempting to transmit data toward the TSI. The TSI aborts the operation.

Bit 14 Parity error over the low byte of the subunit bus while the SMP was attempting to transmit data toward the TSI. The TSI aborts the operation.

Bit 13 Parity error over the high byte of the internal subunit bus while attempting to transmit data toward the SMP.

Bit 12 Parity error over the low byte of the internal subunit bus while attempting to transmit data toward the SMP.

Bit 11 Parity error on the nibble bus at the output for DLI odd time slot data.

Bit 10 Parity error on the nibble bus at the output for the DLI even time slot data.

Bit 9 Parity error on the nibble bus at the input from the DLI odd time slot data.

Bit 8 Parity error on the nibble bus at the input from the DLI even time slot data.

Bit 7 Parity error in one of the time slot counters. Both the control time slot counter and the TSI time slot counter can flag the error.

Bit 6 Parity error on the subunit address bus. If this error occurs on an SMP initiated write cycle, the TSI will abort the operation.

Bit 5 Parity error in the read modify write circuit for the high byte of control RAM data.

Bit 4 Parity error in the read modify write circuit for the low byte of control RAM data.

Bit 3 Parity error in the read logic for the high byte of control RAM B.

Bit 2 Parity error in the read logic for the low byte of control RAM B.

Bit 1 Parity error in the read logic for the high byte of control RAM A.

Bit 0 Parity error in the read logic for the low byte of control RAM A.

8.2.6.8 TSI Error Source Register 2

Errors detected in the TSI error source register 2 are associated with the data interfaces, digital service unit (DSU) interface, SP interface, control RAM C and alternate data RAM, attenuation ROM, and control RAM E (even locations only).

This a 16-bit, read-only register. Any write to this register causes it to clear. Setting any bit in this register activates the TSI interrupt lead until the error condition is resolved and the register is cleared.

Table 8.2-12 defines the bit layout of TSI error source register 2.

When an interrupt is generated from this register, bit 2 (TSI errors) is set in the PIC C register. (See Table 8.2-7.)

Table 8.2-12 — TSI Error Source Register 2

Bit	Function	Bit	Function
15	Control address	7	DI B output
14	Attenuator error	6	DI A output
13	LDSU B	5	Control RAM E even
12	LDSU A	4	Unused
11	Control RAM C high (odd)	3	TSI AUTISSing (no interrupt generated)
10	Control RAM C high (even)	2	SP input
9	Latched time slot count	1	Alternate data odd
8	Control RAM C low	0	Alternate data even

This section describes the TSI error source register 2 bits.

Bit 15 Parity error on the TSI control RAM address bus.

Bit 14 Parity error while comparing input data, address, and output data in the digital attenuator circuit.

Bit 13 Parity error on the data returning to the TSI from SG 1 of the LDSU. This error usually implicates the LDSU.

Bit 12 Parity error on the data returning to the TSI from SG 0 of the LDSU. This error usually implicates the LDSU.

Bit 11 Parity error in the read logic for the odd locations of the high byte of control RAM C.

Bit 10 Parity error in the read logic for the even locations of the high byte of control RAM C.

Bit 9 Parity error in the latched time slot count circuit. This is a serious error because this count is used as both address and data in several areas of the alternate data RAM.

Bit 8 Parity error in the read logic for the low byte of control RAM C.

Bit 7 Parity error on the nibble bus going out to the odd data interface (DI) board.

Bit 6 Parity error on the nibble bus going out to the even DI board.

Bit 5 Parity error in the read logic for the even locations of control RAM E.

Bit 3 This TSI has invoked automatic time slot switching (AUTISS) on at least one time slot for 256 consecutive frames. This bit is used as a maintenance indicator only and does not generate an interrupt.

Bit 2 Parity error on the nibble bus from the SP.

Bit 1 Parity error on the odd time slot nibble bus from the alternate data RAM at the input to the data selector (DS).

Bit 0 Parity error on the even time slot nibble bus from the alternate data RAM at the input to the DS.

8.2.6.9 TSI Error Source Register 3

TSI error source register 3 is a 16-bit register, though currently only 10 of the bits are used.

- Bits 0-3 indicate a specific error condition.
- Bits 8-13 are used as a pointer to the DI error buffer.
- Bits 4-7 are unused.

Table 8.2-13 defines the bit layout of TSI error source register 3.

When an interrupt is generated from this register, bit 2 (TSI errors) is set in the PIC C register. (See Table 8.2-7.)

Table 8.2-13 — TSI Error Source Register 3

Bit	Function
3	Control RAM D odd
2	Control RAM D even
1	Control RAM E odd
0	Data Selector (DS)

This section describes the TSI error source register 3 bits.

Bit 3 Parity error in the read logic for the odd locations of control RAM D.

Bit 2 Parity error in the read logic for the even locations of control RAM D.

Bit 1 Parity error in the read logic for the odd locations of control RAM E.

Bit 0 Parity failure in the data selector (DS) circuit.

8.2.6.10 DLI Error Source Register 1

Table 8.2-14 defines the bit layout of DLI error source register 1.

When an interrupt is generated from this register in DLI 0, bit 0 (DLI 0 errors) is set in the PIC C register; when an interrupt is generated from this register in DLI 1, bit 1 (DLI 1 errors) is set in the PIC C register. (See Table 8.2-7.)

Table 8.2-14 — DLI Error Source Register 1

Bit	Function
7	Phase lock loop slip
6	B link in frame
5	B link transmit parity error
4	B link receive parity error
3	Parity error on module processor interface
2	A link in frame
1	A link transmit parity error
0	A link receive parity error

This section describes the DLI error source register 1 bits.

Bit 7 Slip detected between the two 32 MHz clocks in the DLI.

There are two clocks in the DLI. One is driven by the data transitions at the input to the receive side, and the other resides in the digital phase lock loop area. There is a 90 degree phase shift between the two clocks, so that a set of data written into the "elastic buffer" is not read out until 30-plus µsec later. The slip detector checks that the second clock is exactly 90 degrees shifted from the first clock.

Bits 6, 5, 4 These bits are related and pertain to the odd (i.e., B) NCT link.

Bit 3 The SMP sends orders to the DLI. Since the operation may be a read or write of some register, the address and data bits accompany the operation list. Bit 3 is set when bad parity is detected over this data.

Bits 2, 1, 0 These bits are related and pertain to the even (i.e., A) NCT link.

8.2.6.11 DLI Error Source Register 2

Table 8.2-15 defines the bit layout of DLI error source register 2.

When an interrupt is generated from this register in DLI 0, bit 0 (DLI 0 errors) is set in the PIC C register; when an interrupt is generated from this register in DLI 1, bit 1 (DLI 1 errors) is set in the PIC C register. (See Table 8.2-7.)

Table 8.2-15 — DLI Error Source Register 2

Bit	Function
7	Receive message time slot parity error link B
6	Receive message time slot parity error link A
5	Transmit message time slot parity error link B
4	Transmit message time slot parity error link A
3	Buffer error link B
2	Buffer error link A
1	B link clock circuit error
0	A link clock circuit error

This section describes the DLI error source register 2 bits.

Bits 7, 6, 5, 4 Parity error detected on a control time slot.

Bits 3, 2 Buffer errors, NCT link A or B.

Bits 1, 0 Clock circuit error. The clock divides down, providing all of the required frequencies throughout the DLI. It finally divides down to an 8 KHz sync that is compared to a reference. A mismatch will set one of these bits.

8.2.7 *Motorola* MC68XXX Processor Family Distinctions

8.2.7.1 Family of *Motorola* MC68XXX Processors

The *Motorola* MC68XXX family of microprocessors consists of several generations of processor chips: MC68000, MC68012, MC68020, MC68040, and MC68060. All are upwardly compatible.

8.2.7.2 *Motorola* MC68000 Processor

The *Motorola* MC68000 processor was the earliest of the MC68XXX based processors. It had a 24-bit address bus, 16-bit data bus, eight 32-bit data registers, and eight 32-bit address registers.

8.2.7.3 *Motorola* MC68012 Processor

The *Motorola* MC68012 processor differs from its predecessor in that it has:

- a 31-bit address bus
- loop-mode caching
- a larger instruction set.

8.2.7.4 *Motorola* MC68020 Processor

The *Motorola* MC68020 processor was an improvement on the MC68012 processor, and provides:

- a 32-bit address bus
- a 32-bit data bus
- a 256-byte instruction cache (direct-mapped)
- a microcoded coprocessor interface
- fewer alignment restrictions (only instructions need be word aligned)
- additional control registers and addressing modes
- a further enlarged instruction set
- two additional stack pointers
- additional status register bits.

The SMP20 (SMP using the *Motorola* MC68020 processor chip) uses a level 6 interrupt for fast pump operations. Fast pump is used to pump RAM memory following a power up of the processor or for a full initialization. Data is pumped in block sizes up to 256 K. If they are available, up to 32 time slots can be used for the pump operation.

Minor (but significant) changes were made to a number of registers:

- The memory error source register now uses bits 6 and 7. (See Table 8.2-4.)

  When set, bit 7 indicates multiple cache banks were "hit" for one access of memory.

  Bit 6 is used to report correctable bit errors.

- Hardware error source register bit 2 is now used to indicate a multiple response error, i.e., more than one board responded to an operation. Previously, bit 2 indicated an I/O bus parity error. (See Table 8.2-3.)
- Bit 1 of the software error source register is now used to indicate "no response" to a command. Previously, bit 1 indicated an invalid I/O operation. (See Table 8.2-2.)

8.2.7.5 *Motorola* MC68040 Processor

The *Motorola* MC68040 processor is an evolutionary step in switch module architecture. The core hardware consists of the:

- microprocessor core (CORE40) pack
- 32 Mbyte dynamic memory (MEM32) pack
- bus service node (BSN) pack
- local system bus (LS).

The interrupt structure for SMP40 is similar to the structure of SMP20, except the SMP40 uses six interrupt levels. See Figure 8.2-9 for the interrupt hierarchy.

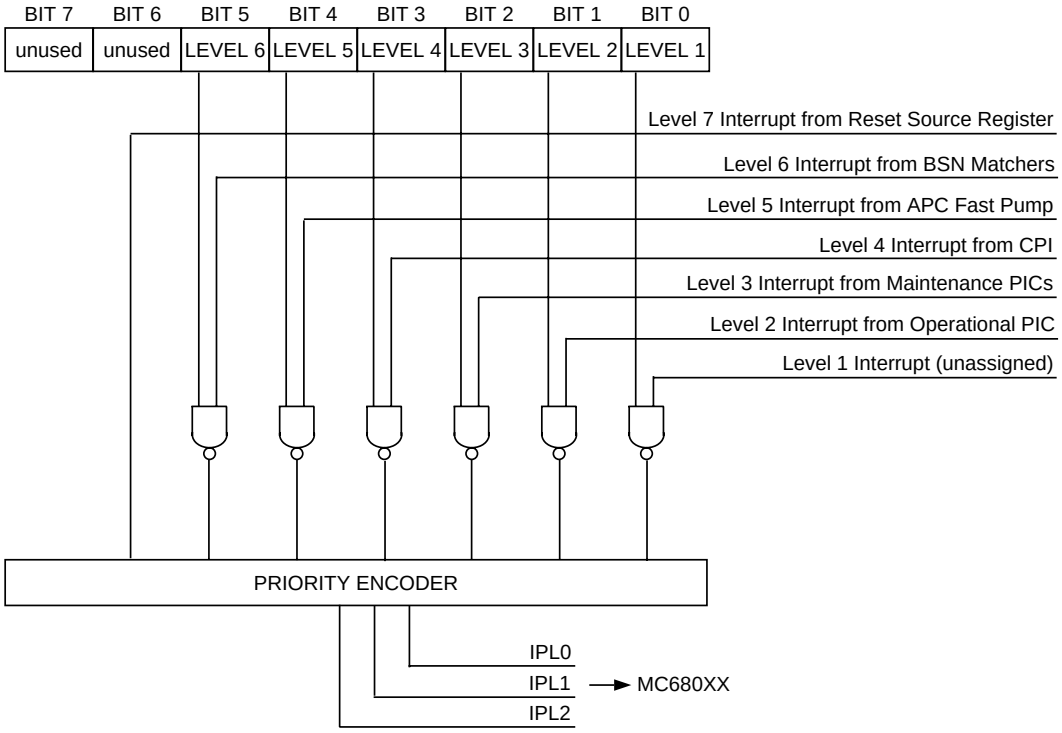


Figure 8.2-9 — Interrupt Hierarchy

The level 7 interrupts are sourced by any unmasked bit set in the reset source register, which is a collection of the truly unmaskable errors (sanity timer, CPI, etc.) and the maskable summary errors from the hardware and software error source registers. All level 7 interrupts are autovectored by the hardware. Figure 8.2-10 shows the level 7 interrupt hierarchy.

The level 6 interrupt is allocated to the address/data/operational pattern matchers on the BSN circuit pack. This interrupt is autovectored by the CORE40.

The level 5 interrupt is reserved for the fast pump mechanism on the application controller pack (APC). This autovectored interrupt is used to indicate that a header

has been detected by the bootstrapper circuit and that the microprocessor must program the fast pump direct memory access (DMA) controller.

The level 4 interrupt is used by the central processor intervention (CPI) gate array on the BSN. This autovectored interrupt is generated for all operational CPI messages except CPI reset orders received by the gate array.

The level 3 interrupt is generated by the programmable interrupt controllers for the generation of peripheral maintenance interrupts. The interrupt is not autovectored and reads an interrupt vector from the programmable interrupt controllers (PIC B and PIC C) resident on the BSN.

A level 2 interrupt is created for SMP40 to accommodate the operational interrupts. The only interrupt at this level is the 10 ms timer interrupt. The interrupt is not autovectored and reads an interrupt vector from the programmable interrupt controller (PIC-A) on the BSN. Additional level 2 interrupts in PIC A are allocated to RS bus packs and miscellaneous timers. These interrupts are presented to the microprocessor at level 2 when the I/O timer is not running.

The Level 1 interrupt is not assigned.

The settings of the interrupt priority levels (IPLs) for all 6 interrupt levels are shown here:

Interrupt Level	IPL2	IPL1	IPL0
7	1	1	1
6	1	1	0
5	1	0	1
4	1	0	0
3	0	1	1
2	0	1	0

Note: Register layouts for all registers can be obtained from the header files available through the on-line listing procedure described in Section 2. From the LISTINGS MENU, select item 2, Source File Name and enter the file name associated with each register, such as SMmp_xxxxx.h, where xxxxx is the unique register file name.

The system responds with the full path name, including the register file name, such as /.../.../.../hdr/smim/SMmp_xxxxx.h and then displays the register layout.

To locate the register layout for any register, look at the file named SMmp_reg.h. This file lists all registers and the unique register layout file name for each.

For reader convenience, the register layouts shown in this manual contain the unique register file name.

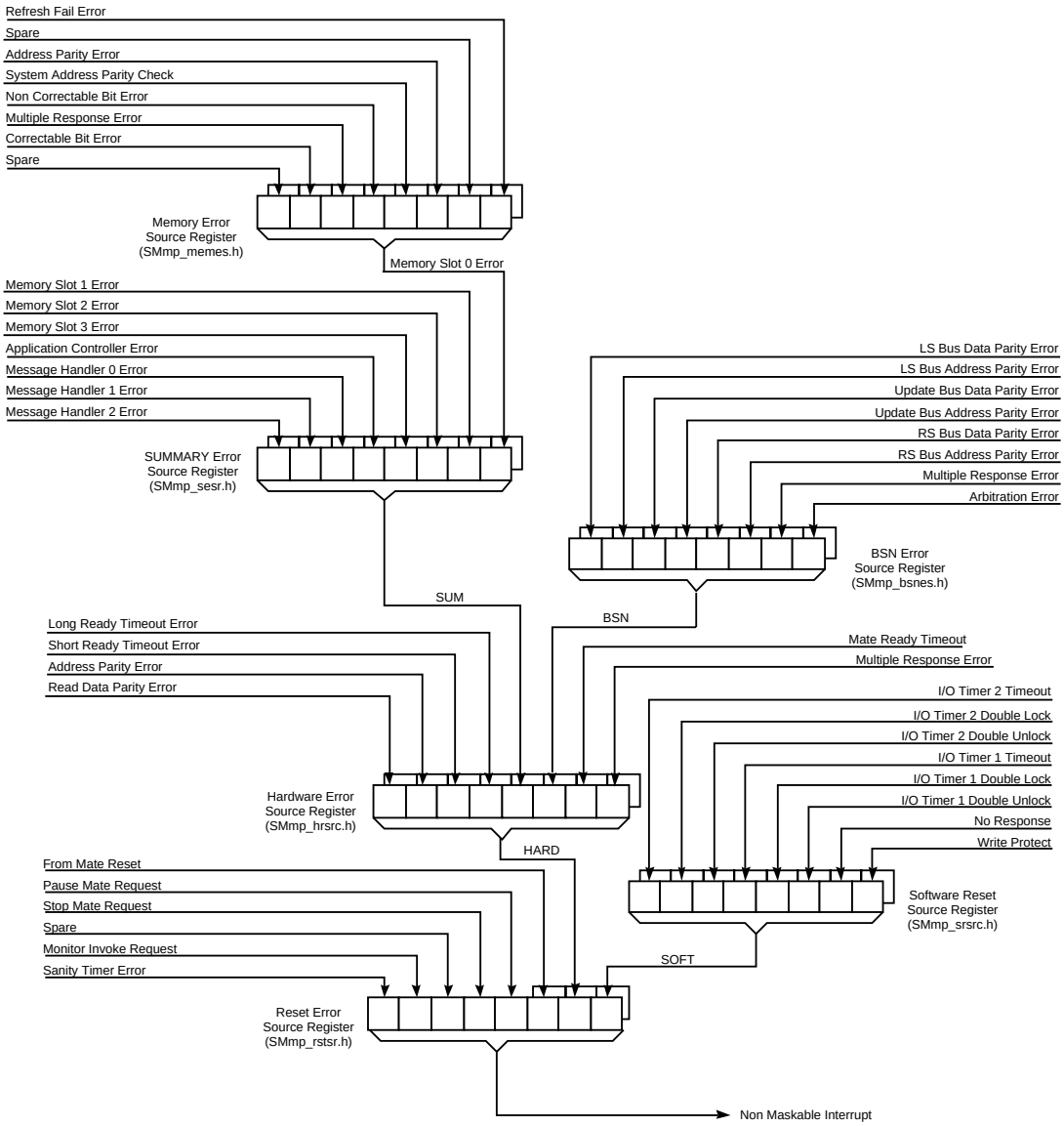


Figure 8.2-10 — SMP40 Non-maskable Interrupt Hierarchy

The reset source register summarizes the seven non-maskable interrupt sources to the microprocessor. It operates in conjunction with the reset source mask register to generate an NMI if any of the specific conditions are met. Only the hardware, software and from mate reset bits may be masked by the corresponding bit in the reset source mask register. The other four bits (SANTIM, SMITS, STOP and PAUSE) **are not** maskable. In addition, the mask bit for the from mate reset is only functional in an active (A-FF set, and **RUNNING**) processor (this means that the from mate reset **cannot** be masked in a non-active processor).

The hardware, software, and summary error source registers (ESRs) function identical to previous SMPs. The summary ESR reports to the hardware ESR and the hardware and software ESRs report directly to the reset source register.

Each MEM32 is equipped with a memory error source register for latching pack related errors. Each MEM32 pack provides a single error lead that feeds the summary error source register on the circuit pack. Each bit in the register is maskable by the corresponding bit in the memory error mask register.

The bus service node (BSN) pack contains a BSN error source register to collect address and data parity errors for each of three buses terminating on it. Each bit is maskable by the corresponding bit in the BSN mask register. All mask bits are set by the hardware reset signal.

The programmable interrupt controller - A (PIC A) register provides a reporting mechanism for all SMP40 operational interrupts. See Figure 8.2-11 for a bit description. In the SMP20 this register generates a level 4 interrupt, but in the SMP40 this is a level 2 interrupt.

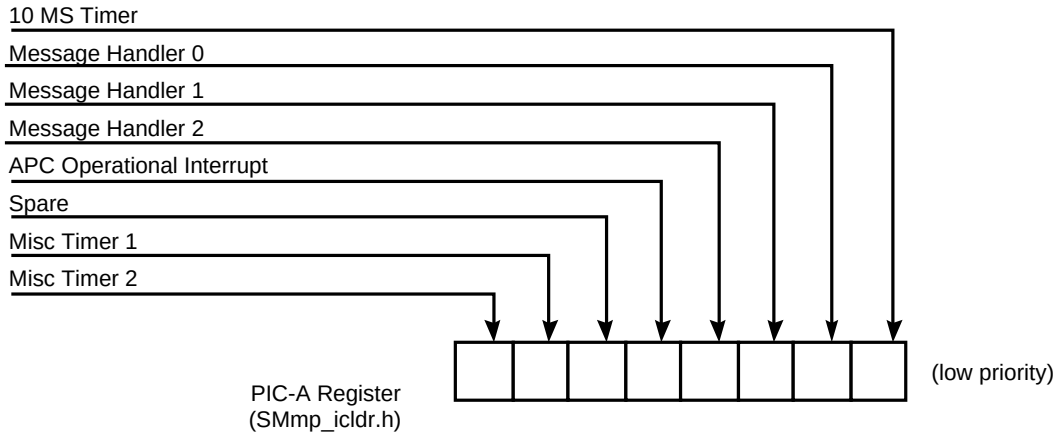


Figure 8.2-11 — PIC A Register

Exhibit 8.2-1 — PIC A Register: File SMmp_icldr.h

```
/*MPICLDR*/

          /*INTERRUPT CONTROLLER - LOWEST PRIORITY*/
/* ____________________________________________________________      */
/*|   7  |   6  |   5  |   4  |   3  |   2  |   1  |   0  |     */
/*|      |      |      |      |      |      |      |      |     */
/*|MTINT |SDLCI |SDBTI |SDATI |SDBRI |SDARI |10MSI |SDCRI | (A) */
/*|      |      |      |      |      |      |      |      |     */
/*|-------------------------------------------------------|     */
/*|      |      |      |      |      |      |      |      |     */
/*|MTINT |MT2IN |      |      |      |QUICL |10MSI |QUICH | (D) */
/*|      |      |      |      |      |      |      |      |     */
/*|-------------------------------------------------------|     */
/*|      |      |      |      |      |      |      |      |     */
/*|MT2IN |MTINT |      |APCOI |MH2OI |MH1OI |MH0OI |10MSI | (K) */
/*|      |      |      |      |      |      |      |      |     */
/* ------------------------------------------------------       */

/*  Exists for:                 */
/*  (A) SMP12                    */
/*      SMP23, SMP23X, SMP23CDM  */
/*      MCTU2                    */
/*  (D) MCTU3                    */
/*  (K) SM-2000                  */
```

```
/*  O:  offset        */
/*  M:  mask          */
/*  V:  active value  */


/*  Bit Description   */

#if (ASM || EES)
#define _ICLDR 0xea4
#else
#define _ICLDR 0x20080
#endif


#if (ASM)
#define OMPMTINT _ICLDR /*misc.timer interrupt*/
#define MMPMTINT 0x80
#define VMPMTINT 0x1

#define  OMPSDLCI _ICLDR /*Combined SDLC interrupt*/
#define  MMPSDLCI 0x40   /*This exists in TN872 only*/
#define  VMPSDLCI 0x01   /*It was removed with TN1617*/

#define  OMPMT2IN _ICLDR /*misc. timer 2 interrupt - MCTU3*/
#define  MMPMT2IN 0x40
#define  VMPMT2IN 0x01

#define OMPSDBTI _ICLDR /*sdlcb transmit interrupt*/
#define MMPSDBTI 0x20
#define  VMPSDBTI 0x1

#define OMPSDATI _ICLDR /*sdlca transmit interrupt*/
#define MMPSDATI 0x10
#define  VMPSDATI 0x1

#define OMPSDBRI _ICLDR /*sdlcb receive interrupt*/
#define MMPSDBRI 0x08
#define  VMPSDBRI 0x1

#define OMPSDARI _ICLDR /*sdlca receive interrupt*/
#define MMPSDARI 0x04
#define  VMPSDARI 0x1

#define OMPQUICL _ICLDR /*QUICC RQOUT interrupt low - MCTU3*/
#define MMPQUICL 0x04
#define  VMPQUICL 0x1

#define OMP10MSI _ICLDR /*10 ms timer interrupt*/
#define MMP10MSI 0x02
#define  VMP10MSI 0x1

#define OMPSDCRI _ICLDR /*Combined SDLC receive interrupt*/
#define MMPSDCRI 0x01   /*This exists in TN1617 and */
#define  VMPSDCRI 0x1   /*UN518 (SMP20) only*/

#define OMPQUICH _ICLDR /*QUICC RQOUT interrupt high - MCTU3*/
#define MMPQUICH 0x01
#define  VMPQUICH 0x1
#else


#define OMPMT2IN _ICLDR  /*Misc timer 2(billing counter) interrupt*/
#define MMPMT2IN 0x80
#define VMPMT2IN 0x1

#define OMPMTINT _ICLDR  /*Misc timer 1(10 ms timer) interrupt*/
#define MMPMTINT 0x40
```

```
#define VMPMTINT 0x1

#define OMPAPCOI _ICLDR  /*APC operational interrupt*/
#define MMPAPCOI 0x10
#define VMPAPCOI 0x1

#define OMPMH2OI _ICLDR  /*MH 2 operational interrupt*/
#define MMPMH2OI 0x08
#define VMPMH2OI 0x1

#define OMPMH1OI _ICLDR  /*MH 1 operational interrupt*/
#define MMPMH1OI 0x04
#define VMPMH1OI 0x1

#define OMPMH0OI _ICLDR  /*MH 0 operational interrupt*/
#define MMPMH0OI 0x02
#define VMPMH0OI 0x1

#define OMP10MSI _ICLDR  /*10 ms timer interrupt*/
#define MMP10MSI 0x01
#define VMP10MSI 0x1

#endif



 /*NOTES:  -  unlocked read/write
data register
This register represents the contents of the interrupt
service, interrupt mask, interrupt request, and auto
clear registers. The register selected to be read must
be preselected by first writing to the command register[ff41]
to select the desired register[mode bits 5, 6], then reading
this register according to the chart below:
a0 - interrupt service register
a4 - interrupt mask register
a8 - interrupt request register
ac - auto clear register

  MCTU3:
The QUICC will only generate one interrupt, so the two QUICC
interrupts (QUICL and QUICH) are actually the same interrupt,
both can be masked/enabled simultaneously.  Thus care must be
taken when enabling them both at the same time (possibly dual
interrupts) or when switching priorities (lost interrupts),
as unexpected results may happen when a QUICC interrupt comes
while no QUICH or QUICL bit is active. So when switching
priorities, it's safe to toggle both bits (QUICH and QUICL) in
the same instruction.
*/
```

The PIC B and PIC C registers report up to 16 maintenance interrupts. The functionality is the same as the SMP20, but the interrupt sources have changed because of differences in the SMP40 architecture. These PICs generate a level 3 interrupt. See Figures 8.2-12 and 8.2-13 for bit descriptions.

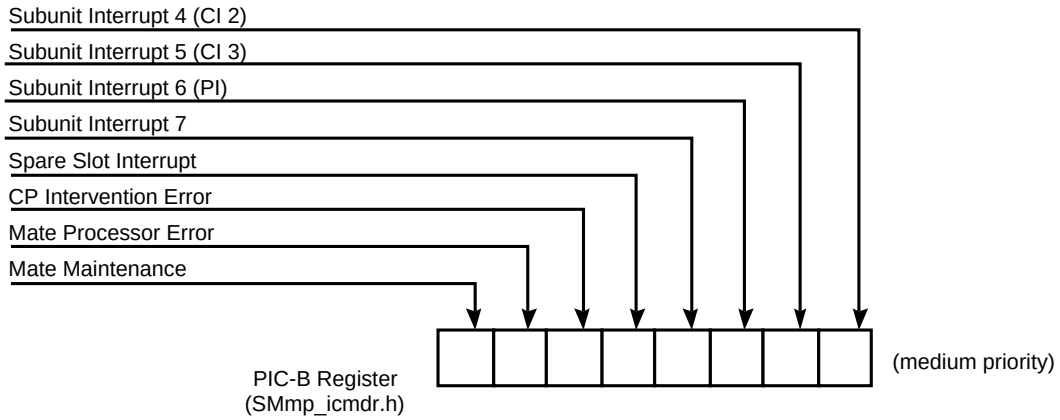


Figure 8.2-12 — PIC B **Register**

Exhibit 8.2-2 — PIC B **Register: File** SMmp_icmdr.h

```
/*MPICMDR*/

         /*INTERRUPT CONTROLLER - MEDIUM PRIORITY*/
/* ____________________________________________________________      */
/*|   7  |   6  |   5  |   4  |   3  |   2  |   1  |   0  |     */
/*|      |      |      |      |      |      |      |      |     */
/*|MPRIN |MCINT |      |      | CPIE |CBERR |CI5IN |CI4IN | (A) */
/*|      |      |      |      |      |      |      |      |     */
/* ------------------------------------------------------      */
/*|      |      |      |      |      |      |      |      |     */
/*|MPRIN |MCINT |MRWRI |AFFCI | CPIE |CBERR |CI5IN |CI4IN | (D) */
/*|      |      |      |      |      |      |      |      |     */
/* ------------------------------------------------------      */
/*|      |      |      |      |      |      |      |      |     */
/*|MPRIN |MCINT |CPIE  |      |      |PIIN  |CI3IN |CI2IN | (K) */
/*|      |      |      |      |      |      |      |      |     */
/* ------------------------------------------------------      */

/*  Exists for:                */
/* (A) SMP12                    */
/*     SMP23, SMP23X, SMP23CDM */
/*     MCTU2                    */
/* (D) MCTU3                    */
/* (K) SM-2000                    */


/*  O:  offset        */
/*  M:  mask          */
/*  V:  active value  */


/*  Bit Description   */

#if (ASM  || EES)
#define _ICMDR 0xea8
#else
#define _ICMDR 0x200c0
#endif

#if (ASM)
#define OMPMPRIN _ICMDR /*mate peripheral interrupt [a maintenance*/
#define MMPMPRIN 0x80     /*interrupt occurred on the mate side]*/
```

```
#define  VMPMPRIN 0x01

#define OMPMCINT _ICMDR /*mate controller interrupt [a non maskable*/
#define MMPMCINT 0x40     /*interrupt occurred on the mate side]*/
#define  VMPMCINT 0x01

#define OMPMRWRI _ICMDR /*mate read while running - MCTU3*/
#define MMPMRWRI 0x20
#define  VMPMRWRI 0x01

#define OMPAFFCI _ICMDR /*A-FF clear interrupt - MCTU3*/
#define MMPAFFCI 0x10
#define  VMPAFFCI 0x01

#define OMPCPIE  _ICMDR /*cp intervention interrupt*/
#define MMPCPIE  0x08
#define  VMPCPIE  0x01

#define OMPCBERR _ICMDR /*correctable bit error*/
#define MMPCBERR 0x04
#define  VMPCBERR 0x01

#define OMPCI5IN _ICMDR /*CI 5 interrupt*/
#define MMPCI5IN 0x02
#define  VMPCI5IN 0x01

#define OMPCI4IN _ICMDR /*CI 4 interrupt*/
#define MMPCI4IN 0x01
#define  VMPCI4IN 0x01
#else


#define OMPMPRIN _ICMDR /*mate peripheral interrupt [a maintenance*/
#define MMPMPRIN 0x80     /*interrupt occurred on the mate side]*/
#define  VMPMPRIN 0x01

#define OMPMCINT _ICMDR /*mate controller interrupt [a non maskable*/
#define MMPMCINT 0x40     /*interrupt occurred on the mate side]*/
#define  VMPMCINT 0x01

#define OMPCPIE  _ICMDR /*cp intervention interrupt*/
#define MMPCPIE  0x20
#define  VMPCPIE  0x01

#define OMPPIIN _ICMDR /*Packet interface interrupt*/
#define MMPPIIN 0x04
#define VMPPIIN 0x01

#define OMPCI3IN _ICMDR /*CI 3 interrupt*/
#define MMPCI3IN 0x02
#define  VMPCI3IN 0x01

#define OMPCI2IN _ICMDR /*CI 2 interrupt*/
#define MMPCI2IN 0x01
#define  VMPCI2IN 0x01
#endif



 /*NOTES:  -  data register
unlocked read/write
This register represents the contents of the interrupt
service, interrupt mask, interrupt request, and auto
clear registers. The register selected to be read
must be preselected by first writing to the command register[ff45]
to select the desired register[mode bits 5, 6], then reading this
register according to the chart below:
a0 - interrupt service register
a4 - interrupt mask register
```

```
a8 - interrupt request register
ac - auto clear register

MCTU3: The AFFCI bit exists for SM-2000 in the MPICHDR register.
This is only a bit move for the MCTU3.

Any unmasked maintenance interrupt occurring on a standby processor
will show as interrupt 7 in the active processor.
*/
```

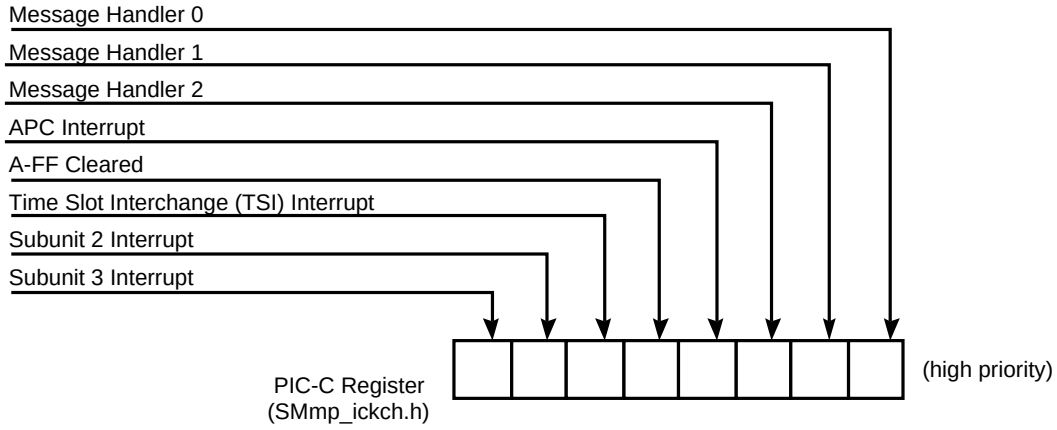


Figure 8.2-13 — PIC C **Register**

Exhibit 8.2-3 — PIC C **Register: File** SMmp_ichdr.h

```
/*MPICHDR*/

         /*INTERRUPT CONTROLLER - HIGHEST PRIORITY*/
/* ___________________________________________________________             */
/*|   7   |   6   |   5   |   4   |   3   |   2   |   1   |   0   |        */
/*|       |       |       |       |       |       |       |       |        */
/*|CI3IN |PIIN  |CI1IN |CIOIN |SPINT |TSIIN |DLI1I |DLIOI | Pre-SMP40 */
/*|       |       |       |       |       |       |       |       |        */
/* -----------------------------------------------------             */
/*|       |       |       |       |       |       |       |       |        */
/*|CI1IN |CIOIN |TSIIN |AFFCI |APCIN |MH2IN |MH1IN |MHOIN | SMP40     */
/*|       |       |       |       |       |       |       |       |        */
/* -----------------------------------------------------             */

/*  O:  offset        */
/*  M:  mask          */
/*  V:  active value  */


/*  Bit Description   */

#if (ASM || EES)
#define _ICHDR   0xeac
#else
#define _ICHDR   0x200c8
#endif

#if (ASM)
#define OMPCI3IN _ICHDR /*control interface 3 interrupt*/
#define MMPCI3IN 0x80
```

```
#define  VMPCI3IN 0x1

#define OMPCSCIN OMPCI3IN /* CSC interrupt - See Note 1 */
#define MMPCSCIN MMPCI3IN
#define VMPCSCIN VMPCI3IN

#define OMPPIIN  _ICHDR /*PI  interrupt*/
#define MMPPIIN 0x40
#define  VMPPIIN  0x1

#define OMPCI1IN _ICHDR /*control interface 1 interrupt*/
#define MMPCI1IN 0x20
#define  VMPCI1IN 0x1

#define OMPCI0IN _ICHDR /*control interface 0 interrupt*/
#define MMPCI0IN 0x10
#define  VMPCI0IN 0x1

#define OMPSPINT _ICHDR /*signal processor interrupt*/
#define MMPSPINT 0x08
#define  VMPSPINT 0x1

#define OMPTSIIN _ICHDR /*timeslot interchange interrupt*/
#define MMPTSIIN 0x04
#define  VMPTSIIN 0x1

#define OMPDLI1I _ICHDR /*data link 1 interrupt*/
#define MMPDLI1I 0x02
#define  VMPDLI1I 0x1

#define OMPDLI0I _ICHDR /*data link 0 interrupt*/
#define MMPDLI0I 0x01
#define  VMPDLI0I 0x1
#else


#define OMPCI1IN _ICHDR  /*control interface 1 interrupt*/
#define MMPCI1IN 0x80
#define  VMPCI1IN 0x1

#define OMPCI0IN _ICHDR  /*control interface 0 interrupt*/
#define MMPCI0IN 0x40
#define  VMPCI0IN 0x1

#define OMPTSIIN _ICHDR  /*timeslot interchange interrupt*/
#define MMPTSIIN 0x20
#define  VMPTSIIN 0x1

#define OMPAFFCI _ICHDR  /*A-FF cleared interrupt*/
#define MMPAFFCI 0x10
#define VMPAFFCI 0x1

#define OMPAPCIN _ICHDR  /*APC interrupt*/
#define MMPAPCIN 0x08
#define VMPAPCIN 0x1

#define OMPMH2IN _ICHDR  /*MH2 interrupt*/
#define MMPMH2IN 0x04
#define VMPMH2IN 0x1

#define OMPMH1IN _ICHDR  /*MH1 interrupt*/
#define MMPMH1IN 0x02
#define VMPMH1IN 0x1

#define OMPMH0IN _ICHDR  /*MH0 interrupt*/
#define MMPMH0IN 0x01
#define VMPMH0IN 0x1

#endif
```

```
 /*NOTES:  -  data register

General Comment:
unlocked read/write
This register represents the contents of the interrupt service,
interrupt mask, interrupt request, and auto clear registers.
The register selected to be read must be preselected by first
writing to the command register[ff49] to select the desired
register [mode bits 5, 6], then reading this register according
to the chart below:
a0 - interrupt source register
a4 - interrupt mask register
a8 - interrupt request register
ac - auto clear register

Note 1: In an AWS_SM, the CSC will occupy the same slot as
CI3 would have.  The bit definitions for the CSC
(MPCSCIN) will share the CI3 bit defines.
*/
```

The interrupt enable register (IER) and read interrupt status register (RINTS) are SMP40 registers.

The interrupt enable register provides a single location to mask each level of interrupt independently. Previous SMPs had various interrupt mask bits scattered throughout the system (generally associated with the circuitry responsible for generation of the interrupt); the SMP40 consolidates these mask bits into a single register. See Figure 8.2-14.

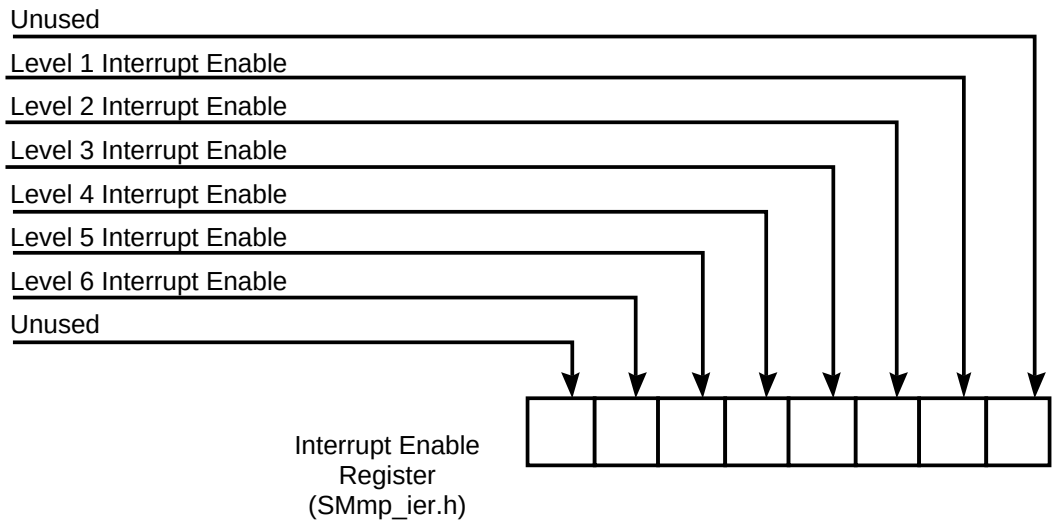


Figure 8.2-14 — Interrupt Enable Register

The read interrupt status register (RINTS) makes the pre-enabled status of the interrupt signals available to the microprocessor. The RINTS register gives software the ability to determine if a specific level of interrupt is pending prior to enabling that level. See Figure 8.2-15.

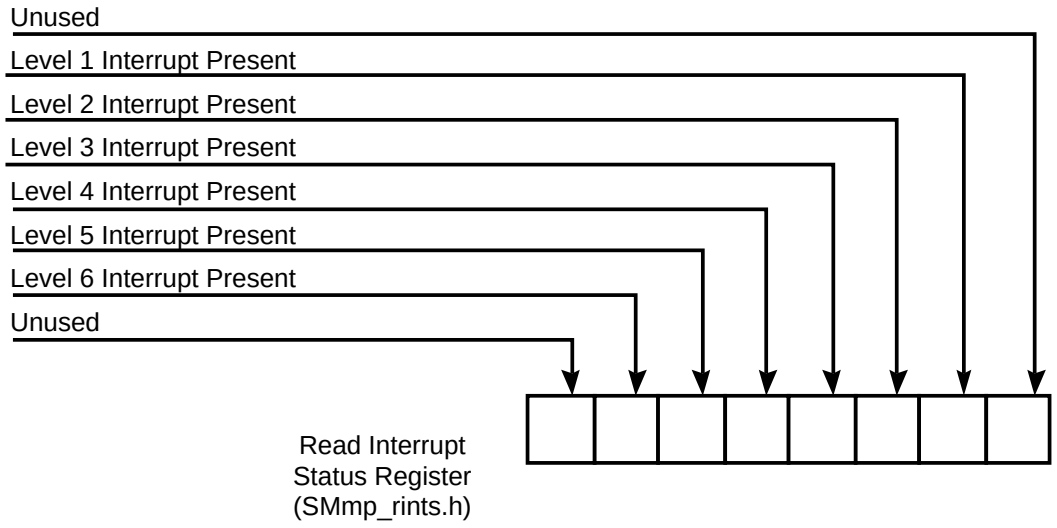


Figure 8.2-15 — Read Interrupt Status Register

There are five registers that report processor status. The controller status 1, controller status 2, and active status registers are carry-overs from previous SMPs. The auxiliary status register and test utility status register are new additions for SMP40.

The auxiliary status register provides additional miscellaneous status. See Figure 8.2-16 for a definition of the active bits.

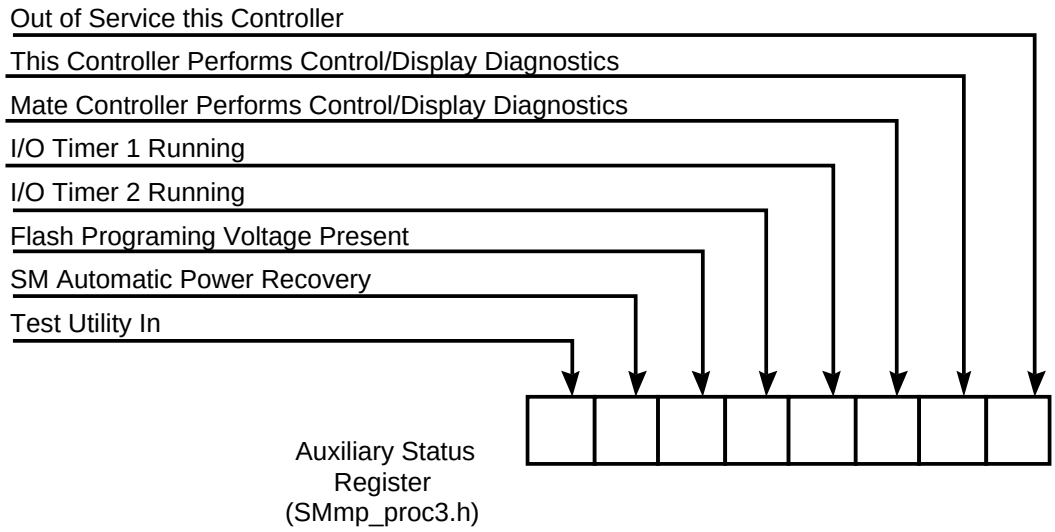


Figure 8.2-16 — Auxiliary Status Register

The test utility status register consolidates the status from the test utility bus. This eight-bit read-only register is shown in Figure 8.2-17.

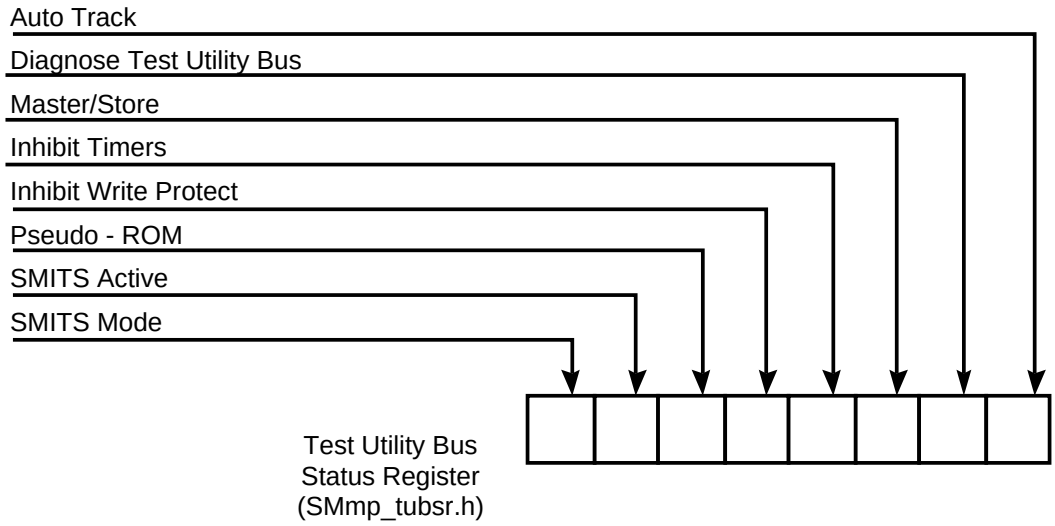


Figure 8.2-17 — Test Utility Bus Status Register

8.2.7.6 *Motorola* MC68060 Processor

The CORE60 circuit pack is the second generation CORE processor pack for the SM-2000 switch module processor. It can replace the CORE40 circuit pack in the module controller timeslot interchanger model 4 (SMPU4 [SM2000]). The CORE60 circuit pack contains all the functionality of the CORE40 circuit pack.

The CORE60 circuit pack is implemented around the *Motorola* MC68060 microprocessor. The *Motorola* MC68060 processor provides:

- An integral memory management unit (MMU)
- Two 8 KByte cache memory systems
- An internal branch target cache for improved branch performance

The CORE60 circuit pack also provides the following performance enhancing features:

- A 2 MByte direct mapped unified instruction and data cache
- A dynamic RAM subsystem with EDC (64 MByte with CORE60; 64 or 128 MByte with CORE60MM)
- A write posting buffer
- An enhanced I/O subsystem with optimized write strobes.

The interrupt structure for the CORE60 is similar to the structure of CORE40. See Figure 8.2-9. A major difference is the addition of the CORE error source register (`CORES`) and the CORE auxiliary error source register (`CAXES`). See Figure 8.2-18 for interrupt hierarchy.

Note: Register layouts for all registers can be obtained from the header files available through the on-line listing procedure described in Section 2. From the `LISTINGS MENU`, select item 2, `Source File Name` and enter the file name associated with each register, such as `SMmp_xxxxx.h`, where `xxxxx` is the unique register file name.

The system responds with the full path name, including the register file name, such as `/.../.../.../.../hdr/smim/SMmp_xxxxx.h` and then displays the register layout.

To locate the register layout for any register, look at the file named `SMmp_reg.h`. This file lists all registers and the unique register layout file name for each.

For reader convenience, the register layouts shown in this manual contain the unique register file name.

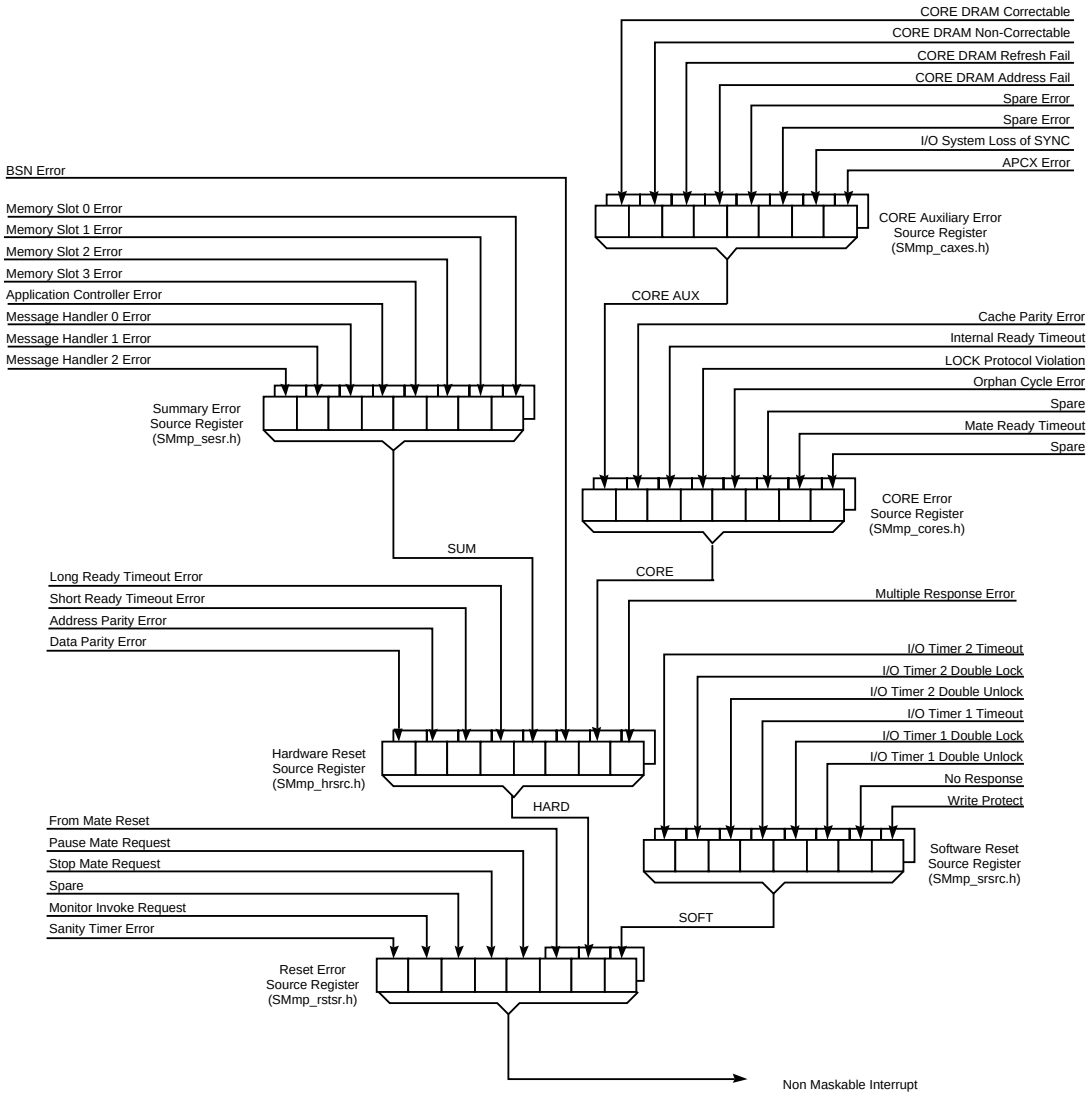


Figure 8.2-18 — SMP60 Non-maskable Interrupt Hierarchy

The hardware, software, summary, CORE, and CORE auxiliary ESRs function identically. The CAXES (CORE auxiliary ESR) reports to the CORES (CORE ESR). The summary and CORES ESRs report to the hardware ESR. The hardware and software ESRs report to the reset source register. Each bit in each of the five ESRs has the ability to be set by externally detected sources (if specified) or by software. Each bit also has an associated mask bit in the appropriate mask register. If a mask bit is set, the error bit may be set, but the error will not propagate beyond the register. If a mask bit is not set and an error bit is set (by hardware or software), the error will be propagated to the next level register.

The interrupt enable and the read interrupt status registers function the same as in the SMP40.

There are six registers which report processor status. The proc 1 (controller status 1), proc 2 (controller status 2), and active status register are carry-overs from SMPs prior to SMP40. The proc 3 (auxiliary status) register and test utility status register were new additions to SMP40. The CORE60 control and status register is new to the SMP60. The CORE60 control and status register provides control and status for the CORE60 functionality. See Figure 8.2-19.

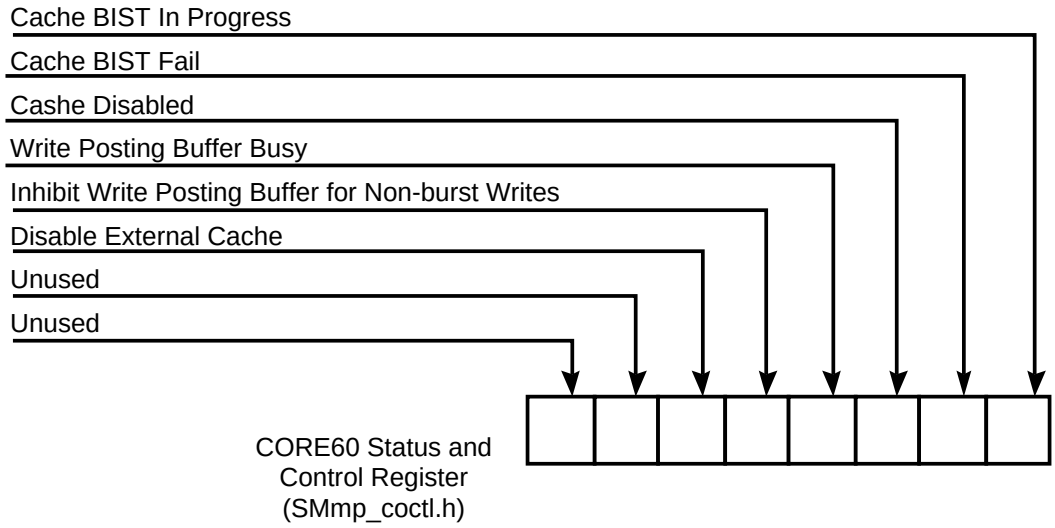


Figure 8.2-19 — CORE60 Status and Control Register

8.2.8 Interrupt Masking

8.2.8.1 Interrupt Masking

Interrupt masking prevents an interrupt from notifying the system of its occurrence. Although the condition that caused the interrupt may still exist, the interrupt (and thus its source) is ignored.

Each interrupt register that contains maskable interrupts has an associated mask register. A bit set in the mask register causes the corresponding interrupt in the interrupt register to be masked (i.e., ignored). Conversely, if a bit is not set in the mask register, its associated interrupt in the interrupt register is "allowed."

Interrupts in peripheral units can be masked at the unit level, thus preventing them from propagating to the CI. Interrupts from the peripheral units to the CI can be masked in the CI by setting the appropriate bits in the CI mask registers.

Some interrupt registers and their mask registers share the same address. In these cases, the interrupt register is a read-only register and the mask register is a write-only register. A read operation causes the interrupt register to be read, while a write operation to the same address causes the mask register to be written. (Interrupt and mask registers in the SMP do not share addresses.)

Since the mask register cannot be read, any software needing to know its current state must consult a software image of that mask register. Software images of mask registers are kept in relations in the database and updated each time the mask register is written.

8.2.8.2 Status Register

Three bits of the 16-bit status register are used to control the interrupt system.

The module processor compares the level of an interrupt request with the setting of the interrupt mask in bits 8-10 of the status register. The interrupt will not be recognized unless it is of a higher level than the mask level set in the status register.

The status register is also used to store other information about the status of the processor. For example, the conditional branch instruction refers to the status register to determine the results of previous operations. Bits 0 through 7 of the register comprise the "user byte" and can be read or modified by programs executing in either the user or supervisor mode. Bits 8 through 15 comprise the "system byte" and can be modified only by programs that execute in the supervisor mode. See Figure 8.2-20.

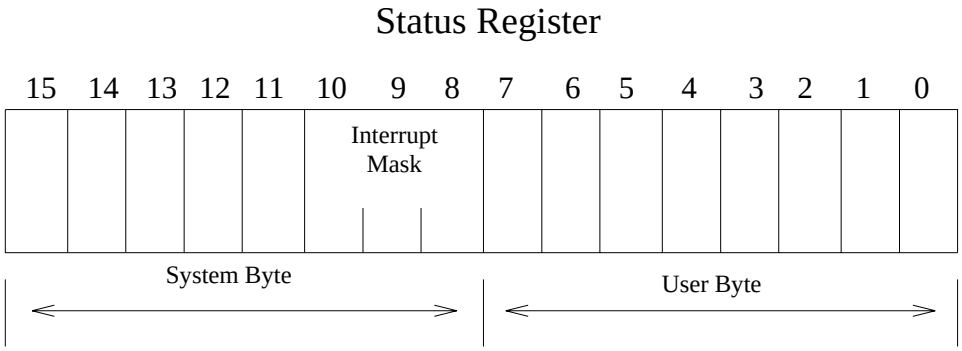


Figure 8.2-20 — Status Register and Interrupt Masking Structure

Level of Interrupt to be Masked	Interrupt Mask		
	Bit 10	Bit 9	Bit 8
1 to 7	1	1	1
1 to 6	1	1	0
1 to 5	1	0	1
1 to 4	1	0	0
1 to 3	0	1	1
1 to 2	0	1	0
1	0	0	1
All levels enabled	0	0	0

System Byte		User Byte	
Bit	**Description**	**Bit**	**Description**
15	The trace or T1 (*Motorola* MC68020 processor) bit is a hardware aid for debuggers. If bit 15 is set, an exception takes place at the end of each instruction.	7	Unused
14	The T0 (*Motorola* MC68020 processor) bit. If bit 14 is set, a trace occurs on change of flow. Bits 15 and 14 cannot both be set at the same time.	6	Unused
13	The supervisor bit is used to regulate access to certain instructions and to the system byte of the status register.	5	Unused
12	The master/interrupt (*Motorola* MC68020 processor) bit allows multitask operating systems to use an interrupt or additional stacks in the supervision state.	4	The extended bit is always set to the same state as the carry bit, bit 0, whenever it is affected by an instruction. This bit is provided for use in the multiprecision arithmetic operations.
11	Unused	3	The negative bit is set if the high order bit of a result is set.
10	Interrupt mask	2	The zero bit is set only when an arithmetic operation produces a zero or reset with a non-zero result.
9	Interrupt mask	1	The overflow bit represents overflow from an operation yielding a condition where the result cannot be represented.
8	Interrupt mask	0	The carry bit holds the carry from the high bit produced by arithmetic operations or shifts. Logical operations, moves, multiples and divides clear the bit.

8.3 INTERRUPT RECEIVE ONLY PRINTER (ROP) OUTPUT

8.3.1 AM Interrupt ROP Output

Report messages (REPT) are used to provide data and information on error interrupt or system initialization on the 3B20D computer. They are output at the time of the error and provide a sequence number which can be used to locate additional data that is related to the error.

Supplementary information is generally output after the processor has stabilized. Detailed information is saved on disk in a number of error history files known as logfiles. Each of these logfiles is associated with a specific type of error.

The following is an example of a report message:

```
REPT CU 0 ERROR INTERRUPT X'08 X'21
```

This message reports that a noncorrectable parity failure (as indicated by the X'08) occurred in CU 0. The X'21 is a sequence number that can be used to associate related ROP output. For a complete description of error messages, refer to the 235-600-750, *Output Messages Manual*.

8.3.2 SM Interrupt ROP Output

First Message

```
S570-65669 90-10-06 00:00:30 030742 INT FIVE I
REPT SM=9,0 HWLVL=0 SWLVL=RPI CIO TIME-OUT-ERROR EVENT=6410 COMPLETED
  HW-ERR FAILING ADDR=H'5d8 PROCESS:BG=31,0,RPI CM=NONE, FG=NONE,,
```

The INT output message provides the following information:

Line 1:

- S570-65669 is the utility number and the process ID that caused this message to be printed. It is not related to the process that was executing at the time of the interrupt.
- The remainder of the first line is the date, time, and message sequence number.

Line 2:

- SM=9,0 specifies that SM 9, controller 0 generated this interrupt. Side 0 registers are printed after this message.
- HWLVL=0 is the level of hardware escalation attempted by the system to recover from the fault.
- SWLVL=RPI is the software action taken to recover from the fault; here, return to point of interrupt.
- CIO TIME-OUT-ERROR indicates the type of fault. A peripheral failed to reply to a CI message before the timeout counter in the receive sequencer circuitry reached its terminal count.
- EVENT=6410 is the event number. This is the system integrity (SI) number used for tracking and correlating related output messages.

Line 3:

- HW-ERR indicates that a hardware error is suspected. If a software error was suspected, this field would have been SW-ERR.

- The FAILING ADDR usually points to the address of the executing program. In this case, however, it is the address of some peripheral device. And, in the case of a WRITE-PROT-ERR, the FAILING ADDR is the address at which the program attempted an invalid write.

Second Message

```
S570-65669 90-10-06 00:00:37 030745 INT_MON FIVE I
REPT SM=9,0 HWLVL=0 SWLVL=RPI EVENT=6410 COMPLETED
  CIO TIME-OUT-ERROR
  HW-ERR FAIL-ADDR==H'5d8 ROM-UNK   DATA-BUS=H'0 TIME=0:0.2
  PROCESS:BG=31,0,RPI CM=NONE, FG=NONE,,          NORMAL
  ORIG.-HW-STATUS: MC0: ACT   MC1: STBY
  FINAL-HW-STATUS: MC0: ACT   MC1: STBY
  PREVIOUS TYPE/COUNT:  119    0
  SHADOW TYPE/COUNT:    71    36864
  AUX DATA:             H'0    H'200      H'0     H'0
  ESCALATION-COUNTS:    H'0    H'0        H'0     H'0
```

The first three lines of the INT_MON output message contain information that was also found in the INT output message described previously. The remainder of the message provides the following:

- Line 4: Contains the failing address and the data present on the data bus. The time information field refers to the minutes, seconds, and tenths of seconds past the hour when the interrupt was triggered.
- Line 5: Provides information about the background and foreground processes and the recovery action taken by the system (if any).
- Lines 6, 7: Reflect the original and final hardware status of the two module controllers.
- Line 8: Gives the recovery progress type when the stimuli occurred, and the recovery progress counter within the recovery progress type when the stimuli occurred; both values are given in decimal.
- Line 9: SHADOW TYPE is a snapshot of the recovery progress type at the last recovery progress check, and COUNT is a snapshot of the recovery progress counter at the last recovery progress check; both values are in decimal.
- Line 10: AUX DATA, with four fields:
  1. Unused
  2. Reset counter shadow (lower four bits)
     Requested TTY level (middle four bits)
     Number of links up (upper four bits)
  3. Pump return code from the previous initialization
  4. SMrcvy_level, highest level of most recent hardware recovery (least significant 8 bits)
     SMoldrcvy_level, shadow hardware recovery level (next 8 bits).
- Line 11: ESCALATION-COUNTS, with four fields:
  1. The SI recovery variables, packed into 32 bits:
     INallcntr count of all stimuli (upper 16 bits)
     INrpicntr count of RPI stimuli (lower 16 bits)
  2. The SI recovery variables, packed into 32 bits:
     INsppcntr count of interrupt requested SPP stimuli (upper 16 bits)

```
INdefcntr count of assert requested SPP stimuli (lower 16 bits)
```

3. The SI recovery variables, packed into 32 bits:

   ```
   INausppcntr count of audit requested SPP stimuli (upper 16 bits)
   INdausppcntr count of deferred SPP stimuli (lower 16 bits)
   ```

4. The SI recovery variables, packed into 32 bits:

   ```
   Automatic initialization escalation reason code (upper 16 bits)

   Time call processing was turned off due to SPPs and directed audits;
   each peg count represents 30 milliseconds (lower 16 bits).
   ```

Third Message

This message shows the contents of the *Motorola*[1] MC68XXX processor registers at the time of the interrupt.

The program counter (PC) register shows the point of interrupt. Typically, the program counter is two assembly instructions beyond the instruction which caused the interrupt.

```
S570-65669 90-10-06 00:00:45 030748 INT_MON FIVE I
REPT SM=9  HARDWARE CONTEXT STANDARD LSM EVENT=6410
  68012-REGISTERS: SSP=H'22bfc   PC=H'1a0c46         SR=H'4
          USP=H'8dde2   FP=H'8dde6      A5=H'0       A4=H'0
          A0=H'6877fc   A1=H'20100      A2=H'0       A3=H'0
          D0=H'80000    D1=H'20000      D2=H'1f      D3=H'0
          D4=H'0        D5=H'0          D6=H'5d80b00 D7=H'0
  PIC-REGISTERS:     HI       MED       LOW
              IRR:  H'90      H'80      H'c2
              IMR:  H'c0      H'3b      H'c0
              ISR:  H'10      H'0       H'0
```

In the `PIC-REGISTERS` section,

`HI` = PIC C
`MED` = PIC B
`LOW` = PIC A
`IRR` = interrupt request register, holds interrupt history
`IMR` = interrupt mask register, masks out the unused PIC bits
`ISR` = interrupt service register, the contents of each PIC register

Fourth Message

This message reveals the contents of the hardware registers associated with the failing hardware. The registers that will appear here depends on the type of fault detected by the processor. In this case, the registers are associated with CI 0.

```
S570-65669 90-10-06 00:01:01 030754 INT_MON FIVE I
REPT SM=9 CI 0 HW REGS EVENT=6410
   ERSRC=H'20 MCTRL=H'2 PERAD=H'5d8 PDATA=H'b00
   HLINH=H'b0 LLINH=H'0 HRINH=H'ff LRINH=H'ffff
```

Fifth Message

1. Registered trademark of Motorola Inc.

The user stack trace message contains the return address of each function call leading up to the point of interrupt. The sequence of the function calls is from right to left. The most recent address on the stack should match either the FAIL-ADDR or the value in the PC register.

```
S570-65669 90-10-06 00:01:09 030759 INT_MON FIVE I
REPT SM=9 STACK TRACE ENV=OSDSM SRC=FR EVENT=6410
  USER: 001A0C46 0033663C 00335F1C 0033642C 00335FC8 001F96E0
```

Sixth Message

This message contains two stack frames. The data in the first stack frame belongs to the interrupted function; the second stack frame belongs to the function that called the interrupted function.

The FUNC ADDR of the stack frames are the same as the first two addresses in the user stack trace message (fifth message).

```
S570-65669 90-10-06 00:01:23 030764 INT_MON FIVE I
REPT  SM=9   STACK FRAME ENV=OSDSM SRC=FR EVENT=6410
  FUNC ADDR:   H'1a0c46
  PARAMETERS:  05D8000B  00000000  DE9A005E  D43C0008  02000055
               001B0010  0012000A  11EA126A  DE9A0008  DE6600C5
  LOCAL DATA:  00000000  008C0005  000305DE  00090380  05002000
               02008CDE  00000000  8C000003  64B04600  E8DD0800
               00000000  5E00E8AB  6600C500  1C002B00  E4DD0800
               66DE0800  867D2C00  E8DD0800  FFFF3CD4  0000B88F
               010066DE  08000000  DF0A0000  00000000  00000300
               000006DF  C5000200  66DE0800  00000000  00000000
               00000000  1A8C7400  00000000  00000000  0400D6DD
               080004DF  C500A8C7  2C00BCDD  08003500  E89D3300
               3300B8DE  0800B88F  5E000500  0200C500  000096FF

S570-65669 90-10-06 00:01:27 030767 INT_MON FIVE I
REPT  SM=9   STACK FRAME ENV=OSDSM SRC=FR EVENT=6410
  FUNC ADDR:   H'33663c
  PARAMETERS:  0008DE8A  00000000  00000000  00000000  00000000
               00010000  0000FF83  00000000  1FCC0000  00DACC00
  LOCAL DATA:  0000008C  00050003  00000500  0000C500  00000000
               9BD30000  0C060000  00000000  00000000  010050E1
               C50066DE  08009ADE  6A12EA11  0A001200  10001B00
               55000002  08003CD4  5E009ADE  00000000  0B00D805
               3C663300  3EDE0800  00000000  008C0005  000305DE
               00090380  05002000  02008CDE  00000000  8C000003
               64B04600  E8DD0800  00000000  5E00E8AB  6600C500
               1C002B00  E4DD0800  66DE0800  867D2C00  E8DD0800
               FFFF3CD4  0000B88F  010066DE  08000000  DF0A0000
```

8.4 INTERRUPT ANALYSIS EXAMPLE — HARDWARE

Refer to Exhibit 8.4-1 in the following description of the steps used to analyze an interrupt caused by a hardware error.

1. Determine which SM and which controller reported the error.

   The information field SM=9,0 indicates SM 9, controller 0 is reporting the error.

2. Determine the module controller status before and after the interrupt.

   Since there are no differences between the ORIG-HW-STATUS and the FINAL-HW-STATUS, the status and configuration of the controllers remained unchanged. Module controller 0 was and is active, and module controller 1 was and is standby.

3. Determine the cause of the interrupt.

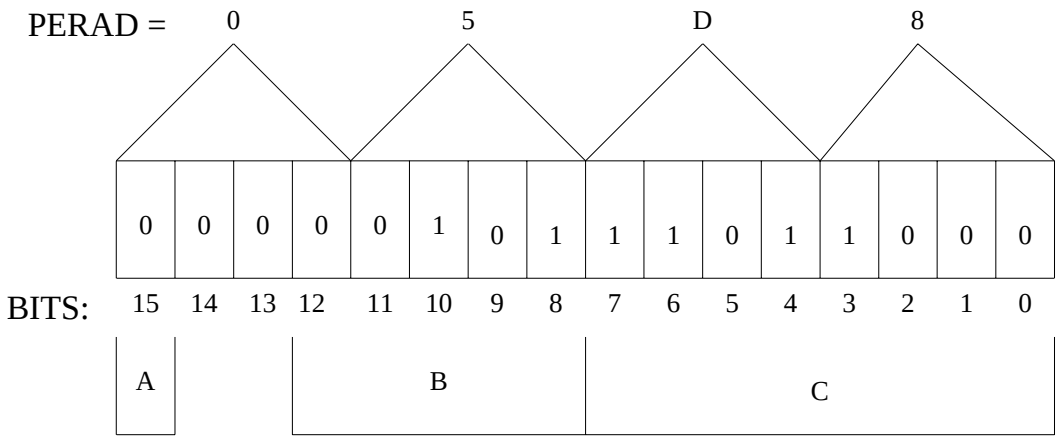
   The interrupt was due to a TIME-OUT-ERROR reported by CI 0. The state of bit 5 in the CI error source register (ESR) confirms this.

   When set, bit 5 in the CI ESR reports that a peripheral failed to reply to a CI requested operation before the timeout counter in the receive sequencer circuitry reached its terminal count.

4. Determine whether the error is software- or hardware-related.

   HW-ERR indicates that this interrupt was due to a hardware error.

5. Refer to the dump of the CI hardware registers (CI 0 HW REGS). The PERAD register contains the peripheral address of the suspected unit. The format of this register is shown in Figure 8.4-1.



A = RWPAB (read/write peripheral address bit)
B = PICB (peripheral interface control bus number)
C = PAB (peripheral address bits)

Figure 8.4-1 — PERAD Register

Bit 15 (RWPAB) specifies a read (E.15=0) or write (E.15=1) operation. Bits 12-8 define the PICB number, bits 7-3 select a board in the peripheral unit, and bits 2-0 specify a register on the board.

Bits 12-8 of the PERAD indicate that PICB 5 was selected. Use ODBE to review the relation PICB in SM 9 to identify the suspected unit:

```
      picb = 5 <<< key
  unitnum = 0
 unittype = SMDCLU
```

Thus PICB 5 is associated with digital carrier line unit (DCLU) 0 in SM 9. Further investigation of bits 7-3 of the PERAD register is needed to identify the subscriber digital facility interface (SDFI) board that was selected.

6. Determine the program address at which the interrupt occurred.

   As indicated by the PC, the interrupt occurred while executing the instruction at or near address H'1a0c46. Use the UPD:FTRC input message to convert this address to a function name.

Exhibit 8.4-1 — Hardware Interrupt Example

```
S570-65669 90-10-06 00:00:30 030742 INT FIVE I
REPT SM=9,0 HWLVL=0 SWLVL=RPI CIO TIME-OUT-ERROR EVENT=6410 COMPLETED
  HW-ERR FAILING ADDR=H'5d8 PROCESS:BG=31,0,RPI CM=NONE, FG=NONE,,

S570-65669 90-10-06 00:00:37 030745 INT_MON FIVE I
```

```
REPT SM=9,0 HWLVL=0 SWLVL=RPI EVENT=6410 COMPLETED
  CIO TIME-OUT-ERROR
  HW-ERR FAIL-ADDR=H'5d8 ROM-UNK  DATA-BUS=H'0 TIME=0:0.2
  PROCESS:BG=31,0,RPI CM=NONE, FG=NONE,,       NORMAL
  ORIG.-HW-STATUS: MC0: ACT   MC1: STBY
  FINAL-HW-STATUS: MC0: ACT   MC1: STBY
  PREVIOUS TYPE/COUNT:  119    0
  SHADOW TYPE/COUNT:    71    36864
  AUX DATA:             H'0    H'200     H'0    H'0
  ESCALATION-COUNTS:    H'0    H'0       H'0    H'0

S570-65669 90-10-06 00:00:45 030748 INT_MON FIVE I
REPT SM=9 HARDWARE CONTEXT STANDARD LSM EVENT=6410
  68012-REGISTERS:   SSP=H'22bfc    PC=H'1a0c46          SR=H'4
             USP=H'8dde2     FP=H'8dde6     A5=H'0       A4=H'0
             A0=H'6877fc     A1=H'20100     A2=H'0       A3=H'0
             D0=H'80000      D1=H'20000     D2=H'1f      D3=H'0
             D4=H'0          D5=H'0         D6=H'5d80b00 D7=H'0
  PIC-REGISTERS:     HI      MED      LOW
              IRR:  H'90     H'80     H'c2
              IMR:  H'c0     H'3b     H'c0
              ISR:  H'10     H'0      H'0

S570-65669 90-10-06 00:01:01 030754 INT_MON FIVE I
REPT SM=9 CI 0 HW REGS EVENT=6410
  ERSRC=H'20 MCTRL=H'2 PERAD=H'5d8 PDATA=H'b00
  HLINH=H'b0 LLINH=H'0 HRINH=H'ff LRINH=H'ffff

S570-65669 90-10-06 00:01:09 030759 INT_MON FIVE I
REPT SM=9 STACK TRACE ENV=OSDSM SRC=FR EVENT=6410
  USER: 001A0C46 0033663C 00335F1C 0033642C 00335FC8 001F96E0

S570-65669 90-10-06 00:01:23 030764 INT_MON FIVE I
REPT  SM=9   STACK FRAME ENV=OSDSM SRC=FR EVENT=6410
  FUNC ADDR:   H'1a0c46
  PARAMETERS:  05D8000B  00000000  DE9A005E  D43C0008  02000055
               001B0010  0012000A  11EA126A  DE9A0008  DE6600C5
  LOCAL DATA:  00000000  008C0005  000305DE  00090380  05002000
               02008CDE  00000000  8C000003  64B04600  E8DD0800
               00000000  5E00E8AB  6600C500  1C002B00  E4DD0800
               66DE0800  867D2C00  E8DD0800  FFFF3CD4  0000B88F
               010066DE  08000000  DF0A0000  00000000  00000300
               000006DF  C5000200  66DE0800  00000000  00000000
               00000000  1A8C7400  00000000  00000000  0400D6DD
               080004DF  C500A8C7  2C00BCDD  08003500  E89D3300
               3300B8DE  0800B88F  5E000500  0200C500  000096FF

S570-65669 90-10-06 00:01:27 030767 INT_MON FIVE I
REPT  SM=9   STACK FRAME ENV=OSDSM SRC=FR EVENT=6410
  FUNC ADDR:   H'33663c
  PARAMETERS:  0008DE8A  00000000  00000000  00000000  00000000
               00010000  0000FF83  00000000  1FCC0000  00DACC00
  LOCAL DATA:  0000008C  00050003  00000500  0000C500  00000000
               9BD30000  0C060000  00000000  00000000  010050E1
               C50066DE  08009ADE  6A12EA11  0A001200  10001B00
               55000002  08003CD4  5E009ADE  00000000  0B00D805
               3C663300  3EDE0800  00000000  008C0005  000305DE
               00090380  05002000  02008CDE  00000000  8C000003
               64B04600  E8DD0800  00000000  5E00E8AB  6600C500
               1C002B00  E4DD0800  66DE0800  867D2C00  E8DD0800
               FFFF3CD4  0000B88F  010066DE  08000000  DF0A0000
```

8.5 INTERRUPT ANALYSIS EXAMPLE — SOFTWARE

In the following example, a software update installed a new version of function FCoo_ans(). As shown by this output message, the new address of the function is H'561C5C.

```
S107-281477271 90-11-14 06:36:24 038035 MAINT YRDLYL0-2V1HD0
UPD APPLY - FCoo_ans NEW ADDRESS 0x00561c5c
```

A short time later, an MP WRITE-PROT-ERR occurred. This error, which indicates that a write was attempted to a write-protected address, usually implies a software problem.

For the moment, ignore the FAILING ADDR pointed to by the output message. This is the address to which the program attempted an invalid write not the address of the failing instruction.

Start by locating the value of the PC in the accompanying hardware register dump. Here, the PC contains H'5623ca, which is most likely one or two instructions beyond the actual point of the interrupt. It is necessary to examine the program flow to reconstruct the sequence of events that resulted in the write protect violation.

```
S570-22741062 90-11-14 06:50:32 038046 SINIT YRDLYL0-2V1HD0
REPT SM=8,1 HWLVL=1 SWLVL=SPP MP WRITE-PROT-ERR EVENT=12620 COMPLETED
  SW-ERR FAILING ADDR=H'57001f PROCESS:BG=398,156,PURGED CM=NONE, FG=NONE,,

S570-22741062 90-11-14 06:50:55 038049 LSPPIN YRDLYL0-2V1HD0
REPT SM=8  HARDWARE CONTEXT BASIC LSM EVENT=12620
  68000-REGISTERS:    SSP=H'22bfc     PC=H'5623ca       SR=H'0
            USP=H'aed3e      FP=H'aefee      A5=H'a41c04    A4=H'770840
            A0=H'4e56ffdc    A1=H'74559c     A2=H'cd438     A3=H'6b47ee
            D0=H'fed4        D1=H'644e20     D2=H'0     D3=H'0
            D4=H'21  D5=H'0  D6=H'0  D7=H'c0
  PIC-REGISTERS:       HI      MED     LOW
              IRR:   H'80    H'40    H'ce
              IMR:   H'c0    H'3b    H'c0
              ISR:   H'0     H'0     H'0
```

Request a function update trace (UPD:FTRC), and subtract the starting address of the function from H'5623ca (PC). The remainder will be used as an index into the program listings to locate the point of failure. Since the starting address of the assembly code differs from the starting address of the function on the *5ESS*® switch, the following computations must be performed:

```
H'5623CA   Program Counter (value of PC register)
- 561C5C   Function Starting Address (From UPD:FTRC response)
--------
   H'76E   Offset


  H'224C   Assembly Starting Address (program listings)
+  H'76E   Offset
--------
  H'29BA   Point of Interrupt (program listings)
```

To ensure that this portion of the program listings is in sync with the code that resides on the switch, use generic utilities to dump disassembled code in the area of the interrupt and compare the two:

```
dump:ut, sm=8, addr=h'56238a, l=128, dis;
```

The program listing shows the following:

```
FCoo_ans()
        224c: 4e56 fd52          link     %fp,$-0x2ae
```

```
[40]    2250: 3f3c 0093          move     $0x93,-(%sp)
        .
        .
        .
[345]   299e: 3f3c 0004          move     $0x4,-(%sp)
        29a2: 4eb9 0000 0000     jsr      0x0
        29a8: 548f               addl     $0x2,%sp
[347]   29aa: 2079 0000 0000     moveal   0x0,%a0
        29b0: 0228 00bf 0043     andb     $-0x41,0x43(%a0)
[350]   29b6: 3f3c 001c          move     $0x1c,-(%sp)
        29ba: 303c 008c          move     $0x8c,%d0
        .
        .
        .
```

Examination of the code reveals that the error occurred at local address 29b0 while executing the andb instruction. This instruction ANDs the contents of a data register and an effective address; the results are left in the effective address (A0 + H'43 = H'57001F). It was the attempt to write the results of this operation at address H'57001F that caused the write protect violation. Of course, the analysis has just begun, because the source of the data in register A0 must now be investigated.

```
A0 = 4E56FFDC
    +      43
     --------
     4E57001F
Mask   FFFFFF
     ---------
     0057001F
```

0057001F

is the FAILING ADDR in the output message.

The C code is shown in Exhibit 8.5-1.

Exhibit 8.5-1 — FCoo_ans() Function Example

```
/*  FCoo_ans(reason, dummy)        operator trunk origination answer
 *  Receives control from the FCoo_call function
 *  At this point, setup complete has been received from
 *  the terminating terminal process, the talking path has
 *  been closed, and the operator should be hearing audible
 *  ringing tone. Upon receipt of an answer message from the
 *  terminating process, control will be passed to the
 *  operator origination talk (FCoo_talk) function.
 */
FCoo_ans(reason, dummy)

short reason;
long dummy;  /* needed because OS passes 4 bytes on restart */
{
        struct{
               OSMSGHEAD msghead;
               union{
                      struct   mgREANS          reans;
                      struct   mgCLR_BACK       clb;
                      struct   mgINTERRUPT      intr;
                      struct   mgPATH_REL       rel;
                      struct   mgANS_NOCHG      ann;
                      struct   mgANS_CHG        anc;
                      struct   mgTERM_FAIL      termf;
                      struct   mgTIMER          timer;
                      struct   mgWINK           wink;
                      struct   mgONHOOK         onhk;
                      struct   mgRT_FAIL        rtf;
```

```
                    struct   mgRECALL         recall;
                    struct   mgANSPU          anspu;
                    struct   mgTASUCCESS      tas;
                    struct   mgTAFAIL         taf;
                } text;
        } msg;                    /* message buffer */

       struct{
              OSMSGHEAD msghead;
              union {
                      struct mgRT_REQ rtreq;
                      struct mgRT_GEN rtgen;
                   } text;
              } rr_msg;          /* route request message */

    RET_VAL          fcr;       /* FC return */
    RET_VAL          osr;       /* OS return */
    RET_VAL          pcr;       /* PC return */
    FC_DCDB          dcdb;      /* Digit collection data block */
    ASTSTAMP         tstamp;    /* ASDISCON time stamp */
    FCTGSRPGBK       *tgsr_ptr; /* pointer to tgsr_data structure */

                   /* initialize dcdb and rt_req */
[40]        FCdcdb_init( &dcdb, &rr_msg.text.rtreq, MGRT_REQ );

            /* this process is given control by the operating system */
            /* when a message arrives. */

[46]        msg.msghead.length = sizeof(msg.text); /* initialize length field */
[47]        osr = OSWGETMSG(&msg, 0L); /* ask for message */
[48]        if ( osr != OSMSG ){
[49]        ASINCOMPLETE( FCGETCID(), AS_TCL);
[50]        PHrel ( FCPATH0 );
[51]                FP_TIDLE ( FCSUICIDE );
                    /* does not return */
            }
[55]        switch(msg.msghead.type){
            .
            .
            case MGANS_CHG:
            case MGANS_NOCHG:
            /* terminating party has answered,
             * relay offhook to operator
                     */
[345]               LPsig(PCOFFHK);
                    /* update supervision to off-hook in PCBLA */
[347]               FCSETSSUPV(FCSND_OFF);
                    /* enable operator flash (winks) */
                    LPepulse(PCONHK, (unsigned short) FCI_MIN/10,
[350]                        (unsigned short) FCI_MAX/10, PCWINKON);
```

Software Analysis Guide

CONTENTS PAGE

9. SINGLE PROCESS PURGE (SPP) 9-1
9.1 INTRODUCTION TO SINGLE PROCESS PURGE (SPP) 9-1
9.2 SPP RECEIVE ONLY PRINTER (ROP) OUTPUT 9-1
9.3 SPP EXAMPLE. 9-2

9. SINGLE PROCESS PURGE (SPP)

9.1 INTRODUCTION TO SINGLE PROCESS PURGE (SPP)

A single process purge (SPP) is used when a severe error is detected which prevents the system from continuing to function. An SPP's primary objective is the restoration of a software configuration that can support call processing.

The SPP is the second level of processor initialization after the Return to Point of Interrupt (RPI). It is responsible for terminating (and restarting, if appropriate) a target process or job (such as killing terminal and system processes). Terminal processes are used for event processing (such as a call, terminal maintenance) and do not restart after being purged. A system process exists for a long duration and is not created or terminated dynamically, it can manage requests from more than one terminal process. If a system process is purged, OSDS will recreate it.

SPPs may be triggered by various mechanisms as follows:

- Hardware interrupts
- Hardware resets
- Asserts
- Audits
- Manual request by maintenance personnel.

SPPs may also cause escalation to a high level of recovery (such as, selective initialization). Consult 235-105-250, *System Recovery Manual* for more details concerning recovery escalation.

9.2 SPP RECEIVE ONLY PRINTER (ROP) OUTPUT

As a result of a SPP, the following messages are printed on the ROP:

- Stimulus - `INIT:AM-LVL`, `INIT:CMP-LVL`, `INIT:SM-LVL-EVENT`
- Stack trace - `REPT:STACK-TRACE`
- Stack frame - `REPT:STACK-FRAME`
- Data dump - `REPT:DATA`

  The data dump messages are only printed if the target of the purge was an OSDS process. In which case the purged process's PCB (Process Control Block) and PCBLA (Process Control Block Link Area) are dumped using this message. Otherwise (if OSDS interject was purged), these messages are not printed.

- PMDB dump - `REPT:PMDB`

  The PMDB dump messages are only printed if the target of the purge was an OSDS terminal process, in which case a `PMDB-IN` and `PMDB-OUT` message may be printed to show the contents of the message the process received or was constructing.

For detailed descriptions of these messages, see the 235-600-700, *Input Messages Manual* and 235-600-750, *Output Messages Manual*.

9.3 SPP EXAMPLE

The analysis of an SPP and an assert are very similar. Both consist of analyzing the stack trace information to determine the software causing the ROP output. The main difference is that the root of an assert stack trace will be an assert macro, while the root of an SPP stack trace will be the piece of code causing the SPP. This may be an assert or code causing a processor fault or some other problem. Due to this commonality, the reader is referred to "Stack Trace Debugging" Section 5.1.3, in this manual.

Software Analysis Guide

CONTENTS PAGE

10. AUDIT ANALYSIS 10-1
10.1 AUDITS OVERVIEW 10-1
10.1.1 Application Audits 10-1
10.1.2 *UNIX* RTR System Audits 10-6
10.1.3 Static Data Audits 10-6
10.2 USING AUDITS. 10-10
10.2.1 *UNIX* RTR System Audits 10-10
10.2.2 SODD Audits 10-11
10.2.3 Application Audit Analysis 10-11

LIST OF TABLES

Table 10-1 — Summary of Audit Scheduling Mechanisms 10-4

Table 10-2 — Summary of Audit Scheduling Effects 10-5

Table 10-3 — Audit Input Message Summary 10-5

Table 10-4 — SODD Audit Responses to Input Messages 10-9

Table 10-5 — Error Code: MODATT 10-19

10. AUDIT ANALYSIS

Section 10 provides an overview of audits and the process for analyzing audit messages. Refer to the 235-600-400, *Audits Manual* for more detailed information, including specific audit descriptions.

10.1 AUDITS OVERVIEW

There are three different kinds of audits:

- **Application audits** are programs that verify the consistency of dynamic data used by *5ESS*® switch application code.
- ***UNIX*[1] RTR system audits** are programs that verify the consistency of data associated with the control unit (CU), equipment configuration database (ECD), file manager (FMGR), memory manager (MMGR), and other *UNIX* RTR system data in the administrative module (AM).

  Refer to the *Input Messages Manual*, 235-600-700, and *Output Messages Manual*, 235-600-750, for more information about system audit-related messages.
- **Static data audits** are programs that verify the consistency of static office-dependent data (ODD). For more information about SODD audits, refer to the *SODD Audits Manual* (236-600-410).

10.1.1 Application Audits

References to "audits" in section 10.1.1 refer to application audits only.

Application audits verify the consistency of dynamic data. The existence of an audit does not guarantee that the data structure it checks is consistent all the time, but it does mean that the audit eventually detects inconsistencies and attempts to recover from them.

By detecting and correcting data inconsistencies, audits help prevent problems such as resource failures or data errors, which could cause call processing or other switch activity to fail or run in a degraded manner. By providing a recovery mechanism that can be invoked by asserts or single process purges, audits help prevent escalation to higher levels of initialization.

Generally, audits report all errors they detect on the read-only printer (ROP). The exceptions are due to brevity control restrictions and the printing status of audit message classes on the processor involved. (Refer to the *Input Messages Manual*, 235-600-700 and *Output Messages Manual*, 235-600-750 for more information on brevity control and message classes).

Audit error reports may be followed by one or more reports that include additional debugging data associated with the error being reported. (In the 235-600-400 *Audits Manual*, refer to the error codes listed in the Error Handling section of the Audit descriptions and in the Driver Error code manual page [DRERRCDS] for more information about specific error codes.) When an audit finishes, an audit completion report is produced that states the total number of errors the audit found.

Audits do not take corrective action without reporting an error on the ROP, unless the audit is running as part of a selective initialization or in post-initialization mode.

---

1. Registered trademark of The Open Group in the United States and other countries.

Audit recovery may include requesting a single process purge or the scheduling of other *related* audits (audits of data structures associated with the data being recovered). A common event number is used for all ROP reports associated with the same stimulus. For example, an assert might request an audit that might, in turn, schedule other related audits. The assert reports and all audit reports would contain the same event number. Technicians can use this information when analyzing ROP output.

If an audit is unable to correct an error without human intervention, the audit prints a *manual action* error report on the ROP. Manual action error reports are marked with an `A` in the left margin of the first line of the report. The audit continues to find (and report) the same manual action error until the problem is corrected by office personnel.

10.1.1.1 Error Report Descriptions

Each audit error report description in the *Audits Manual* includes the following sections:

- **Error Description** explains what the error is.
- **Possible Error Effect** describes what might happen if the error went uncorrected.
- **Corrective Action Taken** describes what the audit does to try to correct the error.
- **Manual Action Required** specifies the actions office personnel must take to recover from the error. It contains information only if the audit is unable to correct the error without human intervention; otherwise, if no manual action is needed, it says *None*.
- **Dump Description** describes the data included in the error report. Five fields are always included: *Error Address*, *Bad Data*, *Good Data*, *Logical Key*, and *Dump*. These fields contain information relevant to understanding the error being reported.

  The field names do not always indicate the kind of data they contain. For example, the error address may not be an address, and bad data may not be *bad* data. Also, additional optional data dumps may be provided; if so, then *Dump* is listed as *Dump 1*, and succeeding optional data dumps are listed as *Dump 2*, *Dump 3*, and so on. The error description explains what each field contains for the particular error being reported.

  All data dumps are unformatted and printed in hexadecimal.

10.1.1.2 Error Report Formats

There are two basic types of audit error reports.

- The first is for errors that the audit will attempt to recover itself (see the **Corrective Action Taken** field in the audit error report description):

```
AUD  SM=2  CR  ERROR-CODE=NONEPROC  EVENT=326
ERROR-ADDR=H'33c250  BAD-DATA=H'82026d
LOG-KEY=H'47  GOOD-DATA=H'0
```

- The second is for manual action errors (see the **Manual Action Required** field in the error report description):

```
A  AUD  DAP=OKP  TKQUE  ERROR-CODE=PORT  EVENT=756
   ERROR-ADDR=H'0  BAD-DATA=H'0
   LOG-KEY=H'a10e0002  GOOD-DATA=H'0
```

Refer to the *Output Messages Manual*, 235-600-750, for more complete descriptions of these and other audit output messages.

10.1.1.3 Audit Environments

Application audits run in the following *5ESS* switch environments:

- CMP (communications module processor)
- MH (message handler)
- MSGS (message switch)
- OKP AM (operational kernel process in the administrative module)
- ONTC (office network and timing complex)
- PH (protocol handler)
- PI (packet switching interface processor)
- QGP (quad gateway processor)
- SM (switching module)
- SMKP AM (switch maintenance kernel process in the administrative module)

Refer to the APP:AUDITS appendix in the Appendixes section of the *Output Messages Manual*, 235-600-750, for a list of audits that exist in each environment.

Most audits are *intraprocessor*; that is, they only check data within one of the environments. Intraprocessor audits with the same name may exist in more than one environment, but each instance is independent of the others. For example, the OKP AM, SMKP AM, and SM environments each have a process control block (PCB) audit that checks the same kind of data. However, each version is independent of the others and makes checks only for its own environment.

All other audits are *interprocessor*; that is, they check data between two of the environments (typically the CMP and SM environments with the CMP in control). In this case, the same audit name in different environments refers to different pieces of the same audit. There are only a few interprocessor audits.

10.1.1.4 Audit Scheduling

Audits run in different modes (refer to Table 10-1):

- **Routine audits** are scheduled automatically when the processor has no higher priority work. They are *segmented* (that is, they periodically give up control of the processor to let higher priority work be performed).
- **Elevated audits** run only by request. They may be scheduled by an input message (refer to section 10.1.1.5 for more information), or by internal requests (from asserts, internal interfaces, single process purges, or for related recovery when another audit has detected an error).

  Elevated audits run segmented and preempt execution of any routine audits.
- **Directed audits** are run only by an internal request from an assert, through an internal interface, or by escalation of a routine or elevated audit through an audit failure code.

Directed audits run *unsegmented*; that is, they do not give up control of the processor until they are finished, and preempt execution of both routine and elevated audits.

Directed audits are only used for system-critical resources. Because they run unsegmented, all other processing (including call processing) is blocked until they are finished.

- **Selective initialization**, or **post initialization mode audits**, are a fixed set (and sequence) of audits scheduled by system integrity during or immediately after a selective initialization to correct errors in dynamic data that is not being reinitialized. The audits are run unsegmented and do not generate any reports. This is the only exception to the rule that audits do not take any action without reporting an error.

Audits can be inhibited (prevented from running) by an input message. However, even when an audit is inhibited, it can be executed manually.

If an audit is scheduled through an input message, an audit completion report is always sent on the ROP. In all other cases, a report is sent only if the audit has detected an error. Even if an assert schedules an audit, there is no evidence on the ROP that the audit was run if the audit did not detect any errors.

If an audit is scheduled by an internal request, all reports from that audit are associated with an event number supplied by the requester. In particular, an audit scheduled by an assert or a single process purge has the same event number as the assert or single process purge.

Related audits also use the same event number. When an audit completes, if it has found errors, audits of data related to the data just recovered are scheduled (for example, the related structure may have a linkage to the structure just recovered). Related audits run elevated and use the same event number as the original audit. Since related audits are *not* scheduled when an audit completes with no errors, this mechanism causes recovery to ripple out from the original stimulus and die out when recovery actions have been successful.

Table 10-1 — Summary of Audit Scheduling Mechanisms

Scheduling Source	Routine Audits	Elevated Audits	Directed Audits
Input Messages	*No*	*Yes*	*No*
Asserts	*No*	*Yes*	*Yes*
Single Process Purges	*No*	*Yes*	*No*
Related Audits	*No*	*Yes*	*No*
Internal Interfaces	*No*	*Yes*	*Yes*
Audit Failure Codes	*No*	*No*	*Yes*
Automatic Scheduling	*Yes*	*No*	*No*

Table 10-2 summarizes the effect of each type of audit scheduling on other switch activity.

Table 10-2 — Summary of Audit Scheduling Effects

Routine Audits	Elevated Audits	Directed Audits	Selective Initialization
None.	Minimal background delays.	Disables call processing until the audit completes.	Helps preserve stable calls and maintenance states through a selective initialization.

10.1.1.5 Input Message Information

The Audit Input Message Summary, Table 10-3, lists the input messages available for application audits in each environment.

Table 10-3 — Audit Input Message Summary

Input Message	Operation

`AUD:{audit},SM=#`
`AUD:{audit},cmp={0|1}`
`AUD:{audit},env={OKP|SMKP}`
	Run an audit

`INH:AUD={audit},SM=#`
`INH:AUD={audit},CMP=#`
`INH:AUD={audit},ENV={OKP|SMKP}`
	Inhibit an audit[a]

`ALW:AUD={audit},SM=#`
`ALW:AUD={audit},CMP=#`
`ALW:AUD={audit},ENV={OKP|SMKP}`
	Allow an audit[a]

`STP:AUD,SM=#`
`STP:AUD,CMP=#`
`STP:AUD,ENV={OKP|SMKP}`
	Stop execution[b]

Note(s):
a. `ALL` can be substituted for `audit`. This will inhibit or allow all audits in the processor with one input message.
b. Will only stop execution of the currently running audit. If no audit is currently running, the input message has no impact.

10.1.1.6 Audit Error Report Analysis

Occasionally, an audit detects (and reports) an error that requires human intervention to be corrected. Until office personnel have taken the required corrective action, the error continues to be re-detected and re-reported. Typically, the intervention needed is a change to the ODD.

Errors that are known to require human intervention, or manual action, for correction are indicated by an `A` in the left margin of the first line of the audit error report. By examining the audit report(s), ODD population rules, and the ODD itself, office personnel can determine exactly what should be done to correct the inconsistency.

Other audit reported errors (that is, those not marked with the `A`) are almost always corrected by the audit itself. In cases where the audit is *not* able to correct the error

(or the error is repeatedly reintroduced), the same audit(s) reports the same error(s) on the same data across multiple events or multiple audit invocations. Such repetitions should not be confused with either of the following:

- single events that include more than one instance of a particular error report or set of error reports
- multiple appearances of the same audit reporting different errors in each appearance, or reporting the same errors on different data each time

As long as reports associated with an event eventually stop appearing and do not recur, recovery has been taken automatically. Technicians should refer questions or concerns about particular sets of audit error reports to their next line of support.

10.1.2 *UNIX* RTR System Audits

UNIX RTR system audits are run under the control of the *UNIX* RTR operating system.

System audits are identified by the family name, the family number, and the audit instance name.

System audits are categorized by the nature of the audit such as file manager, memory manager, and message buffer. Each audit category is referred to as a family, for example, the file manager (FMGR) family. The audit also contains a family number such as FMGR 3 or FMGR 5 (file table audit and internal capability table audit, respectively.)

The audit instance name is necessary when a single audit is used to audit more than one software structure. For example, the file system audits have an instance name for each file system. The instance name is the name of the file system to be audited. An audit with only one instance needs only the family name and family number for identification. An audit with more than one instance must be identified by family name, family number, and instance name.

Refer to the *Input Messages Manual*, 235-600-700, and *Output Messages Manual*, 235-600-750, for more information about system audit-related messages.

10.1.3 Static Data Audits

The static ODD (SODD) audit verifies that the data stored in the base relations of the ODD conform to the database population rules. (Refer to the 235-600-410, *Static Office Dependent Data [SODD] Audits Manual*.) Execution of the audit is initiated by a manual input message. The audit can also be invoked directly to verify a single database component. Data errors detected by the audit are logged in files. A manual input message is required to display the contents of an error file in a readable form. Data errors are not automatically corrected.

The EXC:AUD-SODD input message invokes the SODD audit. The audit is activated either at the time of the input message, or at the time specified in its schedule parameter block. Once activated, the execution of the audit proceeds until the allocated duration has elapsed. The audit process is then suspended and remains dormant until the input message is re-executed. The audit process then resumes from the point at which it was suspended. The normal operational scope of the audit is to run all available components.

The SODD audit is executed in a cyclic mode. During a single cycle, all the relations within the scope of the audit are examined. When the end of the cycle is reached, an audit completion report is printed and a new cycle is started.

In a typical scenario, the audit process is activated periodically by a scheduled input message, and during each period of activity, the execution of the audit cycle is continued from the previous point of suspension. The duration of each of the active periods is specified by the originating input message, and the times of day at which the periods start are specified in a (standard) schedule parameter block entered with the input message.

Controlled by a process operating in the AM, the audit is applied to the data bases that are currently active in the AM and the SMs. Unless restrictions are imposed, all tuples and parameters for which audit products are available will be audited with respect to the applicable population rules.

When the audit is applied to a partitioned base relation that is distributed among the AM and the SMs (or among the SMs), the required audit processes are executed concurrently on the AM. One instance of a redundant relation is separately audited with respect to the population rules, and the consistency of the redundant data is not verified. During the auditing of a relation, the integrity of the data involved is not influenced by RC/V activity.

The AM collects and logs error data. For each 24-hour period, two error logs are maintained: one for the current audit cycle, and the other for the previous cycle. The erroneous data is not corrected.

When a specific component of the audit is requested manually, the request is queued behind any previous requests. The subsequent processing of the manual request queue depends on the situation prevailing at the time. If the SODD audit is active (and there is sufficient time remaining), manually requested audits are executed as soon as the current audit function has finished. If the SODD audit is inactive, manually requested audits are executed with the highest priority when the audit activity is resumed.

After an audit has been executed by manual request, it will not be executed again in the current cycle. When made in the preemptive mode, a manual request stops the active audit (provided that it was not requested manually) and invokes the requested audit function immediately. The preemptive mode also enables a manually requested audit to execute when the entire SODD audit is inhibited.

10.1.3.1 Operational Aspects

To execute a complete audit cycle, an interval of audit activity is initiated periodically by a scheduled input message. The duration of the interval is determined by an input message parameter, and the starting time is specified by a schedule parameter block attached to the input message. Note that a typical audit cycle requires several weeks to complete, depending on the size of the office. There is only a logfile for the current cycle and the previous cycle. Therefore, the technician must check the logfile each cycle. When there are errors, the technician must store the logfile before it is overwritten.

The `Glmaxauds` parameter controls a number of allowed concurrent audit processes (one per processor). The concurrent processors are different processors for the same relation component. The valid range of the parameter is 1 - 3, and the default value, which is 3, cannot be changed with RC/V. This parameter's value is checked when the SODD control process is restarted by user level automatic restart process (ULARP).

A report is printed when execution of an audit cycle or a manually requested audit component has completed, or when auditing is suspended at the end of an interval. This report is also printed when the execution of the audit is stopped manually. Suspend and resume reports occur when inhibit and allow messages (without COMP) are processed.

Note that the SODD audit has no error correction mechanism. Errors detected by the audit must be corrected by hand, using RC/V procedures.

Table 10-4 shows what the SODD audit does with input messages in various circumstances.

Table 10-4 — SODD Audit Responses to Input Messages

Input	State	Action
general INH	preemptive audit running	set flag to stop further dispatches
general INH	all other states	set flag, kill any running product
specific INH	this component running	set component flag, kill running
specific INH	all other states	set component flag
general EXEC	DUR < current end time	NG ack
general EXEC	DUR > current end time	reset duration
general EXEC	general INH	reset duration, send warning
specific EXEC	expired duration	queue component, send warning
specific EXEC preempt	manual audit running	queue component, send warning
specific EXEC preempt	nonmanual audit running	queue component, kill running
specific EXEC nonpreempt	an audit running	queue behind previous manual audits
specific EXEC nonpreempt	general INH	queue component, send warning
specific EXEC preempt	general INH	run component
specific EXEC nonpreempt	specific INH of same	NG ack
specific EXEC preempt	specific INH of same	NG ack
general STP	is a current duration	kill duration, kill running audit
general STP	no current duration	NG ack
specific STP	this component running now	kill and requeue, "killed" msg[a]
specific STP	not running now + queued as manual	requeue, "canceled" msg[a]
specific STP	not running now + not queued as manual	NG ack

Note(s):

a. Requeue using this algorithm: if not run this cycle, requeue as part of a full audit; else, if run successful once or run more than once, unqueue; else, requeue at tail.

In the case of CP-ONLY and REDUNDANT relations, the log shows the completion of processors 0 and 193 (193 means redundant). Any error messages coming from the execution of these processors show processor –1 (the database manager uses –1 to mean any processor).

Specifying PROC in an input message for a redundant component has *no* effect on the behavior of the component audit. The input messages give an OK acknowledgment, but have no effect.

10.2 USING AUDITS

The audit system detects, isolates, and corrects errors in the data structures involved in *5ESS* switch software. It corrects broken linkages, restores lost data resources, and helps identify the location of problems in the source code. The more that is known about the structure of the *5ESS* switch software, particularly of the modified relational database, the more effectively audits can be used for troubleshooting. When used for maintenance work in conjunction with other diagnostic tools (such as recent change check programs and *UNIX* system diagnostic reports), audit reports, along with other reports, can help the switch personnel reconstruct the series of events that caused the problem.

The audit system begins cycling through the audits when the system software is initialized. When there is a heavy call processing load, however, the audit cycle slows down. The operator can also execute a given audit, causing it to run sooner than it would have as part of the cycle. To execute an audit, the operator types an input message requesting the audit. When the audit completes, a summary report is printed. If any errors are discovered, reports describing the errors are also printed. Error reports and summary reports are also generated when the audit runs as part of the cycle. These reports may be printed on the ROP or directed to a logging file. Unlike the reports printed due to operator requests, reports for audits executing as part of the cycle are only generated when the audit discovers errors. If manual action is required to correct an error, the report is printed unconditionally on the ROP with an A in the priority of action field.

10.2.1 *UNIX* RTR System Audits

10.2.1.1 Purpose

Section 10.2.1 explains which input messages query and request software integrity services using *UNIX* real-time reliable (RTR) system audits. *UNIX* RTR system audits are a part of the system integrity subsystem which is a collection of programs, processes, commands, database records, and performance records that work together to ensure the reliability of the operating system software.

10.2.1.2 Types of *UNIX* RTR System Audits

There are seven input messages that give the user the following audit capabilities.

- Allow the routine execution of audits (ALW:AUD).
- Request audits (AUD). There are twelve types of audits that can be requested.

  CUMEM = control unit memory comparison audit

  CUSTAT = control unit status audit

  ECD = equipment configuration database audit

  ECDOWN = equipment configuration database owner audit

  FMGR = file manager audit

  FSBLK = file system block audit

  FSCMPT = file system compaction audit

  FSLINK = file system link audit

MMGR = memory manager audit

MSGBUF = message buffer audit

PMS = plant measurements system database audit

PROAD = process administration audit

For more information on these audits, refer to the Audit Descriptions section in the *Audits Manual* (235-600-400) and the input message pages in the *Input Messages Manual* (235-600-700).

- Inhibit the routine execution of audits (INH:AUD)
- Determine the status of audits (OP:AUD)
- Extract error information for audits (OP:AUDERR)
- Analyze data produced by file system audits (OP:FNAME)
- Stop audits from execution (STOP:AUD)

For information on the use of these input messages, refer to the *Input Messages Manual* (235-600-700).

10.2.2 SODD Audits

10.2.2.1 Purpose

The SODD audit detects errors in the static office-dependent database (SODD). It verifies relations, performs cross checks of attributes, and reports findings to switch personnel for resolution. For more information about SODD audits, refer to the *SODD Audits Manual* (236-600-410).

10.2.3 Application Audit Analysis

10.2.3.1 Purpose

Application audits verify the consistency of dynamic data. In all cases, the goal is to prevent or to help recover from problems that can cause switch functionality to fail or to run in a degraded manner.

10.2.3.2 Types of Audit Error Reports

There are two basic types of audit error reports.

- Errors that the audit will attempt to recover
- Errors that require manual action

Occasionally, in the course of reading static data needed to do a check, an audit will detect static data inconsistency (for example, between two static relations). In that case, the audit will report the error using a "manual action" error code. The error report is printed with an A in the left column of the first line to indicate that action is needed by switch personnel. The description in the *Audits Manual* provides specific information on what data is in error. These errors will be reported every time the audit runs until the static data is manually corrected.

10.2.3.3 Types of Data Audited

Not every piece of dynamic data is checked by an audit. Criteria for auditing include:

- What data errors are possible
- What the effect of an error is, including the severity and scope

- What other mechanisms might detect and/or correct the error
- If the condition can be detected by an audit
- If the condition can be corrected by an audit

The types of data checked will often determine the audit name. For example, the PCB audit checks process control blocks. The PORTLA audit checks tuples of the RLPORTLA relation.

Note: Because data linkages are audited from both directions and sometimes other consistency checks are appropriate, multiple audits may reference the same pieces of data.

10.2.3.4 Types of Audit Checks

Audit checks can be categorized by type:

- Linkage consistency
- Data key validation
- Resource ownership and availability
- Dynamic vs. static data consistency
- Hardware vs. software consistency
- Other data consistency
- Data configuration consistency (such as "parent/child" state consistency)
- Stuck in a transient condition
- Quarantine (recover an error detected elsewhere)

To determine the type of error, switch personnel must refer to the audit description and should not make assumptions based on the error code used in the report.

10.2.3.5 Audit Drivers

An audit driver is a set of code shared by multiple audits that controls segment breaks, performs syntactic checks, schedules semantic checks, and handles errors. Because the code is shared, a common set of error codes can be output by many different audits. If an audit calls a driver, the following message is included in the audit description:

See Driver Error Codes (DRERRCDS), located in the Audit Drivers section of the 235-600-400, Audits Manual for any error codes not found below.

10.2.3.6 Analyzing Audit Reports

10.2.3.6.1 Analysis Procedure

When attempting to reconstruct a scenario that includes audit reports, switch personnel should consider all available information about known software or hardware problems, activity on the switch when the problem first appeared (such as recent change, growth or degrowth procedures, diagnostics, and software update applications).

This discussion should be considered a starting point. It is not (and could not be) exhaustive. The most effective way to learn to debug is to practice. Refer to "Analysis Examples," section 10.2.3.8 for more detailed examples.

Analysis of audit reports generally includes the following steps:

1. Understanding the meaning of the report.
2. Determining the source of the audit report.
3. Placing the report in context.

10.2.3.6.1.1 Understanding the Meaning of the Report

Look up the error description in the *Audits Manual*. Find the section covering the audit reporting the error, and search the alphabetical list of error code descriptions for the one that was reported.

Note: In some cases, the same error report is used in the same way by multiple audits and so is described in a separate section of the manual (refer to the "Audit Drivers" section). In that case, the audit sections will each have a reference to the shared section.

The specific reason for the report may depend on data included in the report. The "Extracting Data from an Audit Report," Section 10.2.3.6.2.1 describes how to extract such data from audit reports.

10.2.3.6.1.2 Determining the Source of the Audit Report

When analyzing audit reports, switch personnel should first try to determine the source of the audit. The audit message class can help. Audit message classes include:

`AUDT` = Audits run due to a manual request

`AUDTMON` = Audits scheduled as part of an ongoing event

`AUDTFST` = Audit error reports that are the start of a new event

Examples:

1. If the message class of the report is `AUDTMON`, the audit report must be a side effect of something else such as an assert, single process purge, or other audit.
2. If the message class is `AUDTFST`, the audit report is the start of a new event. However, even here it is possible that the report is due to a problem that was also detected by an assert (which did not happen to schedule the audit) or due to other audit recovery.

10.2.3.6.1.3 Placing The Report in Context

Often, audit error reports are just part of a larger set of reports. Using the data provided in the audit reports to reconstruct the relationships between the various data structures before and after each audit error can be very useful.

The following sections discuss some common scenarios.

10.2.3.6.1.3.1 Audits Caused by an SPP or Assert

An assert may fire and schedule an audit that finds errors. As part of its recovery, the audit may purge a process. The SPP code may then schedule additional audits, and so on.

The reports from all of these actions are typically tied together by a single event number (the exceptions are due to multiple triggers for the same audit and missing related audit specifications). For various reasons (such as differing message classes and priorities), messages do not always print in the same order that the trigger events

occurred. Therefore, it is important to try to reconstruct the sequence of events on the switch that resulted in the reports that were generated.

Example:

The audit report in question is an SMXRF audit DEADPID error and the dead process found in the rlSMXRF tuple was purged in an earlier event. In that case, it is safe to conclude that the audit error was a side effect of the SPP and further investigation should focus on the SPP itself.

10.2.3.6.1.3.2 Recurring Audits

Sometimes, the same set of audits may report errors over and over. There are many possible causes for this. There may be static ODD inconsistencies causing two or more audits to view the same data differently, causing them to "roll" back and forth, each undoing the recovery the other has done. Inconsistencies between the database dictionaries and dynamic access relation head tables can also cause recurring audits.

10.2.3.6.1.3.3 Sporadic Recurring Audits

In other scenarios, the same audit may report errors sporadically but consistently over time. These cases can be difficult to tie to any specific event. Here again there are many possible causes, including application code problems in rarely-used legs of code, transient or "race" conditions between the audit and application code, or even wild writes from some unrelated piece of code. If a race condition is suspected, one option is to inhibit the audit in question for some time and then request it manually. If the number of errors it then finds indicates that they have been accumulating, there probably isn't a race condition. If the audit finds few or no errors, it is possible that a race condition exists.

10.2.3.6.1.3.4 Recurring Errors

It is also possible that the same error is introduced over and over. If the appearance of the error can be tied to some other event (such as diagnostics, an assert, or a hardware failure), it is possible that the other event is the root cause. However, sometimes a correlation is coincidental and not due to cause and effect.

10.2.3.6.1.3.5 Non-recurring Errors

Sometimes an audit report cannot be tied to another event (such as an assert, SPP, or other audit recovery) and does not recur. Even if it is not possible to determine the root cause, switch personnel can still use the *Audits Manual* to understand the meaning of the report and keep that information in mind when investigating other problems.

10.2.3.6.2 Analysis Aids

10.2.3.6.2.1 Extracting Data from an Audit Report

Audit reports include three different kinds of reports: the error report, optional data dump reports, and a completion report. The data dump reports provide unformatted data (such as the contents of a relation tuple) that may be needed for debugging. Depending on the software release, data type, and other factors, the dump report may or may not include a label for the data being dumped. However, the error code description in the *Audits Manual* (235-600-400) will specify the data provided by each dump report.

To interpret the audit error report, the corresponding entry must be found in the *Audits Manual*. The meaning of the "Error Address," "Bad Data," "Good Data," and

"Logical Key" fields may vary from error to error. The tags (such as "Bad Data") are not meaningful in and of themselves.

To interpret the dump, it is necessary to have the data layout. This information is available in the *Dynamic Data Manual* (235-600-2xx).

For example, here is an rlCHDB tuple dump:

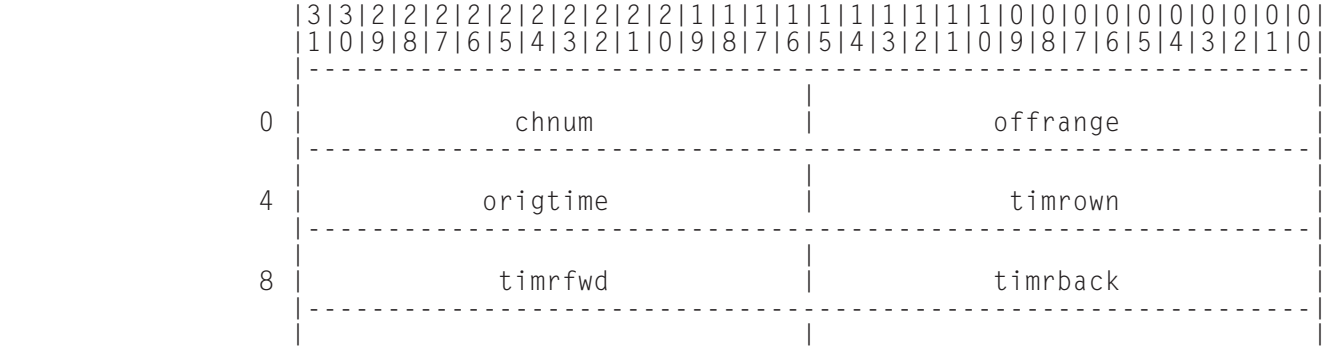
```
AUD SM=7 CHDB DUMP 1 OF 1 EVENT=198
 KEY=rlCHDB TUPLE WHERE
     <the_key>
     =<H'14b>
 BLOCK-ADDR=H'4c904c
 014B0000 00000000 00000000 00000000 00070000 00000000 00000000 00010000
 00000000 00000000 00000000 00000001 01000000 00001000 00000000 00000000
 00000000 00000000 00000000 00000000 00E00000 00001000 00000000 11050000
```

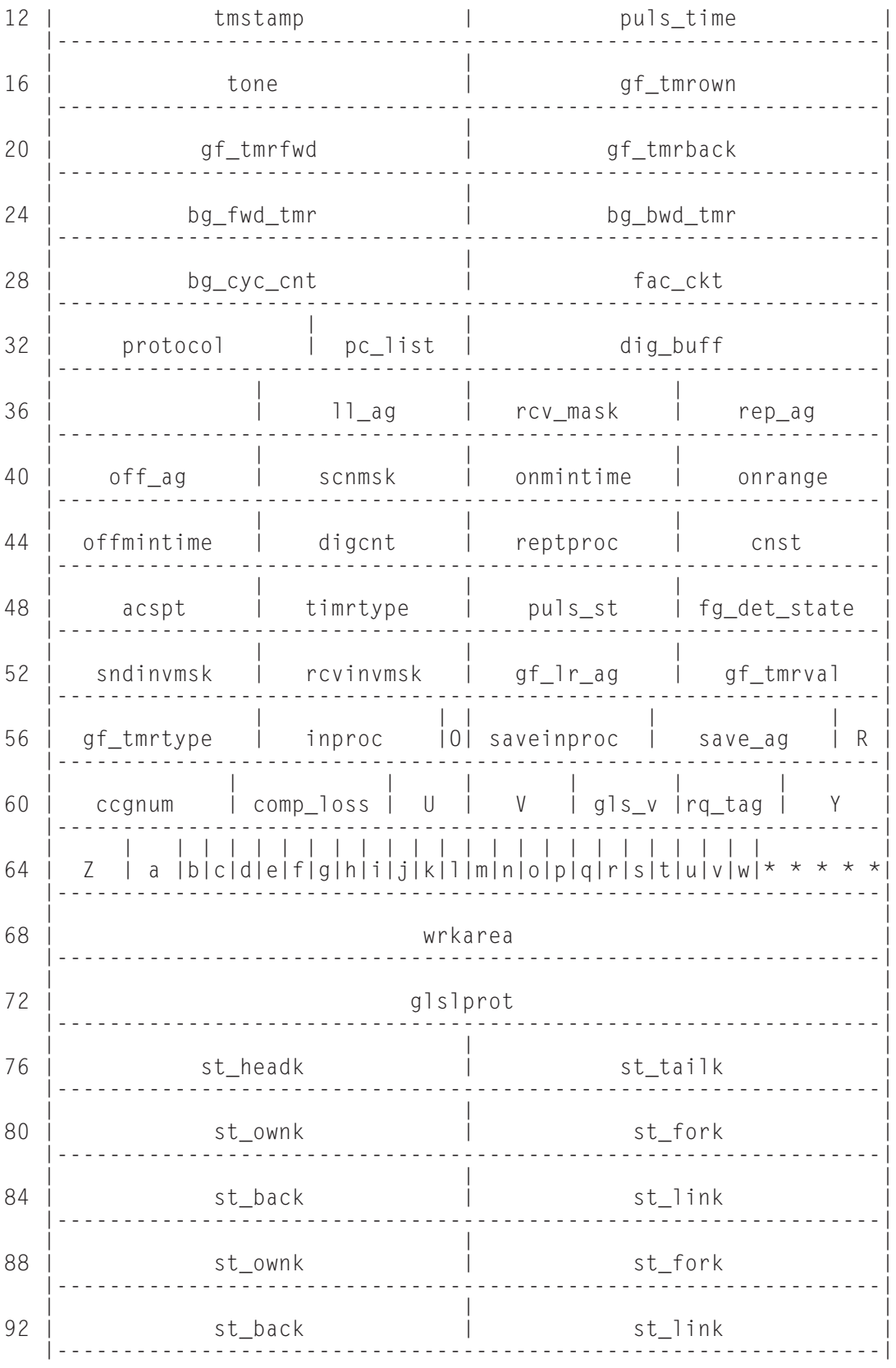
Note that the `BLOCK-ADDR` value is the beginning address of the data being dumped.

From the manual, the layout of an rlCHDB tuple looks like:

```
     struct rlCHDB          rlCHDB
(a)  uint                   rlCHDB.chnum
(b)  int                    rlCHDB.offrange
(c)  uint                   rlCHDB.origtime
(d)  int                    rlCHDB.timrown
(e)  int                    rlCHDB.timrfwd
(f)  int                    rlCHDB.timrback
(g)  uint                   rlCHDB.tmstamp
(h)  uint                   rlCHDB.puls_time
(i)  DMTONE (enum)          rlCHDB.tone
(j)  int                    rlCHDB.gf_tmrown
(k)  int                    rlCHDB.gf_tmrfwd
(l)  int                    rlCHDB.gf_tmrback
(m)  uint                   rlCHDB.bg_fwd_tmr
(n)  uint                   rlCHDB.bg_bwd_tmr
(o)  uint                   rlCHDB.bg_cyc_cnt
(p)  uint                   rlCHDB.fac_ckt
(q)  uint                   rlCHDB.protocol : 10
(r)  DMLIST (enum)          rlCHDB.pc_list : 6
(s)  char                   rlCHDB.dig_buff [3]
(t)  char                   rlCHDB.ll_ag
(u)  char                   rlCHDB.rcv_mask
(v)  char                   rlCHDB.rep_ag
(w)  char                   rlCHDB.off_ag
(x)  char                   rlCHDB.scnmsk
(y)  char                   rlCHDB.onmintime
(z)  char                   rlCHDB.onrange
(A)  char                   rlCHDB.offmintime
(B)  char                   rlCHDB.digcnt
(C)  char                   rlCHDB.reptproc
(D)  char                   rlCHDB.cnst
(E)  char                   rlCHDB.acspt
(F)  char                   rlCHDB.timrtype
(G)  char                   rlCHDB.puls_st
(H)  char                   rlCHDB.fg_det_state
(I)  char                   rlCHDB.sndinvmsk
(J)  char                   rlCHDB.rcvinvmsk
(K)  char                   rlCHDB.gf_lr_ag
(L)  char                   rlCHDB.gf_tmrval
(M)  char                   rlCHDB.gf_tmrtype
(N)  DMINPROC (enum)        rlCHDB.inproc : 7
(O)  DMBOOL (enum)          rlCHDB.reseizflag : 1
(P)  DMINPROC (enum)        rlCHDB.saveinproc : 7
```

```
(Q)  uint                  rlCHDB.save_ag : 7
(R)  DMPCSUPV (enum)       rlCHDB.supvchg : 2
(S)  uint                  rlCHDB.ccgnum : 7
(T)  ulong                 rlCHDB.comp_loss : 6
(U)  DMAUQUAR (enum)       rlCHDB.quarstate : 3
(V)  uint                  rlCHDB.snd_mask : 4
(W)  uint                  rlCHDB.gls_v : 4
(X)  uint                  rlCHDB.rq_tag : 4
(Y)  uint                  rlCHDB.rq_new_tag : 4
(Z)  DMGRPTYPE (enum)      rlCHDB.trk_dir : 3
(a)  uint                  rlCHDB.ireseizflag : 2
(b)  DMBOOL (enum)         rlCHDB.ccstrk : 1
(c)  DMBOOL (enum)         rlCHDB.brgch : 1
(d)  DMBOOL (enum)         rlCHDB.dcstrk : 1
(e)  DMBOOL (enum)         rlCHDB.metallic : 1
(f)  DMBOOL (enum)         rlCHDB.mctqing : 1
(g)  DMBOOL (enum)         rlCHDB.msg_lost : 1
(h)  DMBOOL (enum)         rlCHDB.osps_call : 1
(i)  DMBOOL (enum)         rlCHDB.opcat : 1
(j)  DMBOOL (enum)         rlCHDB.bit_inv : 1
(k)  DMBOOL (enum)         rlCHDB.bit_mskng : 1
(l)  DMBOOL (enum)         rlCHDB.mm_active : 1
(m)  DMBOOL (enum)         rlCHDB.met_trk : 1
(n)  DMBOOL (enum)         rlCHDB.pc_flag : 1
(o)  DMBOOL (enum)         rlCHDB.gf_active : 1
(p)  DMBOOL (enum)         rlCHDB.ack_alw : 1
(q)  DMBOOL (enum)         rlCHDB.prg_alw : 1
(r)  DMBOOL (enum)         rlCHDB.met_ans : 1
(s)  DMBOOL (enum)         rlCHDB.pol_req : 1
(t)  DMBOOL (enum)         rlCHDB.tsi_ram : 1
(u)  uint                  rlCHDB.rep_f : 1
(v)  uint                  rlCHDB.ll_f : 1
(w)  uint                  rlCHDB.outg_f : 1
(x)  ulong                 rlCHDB.wrkarea
(y)  long                  rlCHDB.glslprot
     DMSETOWN (struct)      rlCHDB.chmmo
(z)  uint                  rlCHDB.chmmo.st_headk
(A)  uint                  rlCHDB.chmmo.st_tailk
     DMSETMEM (struct)      rlCHDB.chubm
(B)  uint                  rlCHDB.chubm.st_ownk
(C)  uint                  rlCHDB.chubm.st_fork
(D)  uint                  rlCHDB.chubm.st_back
(E)  uint                  rlCHDB.chubm.st_link
     DMSETMEM (struct)      rlCHDB.chccm
(F)  uint                  rlCHDB.chccm.st_ownk
(G)  uint                  rlCHDB.chccm.st_fork
(H)  uint                  rlCHDB.chccm.st_back
(I)  uint                  rlCHDB.chccm.st_link
```




12
tmstamp
puls_time
16
tone
gf_tmrown
20
gf_tmrfwd
gf_tmrback
24
bg_fwd_tmr
bg_bwd_tmr
28
bg_cyc_cnt
fac_ckt
32
protocol
pc_list
dig_buff
36
ll_ag
rcv_mask
rep_ag
40
off_ag
scnmsk
onmintime
onrange
44
offmintime
digcnt
reptproc
cnst
48
acspt
timrtype
puls_st
fg_det_state
52
sndinvmsk
rcvinvmsk
gf_lr_ag
gf_tmrval
56
gf_tmrtype
inproc
O
saveinproc
save_ag
R
60
ccgnum
comp_loss
U
V
gls_v
rq_tag
Y
64
Z
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
* * * * *
68
wrkarea
72
glslprot
76
st_headk
st_tailk
80
st_ownk
st_fork
84
st_back
st_link
88
st_ownk
st_fork
92
st_back
st_link

With this information, we can find the value of any attribute in the rlCHDB tuple. For example, the tone field is 16 bits long, begins at offset 16, and has a value of 7.

10.2.3.6.2.2 Physical vs. Logical Key Values

The relational database used in the administrative module (AM), switching module (SM) and communications module processor (CMP) environments stores the keys of dynamic relations in two different ways. In the key attributes of the relation, a *logical* value is stored. In the linkage attributes, the *physical* value is stored. The two values are related by a constant offset value defined for the relation:

Logical Key = Physical Key – Offset

Physical keys are non-negative (0 and greater) values. Logical keys may be either positive or negative. A negative logical key indicates that the tuple has been reserved to act as a head cell instead of a data tuple.

10.2.3.7 Manually Scheduling an Audit

10.2.3.7.1 Running an Audit with an Input Message

Although audits run continually, the operator may occasionally want to run specific audits to help locate a problem in either the software or the hardware.

- If a problem is suspected with a particular data structure, the audits associated with the data structure should be run to obtain data dumps that may be analyzed in detail to isolate the fault.
- If a hardware problem is suspected, an audit report may be useful for identifying the fault.

Using audits to discover hardware and software problems is a skill. Proficiency comes with practice and a knowledge of the overall software system. In general, however, the user should be alert to problems in the system revealed by recurring audit reports and assert messages.

To run specific audits, the operator enters an audit input message. The operator may have to turn off brevity controls and logging before running the audit to see all the audit output. Brevity may be turned off for a message class (for example, AUDTFST) or for a processor. The operator should not leave brevity off for too long or the logging file may become overloaded.

Refer to the APP:COMMAND-GRP appendix in the Appendixes section of the *Input Messages Manual* (235-600-700) for a list of all input messages associated with the AUDIT command group.

10.2.3.7.2 System Response to Input Message

If all the information is entered correctly and the system is fully operational, the system acknowledges requests for specific audits with OK or PF (printout follows). If there is a problem, a response of NG (no good) plus a short explanation prints. The system response typically prints within five or ten seconds. For more information on system response to requests for specific audits, refer to the APP:AUD appendix in the Appendixes section of the *Input Messages Manual* (235-600-700).

10.2.3.8 Analysis Examples

Several example analyses are given in this section. Although these examples may refer to real audits and real error codes, they may not reflect the current code or documentation for a given release. They are meant to illustrate scenarios and

troubleshooting techniques. Switch personnel should use the current version of the *Audits Manual* (235-600-400) when attempting to analyze audit reports.

Note: In the following examples, H' refers to a hexadecimal number.

10.2.3.8.1 Case History One: Manual Action Error

In this example, the DD audit repeatedly prints the following error report:

```
A AUD ENV=OKP DD ERROR_CODE=MODATT EVENT=756
   ERROR-ADDR=H'0125a324 BAD-DATA=H'15
   LOG-KEY=H'32          GOOD-DATA=H'32
```

The A in the left margin indicates that this error requires manual action for recovery. This explains why it is reported again and again. It will come out every time the audit runs until the static data is fixed.

The *Audits Manual* (235-600-400) entry for the DD audit, error code MODATT is reproduced here for reference.

Table 10-5 — Error Code: MODATT

Error Code:	**MODATT**
Error Description:	A remote switching module (RSM) from static relation RLHSMRSM (rlHSMRSM.rsmnum[index]) has no corresponding tuple in the RLMODATT relation. That is, a database read failed for relation RLMODATT using rlMODATT.module = rlHSMRSM.rsmnum[index] where the RSM number was valid (!= 0).
Possible Error Effect:	The RSM cannot be made operational.
Corrective Action Taken:	None.
Manual Action Required:	Either RLHSMRSM is corrupt (it contains an incorrect RSM number) or the RSM has not been fully specified in the static ODD. Technicians should determine if the RSM number corresponds to a real SM and either remove the reference from RLHSMRSM or add all missing static ODD via recent change or ODBE. See poprules for RLMODATT and RLHSMRSM for more information.
Dump Description:	
Error Address =	Index into rlHSMRSM.rsmnum[ ].
Bad Data =	NA.
Good Data =	The host SM (HSM) number. This is the key to the rlHSMRSM tuple (rlHSMRSM.hsmnum).
Logical Index =	The RSM number. This is the key to the rlMODATT tuple (rlMODATT.module) for which the database read failed.
Dump =	The rlHSMRSM tuple.

Looking at the audit description in the *Audits Manual* and the data from the error report, the problem can be identified:

1. The static RLHSMRSM relation indicates that H'15 is a valid remote switching module (RSM) number for host switching module (HSM) H'32.
2. The rlMODATT tuple where the key attribute *module* = H'15 does not exist according to the database manager.
3. The static RLHSMRSM and RLMODATT relations are inconsistent. Either SM H'15 is a valid RSM and the rlMODATT tuple for it must be inserted, or it is not a valid RSM number and the rlHSMRSM tuple keyed by *hsm* = H'32 must be updated to reflect that.

10.2.3.8.2 Case History Two: Audits Scheduled from Other Events

At times, audits are scheduled to run elevated as part of other events such as single process purges (SPPs) or asserts. Code attempting to use dynamic data may discover a problem and schedule the appropriate audit to correct the data. In such cases, analysis should start with the assert.

10.2.3.8.2.1 Audit Scheduled from an Assert

In the following example, the first block to appear is from the PCBLA audit but the message class is AUDTMON, indicating that the audit is part of an ongoing event. Looking further down the output, we see that the PCBLA audit was called to run (or "scheduled") from an assert. The audit report prints before the assert because the audit message class is a higher priority than the assert message class. This illustrates the importance of gathering all the reports related to an event before beginning the troubleshooting process.

In general, it is an excellent plan to first look at the assert to analyze the problem. Refer to the *Asserts Manual* (235-600-500) for more information on asserts. The audit reports a QUAR error in the rlPCBLA tuple with BAD-DATA=H'3. However, this particular assert fires due to an inability to follow one tuple to another through the linkages (refer to the *Dynamic Data Manual* [235-600-2xx] for a description). The top function in the stack trace sets the "quarantine" attribute in the rlPCBLA tuple so the audit will find an error and recover the rlPCBLA tuple to the idle list, then it prints the assert and schedules the PCBLA audit to be run. Therefore, in this case, looking at the assert is what is needed rather than looking at the PCBLA audit.

In this example, the RLPCBLA relation has a logical key of –1, which is the idle list for the RLCCBCOM relation. If the stack trace were not available, a search of the code for instances of this assert could be restricted to code dealing with the RLCCBCOM relation and its rlPCBLA owned idle list. In this example, the stack trace can be analyzed to find the function that fired the assert.

```
S570-76 99-03-02 07:51:58 001183 AUDTMON N1 i99004sym
AUD  SM=1   PCBLA      ERROR-CODE=QUAR  EVENT=8
  ERROR-ADDR=H'1fad900  BAD-DATA=H'3
  LOG-KEY=H'ffffffff  GOOD-DATA=H'0

S570-76 99-03-02 07:51:58 001184 AUDTMON N1 i99004sym
AUD  SM=1   PCBLA      DUMP  PART-1  EVENT=8
  ERROR-ADDR=H'1fad900
  KEY=PCBLA H'ffff
  BLOCK-ADDR=H'1fad900
  FFFF0000 00000000 00000000 00000000 00000000 00000000 00000000
  00000000 00000000 00000000 00000000 00000000 00000000 00030000 00000000
  00010000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

```
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 FFFF0000 FFFF0000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000

  S570-76 99-03-02 07:52:07 001186 AUDTMON N1 i99004sym
  AUD  SM=1   PCBLA      DUMP  PART-2  EVENT=8
    ERROR-ADDR=H'1fad900
    KEY=PCBLA H'ffff
    BLOCK-ADDR=H'1fad9c4
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 02630000 00000000 00000000

  S570-76 99-03-02 07:52:10 001187 OVLD N1 i99004sym
*  OP OVRLD SM=1
    REAL TIME NONE
    RESOURCE  CCBCOM
    CONTROLS  DNET
    CONTROLS  AVRT

  S570-76 99-03-02 07:52:13 001189 OVLD N1 i99004sym
  REPT OVERLOAD HISTORY
    MODULE      REALTIME    RESOURCE
    SM=1        NONE        CCBCOM
    END

  S570-76 99-03-02 07:52:15 001190 AUDTMON N1 i99004sym
  AUD  SM=1   PCBLA      DUMP  PART-1  EVENT=8
    ERROR-ADDR=H'1fad900
    KEY=CCBCOM H'1dc
    BLOCK-ADDR=H'22bf630
    01DC0000 00000000 00000000 00000000 01DC4000 00000000 8C000000 FF000000
    00000000 FFFFFFFF FFFF0000 00000000 00000000 00080008 01100000 09060006
    01010103 0A400235 00000235 00970046 00000623 029D0104 00010013 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 01DCFFFF FFFFFFFF 00FF0000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 01DCFFFF FFFFFFFF
    00FF0000

  S570-76 99-03-02 07:52:24 001193 AUDTMON N1 i99004sym
  AUD  SM=1   PCBLA      DUMP  PART-2  EVENT=8
    ERROR-ADDR=H'1fad900
    KEY=CCBCOM H'1dc
    BLOCK-ADDR=H'22bf6f4
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    00000000 00000000 00000000 00000000 00000000 00000000 00000000 0009029B
    00000000 00000000 00000000

  S570-76 99-03-02 07:52:32 001195 AUDTMON N1 i99004sym
  AUD  SM=1   PCBLA      COMPLETED  ERRORS=1  EVENT=8

  S570-76 99-03-02 07:52:42 001197 ASRT N1 i99004sym
  INIT SM=1,1 LVL=RPI EVENT=8 COMPLETED
    DEF-CHK-FAIL=21291  AUD-SCHED=PCBLA  ELEV-MODE
    FAILING-ADDR=H'3c462c  SM-MODE=NORMAL  TIME=45:58.1
    PROCESS: BG=431,5,H'10bb656,RPI  CM=NONE,  FG=NONE,,

  S570-76 99-03-02 07:52:51 001199 ASRTMON N1 i99004sym
  REPT  SM=1   STACK TRACE ENV=OSDSM SRC=DCF EVENT=8
    USER:  003C462C  006637AC  0068F84A  0068EC7E  0068D95C  004BBEC0
           004EE476  010BBB76  00290630

  S570-76 99-03-02 07:52:59 001203 ASRTMON N1 i99004sym
  REPT  SM=1   STACK FRAME ENV=OSDSM SRC=DCF EVENT=8
    FUNC ADDR:   H'3c462c
    PARAMETERS:  FFFF0006  022FB564  0000FFFF  FFFF0229  1ED00008
                 8DC60068  F84A0006  00000793  00000288  00010236
    LOCAL DATA:  00000000  1A95F501  E01E2902  00000000  05000000
                 2B532C46  3C00848D  0800668D  08000000  00000000
```

```
                00000000  00008D00  00000900  00000000  00000000
                00000000  0000188D  0800668D  08000000  00000081
                2902D01F  2902F4D9  FA019A8D  08000081  2902668D
                08002E0E  B000008D  668D0800  668D0800  00000000
                00002B53  96B41F00  668D0800  08000000  8AA5AA00
                03002B53  668D0800  668D0800  00000000  21000000
                00000000  FDFFFFFF  0000204E  00001001  D01E2902

S570-76 99-03-02 07:53:07 001206 ASRTMON N1 i99004sym
REPT  SM=1   STACK FRAME ENV=OSDSM SRC=DCF EVENT=8
  FUNC ADDR:   H'6637ac
  PARAMETERS:  00060000  07930000  02880001  0236002F  00000229
               1EE00229  1ED00000  0000FF4A  00088E7A  0068EC7E
  LOCAL DATA:  D01E2902  FFFFFFFF  000064B5  2F020600  FFFFAC37
               66009E8D  08000000  00001A95  F501E01E  29020000
               00000500  00002B53  2C463C00  848D0800  668D0800
               00000000  00000000  00000000  8D000000  09000000
               00000000  00000000  00000000  188D0800  668D0800
               00000000  00812902  D01F2902  F4D9FA01  9A8D0800
               00812902  668D0800  2E0EB000  008D668D  0800668D
               08000000  00000000  2B5396B4  1F00668D  08000800
               00008AA5  AA000300  2B53668D  0800668D  08000000

S570-76 99-03-02 07:53:15 001208 ASRTMON N1 i99004sym
REPT SM=1  REGISTER DUMP ENV=OSDSM SRC=DCF EVENT=8
  REGISTER DATA= 02298100  00088D9A  01FAD9F4  02291FD0
                 02298100  00000000  00088D66  00088D18
                 00000000  00000000  00000000  00000009
                 0000008D  00000000  00000000  00000000

S570-76 99-03-02 07:53:23 001210 ASRTMON N1 i99004sym
REPT SM=1 PMDB-IN SRC=DCF EVENT=8
  MSG:  TYPE=124  LENGTH=4  PRIORITY=0
  FROM:  PROC=2  PCRID=1  UNIQ=0
  02880001
```

10.2.3.8.2.2 Audit Scheduled from a Single Process Purge (SPP)

In the following example, the single process purge (SPP) appears first, and was caused by a write protect error. In other words, a function attempted to write to a location of memory it is not allowed to touch. The fifth block in the event report is a `STACK TRACE` and all those addresses may be traced back to the line numbers of the called functions. More detail is given about tracking down this problem in “Assert Analysis,” section 5 of this manual.

During an SPP, the process that tried to do the illegal action is purged, a "quarantine" attribute is set in some of the tuples linked to the rlPCBLA tuple, and the associated audits are requested. This ensures that when the audits run, they will find errors and recover the tuples associated with the rlPCBLA tuple. This cleanup can sometimes take quite a while and may result in a number of ROP reports.

Similar to the case in which the audit is scheduled from an assert, the SPP error blocks sometimes appear further down in the event report. Depending on the type of process, there may be associated data or processes that generate further reports.

```
S570-131186 97-07-29 15:10:53 005349 INT A i8.1
REPT SM=4,0,ACT HWLVL=1 SWLVL=SPP EVENT=64  COMPLETED
  MP  WRITE-PROT-ERR  SW-ERR FAILING ADDR=H'8bd
  PROCESS:BG=1210,5,PURGED CM=NONE, FG=NONE,,

S570-131186 97-07-29 15:10:59 005350 INT_MON A i8.1
REPT SM=4,0,ACT HWLVL=1 SWLVL=SPP EVENT=64 COMPLETED
  MP  WRITE-PROT-ERR
  SW-ERR  FAIL-ADDR=H'8bd ROM-WRITE  DATA-BUS=H'0 TIME=8:2.3
  PROCESS:BG=1210,5,PURGED CM=NONE, FG=NONE,,        NORMAL
  ORIG-HW-STATUS: MC0: ACT   MC1: STBY
```

```
  FINAL-HW-STATUS: MC0: ACT   MC1: STBY
  PREVIOUS TYPE/COUNT:  121    0
  SHADOW TYPE/COUNT:    86    2
  AUX DATA:             H'0    H'200    H'0    H'1
  ESCALATION-COUNTS:    H'10000    H'0    H'0    H'0

S570-131186 97-07-29 15:11:09 005352 INT_MON A i8.1
REPT SM=4  HARDWARE CONTEXT SM2000 ORM EVENT=64
  68040-REGISTERS:   SSP=H'100cffc   PC=H'101b02c    SR=H'4
      USP=H'11195d2   FP=H'111963e    A5=H'250d342    A4=H'2387e8c
      A0=H'8bb        A1=H'22e6d90    A2=H'22e87d0    A3=H'250d39a
      D0=H'ffff       D1=H'94 D2=H'0  D3=H'f150a3c
      D4=H'0  D5=H'0  D6=H'0  D7=H'0
  PIC-REGISTERS:      HI      MED     LOW
      IRR:  H'0      H'c0      H'40
      IMR:  H'e      H'3b      H'fe
      ISR:  H'0      H'0       H'0

S570-131186 97-07-29 15:11:13 005354 INT_MON A i8.1
REPT SM=4 MP HW REGS EVENT=64
  ACTSR=H'3     EXCEP=2            PORTA=H'0          PRTBN=H'3b
  PROC1=H'1     PROC2=H'0          PROC3=H'e0
  RSTSR=H'0     HRSRC=H'0          SRSRC=H'0
  RSMSK=H'44    HRSMR=H'0          SRSMR=H'0
  SESR=H'0      BSNES=H'0          APESR=H'c
  SESMR=H'c8    BSNEM=H'0          APESM=H'f00c
  CORES=H'0     CAXES=H'0          COCTL=H'0          CPDSR=H'1e
  CORMR=H'0     CAXEM=H'0          MCIDR=H'1          CPISR=H'f1
  SUBRR=H'ff    MEMBD=H'ffff       CDSR1=H'0
  MBSR10=H'7c   MBSR11=H'7c        MBSR12=H'7c        MBSR13=H'ff
  MBESR0=H'0    MBESR1=H'0         MBESR2=H'82        MBESR3=H'ff
  MBESM0=H'82   MBESM1=H'82        MBESM2=H'82        MBESM3=H'ff
  BUSCR=H'b3    SHADDR=H'0
  SHBCR=H'b3    SHOPR=H'0

S570-131186 97-07-29 15:11:17 005356 INT_MON A i8.1
REPT  SM=4   STACK TRACE ENV=OSDSM SRC=FR EVENT=64
  USER:  0101B02C  0F150A3C  016952C4  01AE6BFA  01AE7B9E  01AE61DE
         01AD7922  01AD6A90  01616D98

S570-131186 97-07-29 15:11:21 005359 INT_MON A i8.1
REPT  SM=4   STACK FRAME ENV=OSDSM SRC=FR EVENT=64
  FUNC ADDR:   H'101b02c
  PARAMETERS:  00000000  00000000  00000000  00000000  022E87D0
               0250D39A  02387E8C  0250D342  016967F0  010D20F2
  LOCAL DATA:  085E6901  64961101  FFFFFFFF  D0872E02  01000000
               2E882E02  ED01D087  2E026A6A  69013696  11010100
               00000000  3802562D  5002D087  ED010003  0504BA04
               0000028D  18002201  0504BA04  00042700  52256101
               32961101  D0872E02  00000000  0E961101  00000000
               00000000  3C0A150F  FFFF1EB0  01013E96  1101D087
               2E025F00  D0872E02  00000000  010094D8  69011101
               CA000100  CD008204  A4E76901  B2951101  7CFF2081
               E6032201  00042700  01000504  BA049095  1101D087

S570-131186 97-07-29 15:11:26 005361 INT_MON A i8.1
REPT  SM=4   STACK FRAME ENV=OSDSM SRC=FR EVENT=64
  FUNC ADDR:   H'f150a3c
  PARAMETERS:  022E87D0  0250D34A  00000000  01119708  01119718
               01AE6BFA  0250D342  04820000  0F150A3C  00000000
  LOCAL DATA:  0196D087  2E029C8C  2E02F220  0D01F067  690142D3
               50028C7E  38029AD3  5002D087  2E020000  00000000
               00000000  00000000  00003C0A  150F7896  1101085E
               69016496  1101FFFF  FFFFD087  2E020100  00002E88
               2E02ED01  D0872E02  6A6A6901  36961101  01000000
               00003802  562D5002  D087ED01  00030504  BA040000
               028D1800  22010504  BA040004  27005225  61013296
               1101D087  2E020000  00000E96  11010000  00000000
               00003C0A  150FFFFF  1EB00101  3E961101  D0872E02
```

```
   S570-131186 97-07-29 15:11:32 005362 INT_MON A i8.1
   REPT SM=4  DATA=PCB,1210 ENV=OSDSM SRC=FR EVENT=64
     ADDR=H'39fdd00
     04BA002A 0A05FFFF FFFFFFFF 00000000 01119596 00800000 FFFFFFFF FFFFFFFF
     00270100 0F150A3C 00000000 00000000 00000000 00000000 022E87D0 0250D39A
     02387E8C 0250D342 011195C8 011195B4 00000000 00000000 00000000 00000000
     00040400 00000000 0078D78C 00800018 FFFFFFFF FFFF0000 0000000B 00000000

   S570-131186 97-07-29 15:11:52 005369 INT_MON A i8.1
   REPT SM=4  DATA=PCBLA-1,1210 ENV=OSDSM SRC=FR EVENT=64
     ADDR=H'39ff400
     04BA0000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
     00000000 00070500 00000000 00000000 00000100 00000000 00000000 01000000
     00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
     00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
     FFFF0000 FFFF0000 00000000 FFFF0000 FFFF0000 FFFF0000 FFFF0000 FFFF0000
     FFFF0000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
     00000000 00000000 00000000 00000000 00000000

   S570-131186 97-07-29 15:11:56 005370 INT_MON A i8.1
   REPT SM=4  DATA=PCBLA-2,1210 ENV=OSDSM SRC=FR EVENT=64
     ADDR=H'39ff4d4
     00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
     05090509 00000000 00000000

   S570-131186 97-07-29 15:12:00 005371 AUDTMON A i8.1
   AUD  SM=4   SMXRF      ERROR-CODE=DEADPID  EVENT=64
     ERROR-ADDR=H'2264fb0  BAD-DATA=H'0
     LOG-KEY=H'41  GOOD-DATA=H'0

   S570-131186 97-07-29 15:12:04 005372 AUDTMON A i8.1
   AUD  SM=4   SMXRF      DUMP    EVENT=64
     ERROR-ADDR=H'2264fb0
     KEY=SMXRF H'41
     BLOCK-ADDR=H'2264fb0
     0041023A 000000A4 00020000 10600000 04BA0405 00000000 000A0000 00000000

   S570-131186 97-07-29 15:12:09 005373 AUDTMON A i8.1
   AUD  SM=4   SMXRF      COMPLETED  ERRORS=1  EVENT=64

   S20B-65603 97-07-29 15:12:12 005374 no_cls A i8.1
*  REPT SOP CANNOT OPEN SCC CHANNEL

   S570-131186 97-07-29 15:12:13 005375 AUDTMON A i8.1
   AUD  SM=4   CCBCOM     ERROR-CODE=QUAR  EVENT=64
     ERROR-ADDR=H'3e68110  BAD-DATA=H'1
     LOG-KEY=H'482  GOOD-DATA=H'0

   S570-131186 97-07-29 15:12:18 005376 AUDTMON A i8.1
   AUD  SM=4   CCBCOM     DUMP  PART-1  EVENT=64
     ERROR-ADDR=H'3e68110
     KEY=CCBCOM H'482
     BLOCK-ADDR=H'3e68110
     04820000 00000000 00000060 04000000 04824000 00000000 8C640000 FF000000
     00000000 FFFFFFFF FFFF0000 00000000 00000000 00080008 01980000 09060007
     010301C3 580003F7 000003F7 00970007 00008000 007816A1 06201000 00000000
     00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
     00000000 00000000 00000000 00000000 0482FFFF FFFFFFFF 00FF0000 00000000
     00000000 00000000 00000000 00000000 00000000 00000000 0482FFFF FFFFFFFF
     00FF0000

   S570-131186 97-07-29 15:12:22 005377 AUDTMON A i8.1
   AUD  SM=4   CCBCOM     DUMP  PART-2  EVENT=64
     ERROR-ADDR=H'3e68110
     KEY=CCBCOM H'482
     BLOCK-ADDR=H'3e681d4
     00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
     00000000 00000000 00000000 023A023A 00000000 00000000 00000000 04C40000
```

```
  00000000 00000000 00000000

S570-131186 97-07-29 15:12:27 005378 AUDTMON A i8.1
AUD  SM=4   CCBCOM     COMPLETED  ERRORS=1  EVENT=64

S570-131186 97-07-29 15:12:31 005381 AUDTMON A i8.1
AUD  SM=4   CDBCOM     ERROR-CODE=MEMBER  EVENT=64
  ERROR-ADDR=H'3f07f40  BAD-DATA=H'1
  LOG-KEY=H'23a  GOOD-DATA=H'0

S570-131186 97-07-29 15:12:36 005383 AUDTMON A i8.1
AUD  SM=4   CDBCOM     DUMP    EVENT=64
  ERROR-ADDR=H'3f07f40
  KEY=CDBCOM H'23a
  BLOCK-ADDR=H'3f07f40
  023A0000 013A0010 00010004 005A0000 00000000 00000000 05090000 00000000

S570-131186 97-07-29 15:12:41 005384 AUDTMON A i8.1
AUD  SM=4   CDBCOM     DUMP  PART-1  EVENT=64
  ERROR-ADDR=H'3f07f40
  KEY=CCBCOM H'482
  BLOCK-ADDR=H'3e68110
  04820000 00000000 00000000 00000000 04824000 00000000 8C640000 FF000000
  00000000 FFFFFFFF FFFF0000 00000000 00000000 00080008 01980000 09060007
  00000000 00000000 000003F7 00970007 00008000 007816A1 06201000 00000000
  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
  00000000 00000000 00000000 00000000 0482FFFF FFFFFFFF 00FF0000 00000000
  00000000 00000000 00000000 00000000 00000000 00000000 0482FFFF FFFFFFFF
  00FF0000

S570-131186 97-07-29 15:12:45 005386 AUDTMON A i8.1
AUD  SM=4   CDBCOM     DUMP  PART-2  EVENT=64
  ERROR-ADDR=H'3f07f40
  KEY=CCBCOM H'482
  BLOCK-ADDR=H'3e681d4
  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00090000
  05240000 00000000 00000000

S570-131186 97-07-29 15:12:49 005387 AUDTMON A i8.1
AUD  SM=4   CDBCOM     ERROR-CODE=ASSOC_EST  EVENT=64
  ERROR-ADDR=H'3dfdf40  BAD-DATA=H'0
  LOG-KEY=H'23a  GOOD-DATA=H'0

S570-131186 97-07-29 15:12:54 005388 AUDTMON A i8.1
AUD  SM=4   CDBCOM     DUMP    EVENT=64
  ERROR-ADDR=H'3dfdf40
  KEY=SMEST H'23a
  BLOCK-ADDR=H'3dfdf40
  023A013A 00980041 00040405 00605001 0026C100 04BA0405 DA380155 00000000

S570-131186 97-07-29 15:12:57 005389 AUDTMON A i8.1
AUD  SM=4   CDBCOM     COMPLETED  ERRORS=2  EVENT=64

S570-131186 97-07-29 15:13:01 005390 AUDTMON A i8.1
AUD  SM=4   CKTDATA    ERROR-CODE=PID  EVENT=64
  ERROR-ADDR=H'3dfd840  BAD-DATA=H'4ba0405
  LOG-KEY=H'202  GOOD-DATA=H'0

S570-131186 97-07-29 15:13:12 005392 AUDTMON A i8.1
AUD  SM=4   CKTDATA    DUMP    EVENT=64
  ERROR-ADDR=H'3dfd840
  KEY=SMEST H'202
  BLOCK-ADDR=H'3dfd840
  02020124 00980041 0004040B 00005001 0026C100 04BA0405 DA380155 00000000

S570-131186 97-07-29 15:13:17 005393 AUDTMON A i8.1
AUD  SM=4   CKTDATA    DUMP    EVENT=64
  ERROR-ADDR=H'3dfd840
```

```
  KEY=SMEST H'246
  BLOCK-ADDR=H'3dfe0c0
  024600D7 00980041 0004040B 00005001 0026C100 04BA0405 DA380155 00000000

S570-131186 97-07-29 15:13:28 005394 AUDTMON A i8.1
AUD  SM=4   CKTDATA    ERROR-CODE=AUOOS  EVENT=64
  ERROR-ADDR=H'3dfd840  BAD-DATA=H'b
  LOG-KEY=H'202  GOOD-DATA=H'2

S570-131186 97-07-29 15:13:33 005395 AUDTMON A i8.1
AUD  SM=4   CKTDATA    DUMP    EVENT=64
  ERROR-ADDR=H'3dfd840
  KEY=SMEST H'202
  BLOCK-ADDR=H'3dfd840
  02020124 00980052 0002040B 00015001 0026C100 FFFF0000 DA380155 00000000

S570-131186 97-07-29 15:13:40 005396 AUDTMON A i8.1
AUD  SM=4   CKTDATA    DUMP    EVENT=64
  ERROR-ADDR=H'3dfd840
  KEY=SMEST H'246
  BLOCK-ADDR=H'3dfe0c0
  024600D7 00980041 0004040B 00005001 0026C100 04BA0405 DA380155 00000000

S570-131186 97-07-29 15:13:49 005398 AUDTMON A i8.1
AUD  SM=4   CKTDATA    ERROR-CODE=ASSOC_TUP  EVENT=64
  ERROR-ADDR=H'3f07840  BAD-DATA=H'66
  LOG-KEY=H'202  GOOD-DATA=H'0

S570-131186 97-07-29 15:13:54 005399 AUDTMON A i8.1
AUD  SM=4   CKTDATA    DUMP    EVENT=64
  ERROR-ADDR=H'3f07840
  KEY=CDBCOM H'202
  BLOCK-ADDR=H'3f07840
  02020000 01240000 01080004 005A0000 00000000 00000000 00710204 00000000

S570-131186 97-07-29 15:14:00 005401 AUDTMON A i8.1
AUD  SM=4   CKTDATA    ERROR-CODE=AUOOS  EVENT=64
  ERROR-ADDR=H'3dfd840  BAD-DATA=H'b
  LOG-KEY=H'202  GOOD-DATA=H'2

S570-131186 97-07-29 15:14:05 005404 AUDTMON A i8.1
AUD  SM=4   CKTDATA    DUMP    EVENT=64
  ERROR-ADDR=H'3dfd840
  KEY=SMEST H'202
  BLOCK-ADDR=H'3dfd840
  02020124 00980052 0002040B 00015001 0026C100 FFFF0000 DA380155 00000000

S570-131186 97-07-29 15:14:09 005407 AUDTMON A i8.1
AUD  SM=4   CKTDATA    DUMP    EVENT=64
  ERROR-ADDR=H'3dfd840
  KEY=SMEST H'246
  BLOCK-ADDR=H'3dfe0c0
  024600D7 00980041 0004040B 00005001 0026C100 04BA0405 DA380155 00000000


S570-131186 97-07-29 15:14:14 005411 AUDTMON A i8.1
AUD  SM=4   CKTDATA    COMPLETED  ERRORS=20  EVENT=64
```

10.2.3.8.3 Case History Three: CKTDATA Audit

This case study illustrates how ROP output generated from a circuit data (CKTDATA) audit is analyzed. The ROP output was generated due to dynamic data inconsistencies detected by the CKTDATA audit. In this case, switch personnel gather the information regarding event 4 provided in the following ROP output.

```
S0-13799 99-04-26 09:22:02 000289 AUDTFST FIVE
   AUD  SM=192   CKTDATA          ERROR-CODE=VERTICAL  EVENT=4
     ERROR-ADDR=H'5bda9f4  BAD-DATA=H'4
```

```
     LOG-KEY=H'91c  GOOD-DATA=H'4

SO-13799 99-04-26 09:22:03 000290 AUDTFST FIVE
   AUD  SM=192   CKTDATA          DUMP  PART 1 OF 1  EVENT=4
     KEY=rlSMEST TUPLE WHERE
         <the_key>
         =<H'91c>
     BLOCK-ADDR=H'5bda9f4
     091C00D6 00980052 0002040B 01215001 0026C100 FFFF0000 042687AC 00000000

SO-13799 99-04-26 09:22:05 000291 AUDTFST FIVE
   AUD  SM=192   CKTDATA          DUMP  PART 1 OF 1  EVENT=4
     KEY=rlSMEST TUPLE WHERE
         <the_key>
         =<H'81c>
     BLOCK-ADDR=H'5bd89f4
     081C00D7 00000052 00000200 00014101 FFFF0000 FFFF0000 042687AC 00000000

SO-13799 99-04-26 09:22:06 000292 AUDTFST FIVE
   AUD  SM=192   CKTDATA          DUMP  PART 1 OF 1  EVENT=4
     KEY=rlSMEST TUPLE WHERE
         <the_key>
         =<H'89c>
     BLOCK-ADDR=H'5bd99f4
     089C00D7 00000052 00000100 00014101 FFFF0000 FFFF0000 042687AC 00000000

SO-13799 99-04-26 09:22:08 000293 AUDTFST FIVE
   AUD  SM=192   CKTDATA          DUMP  PART 1 OF 1  EVENT=4
     KEY=rlSMEST TUPLE WHERE
         <the_key>
         =<H'61c>
     BLOCK-ADDR=H'5bd49f4
     061C013A 00000052 00000200 00014001 FFFF0000 FFFF0000 042687AC 00000000

SO-13799 99-04-26 09:22:10 000294 AUDTFST FIVE
   AUD  SM=192   CKTDATA          DUMP  PART 1 OF 1  EVENT=4
     KEY=rlSMEST TUPLE WHERE
         <the_key>
         =<H'65c>
     BLOCK-ADDR=H'5bd51f4
     065C013A 00000052 00000100 00014001 FFFF0000 FFFF0000 042687AC 00000000

SO-13799 99-04-26 09:22:12 000295 AUDTFST FIVE
   AUD  SM=192   CKTDATA          COMPLETED  ERRORS=1  EVENT=4
```

The CKTDATA audit steps through the static RLCKTDATA relation. The RLCKTDATA relation stores the definition for each circuit that is known by a switching module (SM). For each of these circuits, the CKTDATA audit verifies that the proper corresponding dynamic data exists and that this dynamic data is in a consistent state. For further details on the types of checks the CKTDATA audit makes, please refer to the *Audits Manual* (235-600-400).

The next step the switch personnel should take is to look up the CKTDATA audit VERTICAL error description in the *Audits Manual*. From the *Audits Manual*, the switch personnel will learn that a VERTICAL error is reported when an inconsistency in maintenance state is found between the circuit under audit (the child circuit) and its parent circuit(s).

The *Audits Manual* describes what is dumped in each of the error report fields for a VERTICAL error. ERROR-ADDR contains the address of the rlSMEST tuple for the child circuit, BAD-DATA contains the rlSMEST basic status (bas_stat) of the child circuit, GOOD-DATA contains the number of parents this child circuit has, and LOG-KEY contains the rlSMEST circuit name (unit_id) of the child circuit. Lastly, the *Audits Manual* describes the data dumps that follow a given error report. For VERTICAL

errors, the first report is the rlSMEST tuple corresponding to the child circuit, and all the rest of the data dumps are the rlSMEST tuples for the parent circuits.

All of the audit data dump reports for this case history are of a switching module equipment status table tuple (rlSMEST). The RLSMEST relation is a dynamic relation that contains a tuple per circuit defined in the static RLCKTDATA relation. The dynamic state and activity of a circuit can be obtained from its rlSMEST tuple. The layout of this dynamic data can be found in the *Dynamic Data Manual* (235-600-2xx).

For analyzing VERTICAL errors, the rlSMEST tuple field values needed are the circuit name (unit_id), circuit type (type), basic status (bas_stat), and second qualifier (qual_2) fields. Refer to "Extracting Data from an Audit Report," section 10.2.3.6.2.1 for more information. The switch personnel will need to extract these field values from each of the rlSMEST tuples dumped.

The circuit name is the key to both the rlCKTDATA tuple and the rlSMEST tuple. The circuit type is defined by the DMCKTTYPE enumeration. The basic status represents the maintenance state of the circuit. The second qualifier indicates the reason why the basic status is set to a particular maintenance state. Both the basic status and second qualifier fields are defined by the DMMRASTAT enumeration.

Again, the first report contains the rlSMEST tuple for the circuit under audit, which is the child circuit. The circuit name of the child circuit is H'091C; the circuit type is H'00D6, which corresponds to the PCPHPSIU DMCKTTYPE enumeration value; the basic status is H'04, which corresponds to the SMOOS DMMRASTAT enumeration value; and the second qualifier is H'0B, which corresponds to the SMFE DMMRASTAT enumeration value. So, this child PH circuit is in the out-of-service family of equipment state. “Family of equipment” means that this circuit is out of service because its parent circuit is out of service.

The same analysis should be done for each parent circuit’s rlSMEST tuple. The results of this can be seen in the following table:

Circuit Name	Circuit Type	Basic Status	Second Qualifier
H'091C	PCPHPSIU	SMOOS	SMFE
H'081C	PCSHLFPSIU	SMSTBY	SMSTNUL
H'089C	PCSHLFPSIU	SMACT	SMSTNUL
H'061C	PCCCPSU2	SMSTBY	SMSTNUL
H'065C	PCCCPSU2	SMACT	SMSTNUL

Looking at the table, it can be seen that the parent PSU shelf circuits are running active/stand-by. The grandparent PSU common controller circuits are also running active/stand-by. Therefore, the VERTICAL inconsistency is that the child circuit is in the out-of-service family of equipment state, but none of the parent circuits are out of service.

Recovery for this VERTICAL error is to load the PH circuit on the MRA recovery queue. MRA will then take the PH circuit off of the queue and take the appropriate recovery action to make the child PH circuit’s state consistent with its parents. In this case, MRA restored the PH circuit to the active state. The “restore completed” message is the last message on the ROP output.

10.2.3.8.4 Case History Four: PCBLA Audit

This case history addresses the PCBLA audit. Call processing and terminal and switch maintenance are implemented by means of processes that are associated with tuples of relation RLPCBLA (process control block link area). Each rlPCBLA tuple has links to tuples in other relations that contain all the information about a call, such as ports and time slots used. They also include controls of the call itself, such as call waiting or forwarding data. Incorrect data in an rlPCBLA tuple can cause a variety of problems related to call processing.

The function of the PCBLA audit is to verify the integrity of the data and the link areas. The subsequent audit reports and data dumps are related as indicated by the common EVENT number.

```
S570-65669 95-10-06 22:21:13 050504 AUDTMON FIVE I
AUD  SM=17   PCBLA  ERROR-CODE=DEAD_LNKD  EVENT=1055
  ERROR-ADDR=H'1425900  BAD-DATA=H'4a4
  LOG-KEY=H'3d9  GOOD-DATA=H'0

S570-65669 95-10-06 22:21:21 050512 AUDTMON FIVE I
AUD  SM=17   PCBLA  DUMP    EVENT=1055
  ERROR-ADDR=H'1425900
  KEY=PCBLA H'3d9
  BLOCK-ADDR=H'1425900
  03D90000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
  00000000 FFFF0000 FFFF0000 00000000 FFFF0000 FFFF0000 0000B100 00008000
  01000010 00000000 00000000 00000000 00000000 FFFF0000 00000000 00000000
  00000000 00000000 00000000 00000000 00000000 00000000 00000000 B3C20000

S570-65669 95-10-06 22:21:29 050516 AUDTMON FIVE I
AUD  SM=17   PCBLA  DUMP    EVENT=1055
  ERROR-ADDR=H'1425900
  KEY=PCB H'3d9
  BLOCK-ADDR=H'808e76
  010300B2 031F03B6 FFFF0000 000CBF0A 08B80000 FFFFFFFF FFFFFFFF FFFFFF00

S570-65669 95-10-06 22:21:45 050521 AUDTMON FIVE I
AUD  SM=17   PCBLA  COMPLETED  ERRORS=1  EVENT=1055
```

The previous error code, DEAD_LNKD, indicates that the operating system believes the process associated with this rlPCBLA tuple to be dead, but the linkage fields are not null. This means that the system has resources tied to a process that is no longer active. The purpose of this check is to report the error and then release all of the resources that the dead process is holding. The ERROR-ADDR in the third line of the report is the physical address of the failing tuple, and BAD-DATA is the bad linkage indicator. LOG-KEY is the process number. Two data dumps accompany the PCBLA audit error report: the rlPCBLA tuple and the process control block (PCB) associated with the process.

The process type can be found by looking at the progi field in the PCB dump. Refer to the "Extracting Data from an Audit Report," section 10.2.3.6.2.1 for information on how to determine the value of this attribute.

When the PCBLA audit finds an error, the system automatically calls several other audits that will check the data that the rlPCBLA tuple could be linked to. For example, the port linkage area (PORTLA) contains data for all defined line and trunk ports. When the rlPCBLA tuple is recovered, it breaks the linkage to the rlPORTLA tuple. The PORTLA audit reports the ONE2ONE error. Similarly, other data linked to the rlPCBLA tuple may be detected by additional audits. Typically, these error reports have the same event number.

```
S570-65669 95-10-06 22:22:10 050527 AUDTMON FIVE I
```

```
AUD  SM=17   PORTLA  ERROR-CODE=ONE2ONE  EVENT=1055
  ERROR-ADDR=H'152f040  BAD-DATA=H'2
  LOG-KEY=H'b3c2  GOOD-DATA=H'0

S570-65669 95-10-06 22:22:18 050528 AUDTMON FIVE I
AUD  SM=17   PORTLA  DUMP    EVENT=1055
  ERROR-ADDR=H'152f040
  KEY=PORTLA H'b3c2
  BLOCK-ADDR=H'152f040
  B3C20100 00000A00 00009D9F 04CA3800 00000000 00019261 00000000 03E20000

S570-65669 95-10-06 22:22:26 050530 AUDTMON FIVE I
AUD  SM=17   PORTLA  COMPLETED  ERRORS=1  EVENT=1055
```

The error code ONE2ONE is not described in the PORTLA section of the *Audits Manual* (235-600-400). It can be found in the "Audit Drivers" section of the manual under the heading DRERRCDS. This is because the error was detected by a common program (driver) invoked by the PORTLA audit.

The ONE2ONE error is caused by an invalid one-to-one linkage, meaning that the failing tuple is pointing to a tuple that does not point back. The audit driver re-initializes the rlPORTLA tuple and, if it belongs on an idle link list, links it to the appropriate head cell. The ERROR-ADDR is the physical address of the failing tuple, BAD-DATA is the sub-index to the per-tuple linkage block, and LOG-KEY is the logical key of the tuple in error. The accompanying PORTLA dump is that of the corrupted tuple.

Software Analysis Guide

CONTENTS PAGE

11. DATA COLLECTION AND ANALYSIS 11-1
11.1 PURPOSE 11-1
11.2 ENVIRONMENTAL CONDITIONS 11-1
11.3 SM/CMP/PI/PH EVENT HISTORY 11-1
11.4 FAULT CONDITIONS TO ESCALATE 11-1

11. DATA COLLECTION AND ANALYSIS

11.1 PURPOSE

For each of the areas discussed in this document, an attempt has been made to give a summary outline of the steps that can be taken to correct errors and return the *5ESS*® switch to a sane state. This section lists system environmental conditions that should be accumulated by the user before calling the next level of technical support.

11.2 ENVIRONMENTAL CONDITIONS

When preparing to escalate a problem, all relevant data, information, and read-only printer (ROP) output must be readily available and organized for discussion and analysis. This includes any system environmental conditions that could possibly be pertinent to the fault condition or fault event including associated system outputs (such as system initializations), human/machine interactions (such as growth/degrowth procedures), singular machine processes (such as disk backup), machine configurations (such as AM or SM active/standby status), machine services being supported (such as packet switching), and machine loading (such as heavy, light, resource unavailability). Finally, an event history should be printed.

11.3 SM/CMP/PI/PH EVENT HISTORY

Event histories are data dumps of the latest events for the switching module (SM), the communications module processor (CMP), the packet interface (PI), and the protocol handler (PH). They are printed on the ROP after SM/CMP/PI/PH selective initialization, SM/CMP/PI/PH full initializations, or when manually requested:

`op:postmortem,sm=#`
(where # is the SM number)

`op:postmortem,cmp=0,prim/mate`

In the case of the CMP, the event history is also printed as part of a CMP raise error lead postmortem dump.

Craft personnel can manually request the output of the most recent retained PH/PI event histories by using the commands:

`op:history, psuph=sm#-psu#-shelf#-ph#[,rcvyhst][,eventhst]`
(where `psu` is packet switching unit, `rcvyhst` is recovery history, and `eventhst` is event history)

`op:history,mctsi=sm#-pi#[,rcvyhst][,eventhst]`
(where `mctsi` is module controller time slot interchanger, `rcvyhst` is recovery history, and `eventhst` is event history).

For further information, see `APP:EVENT-HIST` in the 235-600-700, *Input Messages Manual* and 235-600-750, *Output Messages Manual*.

11.4 FAULT CONDITIONS TO ESCALATE

The following list provides a few conditions under which escalation should be considered:

Interrupts — Interrupt errors that are usually caused by software are `ILLEGAL-INSTR`, `PRIVILEGE-VIOL`, `STACK-PROT-ERR`, `TRAP`, and `WRITE-PROT-ERR`. These errors should be reported to the next level of technical support for investigation and correction.

Asserts

Some files have been stripped of symbol information. In this case, the `upd:ftrc` command returns an error message instead of the file and function name. If this occurs, contact the next level of technical support.

Audit

It is possible that events leading to the audit error are not printed on the ROP. In this case, consult the 235-600-400, *Audits Manual* where the audit and error are described in detail. For static and dynamic data inconsistencies, the *Audits Manual* specifies which relations to check. If the correct procedure cannot be found, consult the next level of technical support.

Preventing Future Initializations

For SM initializations, look for the `INITIALIZATION TRIGGER` message to find the general cause of the initialization. The daylog should be dumped from one hour prior to the start of an initialization until one hour after the SM is stable. These daylog messages, along with the initialization trigger message, give a good idea of what occurred. This is the starting point for SM initializations.

To dump an event history that includes the last 240 events leading up to the SM initialization (or 544 events leading up to the SM-2000 initialization), input the command:

`op:postmort,sm=#` (where # is the SM number) and analyze the data collected about the initialization. If the cause of the initialization is unknown, contact the next technical level of support.

RTA DCF

The `RTA DCF` error message gives information about a defensive check failure or assert that occurred in the routing and terminal allocation software. All `RTA DCF` errors should be investigated to find the reason for failure so the appropriate corrective action can be taken. If the reason cannot be found, contact the next technical level of support.

Software Analysis Guide

CONTENTS PAGE

12. OSDS MONITOR 12-1

12.1 OSDS BACKGROUND INFORMATION 12.1-1
12.1.1 OSDS Description 12.1-1
12.1.2 Processes 12.1-1
12.1.2.1 System Processes 12.1-1
12.1.2.2 Terminal Processes 12.1-1
12.1.3 Resources 12.1-1
12.1.3.1 OSDS Resources 12.1-1
12.1.3.2 Process Control Block 12.1-2
12.1.3.3 Process Control Block Link Area 12.1-2
12.1.3.4 Process Message Data Block 12.1-2
12.1.3.5 Stack Control Block. 12.1-2
12.1.3.6 Message Control Block 12.1-3
12.1.3.7 Timer Control Block 12.1-3
12.1.4 Segment Breaks 12.1-3
12.1.5 Processing Levels 12.1-3
12.1.6 OSDS Messages 12.1-4
12.1.7 Feature Execution (FEX) 12.1-4

12.2 OSDS MONITOR OVERVIEW 12.2-1
12.2.1 OSDS Monitor Purpose 12.2-1
12.2.2 Functions 12.2-1
12.2.3 The OSDS Monitor Buffer 12.2-1

12.3 OSDS MONITOR INPUT MESSAGES 12.3-1

12.4 OSDS MONITOR INPUT FLAGS 12.4-1
12.4.1 Section Description 12.4-1
12.4.2 Starting and Stopping The Monitor 12.4-1
12.4.3 Messaging 12.4-1
12.4.4 Per-Event Data 12.4-2
12.4.5 OSDS Usage 12.4-3
12.4.6 Data Dumps 12.4-4
12.4.7 Client Data Dumps. 12.4-4
12.4.8 What To Do Flags 12.4-5
12.4.9 What To Dump Flags (Per-Event Data) 12.4-27
12.4.10 Dumped Data Description 12.4-28
12.4.10.1 Introduction to Dump Description Tables 12.4-28
12.4.10.2 SM and CMP Data Dump Description 12.4-29

12.4.10.3 OKP Data Dump Description 12.4-29
12.4.10.4 SMKP Data Dump Description. 12.4-30
12.4.10.5 MSKP Data Dump Description. 12.4-31

12.5 INPUT COMMANDS TO CLEAR AND DUMP THE BUFFER 12.5-1
12.5.1 Section Description 12.5-1
12.5.2 Zero The Monitor Buffer 12.5-1
12.5.3 Output Data 12.5-1
12.5.3.1 Input Messages for Data Output 12.5-1
12.5.3.2 Output Messages. 12.5-2
12.5.4 Kill A Monitor Job 12.5-2

12.6 USEFUL INPUT MESSAGE SEQUENCES 12.6-1
12.6.1 Section Description 12.6-1
12.6.2 Snap Per-Event Process and Messaging Data 12.6-1
12.6.2.1 Snap Messages, No Processes 12.6-1
12.6.2.2 Snap Messages and Processes 12.6-1
12.6.2.3 Snap Processes, No Messages 12.6-2
12.6.2.4 Snap Processes and Messages for a Port 12.6-2
12.6.2.5 Snap Processes and Messages and/or State Definition Language (SDL) Trace Event Data for Four Ports 12.6-2
12.6.2.6 Snap Processes for a Specific Program ID 12.6-3
12.6.2.7 Snap Processes at a Specific Priority. . . . 12.6-3
12.6.3 Snap Foreground And Messaging Data 12.6-4
12.6.3.1 Snap Foreground Work, All Messages . . . 12.6-4
12.6.3.2 Snap Specific Foreground Entries, No Messages. 12.6-4
12.6.4 Snap Interject and Messaging Data 12.6-4
12.6.4.1 Snap Interject Work, All Messages 12.6-4
12.6.4.2 Snap Interject Work, No Messages 12.6-5
12.6.5 Enable or Disable Monitor on Match or Mismatch . . . 12.6-5
12.6.5.1 Use Input Control Flags 12.6-5
12.6.5.2 Enable Monitor on Data Match 12.6-5
12.6.5.3 Disable Monitor On Data Mismatch. 12.6-5
12.6.5.4 Disable Monitor on PCBLA Match 12.6-6
12.6.6 SDL Trace 12.6-6
12.6.6.1 SDL Trace for a Given SM 12.6-6
12.6.6.2 SDL Trace for a Given Port 12.6-7
12.6.6.3 SDL Trace for an Assert 12.6-7
12.6.6.4 SDL Trace for an SM Overload 12.6-8

12.7 OSDS MONITOR BUFFER LAYOUTS 12.7-1
12.7.1 Section Description 12.7-1

12.7.2 Buffer Layout for The AM 12.7-1
12.7.3 AM Buffer Word Content 12.7-2
12.7.4 Buffer Layout for The SM 12.7-9
12.7.5 SM Buffer Word Content 12.7-9
12.7.6 Buffer Layout for The CMP 12.7-13
12.7.7 CMP Buffer Word Content 12.7-14

12.8 SNAPPED DATA DUMP LAYOUTS 12.8-1
12.8.1 Section Description 12.8-1
12.8.2 DAD Data Dump Layout 12.8-1
12.8.3 DAP Data Dump Layout 12.8-2
12.8.4 F00 Data Dump Layout 12.8-3
12.8.5 F01 Data Dump Layout 12.8-5
12.8.6 F18 Data Dump Layout 12.8-6
12.8.7 F22 Data Dump Layout 12.8-7
12.8.8 F23, F25, F27, F29, F31 Data Dump Layout 12.8-7
12.8.9 HUF, HIJ, HPD, HPY, HMX, HSX, HSP Data Dump Layout 12.8-8
12.8.10 SEG Data Dump Layout 12.8-12

LIST OF FIGURES

Figure 12.7-1 — OSDS Monitor AM Buffer Layout 12.7-2

Figure 12.7-2 — OSDS Monitor SM Buffer Layout 12.7-9

Figure 12.7-3 — OSDS Monitor CMP Buffer Layout. 12.7-14

LIST OF TABLES

Table 12.3-1 — OSDS Monitor Input Messages 12.3-1

Table 12.4-1 — Start and Stop Control Flags 12.4-1

Table 12.4-2 — Messaging Control Flags 12.4-2

Table 12.4-3 — Data Control Flags for Per-Event Data Dumps 12.4-3

Table 12.4-4 — Event Control Flags for Per-Event Data Dumps 12.4-3

Table 12.4-5 — OSDS Usage Control Flags 12.4-4

Table 12.4-6 — Data Dump Control Flags 12.4-4

Table 12.4-7 — Client Data Dump Control Flags 12.4-5

Table 12.4-8 — SM and CMP Data Dump Descriptions 12.4-29

Table 12.4-9 — OKP Data Dump Descriptions 12.4-30

Table 12.4-10 — SMKP Data Dump Descriptions 12.4-31

Table 12.4-11 — MSKP Data Dump Descriptions 12.4-32

Table 12.5-1 — OSDS Monitor Output Messages 12.5-2

Table 12.7-1 — OSDS Monitor AM Buffer Word Structure 12.7-3

Table 12.7-2 — OSDS Monitor SM Buffer Word Structure 12.7-10

Table 12.7-3 — OSDS Monitor CMP Buffer Word Structure 12.7-15

Table 12.8-1 — SM/CMP OSDS Usage Data Layout 12.8-9

Table 12.8-2 — AM OSDS Usage Data Layout 12.8-10

12. OSDS MONITOR

12.1 OSDS BACKGROUND INFORMATION

12.1.1 OSDS Description

The operating system for distributed switching (OSDS) is a real-time operating system designed to efficiently process call originations and terminations. OSDS runs in each switching module (SM), the communication module processor (CMP), and in two kernel processes in the administrative module (AM) the operational kernel process (OKP) and the switch maintenance kernel process (SMKP). The primary function of the operating system is coordinating the many processes necessary to support concurrent calling.

OSDS manages processes by:

- Identifying each process with a process control block (PCB). This block contains information about the status of a process.
- Using a stack to store local variables and return points for function calls.
- Using a program counter to track the location of a process in the program being executed.

A process is an instance of program execution plus the data necessary for the execution. There are two kinds of processes system and terminal.

12.1.2 Processes

12.1.2.1 System Processes

System processes are started at system initialization time, and are not created or terminated dynamically. If a system process is purged, OSDS recreates it. System processes have predefined process IDs. Other processes simply assume that system processes exist when attempting to communicate with them.

System processes typically manage multiple terminal processes. Examples of system processes are scanning and routing.

12.1.2.2 Terminal Processes

Terminal processes typically provide customer services (calls) and terminal maintenance. These types of processes are created and terminated on demand. For example, a terminal process is created when a customer initiates a telephone call and is terminated when the customer hangs up. Terminal processes exist for a finite period of time (unlike system processes).

12.1.3 Resources

12.1.3.1 OSDS Resources

OSDS manages the following resources:

- Process control block (PCB)
- Process control block link area (PCBLA)
- Process message data block (PMDB)
- Stack control block (SCB)
- Message control block (MCB)
- Timer control block (TCB).

12.1.3.2 Process Control Block

The process control block (PCB) contains the relevant information about a process's environment. It is the internal representation of an OSDS process. The PCB stores all of the data that is required to represent and administer the process, including:

- Process ID
- Process state (ready, running, waiting, etc.)
- Process priority
- Associated program name
- Processor registers (including the program counter)
- Stack index
- Process linkage (for linking PCBs into lists)
- Message linkage (for linking MCBs to processes)
- Timer linkage (for linking TCBs to processes).

PCBs are stored in a linear array. The process number is the index into the PCB array (PCBI).

Application programs identify a process by the process ID (OSPID). An OSPID is a unique structure assigned to each process, it contains:

- Processor ID (PCRID)

  The processor in which the process exists.

- Uniqueness

  This field is used to distinguish between new and old (dead) processes that happen to have the same PCBI.

- Process number

  The PCBI of the process.

12.1.3.3 Process Control Block Link Area

Each terminal process in an SM is assigned a data structure called a process control block link area (PCBLA) that links other data blocks to the process. Pertinent data about a call is saved in the PCBLA.

12.1.3.4 Process Message Data Block

Each terminal process in an SM is also assigned a data structure called a process message data block (PMDB). The PMDB is used to send and receive messages. Each PMDB contains two sections one for receiving messages and one for constructing messages. Each section is large enough to store any OSDS message.

12.1.3.5 Stack Control Block

Every running process must have a stack. Each stack is associated with a stack control block (SCB). Each SCB contains, among other things, the stack location and the process that owns the stack. OSDS provides primitives for a process to release its stack while waiting for an event. When the event occurs, a new stack is allocated to the process.

12.1.3.6 Message Control Block

One method that processes use to communicate with each other is sending messages. OSDS stores messages internally using message control blocks (MCBs). Messages destined for a process, but not yet read by that process, are linked to the process's PCB using MCBs. (See "OSDS Messages," Section 12.1.6 for more information.)

12.1.3.7 Timer Control Block

OSDS represents timers using timer control blocks (TCBs). When a process requests a timer, a TCB is linked to the process's PCB. OSDS provides several types of timers including interval timers, cyclic timers, time of day timers, and cyclic time of day timers. OSDS determines which timer should expire next by maintaining a heap data structure containing all of the active timers in the system.

12.1.4 Segment Breaks

Each process, during its execution, has exclusive control of the virtual machine environment in which it resides. OSDS requires that each process manages its own real time breaks. That is, each process must periodically return control to OSDS.

OSDS provides two methods for a process to return control suspending and waiting. To better understand the difference between these two operations, a brief description of how OSDS schedules processes follows.

OSDS maintains eight process ready queues, one for each priority level. To determine the next process to run, OSDS looks down the ready queues in priority order (that is, 7 through 0). The first process that it finds is run next.

When a process *suspends* itself, it is placed at the end of its priority's ready queue. As a result, it blocks all lower priority processes from running.

When a process *waits*, it specifies an event or time period to wait for. When the event occurs or the specified time expires, the process is placed in the appropriate ready queue. While the process is waiting, it is not in a ready queue; therefore, it is not blocking lower priority processes from running.

12.1.5 Processing Levels

OSDS maintains three levels of processing background, interject, and foreground.

- Background

  Process execution (that is, all system and terminal processes) is considered background processing. This is the OSDS environment used most often by applications.

- Interject

  The interject environment is the portion of time between the time that one process returns control to OSDS and the time that OSDS dispatches the next process to run. During this time period, the operating system has control.

  The essential task during interject is to find the next process that should be allowed to run and to give control to that process. In addition, OSDS uses interject to perform message delivery, timer management, signal processing, and OSDS monitor work.

- Foreground

The foreground environment is entered every 10 ms based on a hardware interrupt. The primary function of the foreground environment is to scan for all call originations and terminations, plus perform some process integrity checks.

12.1.6 OSDS Messages

OSDS messages are sent between processes. Each message contains a message header and the message contents. The header contains information about the sending process, the message type, whether the message receiver should handle this message at a higher than normal priority, and the message length.

To receive a message, a process calls an OSDS receive message primitive and provides OSDS with a buffer to copy a message into. Typically, the receiving process waits for some period of time for a message to arrive. Ideally, when a process sends a message to another process, the receiving process has already called an OSDS receive message primitive and is waiting for a message to arrive. In this case, OSDS copies the message from the sending process to the buffer provided by the receiving process when the OSDS receive message primitive was called.

If the receiving process was not waiting for a message (that is, it did not call an OSDS receive message primitive) or more than one message was sent to the receiving process before the process had a chance to run, then OSDS queues the message in an `MCB` and links the `MCB` to the receiving process's `PCB`. The receiving process can later dequeue the messages stored in `MCB`s by calling an OSDS receive message primitive.

12.1.7 Feature Execution (FEX)

In the feature execution (FEX) environment, features are defined in terms of a hierarchy of finite state models. The FEX environment provides external message receiving and distribution services, subprocess scheduling, inter-subprocess communication, and timer services.

FEX uses OSDS messages for communication with external OSDS processes. However, FEX also has internal messages that are unknown to OSDS. FEX messages are sent between FEX subprocesses for different model_ids or FEX subprograms. For example, the subprocess controlling a feature communicates with the subprocess controlling a port using a FEX internal message.

12.2 OSDS MONITOR OVERVIEW

12.2.1 OSDS Monitor Purpose

The OSDS monitor was developed to collect data about the OSDS operating system in the AM, the CMP, and the SMs of the *5ESS*® switch. It allows users to gather performance data and investigate performance problems without the need for specialized tools such as logic analyzers.

12.2.2 Functions

The services provided by the OSDS monitor are loosely divided into six areas:

- Start and stop the monitor based on user requests and system conditions
- Gather data concerning messaging
- Gather per-event data
- Gather data concerning OSDS usage
- Perform general data dumps
- Perform specialized client data dumps.

Each area is discussed in the "OSDS Monitor Input Flags," Section 12.4.

12.2.3 The OSDS Monitor Buffer

The monitor is allocated a buffer of approximately 60 Kbytes. It uses the sections of the buffer for various reasons; however, two sections of the buffer are of main concern the Control section and the Dispatch array.

- The Control section holds flags that are set to indicate the data that the user wants the monitor to gather.
- The Dispatch array is the monitor's general purpose data storage area. The type of data stored in the dispatch array depends on the monitor functions that the user selected. The dispatch array size is approximately 56 Kbytes.

12.3 OSDS MONITOR INPUT MESSAGES

Table 12.3-1 describes the OSDS monitor input messages. For complete details, see the 235-600-700, *Input Messages Manual*.

Table 12.3-1 — OSDS Monitor Input Messages

Message	Description
ALW:MON	Allow the monitor.
CLR:MON	Clear the monitor buffer memory.
INH:MON	Inhibit the monitor.
SET:MON,WTD	Set the monitor control flags, what to do flags, and what to dump flags.
SET:MON,DATA	Initialize the monitor control data.
SET:MON,FCN	Set special function execution flags.
SET:MON,SPEC	Initialize monitor data for timed data dump or special function use.
OP:MON,CTL	Dump the monitor control data.
OP:MON,DSP	Dump selected areas of the monitor buffer or the entire buffer.
OP:MON,PID	Dump the program ID data or operating system data.

12.4 OSDS MONITOR INPUT FLAGS

12.4.1 Section Description

First, this section describes the monitor's six functional areas (as listed in the "OSDS Monitor Overview," Section 12.2) and indicates the specific monitor input message control flags that relate to each function.

Then, a description of the control flags with examples is provided in "What to Do Flags," Section 12.4.8 and "What to Dump Flags," Section 12.4.9.

For complete details on the OSDS monitor input messages and flags, see the 235-600-700, *Input Messages Manual*.

12.4.2 Starting and Stopping The Monitor

The monitor can be started and stopped manually by using input messages, or automatically based on system conditions.

The control flags in Table 12.4-1 relate to starting and stopping the monitor. For a more detailed description of each flag, see "What to Do Flags," Section 12.4.8.

Table 12.4-1 — Start and Stop Control Flags

Flag	Function
BEG	Start writing data at the beginning of the dispatch array (again).
DAT	Check the contents of an address for a match or mismatch condition. Used in conjunction with the SP4 and SP5 flags.
DMH	Check the port in the current process's PCBLA for a match against given port names. Used in conjunction with the SP5 flag.
DOX	Inhibit the monitor or inhibit certain data dumps when the dispatch array becomes full (otherwise, data wraps to the beginning of the dispatch array).
F16	Inhibit the monitor if a real time overload occurs.
F17	Inhibit the monitor if a resource overload occurs.
SP3	Start and stop the monitor automatically.
SP4	Inhibit the monitor on a match or mismatch condition.
SP5	Perform a BEG on a match or mismatch condition.
SP6	Check the current process's PCBLA for a match or mismatch condition. Used in conjunction with the SP4 and SP5 flags.

12.4.3 Messaging

The monitor can dump data related to OSDS messaging and FEX messaging.

The control flags in Table 12.4-2 are related to messaging data. For a more detailed description of each flag, see "What to Do Flags," Section 12.4.8.

Table 12.4-2 — Messaging Control Flags

Flag	Function
F01	Record data concerning the history of MCB usage (that is, MCB allocations and deallocations).
F20	Include message contents in message snaps.
F21	Filter message data to or from a specific processor.
F22	Record SDL trace event data.
F23	Include a timestamp in message data.
F24	Collect counts by message type for OKP messages sent to *UNIX*[a] RTR processes.
F25	Record the sender's program ID, the destination process ID, and the message header for OKP messages sent to *UNIX* RTR processes.
F26	Collect counts by message type for messages from *UNIX* RTR processes to the OKP, CMP, or SMs.
F27	Record the destination process's program ID (if OKP is the receiving process), the destination process ID, and the message header for messages from *UNIX* RTR processes to the OKP, CMP, or SMs.
F28	Collect counts of inter-processor messages received by message type and the sender's processor ID.
F29	Record message data for inter-processor messages received including the destination process's program ID, process number, state and uniqueness, and the message header.
F30	Collect counts of inter-processor or intra-processor messages sent by message type and the destination processor ID.
F31	Record message data for inter-processor or intra-processor messages sent including the sending process's program ID, the destination process ID, and the message header.

a. Registered trademark of The Open Group.

12.4.4 Per-Event Data

The monitor can dump data when certain events occur. The user can specify the data to dump and the events to initiate the dump.

The control flags in Table 12.4-3 indicate the data to dump for a per-event data dump. For a more detailed description of each flag, see "What to Dump Flags," Section 12.4.9.

Table 12.4-3 — Data Control Flags for Per-Event Data Dumps

Flag	Function
AAA	Combine the ACL, APP, AID, and ASA flags.
ACL	Record the value of the clock.
ADM	Record the values of up to six data addresses for the message switch kernel process (MSKP).
ADO	Record the values of up to six data addresses.
ADS	Record the values of up to six data addresses for the SMKP.
AID	Record process priority, program ID, and time spent in the real time segment.
APC	Record the values of up to three data indexes in the current PCBLA.
APP	Record the program ID and PCB index.
ASA	Record the starting address for the process's segment, *UNIX* RTR event flags, OSDS signals, or MSKP specific data.

The control flags in Table 12.4-4 filter the events for which data is dumped in a per-event data dump. For a more detailed description of each flag, see "What to Do Flags," Section 12.4.8.

Table 12.4-4 — Event Control Flags for Per-Event Data Dumps

Flag	Function
DAT	Filter DPD or DPY per-event data based on the value of a memory location.
DIJ	Record per-event data during interject.
DMH	Filter DPD or DPY per-event data based on port names.
DMP	Record per-event data for MSKP job types.
DMX	Record per-event data for MSKP entry or exit.
DPD	Record per-event data filtered on program IDs.
DPY	Record per-event data filtered on process priority.
DSP	Record SMKP per-event data filtered on program IDs.
DSX	Record per-event data for SMKP entry or exit.
DUF	Record per-event data for foreground (SM or CMP), or for OKP entry or exit (AM).
SP6	Filter DPD or DPY per-event data based on values in the running process's PCBLA.

12.4.5 OSDS Usage

The monitor can record information concerning the use of OSDS. Most of this data is related to how long different types of OSDS jobs execute.

The control flags in Table 12.4-5 are related to OSDS usage. For a more detailed description of each flag, see "What to Do Flags," Section 12.4.8.

Table 12.4-5 — OSDS Usage Control Flags

Flag	Function
F18	Record a function back trace for every call to OSREPLACE() and OSRESTART().
HHH	Combine the HUF, HIJ, HPD, and HPY flags.
HIJ	Count entries to and accumulated time in interject (AM, SM, CMP).
HMH	Produce a histogram of segment lengths for various OSDS job types.
HMX	Count entries to and accumulated time in MSKP.
HSP	Count dispatches of and accumulated time in each program ID in SMKP.
HPD	Count dispatches of and accumulated time in each program ID (OKP, SM, CMP).
HPY	Count dispatches of and accumulated time in each OSDS priority level (AM, SM, CMP).
HSX	Count entries to and accumulated time in SMKP.
HUF	Count entries to and accumulated time in OKP and *UNIX* RTR (AM), or count entries to and accumulated time in foreground (SM).
SEG	Record data concerning the number of consecutive segments a program ID runs (OKP, SM, CMP).

12.4.6 Data Dumps

The monitor allows the user to dump raw memory values as well as pseudo raw dumps of OSDS resources.

The control flags in Table 12.4-6 are related to data dumps. For a more detailed description of each flag, see "What to Do Flags," Section 12.4.8.

Table 12.4-6 — Data Dump Control Flags

Flag	Function
F00	Dump memory regions or data from OSDS resource control blocks (PCB, SCB, TCB, or MCB).
SP7	Timed data dump.

12.4.7 Client Data Dumps

The monitor provides a mechanism for switch application code to dump extra debugging information if certain OSDS monitor flags are turned on.

The application code must have hooks in it to dump the additional debugging data. The OSDS monitor simply provides a general purpose mechanism for the application code to use.

The control flags in Table 12.4-7 are related to client data dumps. For a more detailed description of each flag, see "What to Do Flags," Section 12.4.8.

Table 12.4-7 — Client Data Dump Control Flags

Flag	Function
DAD	Stack dump flag.
DAP	Data dump flag.

12.4.8 What To Do Flags

This section describes each of the OSDS monitor what to do flags for the AM, SM, and CMP. The flags are listed alphabetically.

Note: Many of the input flags can be used to control data collection in the AM, CMP, *or* SMs, for example:

```
SET:MON,{AM|SM|CMP},WTD,BEG;
If the command is for the SM, the SM must be specified as SM=x (where x is
the SM number). The CMP is specified as CMP=x-y (where x-y is 0-0 or 0-1).
```

BEG Environment: AM/SM/CMP

Start recording data at the beginning of the dispatch array (again).

Description:

This flag allows the user to move the monitor's write pointer back to the beginning of the dispatch array. This is useful for recording some data, dumping it to the receive only printer (ROP) or a file, then continuing to record data. If the BEG flag is used after the first set of data is dumped and before the monitor is restarted, the monitor will write the second set of data at the beginning of the dispatch array (overwriting the first set) making it easier to dump the second set of data. Effectively, this flag allows the user to restart the monitor without erasing other what to do flags and/or data that were previously established.

Example:

```
SET:MON,{AM|SM|CMP},WTD,BEG;
```

DAD Environment: OKP/SM/CMP

Client stack dump flag.

Description:

This flag is similar to the DAP flag in that switch application code contains hooks to indicate when data should be dumped (see the DAP flag).

When the application hook calls the monitor, the DAD flag causes the stack that the application code is currently executing on to be dumped. See the "Snapped Data Dump Layouts," Section 12.8 for the general layout of the data dumped by this flag.

Example:

```
CLR:MON,{AM|SM|CMP},ALL;
SET:MON,{AM|SM|CMP},DAD,WTD,CTL=ON[,DOX];
SET:MON,{AM|SM|CMP},DATA,AD1=aaaabbbb,AD2=ccccdddd,AD3=eeeeffff,
     AD4=gggghhhh,AD5=iiiijjjj,AD6=kkkkllll;
ALW:MON,{AM|SM|CMP};
```

Where aaaa - llll are event or group codes to "turn on." If less than 12 codes are specified, 0000 must be used to indicate the end of the list of codes.

DAP Environment: OKP/SM/CMP

Client data dump flag.

Description:

This flag (and the DAD flag) allows switch application code to dump additional debugging data. The monitor provides a general purpose mechanism for application code to dump extra data, but has no control over what data the application code chooses to dump. The data is dumped by hooks in the application code.

Each hook in the application code is identified by an event code and a group code. Each hook has an independent event code; however, any number of hooks may have the same group code. The user specifies which hooks dump data by selecting up to 12 event or group codes. Definitions of the event and group codes are found in the global header hdr/os/OSclidump.h.

The DAP flag allows the application code to dump a block of memory into the dispatch array. See the "Snapped Data Dump Layouts," Section 12.8 for the general layout of the data dumped by this flag.

Example:

```
CLR:MON,{AM|SM|CMP},ALL;
SET:MON,{AM|SM|CMP},DAP,WTD,CTL=ON[,DOX];
SET:MON,{AM|SM|CMP},DATA,AD1=aaaabbbb,AD2=ccccdddd,AD3=eeeeffff,
     AD4=gggghhhh,AD5=iiiijjjj,AD6=kkkkllll;
ALW:MON,{AM|SM|CMP};
```

Where aaaa - llll are event or group codes to "turn on." If less than 12 codes are specified, 0000 must be used to indicate the end of the list of codes.

DAT Environment: OKP/SM/CMP

Filter DPD or DPY flag per-event data based on a memory match or mismatch condition.

Description:

This flag must be used in conjunction with the DPD or DPY flag. That is, this flag is not checked unless either DPD or DPY is on. DAT allows the user to filter per-event data dumps based on the contents of a memory location. It also allows the user to start (actually, execute a BEG) or stop the monitor based on the contents of a memory location when DAT is used in conjunction with the SP4 or SP5 flags.

When DAT is used to filter per-event data snaps for the DPD or DPY flags, the user specifies the address to be checked, a mask for the contents of the address, the data to check against, and whether the monitor should check for a match or a mismatch. If the specified condition does not exist, then the DPD or DPY data dump is aborted.

Example:

```
SET:MON,{AM|SM|CMP},WTD,DAT;
SET:MON,{AM|SM|CMP},DATA,PDA=aaaaaaaa,PDM=bbbbbbbb,AD1=cccccccc;
```

Where:

aaaaaaaa - data value to match or mismatch
bbbbbbbb - mask for the data (used as an AND mask)
cccccccc - address to match or mismatch on
0x00000000 + address for a match
0x80000000 + address for a mismatch

Note: The address must be word aligned (that is, end in 0x0, 0x4, 0x8, or 0xc).

Either the DPD or DPY flag must also be set for this example to work (see the DPD or DPY flags).

See the SP4 and SP5 flags for information about how they are used.

DIJ Environment: AM/SM/CMP

Snap per-event data during interject.

Description:

This flag allows the user to snap per-event data when interject begins. The snaps may be filtered based on the segment length that interject ran. See the "Dumped Data Description," Section 12.4.10 for an explanation of various data that may be dumped.

Example:

```
SET:MON,{AM|SM|CMP},WTD,DIJ,[AAA,ACL,APP,AID,ASA,ADO,APC];
SET:MON,{AM|SM|CMP},DATA,PTM=aaaa;
```

Where aaaa is the segment length filter in 125 µsec units. Data will only be recorded if the actual segment time is greater than or equal to the specified value.

DMH Environment: SM

Filter DPD or DPY flag per-event data using port names.

Description:

This flag must be used in conjunction with either the DPD or DPY flag. DMH allows the user to filter the DPD or DPY per-event data dumps based on the port name stored in the PCBLA for the running process. This flag may be used in conjunction with SP5 to "turn on" the monitor when a port match occurs (see the SP5 flag). The user can filter the data based on up to four port names. If the port name in the running process's PCBLA does not match one of the given port names, then the data dump will be aborted.

Example:

```
SET:MON,SM=SM_number,WTD,DMH,{DPD|DPY};
SET:MON,SM=SM_number,DATA,PRT=aaaaaaaa;
```

Where aaaaaaaa is a specific port member number or 0xffff to indicate that up to four port numbers will be specified by the following command:

```
SET:MON,SM=SM_number,SPEC,S00=p1p1,S01=p2p2,S02=p3p3,S03=p4p4;
```

Where p1p1 - p4p4 are the specific port member numbers to trap on. All four port member numbers must be specified; however, if less than four port member numbers are desired, one of the port member numbers may be entered multiple times.

Note: The DPD or DPY flag must be specified in conjunction with this flag; see the DPD or DPY flags for examples of their use.

Warning: The monitor cannot trap data based on four specific port member numbers** and **four specific program IDs (DPD flag) simultaneously. The monitor can only trap data based on one port member number and four program IDs, or four port member numbers and one program ID.

DMP Environment: MSKP

Snap per-event data for MSKP job types.

Description:

This flag allows the user to snap per-event data for MSKP job types. See the "Dumped Data Description," Section 12.4.10 for an explanation of various data that may be dumped.

Example:

```
SET:MON,AM,WTD,DMP,[AAA,ACL,APP,AID,ASA,ADO];
```

DMX Environment: MSKP

Snap per-event data for MSKP entry or exit.

Description:

This flag allows the user to snap per-event data upon entry or exit from MSKP. See the "Dumped Data Description," Section 12.4.10 for an explanation of the various data that may be dumped.

Example:

```
SET:MON,AM,WTD,DMX,[AAA,ACL,APP,AID,ASA,ADO];
```

DOX Environment: AM/SM/CMP

Prevent wrap around of the dispatch array for various flags.

Description:

The DOX flag causes certain data dumps or the monitor itself to be inhibited if the dispatch array becomes full. This list indicates which flags DOX can be used with and the action that will be taken when a full condition occurs.

Flag	Action on Full Condition
DAD	Inhibit the monitor
DAP	Inhibit the monitor
DIJ	Inhibit dumping to the dispatch array
DMP	Inhibit dumping to the dispatch array
DMX	Inhibit dumping to the dispatch array
DPD	Inhibit dumping to the dispatch array
DPY	Inhibit dumping to the dispatch array
DSP	Inhibit dumping to the dispatch array
DSX	Inhibit dumping to the dispatch array
DUF	Inhibit dumping to the dispatch array
F00	Inhibit the monitor
F01	Inhibit the monitor
F18	Inhibit the monitor
F22	Inhibit dumping to the dispatch array
F25	Inhibit dumping to the dispatch array
F27	Inhibit dumping to the dispatch array
F29	Inhibit dumping to the dispatch array
SEG	Inhibit dumping to the dispatch array
SP7	Inhibit dumping to the dispatch array

If the SP5 flag is used with the DAT, SP6, or DMH flags, the DOX flag will be turned on when the DAT, SP6, or DMH condition becomes true (see the DAT, SP6, and DMH flags for more details).

DPD Environment: OKP/SM/CMP

Snap per-event data on process dispatches filtered on program IDs.

Description:

This flag allows the user to snap per-event data when processes are dispatched and filter the data dumps based on a program ID. The user can specify a variety of filters:

- All program IDs
- All program IDs except those running at OSDS priority 0
- A specific program ID
- Up to four specific program IDs.

The user can also filter the data dump based on the segment length that the process runs. See the "Dumped Data Description," Section 12.4.10 for an explanation of the various data that may be dumped.

Example:

```
SET:MON,{AM|SM|CMP},WTD,DPD,[AAA,ACL,APP,AID,ASA,ADO];
SET:MON,{AM|SM|CMP},DATA,PRG=aaaa[,PTM=bbbb];
```

Where:

aaaa is a specific program ID or one of the following values:

h'ff00	-	All program IDs
h'f000	-	All program IDs except those running at OSDS priority 0
h'fff0	-	Up to four specific program IDs specified by the SET:MON,SPEC command

bbbb is a segment length filter in 125 µsec units. Data will only be recorded if the actual segment time is greater than or equal to the specified value.

If aaaa is h'fff0, the four program IDs are specified as follows:

```
SET:MON,{AM|SM|CMP},SPEC,S00=p1p1,S01=p2p2,S02=p3p3,S03=p4p4;
```

Where p1p1 - p4p4 are the specific program IDs to filter on. All four program IDs must be specified; however, if less than four program IDs are desired, one of the program IDs may be entered multiple times.

Warning: The monitor cannot trap data based on four specific program IDs and four port member numbers (DMH flag) simultaneously. The monitor can only trap data based on one port member number and four program IDs, or four port member numbers and one program ID.

DPY Environment: OKP/SM/CMP

Snap per-event data on process dispatches filtered on priority level.

Description:

This flag allows the user to snap per-event data when processes are dispatched and filter the data dumps based on the process's priority. If the process's priority level does not match the specified priority level, then the data dump will be aborted. The user can also filter the data dump based on the segment length that the process runs. See the "Dumped Data Description," Section 12.4.10 for an explanation of the various data that may be dumped.

Example:

```
SET:MON,{AM|SM|CMP},WTD,DPY,[AAA,ACL,APP,AID,ASA,ADO];
SET:MON,{AM|SM|CMP},DATA,PRI=aaaa[,PTM=bbbb];
```

Where:

aaaa is the priority level to filter on (0-7).

bbbb is the segment length filter in 125 µsec units. Data will only be recorded if the actual segment time is greater than or equal to the specified value.

DSP Environment: SMKP

Snap per-event data for SMKP process dispatches filtered on program IDs.

Description:

This flag allows the user to snap per-event data when SMKP processes are dispatched and filter the data dumps based on program ID. The user can specify a variety of filters:

- All program IDs
- All program IDs except those running at OSDS priority 0
- A specific program ID

The user can also filter the data dump based on the segment length that the process runs. See the "Dumped Data Description," Section 12.4.10 for an explanation of the various data that may be dumped.

Example:

```
SET:MON,AM,WTD,DSP,[AAA,ACL,APP,AID,ASA,ADO];
SET:MON,AM,DATA,PSG=aaaa[,PST=bbbb];
```

Where:

`aaaa` is a specific program ID or one of the following values:

h'ff00	-	All program IDs
h'f000	-	All program IDs except those running at OSDS priority 0

`bbbb` is the segment length filter in 125 µsec units. Data will only be recorded if the actual segment time is greater than or equal to the specified value.

`DSX` Environment: SMKP

Snap per-event data on SMKP entry or exit.

Description:

This flag allows the user to snap per-event data when the SMKP is entered or exited. See the "Dumped Data Description," Section 12.4.10 for an explanation of the various data that may be dumped.

Example:

```
SET:MON,AM,WTD,DSX,[AAA,ACL,APP,AID,ASA,ADO,APC];
```

`DUF` Environment: OKP/SM/CMP

Snap per-event data for foreground (SM or CMP), or for OKP entry or exit (AM).

Description:

This flag allows the user to snap per-event data when an SM or CMP exits foreground or when OKP is entered or exited from *UNIX* RTR. The SM and CMP snaps may be filtered based on the segment length that foreground ran. See the "Dumped Data Description," Section 12.4.10 for an explanation of the various data that may be dumped.

Example:

```
SET:MON,{AM|CMP|SM},WTD,DUF,[AAA,ACL,APP,AID,ASA,ADO,APC];
SET:MON,{SM|CMP},DATA,PTM=aaaa;
```

Where aaaa is the segment length filter in 125 µsec units. Data will only be recorded if the actual segment time is greater than or equal to the specified value. This parameter only applies to the SM and CMP.

F00 Environment: OKP/SM/CMP

Dump memory or OSDS resource control blocks.

Description:

This flag allows the user to dump various data into the dispatch array:

- Raw memory (option 1)
- PCB data (option 2)
- SCB data (option 3)
- MCB data (option 4)
- TCB data (option 5)

The user may specify up to 15 dump sets. The monitor will process the sets in order until all sets have been exhausted or until the dispatch array is full. The monitor will be inhibited after the data dumps are complete. All of the dumps are performed in one real-time segment. Each dump set is defined by a three-word entry in the following format:

- Raw memory dump

  word 0 - 1 to indicate a raw memory dump
  word 1 - starting address of the memory to dump
  word 2 - number of bytes to dump

- PCB, SCB, MCB, or TCB dumps

  word 0 - 2, 3, 4 or 5 to indicate a PCB, SCB, MCB, or TCB dump
  word 1 - 0
  word 2 - 0

See the "Snapped Data Dump Layouts," Section 12.8 for the layout of the dumped data.

Example:

```
ALW:MON,{AM|SM|CMP};
SET:MON,{AM|SM|CMP},SPEC,S00=do1,S01=dw1a,S02=dw1b[,S03=do2
     ,S04=dw2a,S05=dw2b],...[,S42=do15,S43=dw15a,S44=dw15b];
SET:MON,{AM|SM|CMP},FCN,F00;
SET:MON,{AM|SM|CMP},WTD,CTL=ON;
```

Where:

do1-do15	-	Dump option (1 - 5) (word 0 in the three-word entry format)
dw1a-dw15a	-	First data word for the dump (word 1 in the three-word entry format)
dw1b-dw15b	-	Second data word for the dump (word 2 in the three-word entry format)

Note: If all 15 dump options are not used, the last dump option should be 0 (zero).

The ALW:MON command must be given before the SET:MON,SPEC command for this flag to work.

F01 Environment: OKP/SM/CMP

Record MCB usage history data.

Description:

This flag allows the user to record a history of MCB usage. The monitor records the number of MCBs used by each process (in an array) and also saves the maximum number of MCBs ever in use. In addition, a dump is made each time one of the following events occurs:

- An MCB is acquired by a process
- A process is dispatched while at least one MCB is in use
- An MCB is released by a process

The monitor may be automatically inhibited when a user-specified MCB usage threshold is surpassed. This flag is useful for trying to determine the events that led to an MCB overload condition.

See the "Snapped Data Dump Layouts," Section 12.8 for the layout of the data recorded by this flag.

Example:

```
SET:MON,{AM|SM|CMP},FCN,F01;
SET:MON,{AM|SM|CMP},SPEC,SOO=aa;
SET:MON,{AM|SM|CMP},WTD,CTL=ON;
ALW:MON,{AM|SM|CMP};
```

Where aa is the MCB usage threshold. If more than aa MCBs are used, the monitor will be inhibited.

F02 - F15 Environment: None.

Spare control flags.

F16 Environment: AM/SM/CMP

Inhibit the monitor on real-time overload.

Description:

If overload control detects a real-time overload and the F16 flag is set, the monitor is automatically inhibited.

F17 Environment: AM/SM/CMP

Inhibit the monitor on resource overload.

Description:

If overload control detects a resource overload and the F17 flag is set, the monitor is automatically inhibited.

F18 Environment: SM

Record a function back trace for every call to OSREPLACE() and OSRESTART().

Description:

The F18 flag allows the user to obtain a function back trace for every call to OSREPLACE() and OSRESTART(). This option is used to track a process's transitions between program IDs.

For each call to OSREPLACE() or OSRESTART(), a dump is made to the dispatch array containing the calling process's ID, a timestamp, the new and old program IDs, the number of addresses in the function back trace, and the function back trace. See the "Snapped Data Dump Layouts," Section 12.8 for the contents of the dump.

Example:

```
SET:MON,SM=SM_number,FCN,F18;
SET:MON,SM=SM_number,WTD,CTL=ON;
ALW:MON,SM=SM_number;
```

F19 Environment: SM

VFSM tracing.

Description:

Virtual finite state machine (VFSM) tracing in GLIB subsystem.

F20 Environment: AM/SM/CMP

Include message contents in message snap.

Description:

This flag causes the contents of the message to be collected with the other message information. The default setting does not collect the message contents.

Note: The F20 flag must be used in conjunction with the F25, F27, F29, or F31 flag.

F21 Environment: OKP/SM/CMP

Filter message data generated by the F29 or F31 flags based on processor number.

Description:

This flag filters message data for a specific processor. Either the F29 flag must be set to filter data for messages received by the identified processor, or the F31 flag must be set to filter data for messages sent by this processor.

Example:

```
SET:MON,{AM|SM|CMP},FCN,F21;
SET:MON,{AM|SM|CMP},SPEC,SO5=<processor#>;
```

F22 Environment: SM

Record SDL trace event data.

F23 Environment: AM/SM/CMP

Snap the clock for per-event message data (F25, F27, F29, and F31 flags).

Description:

This flag causes a timestamp to be added to the per-event message data generated by the F25, F27, F29, and F31 flags.

The clock is a software clock that indicates the number of days since the monitor was last allowed and the number of milliseconds since midnight of the current day. The most significant four bits of the clock (which is 32 bits total) contain the number of days since the monitor was allowed. The other 28 bits contain the number of milliseconds since midnight. The clock is only accurate to 100 ms.

See the "Snapped Data Dump Layouts," Section 12.8 for the location of this clock in the per message event data.

Example:

```
SET:MON,{AM|SM|CMP},FCN,F23;
```

Note: The F23 flag must be used in conjunction with the F25, F27, F29, or F31 flag.

F24 Environment: OKP

Create a histogram of message types sent from the OKP to **UNIX** *RTR processes.*

Description:

This flag allows the user to create a histogram showing the number of messages of each message type that OKP sends to *UNIX* RTR processes. This histogram is stored as an array of longs where each long represents a message type. The index into the array is the message type. The array begins at an offset of 0x800 from the start of the dispatch array. Unlike a per processor messaging histogram (part of the F28 and F30 flags), each long in a per message type histogram represents both the number of messages sent and the number of messages received for a given message type.

- The most significant short in the long (aaaa in 0xaaaabbbb) represents the number of messages received for the given type.
- The least significant short in the long (bbbb in 0xaaaabbbb) represents the number of messages sent for the given type.

The F24 flag only increments the "sent" short of the long.

Note: If the F24 flag is used in conjunction with the F26 flag, double counting may occur because both flags use the same array.

F25 Environment: OKP

Snap per-event message data for OKP messages sent to **UNIX** *RTR processes.*

Description:

This flag allows the user to snap per-event message data for messages sent from the OKP to *UNIX* RTR processes. The user may filter the snapped data based on the sending process's program ID and the message type. The sending process's program ID, the destination process ID, and the message header are written to the dispatch array. See the "Snapped Data Dump Layouts," Section 12.8 for the layout of the dumped data.

Example:

```
SET:MON,AM,FCN,F25;
SET:MON,AM,DATA,PRG=aaaa,AD5=0,AD6=bbbb;
```

Where:

`aaaa` is a specific program ID or one of the following values:

h'ff00	-	All program IDs
`h'f000`	-	All program IDs except those executing at priority 0
`h'fff0`	-	Up to four specific program IDs specified by the `SET:MON,SPEC` command.

`bbbb` is a specific message type or `h'f000` for all message types.

If `aaaa` is `h'fff0`, the four program IDs are specified as follows:

```
SET:MON,AM,SPEC,S00=p1p1,S01=p2p2,S02=p3p3,S03=p4p4;
```

where `p1p1` - `p4p4` are the specific program IDs. If less than four program IDs are desired, set the respective inputs to a value greater than 512.

`F26` Environment: AM

Create a histogram of message types sent from **UNIX** *RTR processes to the OKP, CMP, or SMs.*

Description:

This flag allows the user to create a histogram showing the number of messages of each message type that the OKP or the SMs or the CMP received from *UNIX* RTR processes.

This histogram is stored as an array of longs where each long represents a message type. The index into the array is the message type. The array begins at an offset of `0x800` from the start of the dispatch array.

Unlike a per processor messaging histogram (part of the `F28` and `F30` flags), each long in a per message type histogram represents both the number of messages sent and the number of messages received for a given message type.

- The most significant short in the long (`aaaa` in `0xaaaabbbb`) represents the number of messages received for the given type.

- The least significant short in the long (bbbb in 0xaaaabbbb) represents the number of messages sent for the given type.

The F26 flag only increments the "received" short of the long.

Note: If the F26 flag is used in conjunction with the F24 flag, double counting may occur because both flags use the same array.

F27 Environment: AM

Snap per-event message data for messages sent from **UNIX** *RTR to the OKP, SMs, or CMP.*

Description:

This flag allows the user to snap per-event message data for messages sent from *UNIX* RTR processes to the OKP, the SMs, or the CMP. The user may filter the snapped data based on the message type. If the destination process is in the OKP, the snapped data may also be filtered on the destination process's program ID.

The destination process's program ID (if the destination process is in OKP), the destination process ID, and the message header are written to the dispatch array. See the "Snapped Data Dump Layouts," Section 12.8 for the layout of the dumped data.

Example:

See the example for the F25 flag. The F27 flag is used on the SET:MON,FCN line instead of F25.

F28 Environment: OKP/SM/CMP

Create a histogram of message types received by the processor, and a histogram of processors that sent messages to the local processor.

Description:

Two histograms are created by this flag. The first is a histogram showing the number of messages of each message type that the local processor received from other processors (that is, interprocessor messages).

This histogram is stored as an array of longs where each long represents a message type. The index into the array is the message type. The array begins at an offset of 0x800 from the start of the dispatch array.

Each long in a per message type histogram represents both the number of messages sent and the number received for a given message type.

- The most significant short in the long (aaaa in 0xaaaabbbb) represents the number of messages received for the given type.
- The least significant short in the long (bbbb in 0xaaaabbbb) represents the number of messages sent for the given type.

The F28 flag only increments the "received" short of the long.

The second histogram provides the number of messages that other processors sent to this processor. This histogram is stored as an array of

longs where each long represents a processor. The index into the array is the sending processor number. The array begins at the start of the dispatch array.

Note: If the F28 flag is used in conjunction with the F30 flag, double counting may occur because both flags use the same array.

F29 Environment: OKP/SM/CMP

Snap per-event message data for inter-processor messages received.

Description:

This flag allows the user to snap per message event data for inter-processor messages received in the OKP, CMP, and the SMs.

The user may filter the snapped data based on the destination process's program ID and the message type. In the SM, the user may also filter on the port associated with the destination process.

The destination process's program ID, the destination process ID, and the message header are written to the dispatch array. See the "Snapped Data Dump Layouts," Section 12.8 for the layout of the dumped data.

See the F21 flag to filter on the processor number.

Example:

```
SET:MON,{AM|SM|CMP},FCN,F29;
SET:MON,{AM|SM|CMP},DATA,PRG=aaaa,PRT=bbbb,AD5=0,AD6=cccc;
```

Where:

aaaa is a specific program ID or one of the following values:

h'ff00	-	All program IDs
h'f000	-	All program IDs except those executing at priority 0
h'fff0	-	Up to four specific program IDs specified by the SET:MON,SPEC command

bbbb is a specific port member number or one of the following values:

h'0	-	All port member numbers
h'ffff	-	Up to four specific port member numbers specified via the SET:MON,SPEC command

cccc is a specific message type or h'f000 all message types.

If aaaa is h'fff0, the four program IDs are specified as follows:

```
SET:MON,{AM|SM|CMP},SPEC,S00=p1p1,S01=p2p2,S02=p3p3,S03=p4p4;
```

where p1p1 - p4p4 are the specific program IDs. If less than four program IDs are desired, set the respective inputs to a value greater than 512.

If bbbb is h'ffff, the four port member numbers are specified as follows:

```
SET:MON,SM,SPEC,S00=p1p1,S01=p2p2,S02=p3p3,S03=p4p4;
```

where p1p1 - p4p4 are the specific port member numbers. All four port member numbers must be specified; however, if less than four port member numbers are desired, one of the port member numbers may be entered multiple times.

Note: The PRT - p4p4 parameter is only valid on the SM.

Warning: The monitor cannot filter data based on four specific port member numbers and four specific program IDs simultaneously. The monitor filters data based on one port member number and four program IDs, or four port member numbers and one program ID.

F30 Environment: AM/SM/CMP

Create a histogram of message types sent by the local processor, and a histogram of processors that received messages from the local processor.

Description:

Two histograms are created by this flag. The first shows the number of interprocessor and intraprocessor messages of each message type sent by the local processor. This histogram is stored as an array of longs where each long represents a message type. The index into the array is the message type. The array begins at an offset of 0x800 from the start of the dispatch array.

Each long in the per message type histogram represents both the number of messages sent and the number received for a given message type.

- The most significant short in the long (aaaa in 0xaaaabbbb) represents the number of messages received for the given type.
- The least significant short in the long (bbbb in 0xaaaabbbb) represents the number of messages sent for the given type.

The F30 flag only increments the "sent" short of the long.

The second histogram provides the number of messages that other processors received from this processor. This histogram is stored as an array of longs where each long represents a processor. The index into the array is the receiving processor number. The array begins at the start of the dispatch array.

Note: If the F30 flag is used in conjunction with the F28 flag, double counting may occur because both flags use the same array.

F31 Environment: OKP/SM/CMP

Snap per-event message data for intra- and inter-processor messages.

Description:

This flag allows the user to snap per-event message data for intra- and inter-processor messages sent by the processor. The user may filter the snapped data based on the sending process's program ID and the message type. In the SM, the data may also be filtered based on the port associated with the sending process. The sending process's program ID,

the destination process ID, and the message header are written to the dispatch array. See the "Snapped Data Dump Layouts," Section 12.8 for the layout of the dumped data.

Example:

See the example for the F29 flag. The F31 flag is used on the SET:MON,FCN line instead of F29.

HAP Environment: AM/SM/CMP

Spare control flag.

HHH Environment: AM/SM/CMP

Combine the HUF, HIJ, HPD, and HPY flags.

See the HUF, HIJ, HPD, and HPY flags for details.

HIJ Environment: AM/SM/CMP

Count entries to and accumulated time in interject.

Description:

This flag allows the user to collect data on each entry to interject. For the SM and CMP, the data consists of a count of the number of entries to interject, the accumulated time spent in interject, and a count of the number of entries to interject with no work (OSsignals = 0).

For the AM, the data consists of a count of the number of entries to CM interject and the accumulated time in CM interject, and a count of the number of entries to CNI interject with work to do and the accumulated time in CNI interject. See the "Snapped Data Dump Layouts," Section 12.8 for the location of this data.

Example:

```
SET:MON,WTD,{AM|SM|CMP},HIJ;
```

HMH Environment: AM/SM/CMP

Create a histogram of segment lengths for various OSDS jobs.

Description:

This flag allows the user to construct histograms of segment lengths for various OSDS "jobs" (processes, interject, foreground, SMKP, *UNIX* RTR, for example). The histograms are constructed such that for a specific job type, an array of longs is defined in which each long represents a length of time.

The first long in the array represents segments of 0 time units, the second long represents segments of 1 time unit, the third long represents segments of 2 time units, and so forth. Each time the job runs, a long is pegged indicating how long of a segment the job ran (if the segment length is larger than the maximum length represented in the histogram, the maximum segment length in the histogram is pegged). Over a period of time a histogram is constructed showing how many segments of each length the job ran.

In the SM and CMP, a unit of time is 125 µsec and histograms are made of time spent in interject, foreground, and running processes. In addition, the histogram of time spent in running processes may be filtered based on program ID. This list indicates the histograms that are created for the SM and CMP, the range of segment lengths that are recorded in the histogram, and the index into the dispatch array at which the histogram begins.

Histogram for	Range in ms	Start Index in Dispatch Array
Foreground	0 - 128	0
Interject	0 - 128	0x4000
Processes	0 - 128	0x8000

In the AM, a unit of time is one millisecond and histograms are made of time spent in a wide variety of jobs. As in the SM and CMP, the histogram of time spent running processes (OKP and SMKP) may be filtered based on program ID. This list indicates the histograms that are created for the AM, the range of segment lengths that are recorded in the histogram, and the index into the dispatch array at which the histogram begins.

Histogram for	Range in ms	Start Index in Dispatch Array
OKP	0 - 511	0
UNIX RTR	0 - 255	0x800
OKP processes	0 - 511	0xc00
OKP interject	0 - 127	0x1400
OKP CNI work	0 - 127	0x1600
SMKP	0 - 1023	0x1800
SMKP processes	0 - 1023	0x2800
MSKP event	0 - 255	0x3800
MSKP fault	0 - 255	0x3c00
MSKP interrupt	0 - 255	0x4000

Segment lengths for OKP jobs may contain time spent in *UNIX* RTR priority levels 9 - 15 if these levels interrupt OKP.

Example:

```
SET:MON,{AM|SM|CMP},WTD,HMH;
SET:MON,{AM|CM|CMP},DATA[,PRG=aaaa][,PSG=bbbb];
```

Where:

`aaaa` (required for OKP, SM, and CMP histograms) is a specific program ID or one of the following values:

h'ff00 - All program IDs

`h'f000` - All program IDs except those running at OSDS priority 0

`bbbb` (required for SMKP histograms) is a specific program ID or one of the following values:

h'ff00 - All program IDs
`h'f000` - All program IDs except those running at OSDS priority 0

`HMX` Environment: MSKP

Count entries to and accumulated time in MSKP.

Description:

This flag allows the user to collect data on each exit from MSKP. The data consists of a two word entry that contains a count of the number of entries to MSKP and the accumulated time spent in MSKP. See the "Snapped Data Dump Layouts," Section 12.8 for the location of this data.

Example:

```
SET:MON,WTD,AM,HMX;
```

`HPD` Environment: OKP/SM/CMP

Count dispatches of and accumulated time in each program ID.

Description:

This flag allows the user to collect data for each dispatched program ID. The data consists of a two word entry that contains a count of the number of times the program ID is dispatched and the accumulated time spent in the program ID. The data is stored in an array indexed by the program ID in the monitor buffer (independent of the dispatch array). See the "Snapped Data Dump Layouts," Section 12.8 for the location of this data.

Example:

```
SET:MON,WTD,{AM|SM|CMP},HPD;
```

`HPY` Environment: AM/SM/CMP

Count dispatches of and accumulated time in each OSDS priority level.

Description:

This flag allows the user to collect data for each OSDS priority level (0-7). The data consists of a two word entry that contains a count of the number of times a process of the given priority is dispatched and the accumulated time spent in the process at the given priority. The data is stored in an array indexed by the process priority in the monitor buffer (independent of the dispatch array).

Example:

```
SET:MON,WTD,{AM|SM|CMP},HPY;
```

`HSP` Environment: SMKP

Count dispatches of and accumulated time in each program ID in SMKP.

Description:

This flag allows the user to collect data for each dispatched program ID in SMKP. The data consists of a two word entry that contains a count of the number of times the program ID is dispatched and the accumulated time spent in the program ID. The data is stored in an array indexed by the program ID in the monitor buffer (independent of the dispatch array). See the "Snapped Data Dump Layouts," Section 12.8 for the location of this data.

Example:

```
SET:MON,WTD,{AM|SM|CMP},HSP;
```

HSX Environment: AM

Count entries to and accumulated time in SMKP.

Description:

This flag allows the user to collect data on each exit from SMKP. The data consists of a two word entry that contains a count of the number of entries to SMKP and the accumulated time spent in SMKP.

See the "Snapped Data Dump Layouts," Section 12.8 for the location of this data.

Example:

```
SET:MON,WTD,AM,HSX;
```

HUF Environment: OKP/SM/CMP

Count entries to and accumulated time in OKP and **UNIX** *RTR (AM), or count entries to and accumulated time in foreground (SM).*

Description:

In the AM, this flag allows the user to collect data on each entry to OKP and *UNIX* RTR. The data consists of:

- A count of the number of times OKP was entered (which is the same as the number of times *UNIX* RTR was entered).
- A count of and accumulated time spent in OKP entries in which work was performed (interject or process dispatch).
- A count of the number of entries to OKP that exceeded 100 ms.
- The accumulated time of OKP entries which exceeded 2 ms.
- A count of and accumulated time in *UNIX* RTR entries in which work was performed (that is, segment length exceeded 1 ms).
- A count of the number of idle entries to *UNIX* RTR (that is, segment length was less than 1 ms).

In the SM, this flag allows the user to collect data on each exit from foreground. The data consists of a two word entry that contains a count of the number of entries to foreground and the accumulated time spent in foreground.

See the "Snapped Data Dump Layouts," Section 12.8 for the location of this data.

Example:

```
SET:MON,WTD,{AM|SM|CMP},HUF;
```

SEG Environment: OKP/SM/CMP

Record data concerning the number of consecutive segments a program ID runs.

Description:

This flag allows the user to collect data about the number of consecutive real time segments that a specific program ID runs. Real time segments are considered consecutive if the process *suspends* instead of *waits* between the two segments. By suspending, the process blocks all lower priority processes from running.

The user must specify a program ID and a threshold for the number of real time segments. The program must run at least as many consecutive segments as specified by the threshold before data will be recorded.

The recorded data contains a timestamp, the program ID, the number of consecutive segments, the total time of the consecutive segments, and the elapsed time between the start of the first segment and the end of the last segment. See the "Snapped Data Dump Layouts," Section 12.8 for the complete data layout.

Example:

```
SET:MON,{AM|SM|CMP},WTD,SEG;
SET:MON,{AM|SM|CMP},DATA,PRG=aaaa,PTM=bbbb;
```

Where:

aaaa	-	Program ID to check
bbbb	-	Minimum number of consecutive segments for which data will be recorded

SP1 - SP2 Environment: None

Spare control flags.

SP3 Environment: AM/SM/CMP

Automatically start and stop the monitor.

Description:

This flag allows the user to automatically start and stop the monitor based on the time of day. While the monitor is waiting for the start time to arrive, it is actually on; therefore, the monitor is using system real time while it is waiting for the start time to occur. Once the start time arrives, the monitor internally changes its what to do flags to reflect the operations requested by the user. The user must specify these operations by using the `SET:MON,WTD` and `SET:MON,DATA` commands before the `SP3` option is selected.

Example:

1. Specify the operations that the monitor should perform when automatically started:

   ```
   SET:MON,{AM|SM|CMP},WTD,...;
   ```

```
SET:MON,{AM|SM|CMP},DATA,...;
```

2. Specify when the monitor should automatically start and stop:

```
SET:MON,{AM|SM|CMP},DATA,PDA=h'aabbbbbb,PDM=h'cccccccc;
SET:MON,{AM|SM|CMP},WTD,SP3;
```

Where:

aa	-	Start day in number of days past the current day (current day = 0)
bbbbbb	-	Start time of day in number of seconds past midnight on the start day
cc	-	Stop day in number of days past the current day (current day = 0)
dddddd	-	Stop time of day in number of seconds past midnight on the stop day

3. Allow the monitor to wait for the start time.

```
ALW:MON,{AM|SM|CMP};
```

Note: The parameter format in this example is in hex for clarity.

The SP3 flag may not be used in conjunction with the DAT or SP6 flags because the flags share a common data area (each flag defines data using the PDA and PDM parameters in the SET:MON,DATA command).

SP4 Environment: OKP/SM/CMP

Inhibit the monitor on a match or mismatch condition.

Description:

This flag can only be used in conjunction with the DAT or SP6 flags (see the DAT or SP6 flag). If the condition specified by the DAT or SP6 flag occurs and SP4 is set, the monitor will be turned off.

If both DAT and SP6 are used, the condition that occurs first will cause the monitor to be inhibited.

This flag takes precedence over SP5.

Example:

```
SET:MON,WTD,SP4,{DAT|SP6};
```

SP5 Environment: OKP/SM/CMP

Restart the dispatch array index on a match or mismatch condition.

Description:

This flag can only be used in conjunction with the DAT, SP6, or DMH flags (see the DAT, SP6, or DMH flags). If the condition specified by the DAT, SP6, or DMH flag occurs and SP5 is set, the monitor's write pointer into the dispatch array is reset to the start of the array and the DOX flag is turned on.

If a combination of DAT, SP6, and DMH is used, the condition that occurs first will cause the write pointer to be reset; the SP5, SP6, DMH, and DAT flags will be turned off.

Example:

```
SET:MON,WTD,SP5,{DAT|SP6|DMH};
```

SP6 Environment: SM

Filter per-event data for PCBLA match or mismatch.

Description:

This flag is similar to the DAT flag. It can only be used in conjunction with the DPD or DPY flags; that is, this flag is not checked unless either DPD or DPY is on. SP6 allows the user to filter per-event data dumps based on the contents of the running process's PCBLA. It also allows the user to start (actually, execute a BEG) or stop the monitor based on the contents of the running process's PCBLA when SP6 is used in conjunction with the SP4 and SP5 flags.

When SP6 is used to filter per-event data snaps for the DPD or DPY flags, the user specifies the offset from the start of the running process's PCBLA to be checked, a mask for the contents of the address, the data to check against, and whether the monitor should check for a match or a mismatch. If the specified condition does not exist, the DPD or DPY data dump is aborted.

Example:

```
SET:MON,SM=SM_number,WTD,SP6;
SET:MON,SM=SM_number,DATA,PDA=aaaaaaaa,PDM=bbbbbbbb,LA1=cccccccc;
```

Where:

aaaaaaaa - Data value to match or mismatch
bbbbbbbb - Mask for the data (used as an AND mask)
cccccccc - PCBLA offset to match or mismatch on
0x00000000 + offset for a match
0x80000000 + offset for a mismatch

Note: The offset must be word aligned (that is, end in 0x0, 0x4, 0x8, or 0xc).

Either the DPD or DPY flag must also be set for this example to work (see the DPD or DPY flags). See the SP4 and SP5 flags for information about how they are used.

SP7 Environment: OKP/SM/CMP

Timed data dump.

Description:

This flag allows the user to snap up to 46 memory locations at periodic intervals. The snaps are actually performed at the beginning of interject, so the period between dumps will not be exact; instead, it will be at least the specified period. The snap will occur at the first entry to interject after the specified period has elapsed.

All of the data for this flag is stored in the dispatch array. The 46 addresses that the user wishes to snap are stored in the first 46 longs of

the dispatch array and the periodic dumps begin at the 48th long (an offset of 0xc0 from the start of the dispatch array).

Example:

```
SET:MON,{AM|SM|CMP},WTD,SP7;
SET:MON,{AM|SM|CMP},DATA,PTM=aaaa;
SET:MON,{AM|SM|CMP},SPEC,S00=a00[,S01=a01],...[,S45=a45];
```

Where:

aaaa	-	Interval between snaps in milliseconds. It is recommended that this value not be less than 500 ms
a00-a45	-	The addresses to be snapped. If less than 46 address are specified, the last address must be 0

Note: The addresses must be on short word boundaries (that is, end in 0x0, 0x2, 0x4, 0x6, 0x8, 0xa, 0xc, or 0xe).

The data is snapped as longs (four bytes).

12.4.9 What To Dump Flags (Per-Event Data)

The following list provides a short description and example of each what to dump flag. For more information about each flag, see Tables 12.4-8, 12.4-9, 12.4-10, and 12.4-11 in the "Dumped Data Description," Section 12.4.10 that follows.

AAA Description:

Combine the ACL, APP, AID, and ASA flags.

Example:

```
SET:MON,{AM|SM|CMP},WTD,AAA;
```

ACL Description:

Snap the clock.

Example:

```
SET:MON,{AM|SM|CMP},WTD,ACL;
```

AID Description:

Snap the process priority, program ID, and time in segment.

Example:

```
SET:MON,{AM|SM|CMP},WTD,AID;
```

ADM Environment: MSKP

Description:

Perform the same function as the ADO flag, except for the MSKP.

Note: Only one of the ADO, ADM, and ADS flags may be used at one time in the AM.

ADO Environment: OKP/SM/CMP

Description:

Snap user specified memory addresses (up to six data addresses).

Example:

```
SET:MON,{AM|SM|CMP},WTD,ADO;
SET:MON,{AM|SM|CMP},DATA,AD1=addr1[,AD2=addr2]...[,AD6=addr6];
```

where addr1 - addr6 are the addresses to be snapped. An address must be short word aligned (that is, end in 0x0, 0x2, 0x4, 0x6, 0x8, 0xa, 0xc, or 0xe). A long will be dumped for each address. If all six addresses are not specified, the last address must be 0.

Note: Only one of the ADO, ADM, and ADS flags may be used at one time in the AM.

ADS Environment: SMKP

Description:

Perform the same function as the ADO flag, except for the SMKP.

Note: Only one of the ADO, ADM, and ADS flags may be used at one time in the AM.

APC Environment: SM

Description:

Snap user specified PCBLA memory (up to three words from the PCBLA).

Example:

```
SET:MON,SM=SM_number,WTD,APC;
SET:MON,SM=SM_number,DATA,LA1=idx1[,LA2=idx2][,LA3=idx3];
```

where idx1 - idx3 are the indexes into the running process's PCBLA to be snapped. An index must be short word aligned (that is, end in 0x0, 0x2, 0x4, 0x6, 0x8, 0xa, 0xc, or 0xe). A long will be dumped for each index. If all three indexes are not specified, the last index must be 0.

APP Description:

Snap the program ID and PCB index.

Example:

```
SET:MON,{AM|SM|CMP},WTD,APP;
```

ASA Description:

Snap the starting address for the process's segment, *UNIX* RTR event flags, OSDS signals, or MSKP specific data.

Example:

```
SET:MON,{AM|SM|CMP},WTD,ASA;
```

12.4.10 Dumped Data Description

12.4.10.1 Introduction to Dump Description Tables

Tables 12.4-8, 12.4-9, 12.4-10, and 12.4-11 in this section provide a description of the data that is dumped by each what to dump flag for the various per-event data flags (DIJ, DUF, DSX, DSP, DMX, DMP, DPD, and DPY). There is one table for each environment.

The tables also indicate the format of the dumped data in the dispatch array. The data is dumped in the order shown in the tables (that is, moving left to right in the table is

moving from lower to higher addresses in the dispatch array). Remember that the AAA flag is a combination of the ACL, APP, AID, and ASA flags.

12.4.10.2 SM and CMP Data Dump Description

Table 12.4-8 describes the data that is dumped by each "what to dump" flag for the various per-event data flags in the SM and CMP.

Table 12.4-8 — SM and CMP Data Dump Descriptions

What to Do Flags	What to Dump Flags					
	ACL	APP	AID	ASA	ADO	APC
DUF	Billing clock at entry to foreground	N/A	Foreground segment length	PIC interrupt identifier	User specified memory data dumps, up to 6 longs	N/A
Format:	aaaaaaaa	00000000	005Ftttt	bbbbbbbb	1-6 longs	00000000
DPD	Billing clock at process dispatch	Program ID and PCB number of process	OSDS process priority and segment length	Start address of this segment's execution	User specified memory data dumps, up to 6 longs	User specified PCBLA data dump, up to 3 longs
Format:	aaaaaaaa	ddddpppp	yy5Dtttt	cccccccc	1-6 longs	1-3 longs
DPY	Same as DPD	Same as DPD	Same as DPD	Same as DPD	Same as DPD	Same as DPD
Format:	aaaaaaaa	ddddpppp	yy5Dtttt	cccccccc	1-6 longs	1-3 longs
DIJ	Billing clock at start of interject	N/A	Interject segment length	Value of OSsignals at start of interject	User specified memory data dumps, up to 6 longs	N/A
Format:	aaaaaaaa	00000000	0051tttt	eeeeeeee	1-6 longs	00000000

Key:

aaaaaaaa - Billing clock in 125 µsec units.

bbbbbbbb - PIC interrupt number. For nested interrupts, only the first interrupt number is recorded.

cccccccc - Starting address for the process segment.

dddd - Process program ID.

eeeeeeee - Signal flags on entry to interject; that is, the value of OSsignals.

pppp - PCB number.

tttt - Segment length in 125 µsec units.

yy - Process OSDS priority level.

12.4.10.3 OKP Data Dump Description

Table 12.4-9 describes the data that is dumped by each "what to dump" flag for the various per-event data flags in the OKP.

Table 12.4-9 — OKP Data Dump Descriptions

What to Do Flags	What to Dump Flags				
	ACL	APP	AID	ASA	ADO
DUF (OKP entry)	AM real time clock at OKP entry	N/A	*UNIX* RTR segment time	*UNIX* RTR event flags	User specified memory data dumps, up to 6 longs
Format:	aaaaaaaa	00000000	008Etttt	bbbbbbbb	1-6 longs
DUF (OKP exit)	AM real time clock at OKP exit	N/A	OKP segment time	N/A	User specified memory data dumps, up to 6 longs
Format:	aaaaaaaa	00000000	0083tttt	00000000	1-6 longs
DPD	AM real time clock at process dispatch	Program ID and PCB number of process	OSDS process priority and process segment length	Start address of this segment's execution	User specified memory data dumps, up to 6 longs
Format:	aaaaaaaa	ddddpppp	yy8Dtttt	cccccccc	1-6 longs
DPY	Same as DPD	Same as DPD	Same as DPD	Same as DPD	Same as DPD
Format:	aaaaaaaa	ddddpppp	yy8Dtttt	cccccccc	1-6 longs
DIJ (CM interject entry)	AM real time at start of interject	N/A	CM interject segment length	N/A	User specified memory data dumps, up to 6 longs
Format:	aaaaaaaa	00000000	0081tttt	00000000	1-6 longs
DIJ (CNI interject entry)	AM real time at start of CNI interject	N/A	CNI interject segment length	N/A	User specified memory data dumps, up to 6 longs
Format:	aaaaaaaa	00000000	008Ctttt	00000000	1-6 longs

Key:

aaaaaaaa - AM real time clock in milliseconds.

bbbbbbbb - *UNIX* RTR event flags (defined in hdr/dmert/evflag.h).

cccccccc - Starting address for the process segment.

dddd - Process program ID.

pppp - PCB number.

tttt - Segment length in milliseconds.

yy - Process OSDS priority level.

12.4.10.4 SMKP Data Dump Description

Table 12.4-10 describes the data that is dumped by each what to dump flag for the various per-event data flags in the SMKP.

Table 12.4-10 — SMKP Data Dump Descriptions

	What to Dump Flags				
What to Do Flags	ACL	APP	AID	ASA	ADS
DSX (SMKP entry)	AM real time clock at SMKP entry	N/A	N/A	*UNIX* RTR event flags	User specified memory data dumps, up to 6 longs
Format:	aaaaaaaa	00000000	00BE0000	bbbbbbbb	1-6 longs
DSX (SMKP exit)	AM real time clock at SMKP exit	N/A	SMKP segment time	N/A	User specified memory data dumps, up to 6 longs
Format:	aaaaaaaa	00000000	00B3tttt	00000000	1-6 longs
DSP	AM real time clock at process dispatch	Program ID and PCB number of process	OSDS process priority and process segment length	Start address of this segment's execution	User specified memory data dumps, up to 6 longs
Format:	aaaaaaaa	ddddpppp	yyBDtttt	cccccccc	1-6 longs

Key:

aaaaaaaa - AM real time clock in milliseconds.

bbbbbbbb - *UNIX* RTR event flags (defined in hdr/dmert/evflag.h).

cccccccc - Starting address for the process segment.

dddd - Process program ID.

pppp - PCB number.

tttt - Segment length in milliseconds.

yy - Process OSDS priority level.

12.4.10.5 MSKP Data Dump Description

Table 12.4-11 describes the data that is dumped by each what to dump flag for the various per-event data flags in the MSKP.

Table 12.4-11 — MSKP Data Dump Descriptions

	What to Dump Flags				
What to Do Flags	ACL	APP	AID	ASA	ADM
DMX (MSKP entry)	AM real time clock at MSKP entry	N/A	MSKP program point ID	MSKP specific data, depends on the program point ID	User specified memory data dumps, up to 6 longs
Format:	aaaaaaaa	00000000	ccAE0000	bbbbbbbb	1-6 longs
DMX (MSKP exit)	AM real time clock at MSKP exit	N/A	MSKP segment time and program point ID	MSKP specific data, depends on the program point ID	User specified memory data dumps, up to 6 longs
Format:	aaaaaaaa	00000000	ccA3tttt	bbbbbbbb	1-6 longs
DMP	AM real time clock at MSKP job point	N/A	MSKP program point ID	MSKP specific data, depends on the program point ID	User specified memory data dumps, up to 6 longs
Format:	aaaaaaaa	00000000	ccAD0000	bbbbbbbb	1-6 longs

Key:
aaaaaaaa - AM real time clock in milliseconds.
bbbbbbbb - Data depends on the value of the MSKP program point ID (cc).
cc - MSKP program point ID.
tttt - Segment length in milliseconds.

This table defines the values for bbbbbbbb.

If cc is ...	**Then bbbbbbbb is ...**
0xa3 - 0xa7 0xaa - 0xaf 0xbf	*UNIX* RTR event flags (defined in hdr/dmert/evflag.h).
0xc3	MSCU side number
anything else	00000000

12.5 INPUT COMMANDS TO CLEAR AND DUMP THE BUFFER

12.5.1 Section Description

This section discusses the input commands that are used to perform the following functions:

- Zero the OSDS monitor buffer or portions of the buffer
- Output data to the ROP or to a file in the AM
- Kill a monitor job.

12.5.2 Zero The Monitor Buffer

The following commands are used to zero the entire monitor buffer or portions of the buffer.

- Zero the entire buffer.

  ```
  clr:mon,sm=x,all;
  ```

- Zero the control data area.

  ```
  clr:mon,sm=x,ctl;
  ```

- Zero the peg and time data area (i.e., OSDS usage flags, `HHH`).

  ```
  clr:mon,sm=x,pta;
  ```

- Zero the dispatch array.

  ```
  clr:mon,sm=x,dpa;
  ```

12.5.3 Output Data

12.5.3.1 Input Messages for Data Output

This section presents the messages that are used to write data to the ROP or to a file in the AM, and lists the types of output messages that the OSDS monitor prints to the ROP. The following input messages write data to the ROP or to a file in the AM. If data is to be output to the ROP, use `rop` in the command line. Use `fn="full_file_pathname"` to write data to a file (use the `/cdmp` or the `/unixa/users` directory for the dump).

- Print the formatted control data area.

  ```
  op:mon,sm=x,wtd,ctl,rop;
  ```

  An `OP MON CTL` output message is printed.

  This is a good way to see the flags and data that are set. If the first word (`ACT`) is `0`, the monitor is off.

- Dump the entire buffer (control data, peg and time data, and dispatch array).

  Note: Always send the entire buffer to a file, *never* to the ROP.

  ```
  op:mon,sm=x,dsp,dpa,fn="full_file_pathname";
  ```

- Dump the first *x* words in the dispatch array.

  ```
  op:mon,sm=x,dsp,dpf=number_of_words,rop;
  ```

  An `OP MON DSP` output message is printed.

- Dump a range of words in the dispatch array; that is, from one index in the array to some other index in the array.

  ```
  op:mon,sm=x,dsp,dpr=start_idx-end_idx,rop;
  ```

An OP MON DSP output message is printed.

Note: start_idx and end_idx should be a number of longs (that is, not a byte index into the array).

- Dump the last *x* words in the dispatch array.

  op:mon,sm=x,dsp,dpl=number_of_words,rop;

  An OP MON DSP output message is printed.

- Dump peg and time information for a specific program ID.

  An OP MON PID output message is printed.

  op:mon,sm=x,pid,prs=program_id;

- Dump peg and time information for all program IDs.

  op:mon,sm=x,pid,prs=ALL;

  An OP MON PID output message is printed.

- Dump peg and time information for the operating system (including process priority levels).

  op:mon,sm=x,pid,prt;

  An OP MON PID SM output message is printed for the SM.

  For the AM and CMP, the output messages are OP MON PID AM and OP MON PID CMP, respectively.

12.5.3.2 Output Messages

Table 12.5-1 lists the output messages for the AM, SM, and CMP that are provided by the OSDS monitor. For complete details on the meaning and format of these output messages, see the 235-600-750, *Output Messages Manual*.

Table 12.5-1 — OSDS Monitor Output Messages

Message	Description
OP MON CTL	Control data dump
OP MON DSP	Dispatch array dump
OP MON PID	Peg and time information dump for program IDs
OP MON PID AM	Peg and time operating system data dump (AM)
OP MON PID SM	Peg and time operating system data dump (SM)
OP MON PID CMP	Peg and time operating system data dump (CMP)

12.5.4 Kill A Monitor Job

The following input message is used to kill an OSDS monitor job.

stp:op,mon,sm=x;

12.6 USEFUL INPUT MESSAGE SEQUENCES

12.6.1 Section Description

This section provides the sequence of input messages that are used to perform the following functions:

- Snap per-event process and messaging data
- Snap foreground and messaging data
- Snap interject and messaging data
- Enable or disable the monitor based on a data or PCBLA match or mismatch
- Perform an SDL trace.

12.6.2 Snap Per-Event Process and Messaging Data

12.6.2.1 Snap Messages, No Processes

The following procedure is used to snap data for all messages, but no processes.

1. Zero the entire buffer.
   ```
   clr:mon,sm=x,all;
   ```
2. Set the data information for all program IDs and all message types.
   ```
   set:mon,data,sm=x,prg=h'ff00,ad5=0,ad6=h'f000;
   ```
3. Turn on message snaps.
   ```
   set:mon,fcn,f31,f29,sm=x;
   ```
4. Turn the monitor on to allow data gathering.
   ```
   alw:mon,sm=x;
   ```
5. Turn the monitor off before analyzing the results.
   ```
   inh:mon,sm=x;
   ```

12.6.2.2 Snap Messages and Processes

The following procedure is used to snap data for all messages and all processes.

1. Zero the entire buffer.
   ```
   clr:mon,sm=x,all;
   ```
2. Set the what to do and what to dump (WTD) flags.
   ```
   set:mon,wtd,sm=x,dpd,aaa;
   ```
3. Set the data information for all program IDs and all message types.
   ```
   set:mon,data,sm=x,prg=h'ff00,ad5=0,ad6=h'f000,ptm=0;
   ```
4. Turn on message snaps.
   ```
   set:mon,fcn,sm=x,f29,f31;
   ```
5. Turn the monitor on to allow data gathering.
   ```
   alw:mon,sm=x;
   ```
6. Turn the monitor off before analyzing the results.
   ```
   inh:mon,sm=x;
   ```

12.6.2.3 Snap Processes, No Messages

The following procedure is used to snap data for all processes, but not for messages.

1. Zero the entire buffer.
   ```
   clr:mon,sm=x,all;
   ```
2. Set the WTD flags.
   ```
   set:mon,wtd,sm=x,dpd,aaa;
   ```
3. Set the data information.
   ```
   set:mon,data,sm=x,prg=h'ff00,ptm=0;
   ```
4. Turn the monitor on to allow data gathering.
   ```
   alw:mon,sm=x;
   ```
5. Turn the monitor off before analyzing the results.
   ```
   inh:mon,sm=x;
   ```

12.6.2.4 Snap Processes and Messages for a Port

The following procedure is used to snap data for processes and messages associated with a specific port.

1. Zero the entire buffer.
   ```
   clr:mon,sm=all;
   ```
2. Set the WTD flags.
   ```
   set:mon,wtd,sm=x,dpd,dmh,aaa;
   ```
3. Set the data information for all program IDs, all messages, and a specific port name.
   ```
   set:mon,data,sm=x,prg=h'ff00,ad5=0,ad6=h'f000,
   prt=h'port_name,ptm=0;
   ```
4. Turn on message snaps.
   ```
   set:mon,fcn,sm=x,f29,f31;
   ```
5. Turn the monitor on to allow data gathering.
   ```
   alw:mon,sm=x;
   ```
6. Turn the monitor off before analyzing the results.
   ```
   inh:mon,sm=x;
   ```

12.6.2.5 Snap Processes and Messages and/or State Definition Language (SDL) Trace Event Data for Four Ports

The following procedure is used to snap data for all processes except those running at priority 0, their messages, and SDL feature execution (FEX) traces. The data is filtered on four port names.

1. Zero the entire buffer.
   ```
   clr:mon,sm=x,all;
   ```
2. Set the WTD flags.
   ```
   set:mon,wtd,sm=x,dpd,dmh,aaa;
   ```
3. Set the data information for all program IDs except those at priority 0, all message types except those for processes at priority 0, and four port names.
   ```
   set:mon,data,sm=x,prg=h'f000,ad5=0,ad6=h'f000,
      prt=h'ffff,ptm=0;
   ```

```
set:mon,spec,sm=x,s00=p1p1,s01=p2p2,s02=p3p3,s03=p4p4;
```

where `p1p1 - p4p4` are the four port names to filter on.

Note: For OSDS messages, set all four spare control flags to a non-zero value.

4. Turn on message snaps.
   ```
   set:mon,fcn,f22,f31,f29,sm=x;
   ```
5. Turn the monitor on to allow data gathering.
   ```
   alw:mon,sm=x;
   ```
6. Turn the monitor off before analyzing the results.
   ```
   inh:mon,sm=x;
   ```

12.6.2.6 Snap Processes for a Specific Program ID

The following procedure is used to snap data for processes with a specific program ID and their messages.

1. Zero the entire buffer.
   ```
   clr:mon,sm=x,all;
   ```
2. Set the WTD flags.
   ```
   set:mon,wtd,sm=x,dpd,aaa;
   ```
3. Set the data information.
   ```
   set:mon,data,sm=x,prg=program_id,ad5=0,ad6=h'f000,ptm=0;
   ```
4. Turn on message snaps.
   ```
   set:mon,fcn,f31,f29,sm=x;
   ```
5. Turn the monitor on to allow data gathering.
   ```
   alw:mon,sm=x;
   ```
6. Turn the monitor off before analyzing the results.
   ```
   inh:mon,sm=x;
   ```

12.6.2.7 Snap Processes at a Specific Priority

The following procedure is used to snap data for processes at a specific priority that run longer than 50 ms.

1. Zero the entire buffer.
   ```
   clr:mon,sm=x,all;
   ```
2. Set the WTD flags.
   ```
   set:mon,wtd,sm=x,dpy,aaa;
   ```
3. Set the data information for a specific priority and a segment length of 50 ms (400 × 125 µsec).
   ```
   set:mon,data,sm=x,pri=priority,ptm=400;
   ```
4. Turn the monitor on to allow data gathering.
   ```
   alw:mon,sm=x;
   ```
5. Turn the monitor off before analyzing the results.
   ```
   inh:mon,sm=x;
   ```

12.6.3 Snap Foreground And Messaging Data

12.6.3.1 Snap Foreground Work, All Messages

The following procedure is used to snap data for foreground work and all messages.

1. Zero the entire buffer.
   ```
   clr:mon,sm=x,all;
   ```
2. Set the WTD flags.
   ```
   set:mon,wtd,sm=x,duf,aaa;
   ```
3. Set the data information for all program IDs and all message types.
   ```
   set:mon,data,sm=x,prg=h'ff00,ad6=h'f000,ad5=0,ptm=0;
   ```
4. Turn on message snaps.
   ```
   set:mon,fcn,f31,f29,sm=x;
   ```
5. Turn the monitor on to allow data gathering.
   ```
   alw:mon,sm=x;
   ```
6. Turn the monitor off before analyzing the results.
   ```
   inh:mon,sm=x;
   ```

12.6.3.2 Snap Specific Foreground Entries, No Messages

The following procedure is used to snap data for foreground entries that are longer than 1 ms, but not for messages.

1. Zero the entire buffer.
   ```
   clr:mon,sm=x,all;
   ```
2. Set the WTD flags.
   ```
   set:mon,wtd,sm=x,duf,aaa;
   ```
3. Set the data information for segments longer than 1 ms (8×125 µsec).
   ```
   set:mon,data,sm=x,ptm=8;
   ```
4. Turn the monitor on to allow data gathering.
   ```
   alw:mon,sm=x;
   ```
5. Turn the monitor off before analyzing the results.
   ```
   inh:mon,sm=x;
   ```

12.6.4 Snap Interject and Messaging Data

12.6.4.1 Snap Interject Work, All Messages

The following procedure is used to snap data for interject work and all messages.

1. Zero the entire buffer.
   ```
   clr:mon,sm=x,all;
   ```
2. Set the WTD flags.
   ```
   set:mon,wtd,sm=x,dij,aaa;
   ```
3. Set the data information.
   ```
   set:mon,data,sm=x,prg=h'ff00,ad6=h'f000,ad5=0,ptm=0;
   ```
4. Turn on message snaps.
   ```
   set:mon,fcn,sm=x,f29,f31;
   ```
5. Turn the monitor on to allow data gathering.

```
alw:mon,sm=x;
```

6. Turn the monitor off before analyzing the results.

```
inh:mon,sm=x;
```

12.6.4.2 Snap Interject Work, No Messages

The following procedure is used to snap data for interject work, but not for messages.

1. Zero the entire buffer.

```
clr:mon,sm=x,all;
```

2. Set the WTD flags.

```
set:mon,wtd,sm=x,dij,aaa;
```

3. Set the data information. (The `clr` command should set `ptm` = 0, but this command could also be used.)

```
set:mon,data,sm=x,ptm=0;
```

4. Turn the monitor on to allow data gathering.

```
alw:mon,sm=x;
```

5. Turn the monitor off before analyzing the results.

```
inh:mon,sm=x;
```

12.6.5 Enable or Disable Monitor on Match or Mismatch

12.6.5.1 Use Input Control Flags

The input control flags in the examples in this section must be used in conjunction with the `DPD` flag or the `DPY` flag.

12.6.5.2 Enable Monitor on Data Match

The following procedure is used to enable the monitor on a data match, then snap data for all processes.

1. Zero the entire buffer.

```
clr:mon,sm=x,all;
```

2. Set the WTD flags.

```
set:mon,sm=x,wtd,dat,sp5,dpd;
```

3. Set the data information for all program IDs and the matching data.

```
set:mon,sm=x,data,pda=data_to_match,pdm=mask,
   ad1=addr_of_data,prg=h'ff00,ptm=0;
```

4. Turn the monitor on to allow data gathering.

```
alw:mon,sm=x;
```

5. Turn the monitor off before analyzing the results.

```
inh:mon,sm=x;
```

12.6.5.3 Disable Monitor On Data Mismatch

The following procedure is used to disable the monitor on a data mismatch, and record processes at priority 3 prior to the mismatch.

1. Zero the entire buffer.

```
clr:mon,sm=x,all;
```

2. Set the WTD flags.

```
set:mon,sm=x,wtd,dat,sp4,dpy,aaa;
```

3. Set the data information for priority 3 and the mismatch information. The address must have the high bit set for a mismatch.

```
set:mon,sm=x,data,pda=data_to_mismatch,pdm=mask,
   ad1=addr_of_data,pri=3,ptm=0;
```

4. Turn the monitor on to allow data gathering.

```
alw:mon,sm=x;
```

12.6.5.4 Disable Monitor on PCBLA Match

The following procedure is used to disable the monitor on a PCBLA match, and snap data for all processes and messages prior to the match.

1. Zero the entire buffer.

```
clr:mon,sm=x,all;
```

2. Set the WTD flags.

```
set:mon,sm=x,wtd,sp6,sp5,dpd,aaa;
```

3. Set the data information for all program IDs, all message types, and the matching data.

```
set:mon,sm=x,data,pda=data_to_match,pdm=mask,la1=pcbla_idx,
   prg=h'ff00,ad5=0,ad6=h'f000,ptm=0;
```

4. Turn on message snaps.

```
set:mon,fcn,f31,f29,sm=x;
```

5. Turn the monitor on to allow data gathering.

```
alw:mon,sm=x;
```

12.6.6 SDL Trace

12.6.6.1 SDL Trace for a Given SM

The following procedure is used to perform an SDL trace for all calls on a given SM.

1. Activate the monitor.

   Zero the entire buffer area.

```
clr:mon,sm=x,all;
```

   Set the F22 flag to indicate an SDL trace.

```
set:mon,sm=x,fcn,f22;
```

   Initalize the monitor for an SDL trace of all processes.

```
set:mon,sm=x,data,prg=h'ff00;
```

   Allow the monitor to run.

```
alw:mon,sm=x;
```

2. Make the desired phone calls.

3. Turn the monitor off and dump the trace information.

```
inh:mon,sm=x;
op:mon,sm=x,dsp,dpa,fm="/cdmp/sdltrc";
```

12.6.6.2 SDL Trace for a Given Port

The following procedure is used to perform an SDL trace for all calls associated with a given port.

1. Activate the monitor.

   Zero the entire buffer area.

   ```
   clr:mon,sm=x,all;
   ```

   Set the F22 flag to indicate an SDL trace.

   ```
   set:mon,sm=x,fcn,f22;
   ```

   Initialize the monitor for an SDL trace of all processes associated with the port 83cb.

   ```
   set:mon,sm=x,data,prg=h'ff00,prt=h'83cb;
   ```

   Allow the monitor to run.

   ```
   alw:mon,sm=x;
   ```

2. Make the desired phone calls on the specified port.

3. Turn the monitor off and dump the trace information.

   ```
   inh:mon,sm=x;
   op:mon,sm=x,dsp,dpa,fn="/cdmp/sdltrc";
   ```

12.6.6.3 SDL Trace for an Assert

The following procedure is used to perform an SDL trace for data associated with an assert.

1. Activate the monitor.

   Zero the entire buffer area.

   ```
   clr:mon,sm=x,all;
   ```

   Set the F22 flag to indicate an SDL trace.

   ```
   set:mon,sm=x,fcn,f22;
   ```

   Initialize the monitor for an SDL trace of all processes and all ports.

   ```
   set:mon,sm=x,data,prg=h'ff00;
   ```

   Allow the monitor to run.

   ```
   alw:mon,sm=x;
   ```

2. Enter the generic utilities (UT) breakpoint to inhibit the monitor when the assert occurs. You must know where the assert is fired.

   ```
   when:ut:sm=x,addr=h'yyyy,opc=h'zzzz,hit=1!
   load:ut:sm=x,gvar="SIhistory",size=4,l=4,val=0!
   end:ut:sm=x,when;
   alw:ut:sm=x,util;
   ```

   Where:

yyyy	-	The address where the assert is called. This should be available from the ROP stack back trace (SBT) associated with this assert.
zzzz	-	The opcode associated with the address.

3. Wait for the assert to occur.
4. Dump the SDL trace information.

   ```
   op:mon,sm=x,dsp,dpa,fn="cdmp/sdltrc";
   ```

   In the output, the last entries in the dispatch array show the SDL trace leading up to the assert.

12.6.6.4 SDL Trace for an SM Overload

The following procedure is used to perform an SDL trace for the OSDS dispatch times and data associated with an SM overload condition.

1. Activate the monitor.

   Zero the entire buffer area.

   ```
   clr:mon,sm=x,all;
   ```

   Collect OSDS data.

   ```
   set:mon,sm=x,wtd,hhh,aaa,dpd;
   ```

   Set the appropriate flags.

   ```
   set:mon,sm=x,fcn,f16,f17,f22,f23,f29,f31;
   ```

   Where:

   F16, F17 - Inhibit the monitor on overload
   F22 - Provide an SDL trace
   F23 - Dump the real time clock
   F29, F31 - Provide an OSDS message trace

   Initialize the monitor for an SDL trace of all processes, all ports, and all OSDS messages.

   ```
   set:mon,sm=x,data,prg=h'ff00,ad6=h'f000;
   ```

   Allow the monitor to run.

   ```
   alw:mon,sm=x;
   ```
2. Wait for the overload condition to occur.
3. Dump the SDL trace information.

   ```
   op:mon,sm=x,dsp,dpa,fn="/cdmp/sdltrc";
   ```

12.7 OSDS MONITOR BUFFER LAYOUTS

12.7.1 Section Description

This section provides the OSDS monitor buffer layout and buffer word content for the AM, SM, and CMP.

12.7.2 Buffer Layout for The AM

The OSDS monitor AM buffer layout for the 5E10 software release is depicted in Figure 12.7-1. The OSDS monitor AM buffer is contained in the `OShisarray[]` array. This information is from the file `hdr/os/OSmons.h`.

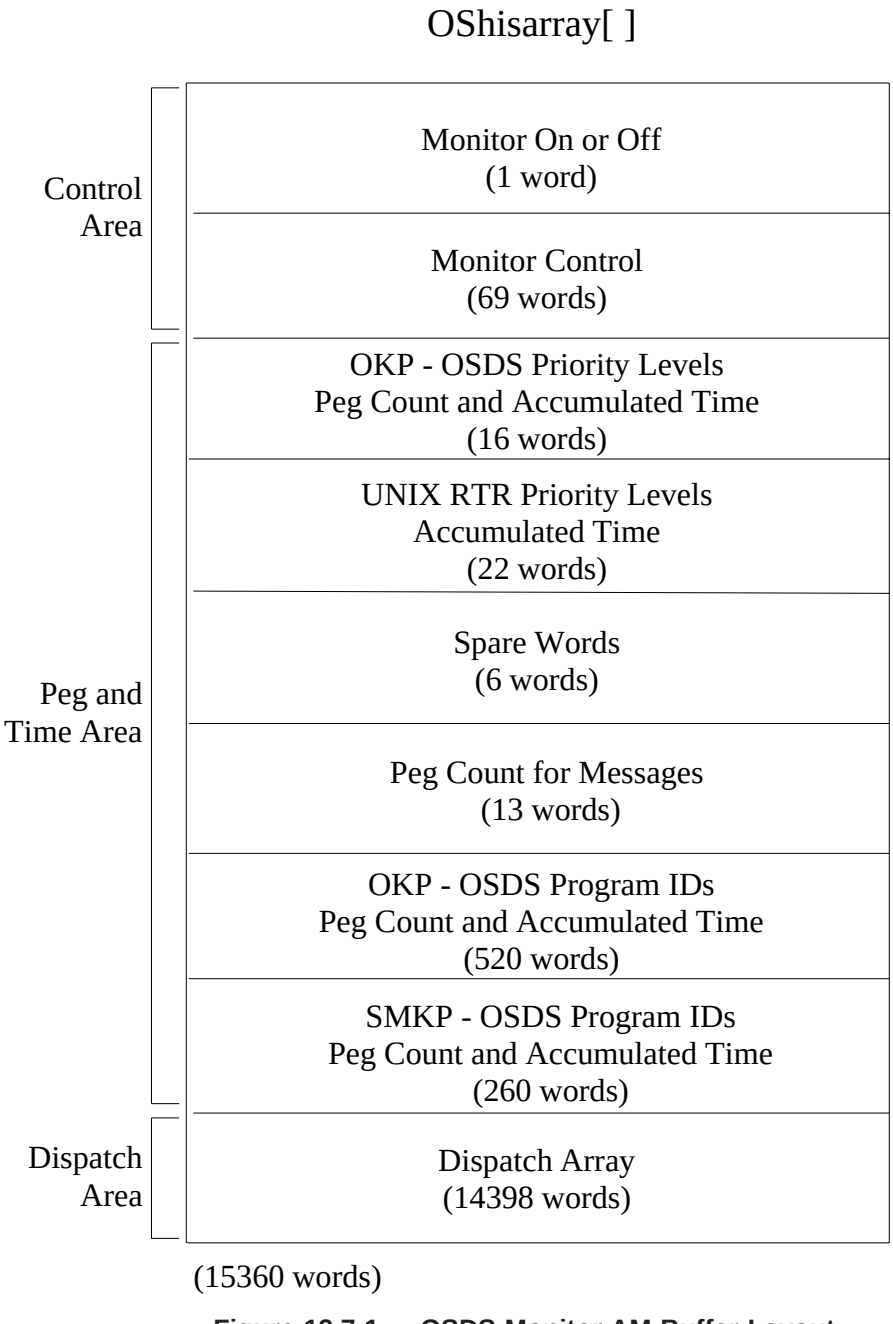


Figure 12.7-1 — OSDS Monitor AM Buffer Layout

12.7.3 AM Buffer Word Content

The AM buffer word content is described in Table 12.7-1.

Table 12.7-1 — OSDS Monitor AM Buffer Word Structure

Word	Symbol	Description
0	OSCONTRL	Master control word for monitor routines: 0 = off, !0 = on
1	OSCINDEX	Byte index into dispatch array
2	OSCHDFLG	What to do flags
3	OSCDDATA	What to dump flags
4	OSCDUMP1	Memory address to dump or 0 - flag AD1
5	OSCDUMP2	Memory address to dump or 0 - flag AD2
6	OSCDUMP3	Memory address to dump or 0 - flag AD3
7	OSCDUMP4	Memory address to dump or 0 - flag AD4
8	OSCDUMP5	Memory address to dump or 0 - flag AD5
9	OSCDUMP6	Memory address to dump or 0 - flag AD6
10	OSCPROGI	Program ID(s) to snap data for on dispatch - flag PRG
11	OSCPRIOR	Priority level to snap data for on dispatch - flag PRI
12	OSCPTIME	Process dispatch time filter value for loading the dispatch array - flagPTM. Dispatch times less than this value will not be snapped.
13	OSCPROGS	SMKP program ID for data dispatch snap - flag PSG
14	OSCSTIME	SMKP process dispatch time filter - flag PST
15	OSCAMDMV	OKP match or mismatch data value - flag PDA
16	OSCAMDMM	OKP match or mismatch mask word, 1 in bits to be matched - flag PDM
17	OSCFWORD	Control word for special function execution - flag CTL
18	OSCFFLAG	Special function execution flags - flag Fii
19	OSCSPAR1	Spare word
20	OSCSPAR2	Spare word
21	OSCSPAR3	Spare word
22	OSCSPAR4	Spare word
23	OSCSPAR5	Spare word
24	OSCSPAR6	Spare word
25	OSCSPAR7	Spare word
26	OSCSPAR8	Spare word
27	OSCSPAR9	Spare word
28	OSCSPARa	Spare word
29	OSCSPARb	Spare word
30	OSCSPARc	Spare word
31	OSCSPARd	Spare word
32	OSAUXENT	Real time clock at entry to *UNIX*[a] RTR
33	OSAOKENT	Real time clock at entry to OKP
34	OSSBGACT	Flag for background loading of dispatch array

See note(s) at end of table.

Table 12.7-1 — OSDS Monitor AM Buffer Word Structure (Contd)

Word	Symbol	Description
35	OSSVFLAG	Save function flag bits word (OSCFFLAG) until time to start monitor
36	OSSVWORD	Save function control word (OSCFWORD) until time to start monitor
37	OSSVSDAY	Record the day the monitor started
38	OSSVSTOD	Record the time of day the monitor started
39	OSSVSEGL	Save process segment length for monitor special functions
40	OSVODNSA	Save next segment address, OSsignals, or PIC interrupt value
41	OSVOPIDP	Save program ID and PCB index (or 0)
42	OSVOPPDT	Save priority, event type, and segment time
43	OSVOCLKB	Save clock at beginning of job
44	OSVODDU1	Save dumped data - address slot 1
45	OSVODDU2	Save dumped data - address slot 2
46	OSVODDU3	Save dumped data - address slot 3
47	OSVODDU4	Save dumped data - address slot 4
48	OSVODDU5	Save dumped data - address slot 5
49	OSVODDU6	Save dumped data - address slot 6
50	OSASKENT	Real time clock at entry to SMKP
51	OSVSDNSA	SMKP save next segment address, OSsignals, or PIC interrupt value
52	OSVSPIDP	SMKP save program ID and PCB index (or 0); or message peg counts
53	OSVSPPDT	SMKP save priority, event type, and segment time; or message peg counts
54	OSVSCLKB	SMKP save clock at beginning of job; or message peg counts
55	OSVSDDU1	SMKP save dumped data - address slot 1; or message peg counts
56	OSVSDDU2	SMKP save dumped data - address slot 2; or message peg counts
57	OSVSDDU3	SMKP save dumped data - address slot 3; or message peg counts
58	OSVSDDU4	SMKP save dumped data - address slot 4; or message peg counts
59	OSVSDDU5	SMKP save dumped data - address slot 5; or message peg counts
60	OSVSDDU6	SMKP save dumped data - address slot 6; or message peg counts
61	OSAMKENT	Real time clock at entry to MSKP

See note(s) at end of table.

Table 12.7-1 — OSDS Monitor AM Buffer Word Structure (Contd)

Word	Symbol	Description
62	OSVMDNSA	MSKP save next segment address, OSsignals, or PIC interrupt value
63	OSVMPIDP	MSKP save program ID and PCB index (or 0)
64	OSVMPPDT	MSKP save priority, event type, and segment time
65	OSVMCLKB	MSKP save clock at beginning of job
66	OSCBEGCL	Start clock in millisecond ticks for time accumulation locations
67	OSCENDCL	Stop clock in millisecond ticks for time accumulation locations
68	OSHPCHUF	Peg count of *UNIX* RTR entries with work to do
69	OSHACHUF	Accumulated time in *UNIX* RTR entries (outside OKP)
70	OSHICHUF	Peg count of *UNIX* RTR idle system entries to OKP
71	OSHICHOX	Peg count of OKP entries
72	OSHPCHOX	Peg count of OKP entries with work to do
73	OSHACHOX	Accumulated time in OKP entries with work to do
74	OSHACOKP	Accumulated time in OKP entries that used > 2 ms
75	OSHTOHOX	Peg count of OKP entries timed out (used > 100 ms)
76	OSHPCHIJ	Peg count of interject entries
77	OSHACHIJ	Accumulated time in interject
78	OSHPCCNI	Peg count of CNI entries with messages
79	OSHACCNI	Accumulated time in CNI processing
80	OSHPCHSX	Peg count of entries to SMKP
81	OSHACHSX	Accumulated time in SMKP
82	OSHPCHMX	Peg count of entries to MSKP
83	OSHACHMX	Accumulated time in MSKP
84	OSHPCAPD	Peg count of entries to the APDL process
85	OSHACAPD	Accumulated time in the APDL process
86	OSHOKAUD	Accumulated time in OKP audits
87	OSHSKAUD	Accumulated time in SMKP audits
88	OSHPCHPY	Peg count for OKP priority 0
89	OSHPCHPY+1	Accumulated time in priority 0
90	OSHPCHPY+2	Peg count for OKP priority 1
91	OSHPCHPY+3	Accumulated time in priority 1
92	OSHPCHPY+4	Peg count for OKP priority 2
93	OSHPCHPY+5	Accumulated time in priority 2
94	OSHPCHPY+6	Peg count for OKP priority 3
95	OSHPCHPY+7	Accumulated time in priority 3
96	OSHPCHPY+8	Peg count for OKP priority 4
97	OSHPCHPY+9	Accumulated time in priority 4

See note(s) at end of table.

Table 12.7-1 — OSDS Monitor AM Buffer Word Structure (Contd)

Word	Symbol	Description
98	OSHPCHPY+10	Peg count for OKP priority 5
99	OSHPCHPY+11	Accumulated time in priority 5
100	OSHPCHPY+12	Peg count for OKP priority 6
101	OSHPCHPY+13	Accumulated time in priority 6
102	OSHPCHPY+14	Peg count for OKP priority 7
103	OSHPCHPY+15	Accumulated time in priority 7
104	OSHACSPY	Elapsed time for SPY data at inhibit
105	OSHACSPY+1	Accumulated time in kernel level at inhibit
106	OSHACSPY+2	Accumulated time in kernel process level at inhibit
107	OSHACSPY+3	Accumulated time in supervisor processes at inhibit
108	OSHACSPY+4	Accumulated time in user processes at inhibit
109	OSHACSPY+5	Accumulated time in idle loop at inhibit
110	OSHACSPY+6	Accumulated time in *UNIX* RTR level 0 at inhibit
111	OSHACSPY+7	Accumulated time in *UNIX* RTR level 1 at inhibit
112	OSHACSPY+8	Accumulated time in *UNIX* RTR level 2 at inhibit
113	OSHACSPY+9	Accumulated time in *UNIX* RTR level 3 at inhibit
114	OSHACSPY+10	Accumulated time in *UNIX* RTR level 4 at inhibit
115	OSHACSPY+11	Accumulated time in *UNIX* RTR level 5 at inhibit
116	OSHACSPY+12	Accumulated time in *UNIX* RTR level 6 at inhibit
117	OSHACSPY+13	Accumulated time in *UNIX* RTR level 7 at inhibit
118	OSHACSPY+14	Accumulated time in *UNIX* RTR level 8 at inhibit
119	OSHACSPY+15	Accumulated time in *UNIX* RTR level 9 at inhibit
120	OSHACSPY+16	Accumulated time in *UNIX* RTR level 10 at inhibit
121	OSHACSPY+17	Accumulated time in *UNIX* RTR level 11 at inhibit
122	OSHACSPY+18	Accumulated time in *UNIX* RTR level 12 at inhibit
123	OSHACSPY+19	Accumulated time in *UNIX* RTR level 13 at inhibit
124	OSHACSPY+20	Accumulated time in *UNIX* RTR level 14 at inhibit
125	OSHACSPY+21	Accumulated time in *UNIX* RTR level 15 at inhibit
126	OSHPMSGI	Peg count of inter-processor messages received
127	OSHPMSGO	Peg count of inter-processor messages sent
128	OSHPMSGL	Peg count of intra-processor messages sent
129	OSHPTUNX	Peg count of messages sent from the OKP to another *UNIX* RTR process
130	OSHPFUNX	Peg count of messages sent from another *UNIX* RTR process to the OKP
131	OSHSPAR1	Spare word
132	OSHSPAR2	Spare word
133	OSHSPAR5	Spare word

See note(s) at end of table.

Table 12.7-1 — OSDS Monitor AM Buffer Word Structure (Contd)

Word	Symbol	Description
134	OSHSPAR6	Spare word
135	OSHSPAR7	Spare word
136	OSHSPAR8	Spare word
137	OSHSPAR9	Spare word
138	OSHSPARa	Spare word
139	OSHSPARb	Spare word
140	OSHSPARc	Spare word
141	OSHSPARd	Spare word
142	OSHSPARe	Spare word
143	OSHSPARf	Spare word
144	OSHSPARg	Spare word
145	OSHSPARh	Spare word
146	OSHSPARi	Spare word
147	OSHSPARj	Spare word
148	OSHSPARk	Spare word
149	OSHPSCTU	Peg count of messages from SMs or the CMP to *UNIX* RTR processes other than OKP
150	OSHGHAIR	Peg count of hairpin messages
151	OSHGIMCP	Peg count of integrity monitor messages
152	OSHGAMCT	Peg count of OSDS clock accuracy messages
153	OSHGDDCP	Peg count of data delivery messages
154	OSHGCCTP	Peg count of CCS messages
155	OSHGBCST	Peg count of OSDS broadcast messages
156	OSHGAMCP	Peg count of AMA messages
157	OSHGPSCP	Peg count of TM AML messages
158	OSHGFALT	Peg count of messages delivered to processes
159	OSHGCMBP	Peg count of messages sent from AM to world
160	OSHGCNIR	Peg count of messages received from CNI ring
161	OSHGCNIS	Peg count of messages sent to the CNI ring
162	OSHGRTFD	Peg count of status forward messages
163	OSHGRTGN	Peg count of route generation messages
164	OSHGTSRE	Peg count of timeslot release forward messages
165	OSHGPATH	Peg count of path established messages
166	OSHGHORQ	Peg count of handover request messages
167	OSHGCINJ	Peg count of messages passed on to CMinjmsg()
168	OSHPCHPD	Peg count for OKP-OSDS program ID = 0 - flag HPD
169	OSHPCHPD+1	Accumulated time in OKP-OSDS program ID = 0
170	OSHPCHPD+2	Peg count for OKP-OSDS program ID = 1

See note(s) at end of table.

Table 12.7-1 — OSDS Monitor AM Buffer Word Structure (Contd)

Word	Symbol	Description
171	OSHPCHPD+3	Accumulated time in OKP-OSDS program ID = 1
. .	. .	. . . .
. .	. .	. . . .
. .	. .	. . . .
686	OSHPCHPD+518	Peg count for OKP-OSDS program ID = 259
687	OSHPCHPD+519	Accumulated time in OKP-OSDS program ID = 259
688	OSHPCHSP	Peg count for SMKP-OSDS program ID = 0
689	OSHPCHSP+1	Accumulated time in SMKP-OSDS program ID = 0
690	OSHPCHSP+2	Peg count for SMKP-OSDS program ID = 1
691	OSHPCHSP+3	Accumulated time in SMKP-OSDS program ID = 1
. .	. .	. . . .
. .	. .	. . . .
. .	. .	. . . .
946	OSHPCHSP+258	Peg count of SMKP-OSDS program ID = 129
947	OSHPCHSP+259	Accumulated time in SMKP-OSDS program ID = 129
948	OSHSPDAT	Beginning of data storage area for special monitor functions
949	OSCSPC00	Data storage area for special monitor functions - flag S00
950	OSCSPC01	Data storage area for special monitor functions - flag S01
951	OSCSPC02	Data storage area for special monitor functions - flag S02
. .	. .	. . . .
. .	. .	. . . .
. .	. .	. . . .
995	OSCSPC47	Data storage area for special monitor functions - flag S47
996	OSHXXDSP	Data collection area for the process dispatch array
997	OSHXXDSP+1	Data collection area for the process dispatch array
. .	. .	. . . .
. .	. .	. . . .
. .	. .	. . . .
15358	OSHXXDSP+14362	Data collection area for the process dispatch array
15359	OSHXXEND	Last word in the monitor buffer

a. Registered trademark of The Open Group.

12.7.4 Buffer Layout for The SM

The OSDS monitor SM and SM-2000 buffer layout for the 5E10 software release is depicted in Figure 12.7-2. The OSDS monitor SM and SM-2000 buffer is contained in the `SIhistory[]` array.

The information included in this section is for reference only. Each software release contains unique buffer layout information. Consult the `hdr/OS/OSmons.h` file for specific details.

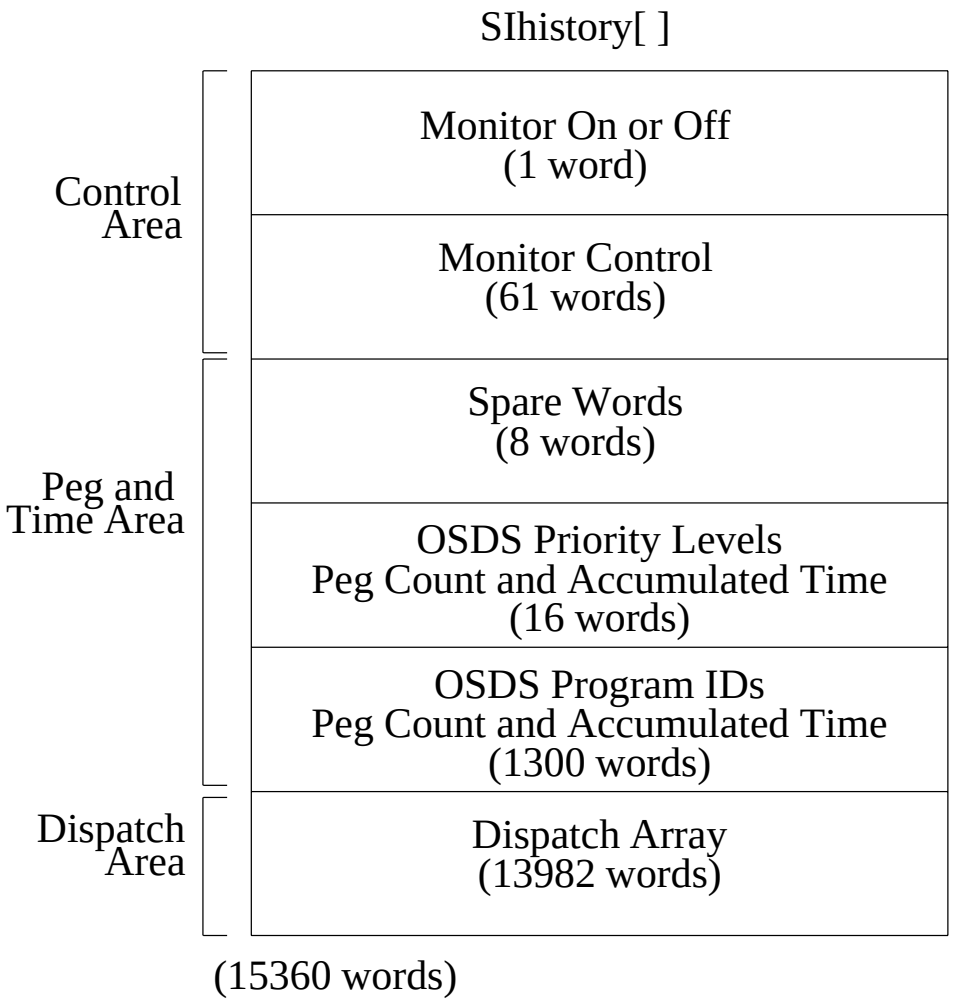


Figure 12.7-2 — OSDS Monitor SM Buffer Layout

12.7.5 SM Buffer Word Content

The SM buffer word content is described in Table 12.7-2.

Table 12.7-2 — OSDS Monitor SM Buffer Word Structure

Word	Symbol	Description
0	OSCONTRL	Master control word for monitor routines: 0 = off, !0 = on
1	OSCINDEX	Byte index into dispatch array
2	OSCHDFLG	What to do flags
3	OSCDDATA	What to dump flags
4	OSCDUMP1	Memory address to dump or 0 - flag AD1
5	OSCDUMP2	Memory address to dump or 0 - flag AD2
6	OSCDUMP3	Memory address to dump or 0 - flag AD3
7	OSCDUMP4	Memory address to dump or 0 - flag AD4
8	OSCDUMP5	Memory address to dump or 0 - flag AD5
9	OSCDUMP6	Memory address to dump or 0 - flag AD6
10	OSCPCBL1	PCBLA index to dump or 0 - flag LA1
11	OSCPCBL2	PCBLA index to dump or 0 - flag LA2
12	OSCPCBL3	PCBLA index to dump or 0 - flag LA3
13	OSCPROGI	OSDS program ID(s) to snap data for on dispatch
14	OSCPRIOR	OSDS priority level to snap data for on dispatch
15	OSCPTIME	Process dispatch time filter value for loading the dispatch array PTM. Dispatch times less than this value will not be snapped.
16	OSCPORTF	Port number for filtering - flag PRT
17	OSCSMDMV	Data match snap data word
18	OSCSMDMM	Data match snap mask word, 1 in bits to be matched
19	OSCSMFRA	Previous function return address
20	OSCFWORD	Special function control word - flag CTL
21	OSCFFLAG	Special function flag bits
22	OSCSPAR1	Spare control word
23	OSCSPAR2	Spare control word
24	OSCSPAR3	Spare control word
25	OSCSPAR4	Spare control word
26	OSCSPAR5	Spare control word
27	OSCSPAR6	Spare control word
28	OSCSPAR7	Spare control word
29	OSCSPAR8	Spare control word
30	OSCSPAR9	Spare control word
31	OSCSPARa	Spare control word
32	OSCSPARb	Spare control word
33	OSCSPARc	Spare control word
34	OSCSPARd	Spare control word
35	OSSVPICV	Foreground PIC interrupt data value

Table 12.7-2 — OSDS Monitor SM Buffer Word Structure (Contd)

Word	Symbol	Description
36	OSSVCLKF	Clock at foreground entry
37	OSSBGACT	Flag for background loading of dispatch array
38	OSSVDNSA	Save next segment address, OSsignals, or PIC interrupt value
39	OSSVPIDP	Save program ID and PCB index (or 0)
40	OSSVPPDT	Save priority, event type, and segment time
41	OSSVCLKB	Save clock at beginning of job
42	OSK2FGST	Save start of foreground using 15.3615 µsec clock (SM-2000)
43	OSK2CLKB	Save segment begin time using 15.3615 µsec clock (SM-2000)
44	OSSVFLAG	Save function flag bits (OSCFFLAG) until time to start monitor
45	OSSVWORD	Save function control word (OSCFWORD) until time to start monitor
46	OSSVSDAY	Record the day the monitor started
47	OSSVSTOD	Record the time of day the monitor started
48	OSSVSEGL	Save the process segment length for monitor special functions
49	OSSVDDU1	Save dumped data - address slot 1
50	OSSVDDU2	Save dumped data - address slot 2
51	OSSVDDU3	Save dumped data - address slot 3
52	OSSVDDU4	Save dumped data - address slot 4
53	OSSVDDU5	Save dumped data - address slot 5
54	OSSVDDU6	Save dumped data - address slot 6
55	OSSVPCL1	Save PCBLA dumped data - address slot 1
56	OSSVPCL2	Save PCBLA dumped data - address slot 2
57	OSSVPCL3	Save PCBLA dumped data - address slot 3
58	OSCBEGCL	Start clock in 125 µsec ticks for time accumulation locations
59	OSCENDCL	End clock in 125 µsec ticks for time accumulation locations
60	OSHPCHUF	Peg count of foreground entries
61	OSHACHUF	Accumulated time in foreground
62	OSHPCHIJ	Peg count of interject entries
63	OSHICHIJ	Peg count of interject entries with no flags
64	OSHACHIJ	Accumulated time in interject processing
65	OSHPMSGI	Peg count of inter-processor messages received by the processor

Table 12.7-2 — OSDS Monitor SM Buffer Word Structure (Contd)

Word	Symbol	Description
66	OSHPMSGO	Peg count of interprocessor messages sent by the processor
67	OSHPMSGL	Peg count of intraprocessor messages sent by the processor
68	OSHACHMH	Spare word
69	OSHSPAR1	Spare word
70	OSHSPAR2	Spare word
71	OSHSPAR3	Spare word
72	OSHSPAR4	Spare word
73	OSHSPAR5	Spare word
74	OSHSPAR6	Spare word
75	OSHSPAR7	Spare word
76	OSHSPAR8	Spare word
77	OSHSPAR9	Spare word
78	OSHSPARa	Spare word
79	OSHSPARb	Spare word
80	OSHSPARc	Spare word
81	OSHSPARd	Spare word
82	OSHSPARe	Spare word
83	OSHSPARf	Spare word
84	OSHSPARg	Spare word
85	OSHSPARh	Spare word
86	OSHSPARi	Spare word
87	OSHSPARj	Spare word
88	OSHSPARk	Spare word
89	OSHPCHPY	Peg count for OSDS priority 0
90	OSHPCHPY+1	Accumulated time in priority 0
91	OSHPCHPY+2	Peg count for OSDS priority 1
92	OSHPCHPY+3	Accumulated time in priority 1
93	OSHPCHPY+4	Peg count for OSDS priority 2
94	OSHPCHPY+5	Accumulated time in priority 2
95	OSHPCHPY+6	Peg count for OSDS priority 3
96	OSHPCHPY+7	Accumulated time in priority 3
97	OSHPCHPY+8	Peg count for OSDS priority 4
98	OSHPCHPY+9	Accumulated time in priority 4
99	OSHPCHPY+10	Peg count for OSDS priority 5
100	OSHPCHPY+11	Accumulated time in priority 5
101	OSHPCHPY+12	Peg count for OSDS priority 6

Table 12.7-2 — OSDS Monitor SM Buffer Word Structure (Contd)

Word	Symbol	Description
102	OSHPCHPY+13	Accumulated time in priority 6
103	OSHPCHPY+14	Peg count for OSDS priority 7
104	OSHPCHPY+15	Accumulated time in priority 7
105	OSHPCHPD	Peg count for program ID 0
106	OSHPCHPD+1	Accumulated time in program ID 0
107	OSHPCHPD+2	Peg count for program ID 1
108	OSHPCHPD+3	Accumulated time in program ID 1
. .	. .	. . . .
. .	. .	. . . .
. .	. .	. . . .
1403	OSHPCHPD+1298	Peg count for program ID 649
1404	OSHPCHPD+1299	Accumulated time in program ID 649
1405	OSHSPDAT	Beginning of data storage area for special monitor functions
1406	OSCSPC00	Special control word - flag S00
1407	OSCSPC01	Special control word - flag S01
. .	. .	. . . .
. .	. .	. . . .
. .	. .	. . . .
1452	OSCSPC47	Special control word - flag S47
1453	OSHXXDSP	Data collection area for the process dispatch array
1454	OSHXXDSP+1	Data collection area for the process dispatch array
. .	. .	. . . .
. .	. .	. . . .
. .	. .	. . . .
15358	OSHXXDSP+13905	Data collection area for the process dispatch array (SM)
15358	OSHXXDSP+75345	Data collection area for the process dispatch array (SM-2000)
15359	OSHXXEND	Last word in the monitor buffer

12.7.6 Buffer Layout for The CMP

The OSDS monitor CMP buffer layout for the 5E10 software release is depicted in Figure 12.7-3. The OSDS monitor CMP buffer is contained in the SIhistory[] array. Each software release contains unique buffer layout information. Consult the hdr/OS/OSmons.h file for specific details.

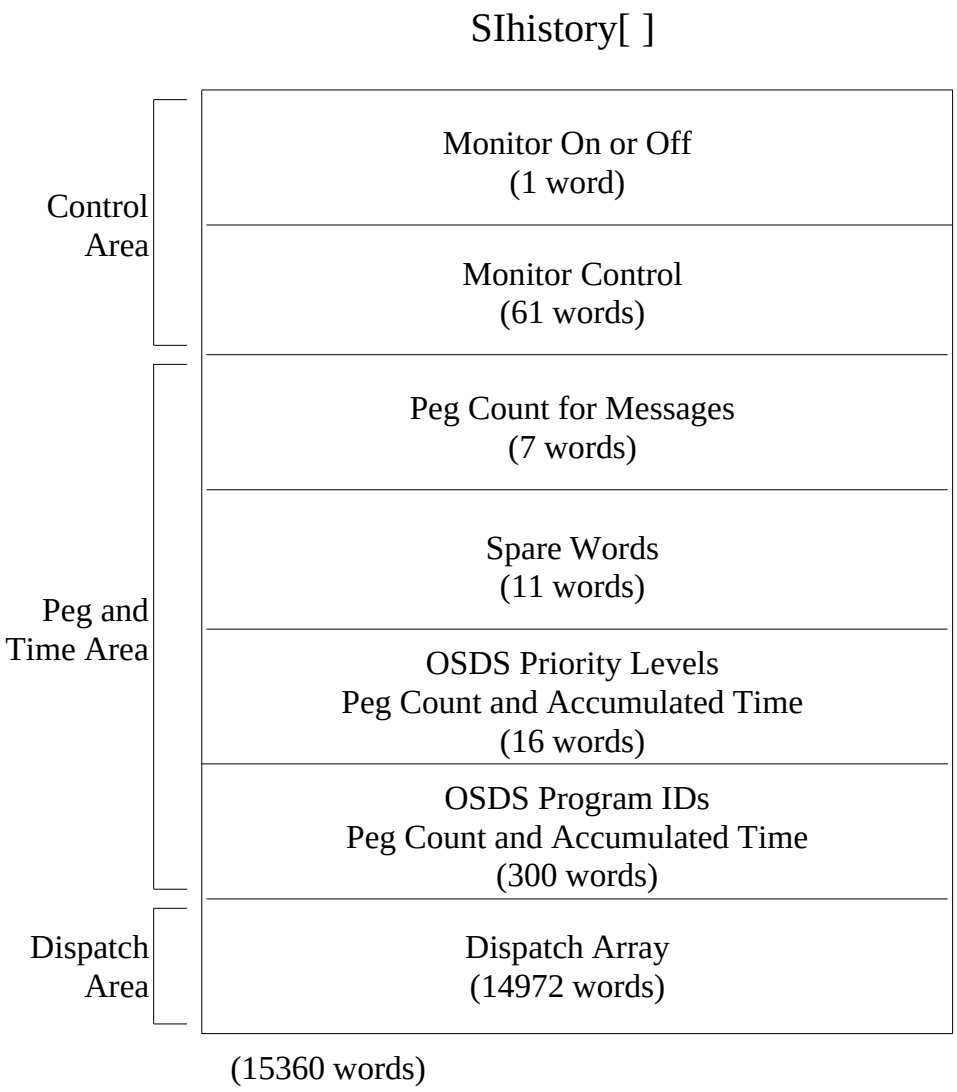


Figure 12.7-3 — OSDS Monitor CMP Buffer Layout

12.7.7 CMP Buffer Word Content

The CMP buffer word content is described in Table 12.7-3.

Table 12.7-3 — OSDS Monitor CMP Buffer Word Structure

Word	Symbol	Description
0	OSCONTRL	Master control word for monitor routines: 0 = off, !0 = on
1	OSCINDEX	Byte index into dispatch array
2	OSCHDFLG	What to do flags
3	OSCDDATA	What to dump flags
4	OSCDUMP1	Memory address to dump or 0 - flag AD1
5	OSCDUMP2	Memory address to dump or 0 - flag AD2
6	OSCDUMP3	Memory address to dump or 0 - flag AD3
7	OSCDUMP4	Memory address to dump or 0 - flag AD4
8	OSCDUMP5	Memory address to dump or 0 - flag AD5
9	OSCDUMP6	Memory address to dump or 0 - flag AD6
10	OSCPCBL1	PCBLA index to dump or 0 - flag LA1
11	OSCPCBL2	PCBLA index to dump or 0 - flag LA2
12	OSCPCBL3	PCBLA index to dump or 0 - flag LA3
13	OSCPROGI	OSDS program ID(s) to snap data for on dispatch
14	OSCPRIOR	OSDS priority level to snap data for on dispatch
15	OSCPTIME	Process dispatch time filter value for loading the dispatch array - flagPTM. Dispatch times less than this value will not be snapped.
16	OSCPORTF	Port number for filtering - flag PRT
17	OSCSMDMV	Data match snap data word
18	OSCSMDMM	Data match snap mask word, 1 in each bit to be matched
19	OSCSMFRA	Previous function return address
20	OSCFWORD	Special function control word - flag CTL
21	OSCFFLAG	Special function flag bits
22	OSCSPAR1	Spare control word
23	OSCSPAR2	Spare control word
24	OSCSPAR3	Spare control word
25	OSCSPAR4	Spare control word
26	OSCSPAR5	Spare control word
27	OSCSPAR6	Spare control word
28	OSCSPAR7	Spare control word
29	OSCSPAR8	Spare control word
30	OSCSPAR9	Spare control word
31	OSCSPARa	Spare control word
32	OSCSPARb	Spare control word
33	OSCSPARc	Spare control word
34	OSCSPARd	Spare control word
35	OSSVPICV	Foreground exception vector number

Table 12.7-3 — OSDS Monitor CMP Buffer Word Structure (Contd)

Word	Symbol	Description
36	OSSVCLKF	Clock at foreground entry
37	OSSBGACT	Flag for background loading of dispatch array
38	OSSVDNSA	Save next segment address, OSsignals, or PIC interrupt value
39	OSSVPIDP	Save program ID and PCB index (or 0)
40	OSSVPPDT	Save priority, event type, and segment time
41	OSSVCLKB	Save clock at beginning of job
42	OSK2FGST	Save start of foreground using 15.3615 µsec clock (SM-2000)
43	OSK2CLKB	Save segment begin time using 15.3615 µsec clock (SM-2000)
44	OSSVFLAG	Save function flag bits word (OSCFFLAG) until time to start monitor
45	OSSVWORD	Save function control word (OSCFWORD) until time to start monitor
46	OSSVSDAY	Record the day the monitor started
47	OSSVSTOD	Record the time of day the monitor started
48	OSSVSEGL	Save process segment length for monitor special functions
49	OSSVDDU1	Save dumped data - address slot 1
50	OSSVDDU2	Save dumped data - address slot 2
51	OSSVDDU3	Save dumped data - address slot 3
52	OSSVDDU4	Save dumped data - address slot 4
53	OSSVDDU5	Save dumped data - address slot 5
54	OSSVDDU6	Save dumped data - address slot 6
55	OSSVPCL1	Save PCBLA dumped data - address slot 1
56	OSSVPCL2	Save PCBLA dumped data - address slot 2
57	OSSVPCL3	Save PCBLA dumped data - address slot 3
58	OSCBEGCL	Start clock in 125 µsec ticks for time accumulation locations
59	OSCENDCL	End clock in 125 µsec ticks for time accumulation locations
60	OSHPCHUF	Peg count of foreground entries
61	OSHACHUF	Accumulated time in foreground
62	OSHPCHIJ	Peg count of interject entries
63	OSHICHIJ	Peg count of interject entries with no flags
64	OSHACHIJ	Accumulated time in interject processing
65	OSHPMSGI	Peg count of interprocessor messages received by the processor

Table 12.7-3 — OSDS Monitor CMP Buffer Word Structure (Contd)

Word	Symbol	Description
66	OSHPMSGO	Peg count of interprocessor messages sent by the processor
67	OSHPMSGL	Peg count of intraprocessor messages sent by the processor
68	OSHGBCST	Peg count of OSDS broadcast messages
69	OSHGCPRT	Peg count of routing (RTA) messages
70	OSHGDDSP	Peg count of data delivery messages
71	OSHGICMP	Peg count of integrity monitor messages
72	OSHGPCNC	Peg count of peripheral control messages
73	OSHGPRGP	Peg count of packet routing messages
74	OSHGFALT	Peg count of messages delivered to processes
75	OSHGCICO	Peg count of interprocessor data sync (IDS) messages
76	OSHGSPR1	Spare word
77	OSHGSPR2	Spare word
78	OSHACHMH	Spare word
79	OSHSPAR1	Spare word
80	OSHSPAR2	Spare word
81	OSHSPAR3	Spare word
82	OSHSPAR4	Spare word
83	OSHSPAR5	Spare word
84	OSHSPAR6	Spare word
85	OSHSPAR7	Spare word
86	OSHSPAR8	Spare word
87	OSHSPAR9	Spare word
88	OSHSPARa	Spare word
89	OSHSPARb	Spare word
90	OSHSPARc	Spare word
91	OSHSPARd	Spare word
92	OSHSPARe	Spare word
93	OSHSPARf	Spare word
94	OSHSPARg	Spare word
95	OSHSPARh	Spare word
96	OSHSPARi	Spare word
97	OSHSPARj	Spare word
98	OSHSPARk	Spare word
99	OSHPCHPY	Peg count for OSDS priority 0
100	OSHPCHPY+1	Accumulated time in priority 0
101	OSHPCHPY+2	Peg count for OSDS priority 1

Table 12.7-3 — OSDS Monitor CMP Buffer Word Structure (Contd)

Word	Symbol	Description
102	OSHPCHPY+3	Accumulated time in priority 1
103	OSHPCHPY+4	Peg count for OSDS priority 2
104	OSHPCHPY+5	Accumulated time in priority 2
105	OSHPCHPY+6	Peg count for OSDS priority 3
106	OSHPCHPY+7	Accumulated time in priority 3
107	OSHPCHPY+8	Peg count for OSDS priority 4
108	OSHPCHPY+9	A108cumulated time in priority 4
109	OSHPCHPY+10	Peg count for OSDS priority 5
110	OSHPCHPY+11	Accumulated time in priority 5
111	OSHPCHPY+12	Peg count for OSDS priority 6
112	OSHPCHPY+13	Accumulated time in priority 6
113	OSHPCHPY+14	Peg count for OSDS priority 7
114	OSHPCHPY+15	Accumulated time in priority 7
115	OSHPCHPD	Peg count for program ID 0
116	OSHPCHPD+1	Accumulated time in program ID 0
117	OSHPCHPD+2	Peg count for program ID 1
118	OSHPCHPD+3	Accumulated time in program ID 1
. .	. .	. . . .
. .	. .	. . . .
443	OSHPCHPD+298	Peg count for program ID 149
444	OSHPCHPD+299	Accumulated time in program ID 149
445	OSHSPDAT	Beginning of data storage area for special monitor functions
446	OSCSPC00	Special control word - flag S00
447	OSCSPC01	Special control word - flag S01
. .	. .	. . . .
. .	. .	. . . .
492	OSCSPC47	Special control word - flag S47
493	OSHXXDSP	Data collection area for the process dispatch array
494	OSHXXDSP+1	Data collection area for the process dispatch array
. .	. .	. . . .
. .	. .	. . . .
15358	OSHXXDSP+14865	Data collection area for the process dispatch array
15359	OSHXXEND	Last word in the monitor buffer

12.8 SNAPPED DATA DUMP LAYOUTS

12.8.1 Section Description

This section provides the data dump layouts for the following input message control flags:

- DAD
- DAP
- F00
- F01
- F18
- F22
- F23, F25, F27, F29, F31
- HUF, HIJ, HPD, HPY, HMX, HSX, HSP
- SEG

Note: For the DIJ, DMP, DMX, DPD, DPY, DUF, DSP, and DSX flags, see the data dump descriptions in the "What To Dump Flags," Section 12.4.9.

12.8.2 DAD Data Dump Layout

The DAD flag causes data blocks specified by hooks in the switch application code along with some header information to be copied to the dispatch array. The DAD flag supports the DOX flag option.

If there is not enough room in the dispatch array and the DOX flag is set, then as much data as possible is copied into the array without wrapping around. If the DOX flag is not set, then the data wraps around overwriting any data previously stored at the beginning of the dispatch array. Wrap around will not be allowed in the header for the data dump, but is legal at any other point in the data dump.

The DAD data dump has the following format:

```
72727272 eeeegggg dddddddd ttttssss
aaaaaaaa xxxxxxxx xxxxxxxx xxxxxxxx
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
   ...      ...      ...      ...
```

Where:

```
72727272 - Tag to identify the start of a dump
    eeee - Event code
    gggg - Group code
dddddddd - Number of days since January 1, 1970
    tttt - Number of milliseconds since midnight
    ssss - Number of bytes dumped
aaaaaaaa - Starting address of the dump
xxxxxxxx - Actual data dumped
```

Note: All data is shown in hex.

12.8.3 DAP Data Dump Layout

The DAP flag causes the used portion of the current stack, the processor registers, and some header information to be copied to the dispatch array when hooks in switch application code indicate that a dump should be made. The DAP flag supports the DOX flag option.

If there is not enough room in the dispatch array and the DOX flag is set, then as much data as possible is copied into the array without wrapping around. If the DOX flag is not set, then the data wraps around overwriting any data previously stored at the beginning of the dispatch array. Wrap around will not be allowed in the header for the data dump or the register dump, but is legal at any other point in the data dump. The DAP data dump has the following format:

```
74747474 eeeegggg dddddddd ttttssss
aaaaaaaa bbbbbbbb cccccccc hhhhhhhh
iiiiiiii jjjjjjjj kkkkkkkk llllllll
mmmmmmmm nnnnnnnn oooooooo pppppppp
qqqqqqqq rrrrrrrr uuuuuuuu vvvvvvvv
wwwwwwww xxxxxxxx xxxxxxxx xxxxxxxx
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
   ...      ...      ...      ...
```

Where:

74747474 - Tag to identify the start of a dump
eeee - Event code
gggg - Group code
dddddddd - Number of days since January 1, 1970
tttt - Number of milliseconds since midnight
ssss - Number of bytes dumped
aaaaaaaa - Starting address of the dump
bbbbbbbb - R0 (AM), A0 (SM/CMP)
cccccccc - R1 (AM), A1 (SM/CMP)
hhhhhhhh - R2 (AM), A2 (SM/CMP)
iiiiiiii - R3 (AM), A3 (SM/CMP)
jjjjjjjj - R4 (AM), A4 (SM//CMP)
kkkkkkkk - R5 (AM), A5 (SM/CMP)
llllllll - R6 (AM), A6 (also known as FP, SM/CMP)
mmmmmmmm - R7 (AM), A7 (also known as SP, SM/CMP)
nnnnnnnn - R8 (AM), D0 (SM/CMP)
oooooooo - AP (AM), D1 (SM/CMP)
pppppppp - FP (AM), D2 (SM/CMP)
qqqqqqqq - SP (AM), D3 (SM/CMP)
rrrrrrrr - N/A (AM), D4 (SM/CMP)
uuuuuuuu - N/A (AM), D5 (SM/CMP)
vvvvvvvv - N/A (AM), D6 (SM/CMP)
wwwwwwww - N/A (AM), D7 (SM/CMP)
xxxxxxxx - Actual stack data dumped

Note: All data is shown in hex.

12.8.4 F00 Data Dump Layout

The F00 flag allows the user to dump memory or OSDS resource control block data to the dispatch array.

For dump option 1 (raw memory data dump), the following format is used:

```
52525252 00000001 aaaaaaaa bbbbbbbb
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
   ...      ...      ...      ...
```

Where:

52525252 - Tag to identify the start of a dump
00000001 - Code indicating the dump option (1)
aaaaaaaa - Starting address of the memory dump
bbbbbbbb - Number of bytes dumped. The actual number of bytes dumped may be less than the requested length if the dump was truncated due to the DOX flag.
xxxxxxxx - Actual data dumped

For dump options 2, 3, or 5 (PCB, SCB, or TCB data dump), the following format is used:

```
52525252 0000000a 00000000 bbbbbbbb
ccccdddd eeeeffgg ccccdddd eeeeffgg
   ...      ...      ...      ...
```

Where:

52525252 - Tag to identify the start of a dump
0000000a - Code indicating the dump option (a = 2, 3, or 5)
bbbbbbbb - Number of PCBs, SCBs, or TCBs that were dumped
cccc - PCB, SCB, or TCB index
dddd - Index of the PCB that owned the PCB, SCB, or TCB
eeee - Program ID associated with the owning PCB
ff - State of the owning PCB
gg - Uniqueness of the owning PCB

For dump option 4 (MCB data dump), the following format is used:

```
52525252 00000004 00000000 bbbbbbbb
ccccdddd eeeeffgg hhhhhhhh iiiijjkk
   ...      ...      ...      ...
```

Where:

52525252 - Tag to identify the start of a dump
00000004 - Code indicating the dump option (4)
bbbbbbbb - Number of MCBs that were dumped
cccc - MCB index
dddd - Index of the PCB that owned the MCB
eeee - Program ID associated with the owning PCB
ff - State of the owning PCB
gg - Uniqueness of the owning PCB

hhhhhhhh iiiijjkk is the message header associated with the MCB, where:

hhhhhhhh - Process ID of the sending process (the "from" PID)
iiii - Message type
jj - Message priority
kk - Message length

Note: All data is shown in hex.

12.8.5 F01 Data Dump Layout

The F01 flag dumps the following information:

- A record of the maximum number of MCBs used
- A histogram of the number of MCBs in use by each process
- Per-event dumps for MCB acquisition, release, and process dispatches when MCBs are in use

The record of the maximum number of the MCBs used is stored as a short (2 bytes) at the start of the dispatch array.

The histogram of the number of MCBs in use by process immediately follows this short at an offset of 0x2 from the start of the dispatch array. The histogram is stored as an array of shorts and is indexed by the PCB number such that process 0's data is stored at offset 0x2 and process *i*'s data is stored at offset $(i + 1) \times 2$ from the start of the dispatch array.

Immediately following the histogram, the per-event data dumps begin. The exact offset depends on the number of PCBs allocated on the processor (OKP: 145, CMP: 150, loaded SM: 1050, standard or basic SM: 650). The per-event data starts at offset $(\#\ of\ PCBs + 1) \times 2$. There are three types of per-event data dumps:

- MCB acquisition
- Process dispatch while an MCB is in use
- MCB release

The format of the MCB acquisition dump is as follows:

```
AAAAaaaa bbbbcccc ddddeeee
```

Where:

AAAA - Tag to identify the start of the dump
aaaa -- Acquiring process's PCB number
bbbb - Message type
cccc - From processor number
dddd - From process's PCB number
eeee - Number of MCBs held by the acquiring process

The data dump format for process dispatch while an MCB is in use is as follows:

```
DDDDaaaa bbbbcccc ccccdddd dddd....
```

Where:

DDDD - Tag to identify the start of the dump
aaaa - PCB number of the process being dispatched
bbbb - Number of idle MCBs
cccccccc - Timestamp (in milliseconds)
dddddddd - Process's segment length in milliseconds
.... - Start of the next dump (not part of this dump)

The format of the MCB release dump is as follows:

```
FEFEaaaa
```

Where:

FEFE - Tag to identify the start of the dump
aaaa - Releasing process's PCB number

Note: All data is shown in hex.

12.8.6 F18 Data Dump Layout

The F18 flag causes a function back trace to be dumped to the dispatch array whenever OSREPLACE() or OSRESTART() is called.

The F18 dump has the following format:

```
52525252 aaaaaaaa bbbbbbbb ccccdddd
eeeeeeee ffffffff gggggggg hhhhhhhh
   ...                   iiiiiiii
            ...
```

Where:

52525252	-	Flag indicating the start of a data dump
aaaaaaaa	-	OSPID of the calling process
bbbbbbbb	-	Timestamp of the call to OSREPLACE() or OSRESTART() (10 ms granularity)
cccc	-	Old program ID
dddd	-	New program ID
eeeeeeee	-	Number of addresses that follow
ffffffff	-	OS primitive's return address
gggggggg	-	Return address of the OS primitive's caller
hhhhhhhh	-	Return address of the caller of the OS primitive's caller
...	-	Remaining return addresses from the stack
iiiiiiii	-	Address of OSSUICIDE() (the last function on the user stack)

12.8.7 F22 Data Dump Layout

The F22 flag dumps SDL trace information. The F22 dump has the following format:

aaaapppp bbbbcccc ttttiiii mmmmssss

Where:

aaaa - Code to identify SDL (7777) or SDLJ (6666)
pppp - Port associated with the process
bbbb - Program ID
cccc - Process ID
tttt - mdb_tag field
iiii - SDL input message type
mmmm - SDL model_id
ssss - SDL state

Note: All data is shown in hex.

12.8.8 F23, F25, F27, F29, F31 Data Dump Layout

The F25, F27, F29, and F31 flags all cause per-event message data to be dumped. All of these flags use the same format with slightly different meanings for some of the fields between the SM/CMP and the AM.

The F23 flag causes a timestamp to be appended to the data dumped by the F25, F27, F29, and F31 flags. The F23 flag causes a long of the following format to be appended to all per-event message data dumps:

aammmmmm

Where:

aa - Number of days since the monitor was allowed

mmmmmm - Number of milliseconds since midnight

The F25, F27, F29, and F31 data dumps all have the following format. There are different meanings for some fields in the different processors.

bbbbcccc ddddeeff gggghhii jjjjkkll

Where:

bbbb	-	Code to identify the dump source 3333 = F25 flag 2222 = F27 flag 5555 = F29 flag 888x = F31 flag Where x is the logical link the message is sent over
cccc	-	Program ID of the receiving process (F27 or F29) or program ID of the sending process (F25 or F31)
dddd	-	PCB number of the receiving process
ee	-	State of the destination process for F29; otherwise, the destination processor ID
ff	-	Uniqueness value in the PCB of the receiving process

gggghhii jjjjkkll is the OSDS message header, where:

gggg	-	PCB number of the sending process (F27, F29, or F31), or the pseudo PID of the sending process (F25, from a *UNIX*[a] RTR process)
hh	-	Processor ID that the sending process is on
ii	-	Uniqueness value of the sending process
jjjj	-	Logical link (bit 15), end-to-end bit (bit 14), OSDS message type (bits13 - 0)
kk	-	Large message indicator (bit 7), software release byte ID indicator (bit 6), compressed message indicator (bit 5), OSDS message priority (bits 2 - 0)
ll	-	Message length

a. Registered trademark of The Open Group.

12.8.9 HUF, HIJ, HPD, HPY, HMX, HSX, HSP Data Dump Layout

The HUF, HIJ, HPD, HPY, HMX, HSX, and HSP flags provide data concerning the number of times certain OSDS jobs are performed and the amount of time spent in the various jobs. The output for these flags is contained in reserved locations in the monitor buffer; that is, the output is not stored in the dispatch array.

Table 12.8-1 and Table 12.8-2 show the locations that the data is recorded in for the SM/CMP and the AM. The "active value" for the locations is the value while the monitor is allowed and collecting data. The "final value" is the value after the monitor has been inhibited. When the monitor is inhibited, various data is copied from scratch locations into its final location.

See "OSDS Monitor Buffer Layouts," Section 12.7, for the actual offsets of the symbols given in Tables 12.8-1 and 12.8-2.

Table 12.8-1 — SM/CMP OSDS Usage Data Layout

Flag	Location	Value
[ab]	OSSVPPDT	Count of originated half calls
[ab]	OSSVCLKB	Count of incoming half calls
[ab]	OSSVDDU1	Count of outgoing half calls
[ab]	OSSVDDU2	Count of terminated half calls
[ab]	OSSVDDU3	Number of POTS terminal processes created
HUF	OSHPCHUF	Count of foreground entries
HUF	OSHACHUF	Accumulated time in foreground
HIJ	OSHPCHIJ	Count of interject entries
HIJ	OSHICHIJ	Count of interject entries when OSsignals = 0
HIJ	OSHACHIJ	Accumulated time in interject when work was done
[a]	OSHPMSGI	Count of interprocessor messages received
[a]	OSHPMSGO	Count of interprocessor messages sent
[a]	OSHPMSGL	Count of local messages delivered
[c]	OSHGPCNC	Accumulator for the number of messages sent to the PCNC process
[c]	OSHGCPRT	Accumulator for the number of messages sent to the CPRT process
[c]	OSHGPRGP	Accumulator for the number of messages sent to the PRGP process
[c]	OSHGICMP	Accumulator for the number of messages sent to the ICMP process
[c]	OSHGDDSP	Accumulator for the number of messages sent to the DDSP process
[c]	OSHGBCST	Accumulator for the number of messages sent to the BCST process
[c]	OSHGFALT	Accumulator for the number of messages handled by processes
HPY[d]	OSHPCHPY	Count of entries to and accumulated time in each OSDS priority level
HPD[d]	OSHPCHPD	Count of entries to and accumulated time in each program ID

Note(s):
a. A count is generated whenever the monitor is turned on.

Table 12.8-1 — SM/CMP OSDS Usage Data Layout (Contd)

Note(s): (Contd)
b. SM only. The count is only valid after the monitor is inhibited (not while the monitor is running).
c. CMP only. Pegged when the monitor is allowed and when it is inhibited,
d. Data is an array starting at the given location. Each element in the array is 2 longs; the first long is a count of entries and the second long is an accumulated time. For the HPY flag, the array is indexed by OSDS priority level. For the HPD flag, the array is indexed by program ID.

Note: All accumulated times are in units of 125 µsec.

Table 12.8-2 — AM OSDS Usage Data Layout

Flag	Location	Value
ab	OSVSDNSA	Count of messages for the IMCP process in OKP
ab	OSVSPIDP	Count of messages for *UNIX* RTR processes
ab	OSVSPPDT	Count of hairpin messages
ab	OSVSDDU1	Count of messages for the AMCT process in OKP
ab	OSVSDDU2	Count of messages for the DDCP process in OKP
ab	OSVSDDU3	Count of messages for the CCTP process in OKP
ab	OSVSDDU4	Count of messages for the BCST process in OKP
ab	OSVSDDU5	Count of messages for the AMCP process in OKP
ab	OSVSDDU6	Count of messages handled by processes
ab	OSVMDNSA	Count of messages handled by PIC 0
ab	OSVMPIDP	Count of messages handled by PIC 1
ab	OSVMPPDT	Count of messages received by CNI
ab	OSVMCLKB	Count of messages sent by CNI
HUF	OSHPCHUF	Count of *UNIX* RTR entries when work was performed
HUF	OSHACHUF	Accumulated time in *UNIX* RTR when work was performed
HUF	OSHICHUF	Count of idle *UNIX* RTR entries
HUF	OSHICHOX	Count of OKP entries
HUF	OSHPCHOX	Count of OKP entries when work was performed
HUF	OSHACHOX	Accumulated time in OKP when work was performed
HUF	OSHACOKP	Accumulated time in OKP entries that were longer than 2 ms
HUF	OSHTOHOX	Count of OKP entries that were longer than 100 ms
HIJ	OSHPCHIJ	Count of CM interject entries
HIJ	OSHACHIJ	Accumulated time in CM interject

See note(s) at end of table.

Table 12.8-2 — AM OSDS Usage Data Layout (Contd)

Flag	Location	Value
HIJ	OSHPCCNI	Count of CNI interject entries
HIJ	OSHACCNI	Accumulated time in CNI interject
HSX	OSHPCHSX	Count of SMKP entries
HSX	OSHACHSX	Accumulated time in SMKP
HMX	OSHPCHMX	Count of MSKP entries
HMX	OSHACHMX	Accumulated time in MSKP
HPY[c]	OSHPCHPY	Count of entries to and accumulated time in each OSDS priority level
[a]	OSHACSPY	*UNIX* RTR SPY times (see the timesnap structure in hdr/dmert/spy.h)
[a]	OSHGHAIR	Accumulator for number of hairpin messages
[d]	OSHGIMCP	Accumulator for the number of messages sent to the IMCP process
[d]	OSHGAMCT	Accumulator for the number of messages sent to the AMCT process
[d]	OSHGDDCP	Accumulator for the number of messages sent to the DDCP process
[d]	OSHGCCTP	Accumulator for the number of messages sent to the CCTP process
[d]	OSHGBCST	Accumulator for the number of messages sent to the BCST process
[d]	OSHGAMCP	Accumulator for the number of messages sent to the AMCP process
[d]	OSHGPSCP	Accumulator for the number of messages sent to the PSCP process
[d]	OSHGFALT	Accumulator for the number of messages handled by processes
[a]	OSHGCMBP	Count of messages sent via the base priority queue (AM to world)
[d]	OSHGCNIR	Accumulator for the number of messages received from the CNI ring
[d]	OSHGCNIS	Accumulator for the number of messages sent to the CNI ring
HPD[c]	OSHPCHPD	Count of entries to and accumulated time in each program ID for OKP
HSP[c]	OSHPCHSP	Count of entries to and accumulated time in each program ID for SMKP
[a]	OSHPMSGI	Count of interprocessor messages received
[a]	OSHPMSGO	Count of interprocessor messages sent
[a]	OSHPMSGL	Count of local messages delivered

See note(s) at end of table.

Table 12.8-2 — AM OSDS Usage Data Layout (Contd)

Flag	Location	Value
[a]	OSHPTUNX	Count of messages sent from OKP to another *UNIX* RTR process
[a]	OSHPFUNX	Count of messages sent from another *UNIX* RTR process to OKP
[a]	OSHPSCTU	Count of messages from SMs or the CMP to *UNIX* RTR processes other than OKP

Note(s):
a. A count is generated whenever the monitor is turned on.
b. The value is only valid after the monitor is inhibited (not while the monitor is running).
c. Data is an array starting at the given location. Each element in the array is 2 longs; the first long is a count of entries and the second long is an accumulated time. For the HPY flag, the array is indexed by OSDS priority level. For the HPD and HSP flags, the array is indexed by program ID.
d. Pegged when the monitor is allowed and when it is inhibited.

Note: All accumulated times are in units of 125 µsec.

12.8.10 SEG Data Dump Layout

The SEG flag causes data concerning the number of consecutive real time segments that a program runs to be dumped.

The SEG data dump has the following format:

```
aaaaaaaa bbbbcccc dddddddd eeeeeeee
```

Where:

aaaaaaaa	-	Timestamp (125 µsec units for the SM/CMP, 1 millisecond for OKP)
bbbb	-	Program ID of the process
cccc	-	Number of consecutive segments
dddddddd	-	Total real time of the consecutive segments (125 µsec units for SM/CMP, 1 millisecond for OKP)
eeeeeeee	-	Elapsed time between the start of the first segment and the end of the last segment (125 µsec units for SM/CMP, 1 millisecond for OKP)

Software Analysis Guide

CONTENTS PAGE

13. OSDS OVERLOAD MONITOR 13-1
13.1 OVERVIEW 13-1
13.2 PCB OVERLOADS 13-1
13.3 SCB OVERLOADS 13-1
13.4 MCB OVERLOADS 13-1
13.5 TCB OVERLOADS 13-1
13.6 OUTPUT FILES 13-1
13.6.1 Output File Location and Layout 13-1
13.6.2 PCB Data Dump Layout 13-2
13.6.3 SCB Data Dump Layout 13-2
13.6.4 MCB Data Dump Layout 13-3
13.6.5 TCB Data Dump Layout 13-3
13.7 UP-LOADING CONTENTION 13-4

13. OSDS OVERLOAD MONITOR

13.1 OVERVIEW

The OSDS overload monitor is a feature present on the AM-OKP, CMP, and SM-2000s that records data whenever an overload occurs on one of the OSDS resources such as process control blocks (PCBs), stack control blocks (SCBs), message control blocks (MCBs), or timer control blocks (TCBs). See "OSDS Monitor," Section 12.2, for information on the OSDS resources.

The data is gathered at the moment when the overload is detected and is written to a file on the AM. The data can be used as a starting point for investigation of the overload. It may not provide enough information to determine the complete cause of the overload, but will at least indicate an area to investigate further via the OSDS monitors.

13.2 PCB OVERLOADS

When a PCB overload occurs, a piece of data is written for every PCB in use. The data consists of the PCB number (also known as, the process number), program ID, current state and uniqueness of the process. See "PCB Data Dump Layout," Section 13.6.2 for the layout of the dump.

13.3 SCB OVERLOADS

When an SCB overload occurs, a piece of data is written for every SCB in use. The data consists of the SCB number, process number, program ID, current state and uniqueness of the owning process. See "SCB Data Dump Layout," Section 13.6.3 for the layout of the dump.

13.4 MCB OVERLOADS

When an MCB overload occurs, a piece of data is written for every MCB in use. The data consists of the MCB number, process number, program ID, current state and uniqueness of the owning process, and the message header of the message stored in the MCB. The message header contains the sending process ID, message type, message priority, and message length. See "MCB Data Dump Layout," Section 13.6.4 for the layout of the dump.

13.5 TCB OVERLOADS

When a TCB overload occurs, a piece of data is written for every TCB in use. The data consists of the TCB number (also known as the timer's system tag), process number, program ID, current state and uniqueness of the owning process, the timer's user tag and the timer's state. See "TCB Data Dump Layout," Section 13.6.5 for the layout of the dump.

13.6 OUTPUT FILES

13.6.1 Output File Location and Layout

The output files for the OSDS overload monitor are all stored on the AM in the /log directory. The file names follow the convention:

OM<proc #>.<uniq #>

Where:

proc # - number of the processor on which the overload occurred (1-192 = SMs/SM-2000s, 193 = OKP, 194 = active CMP, 206 = standby CMP).

uniq # - number to ensure that the file name is unique.

Example:

OM1.456 - overload occurred on SM1

OM193.4 - overload occurred in OKP

A maximum of five OM<proc #>.<uniq #> files can be kept in the /log directory. If five files exist when another overload occurs, the oldest file will be deleted.

13.6.2 PCB Data Dump Layout

A file containing PCB overload information will begin with a dump of:

5252525200000002

This key is followed by an 8 byte dump for each PCB that was in use at the time of the overload. The dump layout is:

0000pppp ggggssuu

Where:

pppp - Process number

gggg - Program ID

ss - Process State

- 0 = EMPTY (not in use)
- 1 = RUNNING (process was running)
- 2 = READY (process was ready to run)
- 3 = WAITING (process was waiting via OSWAIT())
- 4 = RECEIVING (process was waiting via OSWGETMSG())
- 5 = RECTYPE (process was waiting via OSWGETTYPE())
- 6 = RECFROM (process was waiting via OSWGETFROM())
- 7 = RESTARTING (process was waiting via OSRESTART())
- 8 = CMP - BLOCKED (process was blocked by CMP soft switch)
  OKP,SM - LIMBO (transient state)
- 9 = CMP - LIMBO (transient state)
  OKP,SM - not used

uu - Process uniqueness

13.6.3 SCB Data Dump Layout

A file containing SCB overload information will begin with a dump of:

5252525200000003

This key is followed by an 8 byte dump for each SCB that was in use at the time of the overload. The dump layout is:

sssspppp ggggssuu

Where:

ssss - SCB number

pppp - number of the owning process

gggg - program ID of the owning process

ss - state of the owning process (See "PCB Data Dump Layout," Section 13.6.2 for key)

uu - uniqueness of the owning process

13.6.4 MCB Data Dump Layout

A file containing MCB overload information will begin with a dump of:

```
5252525200000004
```

This key is followed by a 16 byte dump for each MCB that was in use at the time of the overload. The dump layout is:

```
mmmmpppp ggggssuu xxxxyyzz ttttrrll
```

Where:

mmmm - MCB number

pppp - number of the owning process

gggg - program ID of the owning process

ss - state of the owning process (See "PCB Data Dump Layout," Section 13.6.2 for key)

uu - uniqueness of the owning process

xxxx - process number of the sending process

yy - processor number of the sending process

zz - uniqueness of the sending process

tttt - message type

rr - message priority (0-7)

ll - message length

13.6.5 TCB Data Dump Layout

A file containing TCB overload information will begin with a dump of:

```
5252525200000005
```

This key is followed by a 12 byte dump for each TCB that was in use at the time of the overload. The dump layout is:

```
ttttpppp ggggssuu xxxx00yy
```

Where:

tttt - TCB number (timer's system tag)

pppp - number of the owning process

gggg - program ID of the owning process

ss - state of the owning process (see "PCB Data Dump Layout," Section 13.6.2 for key)

uu - uniqueness of the owning process

xxxx - timer's user tag

yy - timer's state

0 = EMPTY (not in use)

1 = QTIMEOUT (timer used by OS wait primitives)

2 = QTIMER (one-shot timer)

3 = CTIMER (cyclic timer)

4 = ATIMER (one-shot time-of-day timer)

5 = CATIMER (cyclic time-of-day timer)

6 = LIMBO (transient state)

13.7 UP-LOADING CONTENTION

The OSDS overload monitor uses the same up-loading mechanism as the OSDS monitors. Therefore, only one processor may send overload monitor data or OSDS monitor data to the AM at a time. The overload monitor has a built in mechanism to retry up-loading data in the event that the up-loading mechanism is in use.

A1. ENVIRONMENT TO PATHNAME CROSS REFERENCE

Environment	Pathname	Description
3BSWAB	/bin/3bswab	Swab bytes in files for other machines
ABTAM	/cft/shl/cmds/ABT/AMATAPE	Abort Automatic Message Accounting (AMA) tape writing process
AIM	/prc/aim	Application integrity monitor
ALSAM	/cft/shl/cmds/ALW/AMA/SESSION	Allow AMA session
ALTAM	/cft/shl/cmds/ALW/AMA/AUTOST	Allow automatic AMA tape writing
AMDW	/no5text/prc/amdwic /no5text/prc/amdwoc	AMA message disk writer kernel processes
APDL	/no5text/prc/apdl	Application processor data link process
APPRC	/usr/bin/apprc	Recent Change (RC) maintenance control center
BKCNTL	/no5text/rcv/bkcntl	Office Dependent Data (ODD) backup control *UNIX*[a] RTR process
BOCNTL	/no5text/prc/SIrcbk	RC backup control *UNIX* RTR process
CCSINIT	/no5text/prc/ccsinit	Common Channel Signaling (CCS) initialization
CCpgeupdt	/no5text/hm/CCpgeupdt	CCS
CFILEAM	/cft/shl/cmds/OP/AMA/CONTROLFILE	Output contents of AMA control file
CMKP	/no5text/prc/cmkp	Communication package kernel process
CMP-AP	/no5text/cmp/CMP.out	CMP application processor
CMP-MSGH	/no5text/cmp/CMPMSGH.out	CMP message handler processor
CMP-OUT	/no5text/cmp/CMP.out	
CMPPUMP	/no5text/prc/cmppump	CMP pump process
COTDL		Customer-originated trace data link
CPBKUP	/no5text/rcv/cpbkup	Database backup *UNIX* RTR process for AM
CPDIAGC	/no5text/diag/dgnc/cpdiagc	AM diagnostic control
CPIMCTL		AM-SM control
CPRMV		
CPRS	/no5text/diag/dgnc/cprs	Diagnostic (DG)
CPTLPR	/no5text/diag/dgnc/cptlpr	AM trouble locating procedure process (for CM hardware)
CTRD	/no5text/tm/CTrd	Terminal Maintenance (TM)
CTWR	/no5text/tm/CTwr	TM
DB3BBSTUNX		Redundant bit map recovery process
DBCP3BEDPKG	/usr/bin/odbe-	*UNIX* RTR product for the Office Data Base Editor (ODBE)

See note(s) at end of table.

Environment	Pathname	Description
DBNRGRWUNX		Non-redundant ODD growth
DBODDGRW	/no5text/rcv/oddgrw	AM ODD growth
DBRGRWUNX		Redundant ODD growth
DBSACNVT	/no5text/rcv/sacnvt	Converts an RSM to a stand-alone module
DBUGRWUNX		Unprotected ODD growth
DGCPSUPC	/no5text/prc/DGcpsupc	AM diagnostic supervisor
DGPSUP	/no5text/prc/DGpsup	DG paging supervisor
DMAM		Diagnostic maintenance supervisor
DMON	/no5text/diag/dgnc/dmon	Diagnostic monitor
DWAM	/no5text/prc/amdwic	DG
ECAP	/no5text/as/ECap	EADAS *UNIX* RTR administrative process
ECR3	/no5text/as/ECr3	EADAS *UNIX* RTR high priority channel read process
ECR4	/no5text/as/ECr4	EADAS *UNIX* RTR low priority
ECR5	/no5text/as/ECr5	EADAS *UNIX* RTR read process
ECR6	/no5text/as/ECr6	EADAS *UNIX* RTR read process
ECW3	/no5text/as/ECw3	EADAS *UNIX* RTR high priority channel write process
ECW4	/no5text/as/ECw4	EADAS *UNIX* RTR low priority channel write process
ECW5	/no5text/as/ECw5	EADAS *UNIX* RTR write process
ECW6	/no5text/as/ECw6	EADAS *UNIX* RTR write process
FPUMP	/no5text/prc/fpump	SM fast pump
FTPAM	/no5text/prc/amftpic	AMA file transfer process
GROWTH	/no5text/rcv/smoddbst /no5text/rcv/smoddngre /no5text/rcv/smoddrgrw /no5text/rcv/smoddugrw	ODD growth processes
HMALM	/no5text/hm/HMalm	Human machine alarm process
HMIRA	/no5text/hm/HMira	Human machine input request administrator
HMLOGMAP	/no5text/hm/HMlogmap	Human machine logfile mapping *UNIX* RTR process
HMMCC	/no5text/hm/HMmcc	Human machine master control center (MCC) control process
HMOPN		Human machine *UNIX* RTR process
HMREAD	/no5text/hm/HMiread /no5text/hm/HMoread	Human machine read *UNIX* RTR processes
HMSIP	/no5text/hm/HMsip	Human machine spooler input process

See note(s) at end of table.

Environment	Pathname	Description
HMTIME	/no5text/hm/HMtime	Human machine timing process
INHSAM	/cft/shl/cmds/INH/AMA/SESSION	Inhibit AMA session
INHTAM	/cft/shl/cmds/INH/AMA/AUTOST	Inhibit automatic AMA tape writing
IODRV	/bootfiles/3bsgen.kern	I/O driver kernel process
LBPUMP	/no5text/prc/lbpump	SM little boot pump
LGCNTL	/no5text/rcv/lgcntl	RC log control *UNIX* RTR process
LGCRC	/no5text/rcv/lgcrc	Customer Originated Recent Change (CORC) log control *UNIX* RTR process
LGINITROLL	/no5text/rcv/lginitroll	ODD log roll forward *UNIX* RTR process
LGLOG	/no5text/rcv/lglog	RC log control *UNIX* RTR process
LGROLL	/no5text/rcv/lgroll	RC recovery *UNIX* RTR process
MCRTRC	/usr/bin/mcrtrc	RC
MONAM		AMA monitor process
MOP	/no5text/prc/mop	Mount offline partition process
MSDIAGC		Diagnostic control
MSKP	/no5text/prc/mskp	Message switch kernel process
ODDPAR	/no5text/rev/oddgrow	Parent process of the ODD growth
ODISKAM	/cft/shl/cmds/OP/AMA/DISK	Output AMA disk occupancy information
OGEN	/prc/ogen	AM modified for *5ESS*® switch use
OKP	/no5text/prc/okp	Operational kernel process
OPTPAM	/cft/shl/cmds/OP/AMA/TELEPROCESSING	Output AMA teleprocessing information
OSDSM	/no5text/im/smtxt/IM.out /no5text/im/sm2ktxt/IM.out	OSDS in the SM
OSESAM	/cft/shl/cmds/OP/AMA/SESSION	Output AMA session information
PCTL	/no5text/prc/UPpctl	Program update control
PDSHL.APP	/cft//bin/pdshl.app	Application synchronous craft shell
PDSHLA.APP		Application synchronous craft shell
PLOD	/no5text/prc/plod	Process for loading ODD on to disk
PLOP	/no5text/prc/plop	Process for loading ODD in the Protected Application Segment (PAS)
PMKP	/prc/pmkp	Pump kernel process
PUCR	/no5text/prc/pucr	Pump control process
PUMP	/no5text/ims/pump	SM pump process

See note(s) at end of table.

Environment	Pathname	Description
PUPCI	/no5text/prc/UPpupci	Program update process - craft interface
RCCP3BSRC	/no5text/rcv/RCcp3bsec	RC
RCKP	/no5text/prc/rckp	RC kernel process
RCRMAS		
RINGMON	/no5text/ccs/proc/CCringmon	CCS ring monitor process
RTR		RTR kernel process
SFAM	/cft/shl/cmds/SET/AMA/CONTROL	Set AMA control file information
SIOFFN	/no5text/prc/SIoffn	System integrity offnormal reporting process
SMAPRTS	/no5text/prc/SMaprts	Application real time status
SMBKUP	/no5text/rcv/cpbkup	SM ODD backup *UNIX* RTR process
SMDIMP	/no5text/prc/SMdimp	SM diagnostic input message processor
SMDOMP	/no5text/prc/SMdomp	SM diagnostic output message processor
SMIIAU	/no5text/prc/SMiiau	Switch maintenance inhibit and allow *UNIX* RTR process
SMIMRPT		SM, IM report generator
SMKP	/no5text/prc/smkp	Switch maintenance kernel process
SMNO5FM		*5ESS* switch frame power monitor
SMONL		SM off-normal reporting *UNIX* RTR process
SMPSM	/no5text/prc/SMpsm	Switch maintenance power switch monitor
SMSTO	/no5text/prc/SMstout	
SMSTOUT	/no5text/prc/SMstout	Switch maintenance status output *UNIX* RTR process
STAM		Stop AMA tape writing process
SUOVPRC		
TAPEAM		AMA tape writing process
TERAUX		Export trunk error analysis
TMDAP		TM display administrator
TMSR		
TMSW		
UAXFER		
UCNTL	/no5text/prc/ucntl	AM modified for *5ESS* switch use
UPDISPATCH	/no5text/prc/UPdispatch	Program update *UNIX* RTR process
UPSETIND	/no5text/prc/UPsetind	Program update *UNIX* RTR process
UTCP3B	/no5text/prc/utcp3b	AM utilities *UNIX* RTR process

See note(s) at end of table.

Environment	Pathname	Description
UTLRMAIN	/no5text/prc/WElrmain	AM untilities *UNIX* RTR process
UTLSMAIN	/no5text/prc/WElsmain	AM utilities *UNIX* RTR process
VERTAPEAM	/no5text/prc/amtapeic	Verify the AMA tape
VTAM	/cft/shl/cmds/VFY/AMATAPE	Verify the AMA tape
WTAM	/cft/shl/cmds/CPY/AMATAPE	Write the AMA tape

a. Registered trademark of The Open Group.

A2. IS25, 3B20, AND 3B21 COMPUTER INSTRUCTION LIST (BY MNEMONIC)

This section contains the IS25, 3B20, and 3B21 computer instruction list sorted by mnemonic. The following listing defines the letters used in the syntax column of this appendix.

g	general addressing mode
i	immediate mode
n	nibble mode (4-bit immediate)
m	memory mode (0-9)
r	register number

A more detailed discussion of the instruction set for 3B20D and 3B21D computers can be found in the 3B20D and 3B21D Computer manuals, specifically the 303-028, **UNIX**[1] *RTR Operating System Assembly Language User's Guide*.

Table A2-1 — Mnemonics A Through H

Mnemonic	Opcode	Subcode	Syntax	Function	Type
acj1	74	0	rggm	Add, compare and jump less	IS25
acj1	74	4	mggm	Add, compare and jump less	IS25
acj1	74	8	rggm	Add, compare and jump less	IS25
acj1	74	C	mggm	Add, compare and jump less	IS25
acj1e	74	2	rggm	Add, compare and jump less or equal	IS25
acj1e	74	6	mggm	Add, compare and jump less or equal	IS25
acjle	74	A	rggm	Add, compare and jump less or equal	IS25
acjle	74	E	mggm	Add, compare and jump less or equal	IS25
acjleu	74	3	rggm	Add, compare and jump less or equal unsigned	IS25
acjleu	74	7	mggm	Add, compare and jump less or equal unsigned	IS25
acjleu	74	B	rggm	Add, compare and jump less or equal unsigned	IS25
acjleu	74	F	mggm	Add, compare and jump less or equal unsigned	IS25
acjlu	74	1	rggm	Add, compare and jump less unsigned	IS25
acjlu	74	5	mggm	Add, compare and jump less unsigned	IS25

1. Registered trademark of The Open Group.

Table A2-1 — Mnemonics A Through H (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
acjlu	74	9	rggm	Add, compare and jump less unsigned	IS25
acjlu	74	D	mggm	Add, compare and jump less unsigned	IS25
addb2	20	6	rr	Add (2 oper)	IS25
addb2	21	6	rm	Add (2 oper)	IS25
addb2	22	6	nr	Add (2 oper)	IS25
addb2	23	6	nm	Add (2 oper)	IS25
addb2	24	6	gr	Add (2 oper)	IS25
addb2	25	6	gm	Add (2 oper)	IS25
addb3	68	3	ggr	Add (3 oper)	IS25
addb3	68	B	ggm	Add (3 oper)	IS25
addh2	28	6	rr	Add (2 oper)	IS25
addh2	29	6	rm	Add (2 oper)	IS25
addh2	2A	6	nr	Add (2 oper)	IS25
addh2	2B	6	nm	Add (2 oper)	IS25
addh2	2C	6	gr	Add (2 oper)	IS25
addh2	2D	6	gm	Add (2 oper)	IS25
addh3	69	3	ggr	Add (3 oper)	IS25
addh3	69	B	ggm	Add (3 oper)	IS25
addw2	10	-	rr	Add (2 oper)	OPT
addw2	11	-	nr	Add (2 oper)	OPT
addw2	30	6	rr	Add (2 oper)	IS25
addw2	31	6	rm	Add (2 oper)	IS25
addw2	32	6	nr	Add (2 oper)	IS25
addw2	33	6	nm	Add (2 oper)	IS25
addw2	34	6	gr	Add (2 oper)	IS25
addw2	35	6	gm	Add (2 oper)	IS25
addw3	6A	3	ggr	Add (3 oper)	IS25
addw3	6A	B	ggm	Add (3 oper)	IS25
alsw2	-	-	gm	Arith. left shift (2 oper)	IS25
alsw2	-	-	gr	Arith. left shift (2 oper)	IS25
alsw3	6D	2	ggr	Arith. left shift (3 oper)	IS25
alsw3	6D	A	ggm	Arith. left shift (3 oper)	IS25
andb2	20	0	rr	And (2 oper)	IS25
andb2	21	0	rm	And (2 oper)	IS25
andb2	22	0	nr	And (2 oper)	IS25
andb2	23	0	nm	And (2 oper)	IS25

Table A2-1 — Mnemonics A Through H (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
andb2	24	0	gr	And (2 oper)	IS25
andb2	25	0	gm	And (2 oper)	IS25
andb3	68	0	ggr	And (3 oper)	IS25
andb3	68	8	ggm	And (3 oper)	IS25
andh2	28	0	rr	And (2 oper)	IS25
andh2	29	0	rm	And (2 oper)	IS25
andh2	2A	0	nr	And (2 oper)	IS25
andh2	2B	0	nm	And (2 oper)	IS25
andh2	2C	0	gr	And (2 oper)	IS25
andh2	2D	0	gm	And (2 oper)	IS25
andh3	69	0	ggr	And (3 oper)	IS25
andh3	69	8	ggm	And (3 oper)	IS25
andw2	1B	-	rr	And (2 oper)	OPT
andw2	30	0	rr	And (2 oper)	IS25
andw2	31	0	rm	And (2 oper)	IS25
andw2	32	0	nr	And (2 oper)	IS25
andw2	33	0	nm	And (2 oper)	IS25
andw2	34	0	gr	And (2 oper)	IS25
andw2	35	0	gm	And (2 oper)	IS25
andw3	6A	0	ggr	And (3 oper)	IS25
andw3	6A	8	ggm	And (3 oper)	IS25
arsw2	-	-	gm	Arith. right shift (2 oper)	IS25
arsw2	-	-	gr	Arith. right shift (2 oper)	IS25
arsw3	6D	3	ggr	Arith. right shift (3 oper)	IS25
arsw3	6D	B	ggm	Arith. right shift (3 oper)	IS25
atjnzb	75	0	rgm	Add, test and jump not zero	IS25
atjnzb	75	4	mgm	Add, test and jump not zero	IS25
atjnzb	75	8	rgm	Add, test and jump not zero	IS25
atjnzb	75	C	mgm	Add, test and jump not zero	IS25
atjnzh	75	1	rgm	Add, test and jump not zero	IS25
atjnzh	75	5	mgm	Add, test and jump not zero	IS25
atjnzh	75	9	rgm	Add, test and jump not zero	IS25
atjnzh	75	D	mgm	Add, test and jump not zero	IS25
atjnzw	75	2	rgm	Add, test and jump not zero	IS25
atjnzw	75	6	mgm	Add, test and jump not zero	IS25
atjnzw	75	A	rgm	Add, test and jump not zero	IS25
atjnzw	75	E	mgm	Add, test and jump not zero	IS25
bcc	85	-	m	Branch on carry clear	OPT

Table A2-1 — Mnemonics A Through H (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
bcc	95	-	m	Branch on carry clear	OPT
bcs	86	-	m	Branch on carry set	OPT
bcs	96	-	m	Branch on carry set	OPT
be	82	-	m	Branch equal	OPT
be	92	-	m	Branch equal	OPT
bg	89	-	m	Branch greater	OPT
bg	99	-	m	Branch greater	OPT
bge	83	-	m	Branch greater or equal	OPT
bge	93	-	m	Branch greater or equal	OPT
bgeu	86	-	m	Branch greater or equal unsigned	OPT
bgeu	96	-	m	Branch greater or equal unsigned	OPT
bgu	8B	-	m	Branch greater unsigned	OPT
bgu	9B	-	m	Branch greater unsigned	OPT
bitb	20	3	rr	Bit test	IS25
bitb	21	3	rm	Bit test	IS25
bitb	22	3	nr	Bit test	IS25
bitb	23	3	nm	Bit test	IS25
bitb	24	3	gr	Bit test	IS25
bitb	25	3	gm	Bit test	IS25
bith	28	3	rr	Bit test	IS25
bith	29	3	rm	Bit test	IS25
bith	2A	3	nr	Bit test	IS25
bith	2B	3	nm	Bit test	IS25
bith	2C	3	gr	Bit test	IS25
bith	2D	3	gm	Bit test	IS25
bitw	1D	-	rr	Bit test	OPT
bitw	30	3	rr	Bit test	IS25
bitw	31	3	rm	Bit test	IS25
bitw	32	3	nr	Bit test	IS25
bitw	33	3	nm	Bit test	IS25
bitw	34	3	gr	Bit test	IS25
bitw	35	3	gm	Bit test	IS25
bl	84	-	m	Branch less	OPT
bl	94	-	m	Branch less	OPT
ble	8A	-	m	Branch less or equal	OPT
ble	9A	-	m	Branch less or equal	OPT

Table A2-1 — Mnemonics A Through H (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
bleu	8C	-	m	Branch less or equal unsigned	OPT
bleu	9C	-	m	Branch less or equal unsigned	OPT
blu	85	-	m	Branch less unsigned	OPT
blu	95	-	m	Branch less unsigned	OPT
bne	81	-	m	Branch not equal	OPT
bne	91	-	m	Branch not equal	OPT
bneg	84	-	m	Branch negative	OPT
bneg	94	-	m	Branch negative	OPT
bnneg	83	-	m	Branch not negative	OPT
bnneg	93	-	m	Branch not negative	OPT
bnpos	8A	-	m	Branch not positive	OPT
bnpos	9A	-	m	Branch not positive	OPT
bnz	81	-	m	Branch not zero	OPT
bnz	91	-	m	Branch not zero	OPT
bph	6F	-	-	Breakpoint halt	3B20
bpos	89	-	m	Branch positive	OPT
bpos	99	-	m	Branch positive	OPT
bpt	DA	-	i	Breakpoint trap	3B20
br	80	-	m	Branch	OPT
br	90	-	m	Branch	OPT
bsb	8F	-	m	Branch to subroutine	OPT
bsb	9F	-	m	Branch to subroutine	OPT
bsmo8	46	8	-	BIST and boundary scan	3B21
bsmo9	46	9	-	BIST and boundary scan	3B21
bsmoa	46	a	-	BIST and boundary scan	3B21
bsmob	46	b	-	BIST and boundary scan	3B21
bsmoc	46	c	-	BIST and boundary scan	3B21
bsmod	46	d	-	BIST and boundary scan	3B21
bsmoe	46	e	-	BIST and boundary scan	3B21
bsmof	46	f	-	BIST and boundary scan	3B21
bvc	87	-	m	Branch on overflow clear	OPT
bvc	97	-	m	Branch on overflow clear	OPT
bvs	88	-	m	Branch on overflow set	OPT
bvs	98	-	m	Branch on overflow set	OPT
bz	82	-	m	Branch zero	OPT
bz	92	-	m	Branch zero	OPT

Table A2-1 — Mnemonics A Through H (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
cachi	C6	3	-	Cache initialization	PRIV
cale	7C	-	igi	Call to emulation	PRIV
call	77	0	ig	Call	OPT
call	77	1	ig	Call	OPT
call	78	-	ig	Call	IS25
call	79	-	ig	Call	OPT
call	B9	8	igm	Call	OPT
chrh	D4	3	r	Cache read hit counter	3B20
chrm	D4	4	r	Cache read miss counter	3B20
cinov	C1	B	r	Clear I/O inhibit override	PRIV
clrmd0	E0	8	-	Clear my store error D	PRIV
clrmd1	E0	9	-	Clear my store error D	PRIV
clrod0	E2	8	-	Clear other store error D	PRIV
clrod1	E2	9	-	Clear other store error D	PRIV
cmpb	20	E	rr	Compare	IS25
cmpb	21	E	rg	Compare	IS25
cmpb	22	E	nr	Compare	IS25
cmpb	22	F	rn	Compare	IS25
cmpb	23	E	ng	Compare	IS25
cmpb	23	F	mn	Compare	IS25
cmpb	24	E	gr	Compare	IS25
cmpb	25	E	gg	Compare	IS25
cmph	28	E	rr	Compare	IS25
cmph	29	E	rg	Compare	IS25
cmph	2A	E	nr	Compare	IS25
cmph	2A	F	rn	Compare	IS25
cmph	2B	E	ng	Compare	IS25
cmph	2B	F	mn	Compare	IS25
cmph	2C	E	gr	Compare	IS25
cmph	2D	E	gg	Compare	IS25
cmpn	36	-	ri	Compare	IS25
cmpw	18	-	rr	Compare	OPT
cmpw	19	-	nr	Compare	OPT
cmpw	1A	-	rn	Compare	OPT
cmpw	30	E	rr	Compare	IS25
cmpw	31	E	rg	Compare	IS25
cmpw	32	E	nr	Compare	IS25
cmpw	32	F	rn	Compare	IS25

Table A2-1 — Mnemonics A Through H (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
cmpw	33	E	ng	Compare	IS25
cmpw	33	F	mn	Compare	IS25
cmpw	34	E	gr	Compare	IS25
cmpw	35	E	gg	Compare	IS25
cmpzv	6E	2	riir	Compare field	IS25
cmpzv	6E	6	riim	Compare field	IS25
cmpzv	6E	A	giir	Compare field	IS25
cmpzv	6E	E	giim	Compare field	IS25
cpoff	ED	8	rr	Compare off-line ms to on-line ms	PRIV
cpon	ED	0	rr	Copy on-line main store	PRIV
cs	D2	-	r	Change state	PRIV
decpth	59	-	rr	Decrement pointer by index	IS25
decptw	5A	-	rr	Decrement pointer by index	IS25
diag	C3	-	i	Diagnostics opcode	PRIV
divw2	30	A	rr	Divide (2 oper)	IS25
divw2	31	A	rm	Divide (2 oper)	IS25
divw2	32	A	nr	Divide (2 oper)	IS25
divw2	33	A	nm	Divide (2 oper)	IS25
divw2	34	A	gr	Divide (2 oper)	IS25
divw2	35	A	gm	Divide (2 oper)	IS25
divw3	6C	2	ggr	Divide (3 oper)	IS25
divw3	6C	A	ggm	Divide (3 oper)	IS25
dmioh	C0	5	rrr	Do maint. I/O	PRIV
dmiow	C0	1	rrr	Do maint. I/O	PRIV
doioh	C0	4	rrr	Do I/O	PRIV
doiow	C0	0	rrr	Do I/O	PRIV
enmd	E0	B	-	Enable my store error D reporting	PRIV
enod	E2	B	-	Enable other store error D reporting	PRIV
extzv	6E	1	riir	Extract field	IS25
extzv	6E	5	riim	Extract field	IS25
extzv	6E	9	giir	Extract field	IS25
extzv	6E	D	giim	Extract field	IS25
fidl	C0	8	r	Force channel idle	PRIV
frz	1F	-	rr	Find rightmost zero	3B20
gcc	DC	0	r	Get condition code	3B20

Table A2-1 — Mnemonics A Through H (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
getproc	DC	-	-	Get address of processor rable entry	3B20
haltx	DF	-	i	Halt machine and display	3B20

Table A2-2 — Mnemonics I Through Q

Mnemonic	Opcode	Subcode	Syntax	Function	Type
incpth	57	-	rr	Increment pointer by index	IS25
incptw	58	-	rr	Increment pointer by index	IS25
inctst	B5	-	rir	Increment and test index	IS25
inctst	B6	-	rig	Increment and test index	IS25
inhmd	E0	A	-	Inhibit my store error D reporting	PRIV
inhod	E2	A	-	Inhibit other store error D	PRIV
initgp	A0	00	-	Initialize general purpose registers	3B20
insv	6E	0	riir	Insert field	IS25
insv	6E	4	riim	Insert field	IS25
insv	6E	8	giir	Insert field	IS25
insv	6E	C	giim	Insert field	IS25
iocler	C2	8	rr	I/O clear error	PRIV
ioeack	C2	A	rr	I/O error acknowledge	PRIV
ioiack	C2	9	rr	I/O interrupt acknowledge	PRIV
ioidl	C2	1	r	I/O idle	PRIV
iord	C2	2	rr	I/O receive data	PRIV
ioread	C2	F	rrrr	I/O read user cmd	PRIV
iorint	C2	4	rr	I/O receive interrupt status	PRIV
iorsr	C2	5	rr	I/O receive service request	PRIV
iorst	C2	3	rr	I/O receive status	PRIV
iosrack	C2	B	rr	I/O service request acknowledge	PRIV
iowca	C2	6	rr	I/O write cmd address	PRIV
iowd	C2	7	rr	I/O write data	PRIV
iowdp	C2	C	rr	I/O write data with parity	PRIV
iowt	C2	C	rrrr	I/O write user cmd	PRIV
jbc	72	0	igm	Jump on bit clear	IS25
jbs	72	1	igm	Jump on bit set	IS25
jcc	70	5	m	Jump on carry clear	3B20
jcs	70	6	m	Jump on carry set	3B20

Table A2-2 — Mnemonics I Through Q (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
je	70	2	m	Jump equal	IS25
jg	70	9	m	Jump greater	IS25
jge	70	3	m	Jump greater or equal	IS25
jgeu	70	6	m	Jump greater or equal unsigned	IS25
jgu	70	B	m	Jump greater unsigned	IS25
jioe	73	0	m	Jump on I/O error	3B20
jiom	73	1	m	Jump on I/O maint	3B20
jion	73	2	m	Jump on I/O normal	3B20
jiot	73	3	m	Jump on I/O timeout	3B20
jl	70	4	m	Jump less	IS25
jle	70	A	m	Jump less or equal	IS25
jleu	70	C	m	Jump less or equal unsigned	IS25
jlu	70	5	m	Jump less unsigned	IS25
jmp	70	0	m	Jump	IS25
jne	70	1	m	Jump not equal	IS25
jneg	70	4	m	Jump negative	IS25
jnneg	70	3	m	Jump not negative	IS25
jnpos	70	A	m	Jump not positive	IS25
jnz	70	1	m	Jump not zero	IS25
jpos	70	9	m	Jump positive	IS25
jsb	70	F	m	Jump to subroutine	IS25
jvc	70	7	m	Jump on overflow clear	3B20
jvs	70	8	m	Jump on overflow set	3B20
jz	70	2	m	Jump zero	IS25
lbsmd	46	7	-	BIST and boundary scan	3B21
lbsms	46	6	-	BIST and boundary scan	3B21
llsw2	-	-	gm	Logical left shift (2 oper)	IS25
llsw2	-	-	gr	Logical left shift (2 oper)	IS25
llsw3	6D	0	ggr	Logical left shift (3 oper)	IS25
llsw3	6D	8	ggm	Logical left shift (3 oper)	IS25
lmchb	C5	5	r	Load from on-line mch buffer	PRIV
lmchs	C5	4	r	Load from on-line mch status	PRIV
lrsw2	-	-	gm	Logical right shift (2 oper)	IS25
lrsw2	-	-	gr	Logical right shift (2 oper)	IS25
lrsw3	6D	1	ggr	Logical right shift (3 oper)	IS25

Table A2-2 — Mnemonics I Through Q (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
lrsw3	6D	9	ggm	Logical right shift (3 oper)	IS25
lsm	65	0	gm	Load selected multiple registers	3B20
marf	d6	a	-	Maintenance access - read offline	3B21
marn	d6	8	-	Maintenance access - read online	3B21
mawf	d6	b	-	Maintenance access - write offline	3B21
mawn	d6	9	-	Maintenance access - write online	3B21
mchex	C5	1	-	Maint. channel exec	PRIV
mchin	C5	0	-	Maint. channel init	PRIV
mcomb	40	1	rr	Move complemented	IS25
mcomb	41	1	rm	Move complemented	IS25
mcomb	42	1	nr	Move complemented	IS25
mcomb	43	1	nm	Move complemented	IS25
mcomb	44	1	gr	Move complemented	IS25
mcomb	45	1	gm	Move complemented	IS25
mcomh	48	1	rr	Move complemented	IS25
mcomh	49	1	rm	Move complemented	IS25
mcomh	4A	1	nr	Move complemented	IS25
mcomh	4B	1	nm	Move complemented	IS25
mcomh	4C	1	gr	Move complemented	IS25
mcomh	4D	1	gm	Move complemented	IS25
mcomw	16	-	rr	Move complemented	OPT
mcomw	50	1	rr	Move complemented	IS25
mcomw	51	1	rm	Move complemented	IS25
mcomw	52	1	nr	Move complemented	IS25
mcomw	53	1	nm	Move complemented	IS25
mcomw	54	1	gr	Move complemented	IS25
mcomw	55	1	gm	Move complemented	IS25
memaud	EE	-	rr	Memory audit	PRIV
meminit	EF	-	r	Main store initialization	PRIV
mnegh	48	2	rr	Move negated	IS25
mnegh	49	2	rm	Move negated	IS25
mnegh	4A	2	nr	Move negated	IS25
mnegh	4B	2	nm	Move negated	IS25
mnegh	4C	2	gr	Move negated	IS25

Table A2-2 — Mnemonics I Through Q (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
mnegh	4D	2	gm	Move negated	IS25
mnegw	17	-	rr	Move negated	OPT
mnegw	50	2	rr	Move negated	IS25
mnegw	51	2	rm	Move negated	IS25
mnegw	52	2	nr	Move negated	IS25
mnegw	53	2	nm	Move negated	IS25
mnegw	54	2	gr	Move negated	IS25
mnegw	55	2	gm	Move negated	IS25
modw2	30	C	rr	Modulo (2 oper)	IS25
modw2	31	C	rm	Modulo (2 oper)	IS25
modw2	32	C	nr	Modulo (2 oper)	IS25
modw2	33	C	nm	Modulo (2 oper)	IS25
modw2	34	C	gr	Modulo (2 oper)	IS25
modw2	35	C	gm	Modulo (2 oper)	IS25
modw3	6C	4	ggr	Modulo (3 oper)	IS25
modw3	6C	C	ggm	Modulo (3 oper)	IS25
movaw	66	2	mr	Move address	IS25
movaw	66	A	mm	Move address	IS25
movb	40	0	rr	Move	IS25
movb	41	0	rm	Move	IS25
movb	42	0	nr	Move	IS25
movb	43	0	nm	Move	IS25
movb	44	0	gr	Move	IS25
movb	45	0	gm	Move	IS25
movbbh	40	5	rr	Move bit extended	IS25
movbbh	41	5	rm	Move bit extended	IS25
movbbh	42	5	nr	Move bit extended	IS25
movbbh	43	5	nm	Move bit extended	IS25
movbbh	44	5	gr	Move bit extended	IS25
movbbh	45	5	gm	Move bit extended	IS25
movbbw	40	6	rr	Move bit extended	IS25
movbbw	41	6	rm	Move bit extended	IS25
movbbw	42	6	nr	Move bit extended	IS25
movbbw	43	6	nm	Move bit extended	IS25
movbbw	44	6	gr	Move bit extended	IS25
movbbw	45	6	gm	Move bit extended	IS25
movbhw	48	5	rr	Move bit extended	IS25
movbhw	49	5	rm	Move bit extended	IS25

Table A2-2 — Mnemonics I Through Q (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
movbhw	4A	5	nr	Move bit extended	IS25
movbhw	4B	5	nm	Move bit extended	IS25
movbhw	4C	5	gr	Move bit extended	IS25
movbhw	4D	5	gm	Move bit extended	IS25
movblb	-	-	-	Move block (see moveblkb)	IS25
movblh	-	-	-	Move block (see movblkh)	IS25
movblkb	6B	0	rrr	Move block general	3B20
movblkh	6B	1	rrr	Move block general	3B20
movblkw	6B	2	rrr	Move block general	3B20
movblw	-	-	-	Move block (see movblkw)	IS25
movh	48	0	rr	Move	IS25
movh	49	0	rm	Move	IS25
movh	4A	0	nr	Move	IS25
movh	4B	0	nm	Move	IS25
movh	4C	0	gr	Move	IS25
movh	4D	0	gm	Move	IS25
movthb	37	-	rr	Move truncated	IS25
movtwb	37	-	rr	Move truncated	IS25
movtwb	51	8	rm	Move	IS25
movtwh	51	7	rm	Move	IS25
movw	14	-	rr	Move	OPT
movw	15	-	nr	Move	OPT
movw	26	-	mr	Move	OPT
movw	27	-	mr	Move	OPT
movw	2F	-	nm	Move	OPT
movw	39	-	mr	Move	IS25
movw	3F	-	rm	Move	IS25
movw	50	0	rr	Move	IS25
movw	51	0	rm	Move	IS25
movw	52	0	nr	Move	IS25
movw	53	0	nm	Move	IS25
movw	54	0	gr	Move	IS25
movw	55	0	gm	Move	IS25
movzbh	37	-	rr	Move zero extended	IS25
movzbh	44	3	gm	Move zero extended	IS25
movzbw	37	-	rr	Move zero extended	IS25
movzbw	44	4	gm	Move zero extended	IS25
movzhw	38	-	rr	Move zero extended	IS25

Table A2-2 — Mnemonics I Through Q (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
movznh	42	3	gm	Move zero extended	IS25
movznw	42	4	gm	Move zero extended	IS25
mrf	DD	0	-	Maint. reset function	PRIV
msrf	D6	0	rr	Maint. store read off-line	PRIV
msrn	D6	3	rr	Maint. store read on-line	PRIV
mssrf	D6	2	rr	Maint. store safe read off-line	PRIV
mssrn	D6	5	rr	Maint. store safe read on-line	PRIV
mswf	D6	1	rr	Maint. store safe read off-line	PRIV
mulw2	30	8	rr	Multiply (2 oper)	IS25
mulw2	31	8	rm	Multiply (2 oper)	IS25
mulw2	32	8	nr	Multiply (2 oper)	IS25
mulw2	33	8	nm	Multiply (2 oper)	IS25
mulw2	34	8	gr	Multiply (2 oper)	IS25
mulw2	35	8	gm	Multiply (2 oper)	IS25
mulw3	6C	0	ggr	Multiply (3 oper)	IS25
mulw3	6C	8	ggm	Multiply (3 oper)	IS25
nop	DE	-	-	No operation	3B20
orb2	20	1	rr	Or (2 oper)	IS25
orb2	21	1	rm	Or (2 oper)	IS25
orb2	22	1	nr	Or (2 oper)	IS25
orb2	23	1	nm	Or (2 oper)	IS25
orb2	24	1	gr	Or (2 oper)	IS25
orb2	25	1	gm	Or (2 oper)	IS25
orb3	68	1	ggr	Or (3 oper)	IS25
orb3	68	9	ggm	Or (3 oper)	IS25
orh2	28	1	rr	Or (2 oper)	IS25
orh2	29	1	rm	Or (2 oper)	IS25
orh2	2A	1	nr	Or (2 oper)	IS25
orh2	2B	1	nm	Or (2 oper)	IS25
orh2	2C	1	gr	Or (2 oper)	IS25
orh2	2D	1	gm	Or (2 oper)	IS25
orh3	69	1	ggr	Or (3 oper)	IS25
orh3	69	9	ggm	Or (3 oper)	IS25
orw2	1C	-	rr	Or (2 oper)	OPT
orw2	30	1	rr	Or (2 oper)	IS25
orw2	31	1	rm	Or (2 oper)	IS25

Table A2-2 — Mnemonics I Through Q (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
orw2	32	1	nr	Or (2 oper)	IS25
orw2	33	1	nm	Or (2 oper)	IS25
orw2	34	1	gr	Or (2 oper)	IS25
orw2	35	1	gm	Or (2 oper)	IS25
orw3	6A	1	ggr	Or (3 oper)	IS25
orw3	6A	9	ggm	Or (3 oper)	IS25
ost	D9	-	i	Operating system trap	3B20
patb	D3	-	r	Purge atb	PRIV
pioe	C2	0	r	Poll I/O error status	PRIV
pioi	C2	D	r	Poll I/O interrupt status	PRIV
pior	C2	E	r	Poll I/O service request status	PRIV
popw	64	0	r	Pop word	3B20
popw	64	8	m	Pop word	3B20
psiplse	7E	3	rr	PSI pulse order	PRIV
psipo	7E	0	rrrr	PSI peripheral order	PRIV
psird	7E	1	rr	PSI read register	PRIV
psiwt	7E	2	rr	PSI write register	PRIV
pushaw	63	2	m	Push address	IS25
pushbb	60	3	r	Push bit extended	IS25
pushbb	61	3	n	Push bit extended	IS25
pushbb	62	3	g	Push bit extended	IS25
pushbh	60	4	r	Push bit extended	IS25
pushbh	61	4	n	Push bit extended	IS25
pushbh	62	4	g	Push bit extended	IS25
pushw	2E	-	m	Push	IS25
pushw	60	0	r	Push	IS25
pushw	61	0	i	Push	IS25
pushw	62	0	g	Push	IS25
pushzb	60	1	r	Push zero extended	IS25
pushzb	61	1	n	Push zero extended	IS25
pushzb	62	1	g	Push zero extended	IS25
pushzh	60	2	r	Push zero extended	IS25
pushzh	61	2	n	Push zero extended	IS25
pushzh	62	2	g	Push zero extended	IS25

Table A2-3 — Mnemonics R Through Z

Mnemonic	Opcode	Subcode	Syntax	Function	Type
rbsmd	46	3	-	BIST and boundary scan	3B21
rbsms	46	2	-	BIST and boundary scan	3B21
rcb	67	0	mr	Read and clear	3B20
rcb	67	8	mm	Read and clear	3B20
rch	67	1	mr	Read and clear	3B20
rch	67	9	mm	Read and clear	3B20
rcrefe	D6	8	rrr	Read and clear refresh par err	PRIV
rcw	67	2	mr	Read and clear	3B20
rcw	67	A	mm	Read and clear	3B20
rdblk	C1	2	rrr	Read block	PRIV
rdblki	C1	2	rrr	Read block and clear device interrupt	PRIV
rdblkr	C1	2	rrr	Read block and clear service request	PRIV
rdinhp	C1	B	rrr	Read word and inhib par err	PRIV
rdinhpi	C1	B	rrr	Read word, inhib par err and clear device interrupt	PRIV
rdinhpr	C1	B	rrr	Read word, inhib par err and clear service request	PRIV
rdistk	E1	0	r	Read cache interrupt stack	PRIV
rdmsk	c0	f	-	Read mask	3B21
rdmskb	c0	f	-	Read mask and clear both service request and device interrupt	3B21
rdmski	c0	f	-	Read mask and clear device interrupt	3B21
rdmskr	c0	f	-	Read mask and clear service request	3B21
rdoser	E2	-	rr	Read other store error register	PRIV
rdphy	D5	0	rr	Read with physical address	3B20
rdphyb	d5	2	-	Physical address - read byte	3B21
rdphyh	d5	1	-	Physical address - read half-word	3B21
rdser	E0	-	rr	Read store error register	PRIV
rdsr	D0	-	rs	Read special register	3B20
rdtim	A2	-	rr	Read timers	PRIV

Table A2-3 — Mnemonics R Through Z (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
rdwrd	C1	0	rrr	Read word	PRIV
rdwrdi	C1	0	rrr	Read word and clear device interrupt	PRIV
rdwrdr	C1	0	rrr	Read word and clear service request	PRIV
ret	7B	-	i	Return	IS25
rete	7D	-	ii	Return to emulation	PRIV
rmchb	C5	B	r	Read off-line mch buffer	PRIV
rmchr	C5	9	rs	Read off-line mch register	PRIV
rmchs	C5	A	r	Read off-line mch status	PRIV
rotlw	6D	4	ggr	Rotate left	3B20
rotlw	6D	C	ggm	Rotate left	3B20
rotrw	6D	5	ggr	Rotate right	3B20
rotrw	6D	E	ggm	Rotate right	3B20
rp	D4	2	-	Restore primary	3B20
rrblk	C1	D	rrr	Request to read block	PRIV
rrwrd	C1	C	rrr	Request to read word	PRIV
rsb	71	-	-	Return from subroutine	IS25
rsblk	C1	F	rrr	Request to send block	PRIV
rstat	C1	4	rrr	Read status	PRIV
rstati	C1	4	rrr	Read status and clear device interrupt	PRIV
rstatr	C1	4	rrr	Read status and clear service request	PRIV
rswrd	C1	E	rrr	Request to send word	PRIV
rtb	D8	2	r	Return from breakpoint	3B20
rti	D8	0	-	Return from interrupt	3B20
rto	D8	1	r	Return from ost	3B20
rtt	DC	2	r	Return from interrupt non-standard	3B20
save	7A	-	i	Save	IS25
sbsmc	46	0	-	BIST and boundary scan	3B21
sbsmd	46	1	-	BIST and boundary scan	3B21
scc	DC	l	r	Set condition code	3B20
sds	D4	1	-	Set destination secondary	3B20
sendint	C1	9	r	Send interrupt	PRIV
setmpr	A3	-	r	Measurement pulse in ppr	3B20
sinov	C1	A	r	Send I/O inhibit override	PRIV

Table A2-3 — Mnemonics R Through Z (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
sioh	C0	6	rr	Start untimed I/O	PRIV
siow	C0	2	rr	Start untimed I/O	PRIV
smccp	C5	D	rr	Send mch cmd to off-line mch with parity	PRIV
smcdp	C5	E	rr	Send mch date to off-line mch with parity	PRIV
smchc	C5	6	r	Send mch cmd to off-line mch	PRIV
smchd	C5	7	r	Send mch data to off-line mch	PRIV
sminst	C5	8	rr	Send microinstruction to off-line mch	PRIV
smioh	C0	7	rr	Start untimed maint. I/O	PRIV
smiow	C0	3	rr	Start untimed maint. I/O	PRIV
sss	D4	0	-	Set source secondary	3B20
stat	C0	9	rr	Return channel status	PRIV
stmcbp	C5	C	rr	Store into mch buffer with parity	PRIV
stmchb	C5	3	r	Store into mch buffer	PRIV
stmchc	C5	2	r	Store into mch cmd reg	PRIV
stsm	65	8	gm	Store selected multiple registers	3B20
subb2	20	7	rr	Subtract (2 oper)	IS25
subb2	21	7	rm	Subtract (2 oper)	IS25
subb2	22	7	nr	Subtract (2 oper)	IS25
subb2	23	7	nm	Subtract (2 oper)	IS25
subb2	24	7	gr	Subtract (2 oper)	IS25
subb2	25	7	gm	Subtract (2 oper)	IS25
subb3	68	4	ggr	Subtract (3 oper)	IS25
subb3	68	C	ggm	Subtract (3 oper)	IS25
subh2	28	7	rr	Subtract (2 oper)	IS25
subh2	29	7	rm	Subtract (2 oper)	IS25
subh2	2A	7	nr	Subtract (2 oper)	IS25
subh2	2B	7	nm	Subtract (2 oper)	IS25
subh2	2C	7	gr	Subtract (2 oper)	IS25
subh2	2D	7	gm	Subtract (2 oper)	IS25
subh3	69	4	ggr	Subtract (3 oper)	IS25
subh3	69	C	ggm	Subtract (3 oper)	IS25
subw2	12	-	rr	Subtract (2 oper)	OPT

Table A2-3 — Mnemonics R Through Z (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
subw2	13	-	nr	Subtract (2 oper)	OPT
subw2	30	7	rr	Subtract (2 oper)	IS25
subw2	31	7	rm	Subtract (2 oper)	IS25
subw2	32	7	nr	Subtract (2 oper)	IS25
subw2	33	7	nm	Subtract (2 oper)	IS25
subw2	34	7	gr	Subtract (2 oper)	IS25
subw2	35	7	gm	Subtract (2 oper)	IS25
subw3	6A	4	ggr	Subtract (3 oper)	IS25
subw3	6A	C	ggm	Subtract (3 oper)	IS25
switch	BA	-	irm	Switch	IS25
switcht	BB	-	rm	Switch on case through switch table	IS25
swks	DB	8	-	Switch to kernel stack	PRIV
swps	DB	0	-	Switch to private stack	PRIV
sxl	D7	-	r	Set execution level	PRIV
tarbb	20	5	rr	Test and reset bits	3B20
tarbb	21	5	rm	Test and reset bits	3B20
tarbb	22	5	nr	Test and reset bits	3B20
tarbb	23	5	nm	Test and reset bits	3B20
tarbb	24	5	gr	Test and reset bits	3B20
tarbb	25	5	gm	Test and reset bits	3B20
tarbh	28	5	rr	Test and reset bits	3B20
tarbh	29	5	rm	Test and reset bits	3B20
tarbh	2A	5	nr	Test and reset bits	3B20
tarbh	2B	5	nm	Test and reset bits	3B20
tarbh	2C	5	gr	Test and reset bits	3B20
tarbh	2D	5	gm	Test and reset bits	3B20
tarbw	30	5	rr	Test and reset bits	3B20
tarbw	31	5	rm	Test and reset bits	3B20
tarbw	32	5	nr	Test and reset bits	3B20
tarbw	33	5	nm	Test and reset bits	3B20
tarbw	34	5	gr	Test and reset bits	3B20
tarbw	35	5	gm	Test and reset bits	3B20
tasbb	20	4	rr	Test and set bits	3B20
tasbb	21	4	rm	Test and set bits	3B20
tasbb	22	4	nr	Test and set bits	3B20
tasbb	23	4	nm	Test and set bits	3B20
tasbb	24	4	gr	Test and set bits	3B20

Table A2-3 — Mnemonics R Through Z (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
tasbb	25	4	gm	Test and set bits	3B20
tasbh	28	4	rr	Test and set bits	3B20
tasbh	29	4	rm	Test and set bits	3B20
tasbh	2A	4	nr	Test and set bits	3B20
tasbh	2B	4	nm	Test and set bits	3B20
tasbh	2C	4	gr	Test and set bits	3B20
tasbh	2D	4	gm	Test and set bits	3B20
tasbw	30	4	rr	Test and set bits	3B20
tasbw	31	4	rm	Test and set bits	3B20
tasbw	32	4	nr	Test and set bits	3B20
tasbw	33	4	nm	Test and set bits	3B20
tasbw	34	4	gr	Test and set bits	3B20
tasbw	35	4	gm	Test and set bits	3B20
tio	C0	A	rr	Test I/O complete	PRIV
ucrd	C4	0	r	Utility circuit read	3B20
ucwt	C4	8	r	Utility circuit write	3B20
udivw2	30	B	rr	Unsigned divide (2 oper)	IS25
udivw2	31	B	rm	Unsigned divide (2 oper)	IS25
udivw2	32	B	nr	Unsigned divide (2 oper)	IS25
udivw2	33	B	nm	Unsigned divide (2 oper)	IS25
udivw2	34	B	gr	Unsigned divide (2 oper)	IS25
udivw2	35	B	gm	Unsigned divide (2 oper)	IS25
udivw3	6C	3	ggr	Unsigned divide (3 oper)	IS25
udivw3	6C	B	ggm	Unsigned divide (3 oper)	IS25
umodw2	30	D	rr	Unsigned modulo (2 oper)	IS25
umodw2	31	D	rm	Unsigned modulo (2 oper)	IS25
umodw2	32	D	nr	Unsigned modulo (2 oper)	IS25
umodw2	33	D	nm	Unsigned modulo (2 oper)	IS25
umodw2	34	D	gr	Unsigned modulo (2 oper)	IS25
umodw2	35	D	gm	Unsigned modulo (2 oper)	IS25
umodw3	6C	5	ggr	Unsigned modulo (3 oper)	IS25
umodw3	6C	D	ggm	Unsigned modulo (3 oper)	IS25
umulw2	30	9	rr	Unsigned multiply (2 oper)	IS25
umulw2	31	9	rm	Unsigned multiply (2 oper)	IS25
umulw2	32	9	nr	Unsigned multiply (2 oper)	IS25
umulw2	33	9	nm	Unsigned multiply (2 oper)	IS25
umulw2	34	9	gr	Unsigned multiply (2 oper)	IS25
umulw2	35	9	gm	Unsigned multiply (2 oper)	IS25

Table A2-3 — Mnemonics R Through Z (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
umulw3	6C	1	ggr	Unsigned multiply (3 oper)	IS25
umulw3	6C	9	ggm	Unsigned multiply (3 oper)	IS25
utnop	A1	00	-	Utility nop (in sgs)	3B20
vcall	76	-	m	Vcall	3B20
vtop	A4	-	rr	Virtual to physical translation	3B20
wait	DC	3	-	Wait for interrupt	3B20
waitii	C6	2	r	Wait with interrupts ignored	PRIV
wbsmc	46	4	-	BIST and boundary scan	3B21
wbsmd	46	5	-	BIST and boundary scan	3B21
wcsdump	C6	1	rrr	Writable control store dump	PRIV
wcspump	C6	0	rrr	Writable control store pump	PRIV
wtblk	C1	3	rrr	Write block	PRIV
wtblki	C1	3	rrr	Write block and clear device interrupt	PRIV
wtblkr	C1	3	rrr	Write block and clear service request	PRIV
wtcmd	C1	5	rrr	Write command	PRIV
wtcmdi	C1	5	rrr	Write command and clear device interrupt	PRIV
wtcmdr	C1	5	rrr	Write command and clear service request	PRIV
wtistk	E1	8	r	Write interrupt stack	PRIV
wtmsk	C1	6	rrr	Write mask	PRIV
wtmskb	C1	6	rrr	Write mask and clear both	PRIV
wtmski	C1	6	rrr	Write mask and clear device interrupt	PRIV
wtmskr	C1	6	rrr	Write mask and clear service request	PRIV
wtpar	C1	8	rrrr	Write with parity	PRIV
wtpari	C1	8	rrrr	Write with parity and clear device interrupt	PRIV
wtparr	C1	8	rrrr	Write with parity and clear service request	PRIV
wtphy	D5	8	rr	Write with physical address	3B20
wtphyb	d5	a	-	Physical address - write byte	3B21

Table A2-3 — Mnemonics R Through Z (Contd)

Mnemonic	Opcode	Subcode	Syntax	Function	Type
wtphyh	d5	9	-	Physical address - write half-word	3B21
wtrtn	C1	7	rrr	Write return code	PRIV
wtsr	D1	-	rs	Write special register	PRIV
wtwrd	C1	1	rrr	Write word	PRIV
wtwrdi	C1	1	rrr	Write word and clear device interrupt	PRIV
wtwrdr	C1	1	rrr	Write word and clear service request	PRIV
xorb2	20	2	rr	Exclusive or (2 oper)	IS25
xorb2	21	2	rm	Exclusive or (2 oper)	IS25
xorb2	22	2	nr	Exclusive or (2 oper)	IS25
xorb2	23	2	nm	Exclusive or (2 oper)	IS25
xorb2	24	2	gr	Exclusive or (2 oper)	IS25
xorb2	25	2	gm	Exclusive or (2 oper)	IS25
xorb3	68	2	ggr	Exclusive or (3 oper)	IS25
xorb3	68	A	ggm	Exclusive or (3 oper)	IS25
xorh2	28	2	rr	Exclusive or (2 oper)	IS25
xorh2	29	2	rm	Exclusive or (2 oper)	IS25
xorh2	2A	2	nr	Exclusive or (2 oper)	IS25
xorh2	2B	2	nm	Exclusive or (2 oper)	IS25
xorh2	2C	2	gr	Exclusive or (2 oper)	IS25
xorh2	2D	2	gm	Exclusive or (2 oper)	IS25
xorh3	69	2	ggr	Exclusive or (3 oper)	IS25
xorh3	69	A	ggm	Exclusive or (3 oper)	IS25
xorw2	1E	-	rr	Exclusive or (2 oper)	OPT
xorw2	30	2	rr	Exclusive or (2 oper)	IS25
xorw2	31	2	rm	Exclusive or (2 oper)	IS25
xorw2	32	2	nr	Exclusive or (2 oper)	IS25
xorw2	33	2	nm	Exclusive or (2 oper)	IS25
xorw2	34	2	gr	Exclusive or (2 oper)	IS25
xorw2	35	2	gm	Exclusive or (2 oper)	IS25
xorw3	6A	2	ggr	Exclusive or (3 oper)	IS25
xorw3	6A	A	ggm	Exclusive or (3 oper)	IS25
zeroblock	E1	-	rr	Zeroblock	IS25

A3. *Motorola*[1] MC68000 PROCESSOR FAMILY INSTRUCTION SET

This appendix provides a summary of the *Motorola* MC68000 processor family instruction set and a chart that lists the opcodes and their meanings.

Motorola MC68000 Processor Family Operation Code Map	
0000	Bit Manipulation/MOVEP/Immediate
0001	Move Byte
0010	Move Long
0011	Move Word
0100	Miscellaneous
0101	ADDQ/SUBQ/Scc/DBcc
0110	Bcc/BSR
0111	MOVEQ
1000	OR/DIV/SBCD
1001	SUB/SUBX
1010	(Unassigned)
1011	CMP/EOR
1100	AND/MUL/ABCD/EXG
1101	ADD/ADDX
1110	Shift/Rotate
1111	Coprocessor Interface MC68040 and CPU32 Extensions Type 000 General (for example: FADD, FABS, FMOVE) 001 FDBcc, FScc, FTRAPcc (not supported) 010 FBcc.W 011 FBcc.L (marginally supported) 100 FSAVE 101 FRESTORE 110 Undefined, reserved, not supported 111 Undefined, reserved, not supported

1. Registered trademark of Motorola Inc.

A3.1 ASSEMBLY INSTRUCTIONS A THROUGH E

Instruction	*Motorola*	MC680xx Processor
abcd %dy,%dx	ABCD	00/12/20/30/40/60
abcd –(%ay),–(%ax)		

Description:

Adds the source operand to the destination operand (along with the extend bit), and stores the result in the destination location. The addition is performed using binary-coded decimal arithmetic. The operands can be addressed in two ways:

Data register to data register - the operands are contained in the data registers specified in the instruction.

Memory to memory - the operands are addressed with the predecrement addressing mode using the address registers specified in the instruction.

Operands: `%dy` specifies that the source register is a data register.

`%dx` specifies that the destination register is a data register.

`%ay` used to derive the memory address of the source operand using the predecrement addressing mode.

`%ax` used to derive the memory address of the destination operand using the predecrement addressing mode.

Instruction	*Motorola*	MC680xx Processor
addb $expr,<da>	ADDI.B	00/12/20/30/40/60
addb $expr,<a>	ADDQ.B	
addb %dn,<am>	ADD.B	
addb <d>,%dn	ADD.B	
add $expr,<da>	ADDI.W	
add $expr,<a>	ADDQ.W	
add %dn,<am>	ADD.W	
add <ea>,%an	ADDA.W	
add $<1-8>,%an	ADDQ.W	
add <ea>,%dn	ADD.W	
addl $expr,<da>	ADDI.L	
addl $<1-8>,<a>	ADDQ.L	
addl %dn,<a>	ADD.L	
addl <ea>,%an	ADDA.L	
addl $<1-8>,%an	ADDQ.L	
addl <ea>,%dn	ADD.L	

Description:

Adds the source operand to the destination operand, and stores the result in the destination location.

Operands:

<a>	Alterable addressing modes.
<am>	Alterable memory addressing modes.
<d>	Data addressing modes.
<da>	Data alterable addressing modes.
<ea>	All addressing modes.
$<1-8>	Immediate data in the range of 1 through 8.
%dn	Specifies any of the eight data registers.
%an	Specifies any of the eight address registers.

Programming Note:

When adding a constant to an address register, the `lea` instruction is better than `add` if the constant is in the range of –32768 through 0 or 9 through 32767. For constants of 1 through 8 the `add` instruction generates an `ADDQ` instruction which is better than an `lea`.

When the assembly instruction `add $expr,%dn` is given to the assembler, the `ADDI` encoding is used instead of the `ADD` encoding.

Instruction	*Motorola*	MC680xx Processor
addxb %dy,%dx	ADDX.B	00/12/20/30/40/60
addxb –(%ay),–(%ax)	ADDX.B	
addx %dy,%dx	ADDX.W	
addx –(%ay),–(%ax)	ADDX.W	
addxl %dy,%dx	ADDX.L	
addxl –(%ay),–(%ax)	ADDX.L	

Description:

Adds the source operand to the destination operand along with the extend bit, and stores the result in the destination location.

Data register to data register - operands are contained in data registers specified in the instruction.

Memory to memory - operands are contained in memory and addressed with the predecrement addressing mode using the address registers specified in the instruction.

Operands: `%dy` specifies the source register.

`%dx` specifies the destination register.

`%ax` used to derive the memory address of the destination operand using the predecrement addressing mode.

`%ay` used to derive the memory address of the source operand using the predecrement addressing mode.

Instruction	*Motorola*	MC680xx Processor
andb $expr,%ccr	ANDI to CCR	00/12/20/30/40/60
andb $expr,<da>	ANDI.B	
andb %dn,<am>	AND.B	

Instruction	*Motorola*	MC680xx Processor
andb <d>,%dn	AND.B	
and $expr,<da>	ANDI.W	
and %dn,<am>	AND.W	
and <d>,%dn	AND.W	
and $expr,%sr	ANDI to SR	
andl $expr,<da>	ANDI.L	
andl %dn,<am>	AND.L	
andl <d>,%dn	AND.L	

Description:

Bitwise ANDs the source operand to the destination operand, and stores the result in the destination location.

Operands:

`<am>`	Alterable Memory addressing modes.
`<d>`	Data addressing modes.
`<da>`	Data Alterable addressing modes.
`%ccr`	Condition Code Register.
`%dn`	Data register.
`%sr`	Status Register.
`$expr`	Constant expression.

Programming Note:

When the assembly instruction `and $expr,%dn` is given to the assembler, the `ANDI` encoding is used instead of the `AND` encoding.

Instruction	*Motorola*	MC680xx Processor
asrb %dx,%dy	ASR.B	00/12/20/30/40/60
asrb $expr,%dy	ASR.B	
asr %dx,%dy	ASR.W	
asr $expr,%dy	ASR.W	
asr <ea>	ASR.W	
asrl %dx,%dy	ASR.L	
asrl $expr,%dy	ASR.L	
aslb %dx,%dy	ASL.B	00/12/20/30/40/60
aslb $expr,%dy	ASL.B	
asl %dx,%dy	ASL.W	
asl $expr,%dy	ASL.W	
asl <ea>	ASL.W	
asll %dx,%dy	ASL.L	
asll $expr,%dy	ASL.L	

Description:

Arithmetically shifts the bits of the operand in the direction specified. The carry bit receives the last bit shifted out of the operand. The shift count for the shifting of a register can be specified in two ways:

Immediate - the shift count is specified in the instruction (shift range 1 through 8).

Register - the shift count is contained in a data register specified in the instruction.

The content of memory can be shifted 1 bit only, where the operand size is restricted to a word.

ASL - operand is shifted to left. The number of positions shifted is the shift count. Bits shifted out of the high order bit go to both the carry and the extend bits; zeros are shifted into the low order bit. The overflow bit indicates if any sign changes occur during the shift.

ASR - operand is shifted to right. The number of positions shifted is the shift count. Bits shifted out of the low order bit go to both the carry and extend bits; the sign bit is replicated into the higher order bit.

Operands: `<ea>` specifies the operand to be shifted. Only alterable memory addressing modes are allowed.

`%dy` specifies the data register whose content is to be shifted.

`$expr` or `%dx` specifies shift count or register where shift count is located.

Instruction	*Motorola*	MC680xx Processor
bcc expr	Bcc	00/12/20/30/40/60

Description:

If the specified condition is met, the program execution continues at location program pounter (PC) plus displacement. Displacement is a 2's complement integer which counts the relative distance in bytes. The PC value is the current location plus two.

The *cc* part of this instruction represents one of the following conditions:

CC - Carry clear (unsigned greater than or equal)
CS - Carry set (unsigned less than)
EQ - Equal
GE - Greater than or equal
GT - Greater than
HI - High (unsigned greater than)
LE - Less than or equal
LS - Low or same (unsigned less than or equal)
LT - Less than
MI - Minus
NE - Not equal

PL - Plus
VC - No overflow
VS - Overflow

Operands: `expr` specifies the program symbolic destination when the branch is taken.

Instruction	*Motorola*	MC680xx Processor
bchg %dn,<ea>	BCHG	00/12/20/30/40/60
bchg $expr,<ea>		

Description:

Tests a bit in the destination operand, and reflects the state of the specified bit in the Z condition code. After the test, the state of the specified bit is changed in the destination. If a data register is the destination, the bit numbering is modulo 32 which allows bit manipulation on all bits in a data register. If a memory location is the destination, a byte is read from the location, the bit operation is performed using the bit number modulo 8. The byte is written back to the location. The bit number for this operation is specified in two ways:

Immediate - the bit number is specified in the second word of the instruction.

Register - the bit number is contained in a data register specified in the instruction.

Operands: `<ea>` specifies the destination location. Only data alterable addressing modes are allowed.

`%dn` specifies the data register whose content is the bit number.

`$expr` specifies the bit number.

Instruction	*Motorola*	MC680xx Processor
bclr %dn,<ea>	BCLR	00/12/20/30/40/60
bclr $expr,<ea>		

Description:

Tests a bit in the destination operand, and reflects the state of the specified bit in the Z condition code. After the test, the specified bit is cleared in the destination. If a data register is the destination, the bit numbering is modulo 32 which allows bit manipulation on all bits in a data register. If a memory location is the destination, a byte is read from the location, the bit operation is performed using the bit number modulo 8. The byte is written back to the location. The bit number for this operation can be specified in two ways:

Immediate - the bit number is specified in the second word of the instruction.

Register - the bit number is contained in a data register specified in the instruction.

Operands: `<ea>` specifies the destination location. Only data alterable addressing modes are allowed.

`%dn` specifies the data register whose content is the bit number.

`$expr` specifies the bit number.

Instruction	*Motorola*	MC680xx Processor
bfchg <ea>{offset:width}	BFCHG	20/30/40/60

Description:

Complements the bitfield at the destination location. The N and Z condition codes are set based upon the initial value of the bitfield.

Operands: `<ea>` specifies the destination location. Only data register direct or alterable control addressing modes are allowed.

`offset` is the bitfield offset. Offsets can lie between -2^{31} and 2^{31-1} and indicate the position of the leftmost bit of the bitfield relative to the most significant bit of `<ea>`. Either an immediate value or a data register containing the offset may be used.

`width` is the bitfield width which can lie between 0 and 31 inclusive. A value of 0 indicates an actual width of 32 bits. Either an immediate value or a data register containing the width may be used.

Instruction	*Motorola*	MC680xx Processor
bfclr <ea>{offset:width}	BFCLR	20/30/40/60

Description:

Clears the bitfield at the destination location. The N and Z condition codes are set based upon the initial value of the bitfield.

Operands: `<ea>` specifies the destination location. Only data register direct or alterable control addressing modes are allowed.

`offset` is the bitfield offset. Offsets can lie between -2^{31} and 2^{31-1} and indicate the position of the leftmost bit of the bitfield relative to the most significant bit of `<ea>`. Either an immediate value or a data register containing the offset may be used.

`width` is the bitfield width which can lie between 1 and 32 inclusive. A value of 0 indicates an actual width of 32 bits. Either an immediate value or a data register containing the width may be used.

Instruction	*Motorola*	MC680xx Processor
bfexts <ea>{offset:width},%dn	BFEXTS	20/30/40/60

Description:

Extracts the bitfield at the indicated source location, sign-extends it to 32 bits, and places the results in the destination data register. The N and Z condition codes are set based upon the bitfield value.

Operands: `<ea>` specifies the source location. Only data register direct or control addressing modes are allowed.

`offset` is the bitfield offset. Offsets can lie between -2^{31} and 2^{31-1} and indicate the position of the leftmost bit of the bitfield relative to the most

significant bit of <ea>. Either an immediate value or a data register containing the offset may be used.

width is the bitfield width which can lie between 1 and 32 inclusive. A value of 0 indicates an actual width of 32 bits. Either an immediate value or a data register containing the width may be used.

%dn is the destination data register.

Instruction	*Motorola*	MC680xx Processor
bfextu <ea>{offset:width},%dn	BFEXTU	20/30/40/60

Description:

Extracts the bitfield at the indicated source location, zero-extends it to 32 bits, and places the results in the destination data register. The N and Z condition codes are set based upon the bitfield value.

Operands: <ea> specifies the source location. Only data register direct or control addressing modes are allowed.

offset is the bitfield offset. Offsets can lie between -2^{31} and 2^{31-1} and indicate the position of the leftmost bit of the bitfield relative to the most significant bit of <ea>. Either an immediate value or a data register containing the offset may be used.

width is the bitfield width which can lie between 1 and 32 inclusive. A value of 0 indicates an actual width of 32 bits. Either an immediate value or a data register containing the width may be used.

%dn is the destination data register.

Instruction	*Motorola*	MC680xx Processor
bfffo <ea>{offset:width},%dn	BFFFO	20/30/40/60

Description:

Searches the source location bitfield for the first (i.e., most significant) bit that is set. If one is found, its offset is placed in the destination data register. Otherwise the sum of the offset and width are stored there. The N and Z condition codes are set based upon the bitfield value.

Operands: <ea> specifies the source location. Only data register direct or control addressing modes are allowed.

offset is the bitfield offset. Offsets can lie between -2^{31} and 2^{31-1} and indicate the position of the leftmost bit of the bitfield relative to the most significant bit of <ea>. Either an immediate value or a data register containing the offset may be used.

width is the bitfield width which can lie between 1 and 32 inclusive. A value of 0 indicates an actual width of 32 bits. Either an immediate value or a data register containing the width may be used.

%dn is the destination data register.

Instruction	*Motorola*	MC680xx Processor
bfins %dn,<ea>{offset:width}	BFINS	20/30/40/60

Description:

Inserts the low-order bits of the source data register into the bitfield at the destination location. The N and Z condition codes are set based upon the bitfield value.

Operands: <ea> specifies the destination location. Only data register direct or alterable control addressing modes are allowed.

offset is the bitfield offset. Offsets can lie between -2^{31} and 2^{31-1} and indicate the position of the leftmost bit of the bitfield relative to the most significant bit of <ea>. Either an immediate value or a data register containing the offset may be used.

width is the bitfield width which can lie between 1 and 32 inclusive. A value of 0 indicates an actual width of 32 bits. Either an immediate value or a data register containing the width may be used.

%dn is the source data register.

Instruction	*Motorola*	MC680xx Processor
bfset <ea>{offset:width}	BFSET	20/30/40/60

Description:

Sets all bits of the bitfield at the destination location. The N and Z condition codes are set based upon the initial value of the bitfield.

Operands: <ea> specifies the destination location. Only data register direct or alterable control addressing modes are allowed.

offset is the bitfield offset. Offsets can lie between -2^{31} and 2^{31-1} and indicate the position of the leftmost bit of the bitfield relative to the most significant bit of <ea>. Either an immediate value or a data register containing the offset may be used.

width is the bitfield width which can lie between 1 and 32 inclusive. A value of 0 indicates an actual width of 32 bits. Either an immediate value or a data register containing the width may be used.

Instruction	*Motorola*	MC680xx Processor
bftst <ea>{offset:width}	BFTST	20/30/40/60

Description:

Tests the contents of the bitfield at the destination location and sets the condition codes accordingly. The N and Z condition codes are set based upon the bitfield value. The V and C condition codes are cleared and X is left unchanged.

Operands: <ea> specifies the destination location. Only data register direct or control addressing modes are allowed.

offset is the bitfield offset. Offsets can lie between -2^{31} and 2^{31-1} and indicate the position of the leftmost bit of the bitfield relative to the most significant bit of <ea>. Either an immediate value or a data register containing the offset may be used.

`width` is the bitfield width which can lie between 1 and 32 inclusive. A value of 0 indicates an actual width of 32 bits. Either an immediate value or a data register containing the width may be used.

Instruction	*Motorola*	MC680xx Processor
bkpt $expr	BKPT	12/20/30/40

Description:

Executes a breakpoint acknowledge cycle. For the *Motorola* MC68020 and MC68030 processors, the lower three bits of the immediate value are placed on address lines A2 to A4 and zeros are placed on lines A0 and A1. For the *Motorola* MC68040 processor, an illegal instruction exception is generated. For more information see the appropriate processor manual.

Operands: `$expr` is the breakpoint number. For the *Motorola* MC68012 and MC68040 processors, a debug monitor can look at this number to determine the type of the breakpoint. For the *Motorola* MC68020 and MC68030 processors, this number may be used to address an external device which can place a replacement instruction on the data bus.

Instruction	*Motorola*	MC680xx Processor
bra expr	BRA	00/12/20/30/40/60

Description:

Program execution continues at location (program counter) plus displacement. Displacement is a two's complement integer that counts the relative distance in bytes. The value in the PC is the current location (instruction) plus two.

Operands: `expr` specifies the program symbolic destination when the branch is taken.

Instruction	*Motorola*	MC680xx Processor
bset %dn,<ea> bset $expr,<ea>	BSET	00/12/20/30/40/60

Description:

Tests a bit in the destination operand and reflects the state of the specified bit in the Z condition code. After the test, the specified bit is set in the destination. If the destination is a data register, the bit numbering is modulo 32, which allows bit manipulation on all bits in a data register. If the destination is a memory location, the bit operation is performed using the bit number modulo 8, and the byte is written back to the location. The bit number can be specified in two ways:

Immediate - the bit number is specified in the second word of the instruction.

Register - the bit number is contained in a data register specified in the instruction.

Operands: `<ea>` specifies the destination location. Only data alterable addressing modes are allowed.

`%dn` specifies the data register whose content is the bit number.

`$expr` specifies the bit number.

Instruction	*Motorola*	MC680xx Processor
bsr expr	BSR	00/12/20/30/40/60

Description:

Pushes the address of the instruction immediately following `bsr` instruction onto the stack. Program execution continues at the location (program counter) plus displacement. Displacement is a two's complement integer that counts the relative distance in bytes. The value in the program counter is the current location plus two.

Operands: `expr` specifies the program symbolic destination when the branch is taken.

Instruction	*Motorola*	MC680xx Processor
btst %dn,<ea>	BTST	00/12/20/30/40/60
btst $expr,<ea>		

Description:

Tests a bit in the destination operand and reflects the state of the specified bit in the Z condition code. If the destination is a data register, the bit numbering is modulo 32, which allows bit manipulation on all bits in a data register. If the destination is a memory location, a byte is read from that location. The bit operation is performed using the bit number modulo 8. The byte is written back to the location. The bit number for this operation can be specified in two ways:

Immediate - the bit number is specified in the second word of the instruction.

Register - the bit number is contained in a data register specified in the instruction.

Operands: `<ea>` specifies the destination location. Only data addressing modes are allowed, with the exception of the immediate addressing mode when the bit number is specified by immediate data.

`%dn` specifies a data register whose content is the bit number.

`$expr` specifies the bit number.

Instruction	*Motorola*	MC680xx Processor
casb %dc,%du,<ea>	CAS.B	20/30/40
cas %dc,%du,<ea>	CAS.W	
casl %dc,%du,<ea>	CAS.L	

Description:

Compares the effective address operand to the compare register (`%dc`). If equal, the contents of the update register (`%du`) are copied to the effective address. Otherwise, the contents of the effective address are copied to `%dc`.

Operands: `%dc` specifies the data register to use for the comparison and to update if the comparison fails.

%du specifies the data register with the update value.

<ea> specifies the operand to compare to %dc and update if the comparison succeeds. Only memory alterable modes are allowed.

Instruction	*Motorola*	MC680xx Processor
cas2 %dc1:%dc2,%du1:%du2,(%r1):(%r2)	CAS2.W	20/30/40/60*
cas2l %dc1:%dc2,%du1:%du2,(%r1):(%r2)	CAS2.L	

Description:

Compares the first memory operand (contents at address in %r1) to the first compare register (%dc1) and the second memory operand (contents at address in %r2) to the second compare register (%dc2). If both comparisons succeed, the contents of the update registers (%du1, %du2) are written to the respective memory operand addresses. If either comparison fails, the contents at the memory operand addresses are written to the respective comparison registers.

Operands: %dc1, %dc2 specify data registers to use for the comparisons and to update if either comparison fails.

%du1, %du2 specify the data registers with the update values.

%r1, %r2 specify address or data registers containing the addresses of the memory operands to compare to %dc1 and %dc2 and to update if both comparisons succeed.

Programming Note:

This instruction is not implemented in hardware on the 060. Emulation support is required.

Instruction	*Motorola*	MC680xx Processor
chk <ea>,%dn	CHK.W	00/12/20/30/40/60
chkl <ea>,%dn	CHK.L	20/30/40/60

Description:

Examines the contents of the data register and compares it to the upper bound specified by the effective address. The upper bound is a two's complement integer. If the register value is less than zero or greater than the upper bound contained in the operand word, then the processor initiates exception processing. The vector number 6 is generated to reference the chk instruction exception vector.

Operands: <ea> specifies the upper bound operand word. Only data addressing modes are allowed.

%dn specifies the data register whose contents is checked.

Instruction	*Motorola*	MC680xx Processor
chk2b <ea>,%rn	CHK.B	20/30/40
chk2 <ea>,%rn	CHK.W	
chk2l <ea>,%rn	CHK.L	

Description:

Examines the contents of the register %rn and compares it to the lower

and upper bounds at the location specified by the effective address. The bounds must be two's complement integers with the lower bound being followed by the upper. If the register value is less than the lower bound or greater than the upper then the processor initiates exception processing. The vector number 6 is generated to reference the `chk` instruction exception vector.

Operands: <ea> specifies the address of the bounds. Only control addressing modes are allowed.

`%rn` specifies the register whose contents is checked. If an address register is used and the size of the instruction is byte or word the bounds are sign-extended to 32 bits before the comparisons and are compared against all 32 bits of the register.

Instruction	*Motorola*	MC680xx Processor
cinva [%bc|%dc|%ic|%nc]	CINVA	40/60
cinvl [%bc|%dc|%ic|%nc],(%an)	CINVL	40/60
cinvp [%bc|%dc|%ic|%nc],(%an)	CINVP	40/60

Description:

Invalidates the selected cache lines. Any dirty data is lost. The `cpush` instruction should be used instead when dirty data may not have been written.

The `cinvl` instruction invalidates the cache line matching the physical address contained in `%an`. The `cinvp` instruction invalidates all lines matching the physical page specified by `%an`. The `cinva` instruction invalidates all cache entries.

Operands: `%bc`, `%dc`, `%ic`, `%nc` are cache selectors indicating both caches, the data cache, the instruction cache, or neither cache (i.e., a no-op), respectively.

`%an` contains the physical address associated with the line or lines to be invalidated.

Instruction	*Motorola*	MC680xx Processor
clrb <ea>	CLR.B	00/12/20/30/40/60
clr <ea>	CLR.W	
clrl <ea>	CLR.L	

Description:

Clears destination to all zero bits.

Operands: <ea> specifies the destination location. Only data alterable addresses are allowed.

Programming Note:

The `clr` instruction does a read/write, not just a write.

The easiest way to clear an address register is to subtract it from itself.

Instruction	*Motorola*	MC680xx Processor
cmpb $expr,<ea>	CMPI.B	00/12/20/30/40/60
cmpb <d>,%dn	CMP.B	

Instruction	*Motorola*	MC680xx Processor
cmp $expr,<da>	CMPI.W	
cmp <ea>,%an	CMPA.W	
cmp <ea>,%dn	CMP.W	
cmpl $expr,<da>	CMPI.L	
cmpl <ea>,%an	CMPA.L	
cmpl <ea>,%dn	CMP.L	

Description:

Subtracts the source operand from the destination operand and sets the condition codes according to the results. The destination location is unchanged.

Operands:

`<d>`	Data addressing modes.
`<da>`	Data alterable addressing modes.
`<ea>`	All addressing modes.
`%an`	Address register.
`%dn`	Data register.
`$expr`	Constant expression.

Programming Note:

To test an address register for zero (NULL), use `cmp` instead of `cmpl` since the zero is sign extended.

When the assembly instruction `cmp $expr,%dn` is given to the assembler, the `CMPI` encoding is used instead of the `CMP` encoding.

Instruction	*Motorola*	MC680xx Processor
cmpmb (%ay)+,(%ax)+	CMPM.B	00/12/20/30/40/60
cmpm (%ay)+,(%ax)+	CMPM.W	
cmpml (%ay)+,(%ax)+	CMPM.L	

Description:

Subtracts the source operand from the destination operand, and sets the condition codes according to the results. The destination location is not changed. The operands are always addressed with the post-increment addressing mode using the address registers specified in the instruction.

Operands: `%ay` specifies an address register for the post-increment addressing mode.

`%ax` specifies an address register for the post-increment addressing mode.

Instruction	*Motorola*	MC680xx Processor
cmp2b <ea>,%rn	CMP2.B	20/30/40/60*
cmp2 <ea>,%rn	CMP2.W	
cmp2l <ea>,%rn	CMP2.L	

Description:

Examines the contents of the register %rn and compares it to the lower and upper bounds at the location specified by the effective address. The bounds must be two's complement integers with the lower bound being followed by the upper. If the register value is less than the lower bound or greater than the upper then the C condition code is set. If %rn is equal to either bound the Z condition code is set.

Operands: <ea> specifies the address of the bounds. Only control addressing modes are allowed.

%rn specifies the register whose contents is checked. If an address register is used and the size of the instruction is byte or word the bounds are sign-extended to 32 bits before the comparisons and are compared against all 32 bits of the register.

Programming Note:

This instruction is not implemented in hardware on the 060. Emulation support is required.

Instruction	*Motorola*	MC680xx Processor
cpusha [%bc|%dc|%ic|%nc]	CPUSHA	40/60
cpushl [%bc|%dc|%ic|%nc],(%an)	CPUSHL	40/60
cpushp [%bc|%dc|%ic|%nc],(%an)	CPUSHP	40/60

Description:

Pushes (i.e., writes) and invalidates the selected cache lines. The cpushl instruction pushes and invalidates the cache line matching the physical address contained in %an . The cpushp instruction pushes and invalidates all lines matching the physical page specified by %an. The cpusha instruction pushes and invalidates all cache entries.

Operands: %bc, %dc, %ic, %nc are cache selectors indicating both caches, the data cache, the instruction cache, or neither cache (i.e., a no-op), respectively.

%an contains the physical address associated with the line or lines to be invalidated.

Instruction	*Motorola*	MC680xx Processor
dbcc %dn,expr	DBcc	00/12/20/30/40/60

Description:

This instruction is a looping primitive of three parameters: condition, data register, and displacement. Tests the condition to determine if the termination condition is not true; if it is true, the low order 16 bits of the counter data register are decremented by one. If the result is –1, the counter is exhausted and the execution continues with the next instruction. If the result is not equal to –1, execution continues at the location indicated by the current value of the program counter, plus the sign-extended 16-bit displacement. The value in PC is the address of the displacement word.

The *cc* part of this instruction represents one of the following conditions:

CC - Carry clear (unsigned greater than or equal)

CS - Carry set (unsigned less than)
EQ - Equal
F - Always false
GE - Greater than or equal
GT - Greater than
HI - High (unsigned greater than)
LE - Less than or equal
LS - Low or same (unsigned less than or equal)
LT - Less than
MI - Minus
NE - Not equal
PL - Plus
T - Always true
VC - No overflow
VS - Overflow

Operands: `%dn` specifies the data register which is the counter.

`expr` specifies the program symbolic destination when the branch is taken.

Instruction	*Motorola*	MC680xx Processor
divs <ea>,%dn	DIVS.W	00/12/20/30/40/60

Description:

Divides the destination by the source and stores the result in the destination. The destination operand is long and the source operand is a word. The result is a 32-bit result where:

1. The quotient is in the lower word (least significant 16 bits).
2. The remainder is in the upper word (most significant 16 bits).

Operands: `<ea>` specifies the source operand. Only the data addressing modes are allowed.

`%dn` specifies any of the eight data registers.

Instruction	*Motorola*	MC680xx Processor
divsl <ea>,%dq	DIVS.L	20/30/40/60*

Description:

Divides the destination by the source and stores a long quotient in `%dq`. Both the source and destination are long.

Operands: `<ea>` specifies the source operand. Only data addressing modes are allowed.

`%dq` specifies any of the eight data registers.

Programming Note:

This instruction is not implemented in hardware on the 060. Emulation support is required.

Instruction	*Motorola*	MC680xx Processor
divsl <ea>,%dr:%dq	DIVS.L	20/30/40

Description:

Divides the destination by the source and stores a long quotient in `%dq` and a long remainder in `%dr`. The source is a long operand and the destination is a quad word operand the lower 32 bits of which are stored in `%dq` and the upper in `%dr`.

Operands: `<ea>` specifies the source operand. Only data addressing modes allowed.

`%dr` and `%dq` specify any of eight data registers. They cannot be the same register.

Instruction	*Motorola*	MC680xx Processor
divsll <ea>,%dr:%dq	DIVSL.L	20/30/40

Description:

Divides the destination by the source and stores a long quotient in `%dq` and a long remainder in `%dr`. Both the source and destination (taken from `%dq`) are long.

Operands: `<ea>` specifies the source operand. Only data addressing modes allowed.

`%dr` and `%dq` specify any of eight data registers. They cannot be the same register.

Note: The sign of the remainder is always the same as the dividend, unless the remainder is equal to zero. There are two special conditions:

1. Division by zero causes a *trap*.
2. Overflow can be detected and set before completion of the instruction. If overflow is detected, the condition is flagged, but the operands are unaffected. Overflow occurs in the word form of the instruction if the quotient is larger than a word and in the long form if the quotient is larger than a long.

Instruction	*Motorola*	MC680xx Processor
divu <ea>,%dn	DIVU.W	00/12/20/30/40/60

Description:

Divides the destination by the source and stores the result in the destination. The destination is a long; the source is a word. The operation is performed using unsigned arithmetic. The result is a 32-bit result where:

1. The quotient is in the lower word (least significant 16 bits).
2. The remainder is in the upper word (most significant 16 bits).

Operands: `<ea>` specifies the source operand. Only data addressing modes are allowed.

`%dn` specifies any of the eight data registers.

Instruction	*Motorola*	MC680xx Processor
divul <ea>,%dq	DIVU.L	20/30/40/60*

Description:

Divides the destination by the source and stores a long quotient in %dq. Both the source and destination are long. The operation is performed using unsigned arithmetic.

Operands: <ea> specifies the source operand. Only data addressing modes allowed.

%dq specifies any of eight data registers.

Programming Note:

This instruction is not implemented in hardware on the 060. Emulation support is required.

Instruction	*Motorola*	MC680xx Processor
divul <ea>,%dr:%dq	DIVU.L	20/30/40

Description:

Divides the destination by the source and stores a long quotient in %dq and a long remainder in %dr. The source is a long operand and the destination is quad word operand the lower 32 bits of which are stored in %dq and the upper in %dr. The operation is performed using unsigned arithmetic.

Operands: <ea> specifies the source operand. Only data addressing modes allowed.

%dr and %dq specify any of eight data registers. They cannot be the same register.

Instruction	*Motorola*	MC680xx Processor
divull <ea>,%dr:%dq	DIVUL.L	20/30/40

Description:

Divides the destination by the source and stores a long quotient in %dq and a long remainder in %dr. Both the source and destination (taken from %dq) are long. The operation is performed using unsigned arithmetic.

Operands: <ea> specifies the source operand. Only data addressing modes allowed.

%dr and %dq specify any of eight data registers.

Note:

1. Division by zero causes a *trap*.
2. Overflow can be detected and set before completion of the instruction. If overflow is detected, the condition is flagged, but the operands are unaffected. Overflow occurs in the word form of the instruction if the quotient is larger than a word and in the long form if the quotient is larger than a long.

Instruction	*Motorola*	MC680xx Processor
eorb $expr,%ccr	EORI to CCR	00/12/20/30/40/60

Instruction	*Motorola*	MC680xx Processor
eorb $expr,<da>	EORI.B	
eorb %dn,<d>	EOR.B	
eor $expr,<da>	EORI.W	
eor %dn,<d>	EOR.W	
eor $expr,%sr	EORI to SR	
eorl $expr,<da>	EORI.L	
eorl %dn,<d>	EOR.L	

Description:

Exclusive ORs the source operand to the destination operand and stores the result in the destination location.

Operands:

<d>	Data addressing modes.
<da>	Data alterable addressing modes.
%ccr	Condition code register.
%dn	Data register.
%sr	Status register.
$expr	Constant expression.

Programming Note:

There is no `eor <ea>,%dn` form to match what is available for the `and` and `or` instructions.

Instruction	*Motorola*	MC680xx Processor
exg %rx,%ry	EXG	00/12/20/30/40/60

Description:

Exchanges the contents of two registers (always a 32-bit exchange). The exchange works in three modes: data registers, address register, and a data and an address register.

Operands: `%rx` specifies either a data or an address register depending on the mode.

`%ry` specifies either a data or an address register depending on the mode.

Instruction	*Motorola*	MC680xx Processor
extbl %dn	EXTB.L	20/30/40/60
ext %dn	EXT.W	00/12/20/30/40/60
extl %dn	EXT.L	

Description:

Extends the sign bit of a data register from a byte to a long, byte to a word, or from a word to a long operand depending on the size selected. In the byte to long operation, bit 7 of the designated data register is copied to bits 31:8 of the data register. In the byte to word operation, bit

7 of the designated data register is copied to bits 15:8 of the data register. In the long operation, bit 15 of the designated data register is copied to bits 31:16 of the data register.

Operands: `%dn` specifies the data register whose content is to be sign-extended.

Programming Note:
Word size instructions where the destination is an address register always sign extend the source operand before doing the operation. Sometimes an `ext` instruction can be avoided by assigning a variable to an address register.

A3.2 ASSEMBLY INSTRUCTIONS F THROUGH L

Instruction	*Motorola*	MC680xx Processor
fabs <d>,%fpn	FABS.W	40/60
fabsb <d>,%fpn	FABS.B	40/60
fabsw <d>,%fpn	FABS.W	40/60
fabsl <d>,%fpn	FABS.L	40/60
fabss <d>,%fpn	FABS.S	40/60
fabsd <d>,%fpn	FABS.D	40/60
fabsx <d>,%fpn	FABS.X	40/60
fabsx %fpm,%fpn	FABS.X	40/60
fabsx %fpn	FABS.X	40/60

Description:
Converts source operand to extended precision (if necessary) and stores the absolute value of that number in the destination floating point data register.

Operands:

`<d>`	Data addressing modes.
`%fpm`	Source register. Specifies any of the eight floating point data registers.
`%fpn`	Destination register. Specifies any of the eight floating point data registers. (Also source register if single register syntax is used).

Programming Note:
If a data direct register `%dn` is the data addressing mode in the instruction, mnemonics `fabsd` and `fabsx` are invalid.

Instruction	*Motorola*	MC680xx Processor
fadd <d>,%fpn	FADD.W	40/60
faddb <d>,%fpn	FADD.B	40/60
faddw <d>,%fpn	FADD.W	40/60
faddl <d>,%fpn	FADD.L	40/60
fadds <d>,%fpn	FADD.S	40/60
faddd <d>,%fpn	FADD.D	40/60
faddx <d>,%fpn	FADD.X	40/60

Instruction	*Motorola*	MC680xx Processor
faddx %fpm,%fpn	FADD.X	40/60

Description:

Converts source operand to extended precision (if necessary), adds that number to the number in the destination floating point data register, and stores the result in the destination floating point data register.

Operands:

<d>	Data addressing modes.
%fpm	Source register. Specifies any of the eight floating point data registers.
%fpn	Destination register. Specifies any of the eight floating point data registers.

Programming Note:

If a data direct register %dn is the data addressing mode in the instruction, mnemonics faddd and faddx are invalid.

Instruction	*Motorola*	MC680xx Processor
fbCC label	FBcc.W	40/60
fbCCw label	FBcc.W	40/60
fbCCl label	FBcc.L	40/60

Description:

If the specified condition is met, the program execution continues at location program counter (PC) + displacement. The displacement is a two's-complement integer that counts the relative distance in bytes. The PC value used to calculate the destination address is the address of the branch instruction (current location) plus two. If the displacement size is a word, then a 16-bit displacement is stored in the word immediately following the instruction operation word. If the displacement size is a long word, then a 32-bit displacement is stored in the two words immediately following the instruction operation word.

The *cc* and *CC* parts of this instruction represents one of the following conditions:

CC	cc	Description
eq	EQ	Equal
f	F	false
ge	GE	Greater than or equal
gl	GL	Greater than or less than
gle	GLE	Greater than or less than or equal
gt	GT	Greater than
le	LE	Less than or equal
lt	LT	Less than
ne	NE	Not equal

CC	cc	Description
nge	NGE	Not (greater than or equal)
ngl	NGL	Not (greater than or less than)
ngle	NGLE	Not (greater than or less than or equal)
ngt	NGT	Not greater than
nle	NLE	Not (less than or equal)
nlt	NLT	Not less than
oge	OGE	Ordered greater than
ogl	OGL	Ordered greater than or less than
ogt	OGT	Ordered greater than
ole	OLE	Ordered less than or equal
olt	OLT	Ordered less than
or	OR	Ordered
seq	SEQ	Signaling equal
sf	SF	Signaling false
sne	SNE	Signaling not equal
st	ST	Signaling true
t	T	True
ueq	UEQ	Unordered equal
uge	UGE	Unordered greater than or equal
ugt	UGT	Unordered greater than
ule	UGE	Unordered less than or equal
ult	UGT	Unordered less than
un	UN	Unordered

Operands:

`label` Specifies the program symbol destination (label) when the branch in taken.

Programming Note:

If branch distance is +2, or in other words, the branch is to the following statement, the branch is optimized to an `fnop` instruction.

Long branches are optimized to word branches if the displacement is within a word boundary (range).

Note 1: Instructions with long branches are allowed in the assembler but will not currently load in ITS.

Note 2: When a BSUN exception occurs, the main processor takes a preinstruction exception. If the exception handler returns without modifying the image of the program counter on the stack frame (to point to the instruction following the `FBcc`), then it must clear the cause of the exception (by clearing the NAN bit or disabling the BSUN trap), or the exception will occur again immediately upon return to the routine that caused the exception.

Instruction	*Motorola*	MC680xx Processor
fcmp <d>,%fpn	FCMP.W	40/60
fcmpb <d>,%fpn	FCMP.B	40/60
fcmpw <d>,%fpn	FCMP.W	40/60
fcmpl <d>,%fpn	FCMP.L	40/60
fcmps <d>,%fpn	FCMP.S	40/60
fcmpd <d>,%fpn	FCMP.D	40/60
fcmpx <d>,%fpn	FCMP.X	40/60
fcmpx %fpm,%fpn	FCMP.X	40/60

Description:

Converts source operand to extended precision (if necessary), subtracts that operand from the destination floating point data register and sets the condition codes according to the results. The destination location is unchanged.

Operands:

<d>	Data addressing modes.
%fpm	Source register. Specifies any of the eight floating point data registers.
%fpn	Destination register. Specifies any of the eight floating point data registers.

Programming Note:

If a data direct register %dn is the data addressing mode in the instruction, mnemonics fcmpd and fcmpx are invalid.

Instruction	*Motorola*	MC680xx Processor
fdiv <d>,%fpn	FDIV.W	40/60
fdivb <d>,%fpn	FDIV.B	40/60
fdivw <d>,%fpn	FDIV.W	40/60
fdivl <d>,%fpn	FDIV.L	40/60
fdivs <d>,%fpn	FDIV.S	40/60
fdivd <d>,%fpn	FDIV.D	40/60
fdivx <d>,%fpn	FDIV.X	40/60
fdivx %fpm,%fpn	FDIV.X	40/60

Description:

Converts source operand to extended precision (if necessary), divides that number into the number in the destination floating point data register, and stores the result in the destination floating point data register.

Operands:

<d>	Data addressing modes.
%fpm	Source register. Specifies any of the eight floating point data registers.

`%fpn` Destination register. Specifies any of the eight floating point data registers.

Programming Note:

If a data direct register `%dn` is the data addressing mode in the instruction, mnemonics `fdivd` and `fdivx` are invalid.

Instruction	*Motorola*	MC680xx Processor
fint <d>,%fpn	FINT.W	40/60
fintb <d>,%fpn	FINT.B	40/60
fintw <d>,%fpn	FINT.W	40/60
fintl <d>,%fpn	FINT.L	40/60
fints <d>,%fpn	FINT.S	40/60
fintd <d>,%fpn	FINT.D	40/60
fintx <d>,%fpn	FINT.X	40/60
fintx %fpm,%fpn	FINT.X	40/60
fintx %fpn	FINT.X	40/60

Description:

Converts source operand to extended precision (if necessary), extracts the integer part and converts it to an extended precision floating point number, and stores the result in the destination floating point data register. The integer part is extracted by rounding the extended precision number to an integer using the current rounding mode selected in the floating point control register (FPCR) mode control byte.

Operands:

`<d>` Data addressing modes.

`%fpm` Source register. Specifies any of the eight floating point data registers.

`%fpn` Destination register. Specifies any of the eight floating point data registers. (Also source register if single register syntax is used).

Programming Note:

If a data direct register `%dn` is the data addressing mode in the instruction, mnemonics `fintd` and `fintx` are invalid.

Instruction	*Motorola*	MC680xx Processor
fintrz <d>,%fpn	FINTRZ.W	40/60
fintrzb <d>,%fpn	FINTRZ.B	40/60
fintrzw <d>,%fpn	FINTRZ.W	40/60
fintrzl <d>,%fpn	FINTRZ.L	40/60
fintrzs <d>,%fpn	FINTRZ.S	40/60
fintrzd <d>,%fpn	FINTRZ.D	40/60
fintrzx <d>,%fpn	FINTRZ.X	40/60
fintrzx %fpm,%fpn	FINTRZ.X	40/60
fintrzx %fpn	FINTRZ.X	40/60

Description:

Converts source operand to extended precision (if necessary), extracts the integer part and converts it to an extended precision floating point number, and stores the result in the destination floating point data register. The integer part is extracted by rounding the extended precision number to an integer using the round-to-zero mode, regardless of the rounding mode selected in the floating point control register (FPCR) mode control byte.

Operands:

<d>	Data addressing modes.
%fpm	Source register. Specifies any of the eight floating point data registers.
%fpn	Destination register. Specifies any of the eight floating point data registers. (Also source register if single register syntax is used).

Programming Note:

If a data direct register %dn is the data addressing mode in the instruction, mnemonics fintrzd and fintrzx are invalid.

Instruction	*Motorola*	MC680xx Processor
fmove <d>,%fpn	FMOVE.W	40/60
fmoveb <d>,%fpn	FMOVE.B	40/60
fmovew <d>,%fpn	FMOVE.W	40/60
fmovel <d>,%fpn	FMOVE.L	40/60
fmoves <d>,%fpn	FMOVE.S	40/60
fmoved <d>,%fpn	FMOVE.D	40/60
fmovex <d>,%fpn	FMOVE.X	40/60
fmovex %fpm,%fpn	FMOVE.X	40/60
fmove %fpm,<da>	FMOVE.W	40/60
fmoveb %fpm,<da>	FMOVE.B	40/60
fmovew %fpm,<da>	FMOVE.W	40/60
fmovel %fpm,<da>	FMOVE.L	40/60
fmoves %fpm,<da>	FMOVE.S	40/60
fmoved %fpm,<da>	FMOVE.D	40/60
fmovex %fpm,<da>	FMOVE.X	40/60

Description:

Moves the contents of the source operand to the destination operand. It performs data movement and is also considered an arithmetic instruction since conversion from the source format to the destination format is performed implicitly during the operation.

Memory-to-Register and Register-to-Register Operation: Converts the source operand to extended precision (if necessary) and stores it in the destination floating point data register. Instruction fmove will round the result to the precision selected in the floating point control register (%fpcr). Data addressing modes are valid.

Register-to-Memory Operation: Rounds the source operand to the size of the specified destination format and stores it in the destination operand effective address. Data alterable addressing modes are valid.

Operands:

<d>	Data addressing modes.
<da>	Data alterable addressing modes.
%fpm	Source register. Specifies any of the eight floating point data registers.
%fpn	Destination register. Specifies any of the eight floating point data registers.

Programming Note:
If a data direct register %dn is the data addressing mode in the instruction, mnemonics fmoved and fmovex are invalid.

Instruction	*Motorola*	MC680xx Processor
fmovel <ea>,<fpsysctlreg>	FMOVE.L	40/60
fmovel <fpsysctlreg>,<a>	FMOVE.L	40/60

Description:
Moves the contents of a floating point system control register <fpsysctlreg> (floating point control register: %fpcr, floating point status register: %fpsr or floating point instruction address register: %fpiar) to or from an operand effective address. The floating point status register is changed only if it is the destination.

Memory-to-Register: All addressing modes are valid.

Register-to-Memory: Alterable addressing modes are valid.

Operands:

<ea>	All addressing modes.
<a>	Alterable addressing modes.
<fpsysctlreg>	Specifies any of the three floating point system control registers: %fpcr, %fpsr, %fpiar.

Programming Note:
If an address register %an is an operand in the instruction, the source or destination floating point system control register must be the floating point instruction address register %fpiar.

Instruction	*Motorola*	MC680xx Processor
fmoveml <ea>,<fpsysctlreglst>	FMOVEM.L	40/60
fmoveml <fpsysctlreglst>,<a>	FMOVEM.L	40/60

Description:
Moves one or more 32-bit values into or out of the specified system control registers. Any combination of the three system control registers may be specified. The registers are always moved in the same order, regardless of the addressing mode used and are moved in the following

order, respectively: floating point control register %fpcr, floating point status register %fpsr, and floating point instruction address register %fpiar.

Memory-to-Register: All addressing modes are valid.

Register-to-Memory: Alterable addressing modes are valid.

Operands:

<ea> All addressing modes.

<a> Alterable addressing modes.

<fpsysctlreglst>
Specifies a list of any of the three floating point system control registers: <%fpcr, %fpsr, %fpiar>.

Programming Note:

If an address register %an is an operand in the instruction, the source or destination floating point system control register list must contain only the floating point instruction address register <%fpiar>.

If a data direct register %dn is an operand in the instruction, the source or destination floating point system control register list must contain only one of the floating point system control registers: <%fpcr>, <%fpsr>, <%fpiar>.

Note: If a single register is selected, the opcode is the same as for the fmovel (FMOVE.L) single floating point system control register instruction.

Instruction	*Motorola*	MC680xx Processor
fmovemx <cntl_pincr>,<fpreglist>	FMOVEM.X	40/60
fmovemx <cntl_pincr>,Dn	FMOVEM.X	40/60
fmovemx <fpreglist>,<cntlalt_pdecr>	FMOVEM.X	40/60
fmovemx Dn,<cntlalt_pdecr>	FMOVEM.X	40/60

Description:

Moves one or more extended precision numbers to or from a list of floating point data registers. No conversion or rounding is performed during this operation, and the floating point status register is not affected by the instruction.

Any combination of the eight floating point data registers can be transferred, with the selected registers specified by a user-supplied mask. This mask is an 8-bit number, where each bit corresponds to one register; if a bit is set in the mask, that register is moved. The register select mask may be specified as a static value contained in the instruction or a dynamic value in the least significant eight bits of an integer data direct register.

FMOVEM allows three types of addressing modes: control, predecrement, or postincrement modes.

If the effective address is one of the control addressing modes, the registers are transferred between the processor and memory starting at the specified address up through higher addresses. The order of transfer is from FP0 - FP7.

If the effective address is the predecrement mode, only a register-to-memory operation is allowed; the registers are stored starting at the address contained in the address register down through the lower addresses. The order of transfer is from FP7 - FP0.

If the effective address is the postincrement mode, only a memory-to-register operation is allowed; the registers are loaded starting at the specified address up through higher addresses. The order of transfer is from FP0 - FP7.

Memory-to-Register: Control or postincrement addressing modes are valid.

Register-to-Memory: Control alterable or predecrement modes are valid.

Operands:

`<cntl_pincr>`
Control or postincrement modes.

`<cntlalt_pdecr>`
Control alterable or predecrement modes.

`Dn` Data direct register for dynamic mode.

`<fpreglist>`
Specifies a list of any of the eight floating point data registers to be transferred, where each is either separated by a comma `(,)` delimiter symbol and/or specified as an ordered range of registers as separated by the dash `(-)` delimiter.

Also, an 8-bit mask value can be specified as representation of each register set.

The following, which represent boundary registers, are some examples valid for `<fpreglist>`:

`<0x01>` means `<%fp0>`

`<0x80>` means `<%fp7>`

The following, which represent the same set of registers set, are some examples valid for `<fpreglist>`:

```
<%fp0,%fp1,%fp2,%fp4,%fp6,%fp7>
<%fp0-%fp2,%fp4,%fp6-%fp7>
<%fp7,%fp6,%fp4,%fp2,%fp1,%fp0>
<0xd7>
```

Programming Note:
This instruction provides a useful feature called dynamic register list specification that can significantly enhance system performance. If the calling conventions used for procedure calls use this feature, the number of floating point data registers saved and restored can be reduced.

For predecrement mode, the order of the bitmask is automatically reversed by the assembler. The bitmask at the start of a procedure and at the end of the procedure should be identical when they are specified numerically. This is not true with other assemblers and may lead to confusion.

On the same note as above, there is a similarity between `fmovemx` and `movem` instructions. Please note that the register list bitmask for `fmovemx` is "opposite" to that of `movem` for predecrement and postincrement modes. For `fmovemx`, this difference is compensated within the assembler.

Instruction	*Motorola*	MC680xx Processor
fmul <d>,%fpn	FMUL.W	40/60
fmulb <d>,%fpn	FMUL.B	40/60
fmulw <d>,%fpn	FMUL.W	40/60
fmull <d>,%fpn	FMUL.L	40/60
fmuls <d>,%fpn	FMUL.S	40/60
fmuld <d>,%fpn	FMUL.D	40/60
fmulx <d>,%fpn	FMUL.X	40/60
fmulx %fpm,%fpn	FMUL.X	40/60

Description:

Converts source operand to extended precision (if necessary), multiplies that number by the number in the destination floating point data register, and stores the result in the destination floating point data register.

Operands:

`<d>`	Data addressing modes.
`%fpm`	Source register. Specifies any of the eight floating point data registers.
`%fpn`	Destination register. Specifies any of the eight floating point data registers.

Programming Note:

If a data direct register `%dn` is the data addressing mode in the instruction, mnemonics `fmuld` and `fmulx` are invalid.

Instruction	*Motorola*	MC680xx Processor
fneg <d>,%fpn	FNEG.W	40/60
fnegb <d>,%fpn	FNEG.B	40/60
fnegw <d>,%fpn	FNEG.W	40/60
fnegl <d>,%fpn	FNEG.L	40/60
fnegs <d>,%fpn	FNEG.S	40/60
fnegd <d>,%fpn	FNEG.D	40/60
fnegx <d>,%fpn	FNEG.X	40/60
fnegx %fpm,%fpn	FNEG.X	40/60

Instruction	*Motorola*	MC680xx Processor
fnegx %fpn	FNEG.X	40/60

Description:

Converts source operand to extended precision (if necessary), inverts the sign of the mantissa, and stores the result in the destination floating point data register.

Operands:

<d>	Data addressing modes.
%fpm	Source register. Specifies any of the eight floating point data registers.
%fpn	Destination register. Specifies any of the eight floating point data registers (also source register if single register syntax is used).

Programming Note:

If a data direct register %dn is the data addressing mode in the instruction, mnemonics fnegd and fnegx are invalid.

Instruction	*Motorola*	MC680xx Processor
fnop	FNOP	40/60

Description:

No operation occurs. It is used to force synchronization of the floating point unit with an integer unit or to force processing of pending exceptions.

Programming Note:

fnop is the same instruction as the fb<*cc*>w <*label*> instruction, where <*cc*> = *f* (nontrapping false value 0) and <*label*> = *+2* (which results in a displacement of a word-sized 0).

Instruction	*Motorola*	MC680xx Processor
frestore <pincr_cntl>	FRESTORE	40/60

Description:

Aborts the execution of any floating point operation in progress and loads a new floating point unit internal state from the state frame located at the operand effective address.

Operands: <pincr_cntl> Postincrement or control addressing modes.

Instruction	*Motorola*	MC680xx Processor
fsave <pdecr_cntlalt>	FSAVE	40/60

Description:

Allows the completion of any floating point operation in progress. It saves the internal state of the floating point unit in a state frame located at the operand effective address. After the save operation, the floating point unit is in the idle state, waiting for the execution of the next instruction. Any

floating point operations in progress when and FSAVE is encountered can be completed before the FSAVE executes, saving an IDLE frame.

Operands: <pdecr_cntlalt> Predecrement or control alterable addressing modes.

Instruction	*Motorola*	MC680xx Processor
fsgldiv <d>,%fpn	FSGLDIV.W	40/60
fsgldivb <d>,%fpn	FSGLDIV.B	40/60
fsgldivw <d>,%fpn	FSGLDIV.W	40/60
fsgldivl <d>,%fpn	FSGLDIV.L	40/60
fsgldivs <d>,%fpn	FSGLDIV.S	40/60
fsgldivd <d>,%fpn	FSGLDIV.D	40/60
fsgldivx <d>,%fpn	FSGLDIV.X	40/60
fsgldivx %fpm,%fpn	FSGLDIV.X	40/60

Description:

Converts source operand to extended precision (if necessary), divides that number into the number in the destination floating point data register, and stores the result in the destination floating point data register rounded to single precision (regardless of the current rounding precision).

Both the source and destination operands are assumed to be representable in the single precision format. If either operand requires more than 24 bits of mantissa to be accurately represented, the extraneous mantissa bits are truncated prior to the division; hence, the accuracy of the result is not guaranteed. Furthermore, the result exponent may exceed the range of single precision, regardless of the rounding precision selected in the floating point control register (FPCR) mode control byte.

Operands:

<d>	Data addressing modes.
%fpm	Source register. Specifies any of the eight floating point data registers.
%fpn	Destination register. Specifies any of the eight floating point data registers.

Programming Note:

If a data direct register %dn is the data addressing mode in the instruction, mnemonics fsgldivd and fsgldivx are invalid. This function is undefined for 0/0 and ∞/∞.

The accuracy of the result is not affected by the number of mantissa bits required to represent each input operand since the input operands just change to extended precision. The result mantissa is rounded to single precision, despite the rounding precision selected in the FPCR.

Instruction	*Motorola*	MC680xx Processor
fsglmul <d>,%fpn	FSGLMUL.W	40/60
fsglmulb <d>,%fpn	FSGLMUL.B	40/60
fsglmulw <d>,%fpn	FSGLMUL.W	40/60

Instruction	*Motorola*	MC680xx Processor
fsglmull <d>,%fpn	FSGLMUL.L	40/60
fsglmuls <d>,%fpn	FSGLMUL.S	40/60
fsglmuld <d>,%fpn	FSGLMUL.D	40/60
fsglmulx <d>,%fpn	FSGLMUL.X	40/60
fsglmulx %fpm,%fpn	FSGLMUL.X	40/60

Description:

Converts source operand to extended precision (if necessary), multiplies that number by the number in the destination floating point data register, and stores the result in the destination floating point data register rounded to single precision (regardless of the current rounding precision).

Both the source and destination operands are assumed to be representable in the single precision format. If either operand requires more than 24 bits of mantissa to be accurately represented, the extraneous mantissa bits are truncated prior to the multiplication; hence, the accuracy of the result is not guaranteed. Furthermore, the result exponent may exceed the range of single precision, regardless of the rounding precision selected in the floating point control register (FPCR) mode control byte.

Operands:

`<d>`	Data addressing modes.
`%fpm`	Source register. Specifies any of the eight floating point data registers.
`%fpn`	Destination register. Specifies any of the eight floating point data registers.

Programming Note:

If a data direct register `%dn` is the data addressing mode in the instruction, mnemonics `fsglmuld` and `fsglmulx` are invalid.

The input operand mantissas truncate to single precision before the multiply operation. The result mantissa rounds to single precision despite the rounding precision selected in the FPCR.

Instruction	*Motorola*	MC680xx Processor
fsqrt <d>,%fpn	FSQRT.W	40/60
fsqrtb <d>,%fpn	FSQRT.B	40/60
fsqrtw <d>,%fpn	FSQRT.W	40/60
fsqrtl <d>,%fpn	FSQRT.L	40/60
fsqrts <d>,%fpn	FSQRT.S	40/60
fsqrtd <d>,%fpn	FSQRT.D	40/60
fsqrtx <d>,%fpn	FSQRT.X	40/60
fsqrtx %fpm,%fpn	FSQRT.X	40/60
fsqrtx %fpn	FSQRT.X	40/60

Description:

Converts source operand to extended precision (if necessary) and calculates the square root of that number in the destination floating point data register. This function is not defined for negative operands. FSQRT will round the result to the precision selected in the floating point control register (FPCR).

Operands:

<d>	Data addressing modes.
%fpm	Source register. Specifies any of the eight floating point data registers.
%fpn	Destination register. Specifies any of the eight floating point data registers (also source register if single register syntax is used).

Programming Note:

If a data direct register %dn is the data addressing mode in the instruction, mnemonics fsqrtd and fsqrtx are invalid.

Instruction	*Motorola*	MC680xx Processor
fsub <d>,%fpn	FSUB.W	40/60
fsubb <d>,%fpn	FSUB.B	40/60
fsubw <d>,%fpn	FSUB.W	40/60
fsubl <d>,%fpn	FSUB.L	40/60
fsubs <d>,%fpn	FSUB.S	40/60
fsubd <d>,%fpn	FSUB.D	40/60
fsubx <d>,%fpn	FSUB.X	40/60
fsubx %fpm,%fpn	FSUB.X	40/60

Description:

Converts source operand to extended precision (if necessary), subtracts that number from the number in the destination floating point data register and stores the result in the destination floating point data register.

Operands:

<d>	Data addressing modes.
%fpm	Source register. Specifies any of the eight floating point data registers.
%fpn	Destination register. Specifies any of the eight floating point data registers.

Programming Note:

If a data direct register %dn is the data addressing mode in the instruction, mnemonics fsubd and fsubx are invalid.

Instruction	*Motorola*	MC680xx Processor
ftst <d>	FTST.W	40/60
ftstb <d>	FTST.B	40/60
ftstw <d>	FTST.W	40/60

Instruction	*Motorola*	MC680xx Processor
ftstl <d>	FTST.L	40/60
ftsts <d>	FTST.S	40/60
ftstd <d>	FTST.D	40/60
ftstx <d>	FTST.X	40/60
ftstx %fpm	FTST.X	40/60

Description:

Converts source operand to extended precision (if necessary) and sets the condition code bits according to the data type of the result.

Operands:

`<d>` Data addressing modes.

`%fpm` Source register. Specifies any of the eight floating point data registers.

Programming Note:

If a data direct register `%dn` is the data addressing mode in the instruction, mnemonics `ftstd` and `ftstx` are invalid.

Instruction	*Motorola*	MC680xx Processor
jmp <ea>	JMP	00/20/30/40/60

Description:

Program execution continues at the address specified by the instruction. The address is specified by the control addressing modes.

Operands: `<ea>` specifies the address of the next instruction. Only control addressing modes are allowed.

Instruction	*Motorola*	MC680xx Processor
jsr <ea>	JSR	00/20/30/40/60

Description:

Pushes the address of the instruction immediately following the `jsr` instruction onto the system stack. Program execution continues at the address specified in the instruction.

Operands: `<ea>` specifies the address of the next instruction. Only control addressing modes are allowed.

Instruction	*Motorola*	MC680xx Processor
ldccr <ea>,%ccr	MOVE to CCR	00/20/30/40/60

Description:

Moves the contents of the source operand to the condition codes. The source operand is a word, but only the low-order byte of the word is used to update the condition codes.

Operands: `<ea>` specifies the location of the source operand. Only data addressing modes are allowed.

`%ccr` specifies the condition codes.

Instruction	*Motorola*	MC680xx Processor
ldsr <ea>,%sr	MOVE to SR	00/20/30/40/60

Description:

Moves the contents of the source operand to the status register. The source operand is a word, and all bits of the status registers are affected. Privileged.

Operands: `<ea>` specifies the location of the source operand. Only data addressing modes are allowed.

`%sr` specifies the status register.

Instruction	*Motorola*	MC680xx Processor
lea <ea>,%an	LEA	00/20/30/40/60

Description:

Loads the effective address into the specified address register. This instruction affects all 32 bits of the address register.

Operands: `%an` specifies the address register which is to be loaded with the effective address.

`<ea>` specifies the address to be loaded into the address register. Only control addressing modes are allowed.

Instruction	*Motorola*	MC680xx Processor
link %an,$expr	LINK	00/20/30/40/60
linkl %an,$expr	LINK.L	20/30/40/60

Description:

Pushes the current contents of the specified address register onto the stack. After the push, the address register is loaded from the updated stack pointer. Finally, the sign-extend displacement is added to the stack pointer. The content of the address register occupies two words on the stack.

Operands: `%an` specifies the address register through which the link is to be constructed.

`$expr` specifies the two's complement integer which is to be added to the stack pointer. This may be either a word or long word depending on the size of the instruction.

Instruction	*Motorola*	MC680xx Processor
lpstop $expr	LPSTOP	60

Description:

Low power stop operation. Move immediate number into status register, advance the program counter, and stop fetching instructions. An interrupt or reset causes execution to continue.

Operands: `$expr` is placed in the status register.

Instruction	*Motorola*	MC680xx Processor
lslb %dx,%dy	LSL.B	00/20/30/40/60
lslb $expr,%dy	LSL.B	
lsl %dx,%dy	LSL.W	
lsl $expr,%dy	LSL.W	
lsl <ea>	LSL.W	
lsll %dx,%dy	LSL.L	
lsll $expr,%dy	LSL.L	
lsrb %dx,%dy	LSR.B	00/20/30/40/60
lsrb $expr,%dy	LSR.B	
lsr %dx,%dy	LSR.W	
lsr $expr,%dy	LSR.W	
lsr <ea>	LSR.W	
lsrl %dx,%dy	LSR.L	
lsrl $expr,%dy	LSR.L	

Description:

Shifts the bits of the operand in the direction specified. The carry bit receives the last bit shift out of the operand. The shift count for the shifting of a register is specified in two ways:

Immediate - shift count is specified in instruction (shift range 1 through 8).

Register - shift count is contained in data register specified in instruction.

The content of memory can be shifted only 1 bit where the operand size is restricted to a word.

LSL - the operand is shifted left. The number of positions shifted is the shift count. Bits shifted out of the high-order bit go to both the carry and extend bits; zeros are shifted into the low-order bit.

LSR - the operand is shifted right. The number of positions shifted is the shift count. Bits shifted out of the lower-order bit go to both the carry and extend bits; zeros are shifted into the high-order bit.

Operands: `<ea>` specifies operand to be shifted. Only alterable memory addressing modes are allowed.

`%dy` specifies the data register whose content is to be shifted.

`$expr` or `$dx` specifies shift count or register where count is located.

A3.3 ASSEMBLY INSTRUCTIONS M THROUGH Z

Instruction	*Motorola*	MC680xx Processor
moveb <d>,<ea>	MOVE.B	00/20/30/40/60
move <ea>,%an	MOVEA.W	
move <ea>,<da>	MOVE.W	

Instruction	*Motorola*	MC680xx Processor
movel $bexpr,%dn	MOVEQ	
movel <ea>,%an	MOVEA.L	
movel <ea>,<da>	MOVE.L	

Description:

Moves the contents of the source to the destination location. The data is examined as it is moved and condition codes are set accordingly.

Operands:

<d>	Data addressing modes.
<da>	Data alterable addressing modes.
<ea>	All addressing modes.
$bexpr	Byte-sized constant expression: –128 to +127.
%an	Address register.
%dn	Data register.

Instruction	*Motorola*	MC680xx Processor
movec %rc,%rn	MOVEC	12/20/30/40/60
movec %rn,%rc	MOVEC	

Description:

Copy the contents of the specified control register (%rc) to a specified general purpose register (%rn) or vice-versa. A 32-bit transfer is always done no matter what the size of the control register. Unimplemented bits are read as zeroes. Privileged.

Operands: %rn may be any address or data register.

%rc may be one of the following:

%sfc - Source function code register.

%dfc - Destination function code register.

%cacr - Cache control register.

%usp - User stack pointer.

%vbr - Vector base register.

%caar - Cache address register (20/30 only).

%msp - Master stack pointer.

%isp - Interrupt stack pointer.

%tc - MMU translation control register (40/60 only).

%itt0 - Instruction transparent translation register 0 (40/60 only).

%itt1 - Instruction transparent translation register 1 (40/60 only).

%dtt0 - Data transparent translation register 0 (40/60 only).

`%dtt1` - Data transparent translation register 1 (40/60 only).

`%mmusr` - MMU status register (40/60 only).

`%urp` - User root pointer (40/60 only).

`%srp` - Supervisor root pointer (40/60 only).

`%pcr` - Processor configuration register (60 only).

`%buscr` - Bus control register (60 only).

Instruction	*Motorola*	MC680xx Processor
move16 (%ax)+,(%ay)+	MOVE16	40/60
move16 <absol>,(%an)	MOVE16	40/60
move16 <absol>,(%an)+	MOVE16	40/60
move16 (%an),<absol>	MOVE16	40/60
move16 (%an)+,<absol>	MOVE16	40/60

Description:

Moves the source line (i.e., 16-byte quantity) to the destination line. The lines must be aligned at 16-byte boundaries. Line transfers are performed using burst reads and writes. The address register used in a post-increment mode is incremented by 16 after the move.

Operands:

`%ax`	Source address register.
`%ay`	Destination address register.
`%an`	Source or destination address register.
`<absol>`	Source or destination absolute operand.

Instruction	*Motorola*	MC680xx Processor
movem reglist,<ea>	MOVEM.W	00/20/30/40/60
movem <ea>,reglist	MOVEM.W	
moveml reglist,<ea>	MOVEM.L	
moveml <ea>,reglist	MOVEM.L	

Description:

Transfers selected registers to or from consecutive memory locations, starting at the location specified by the effective address.

Operands: `reglist` specifies which registers are to be transferred.

`<0x0001>--`
means %d0

`<0x0080>--`
means %d7

`<0x0100>--`
means %a0

`<0x8000>--`
means %a7

<ea> specifies the memory address to or from which the registers are to be moved:

1. Register to memory transfer (only control alterable addressing modes, or predecrement mode is allowed).
2. Memory to register transfer (only control addressing modes, or post-increment addressing mode is allowed).

Programming Note:

%d0, %d1, %d2, %a0, %a1 are not saved between function calls. %a6 is the frame pointer and %a7 is the stack pointer. The "biggest" reglist needed for an assembly language function called by a C function is <0x3cf8>.

For predecrement mode addresses the order of the bitmask is automatically reversed by the assembler. The bitmask at the start of a procedure and at the end of the procedure should be identical when they are specified numerically. This is not true with other assemblers and may lead to confusion.

Instruction	*Motorola*	MC680xx Processor
movep %dx,expr(%ay)	MOVEP.W	00/20/30/40/60
movep expr(%ay),%dx	MOVEP.W	
movepl %dx,expr(%ay)	MOVEP.L	
movepl expr(%ay),%dx	MOVEP.L	

Description:

Transfers the data between a data register and alternate bytes of memory, starting at the specified location, and incrementing by two. The higher-order byte of the data register is transferred first, and the low-order byte is transferred last. The memory address is specified using the address register indirect plus displacement addressing mode. If the address is even, all the transfers are made on the high-order half of the data bus; if the address is odd, all the transfers are made on the low-order half of the data bus.

Operands: expr(%ay) specifies the address register indirect, plus the displacement addressing mode.

%dx specifies the data register to or from which the data is to be transferred.

Instruction	*Motorola*	MC680xx Processor
movesb %rn,<ea>	MOVES.B	12/20/30/40/60
movesb <ea>,%rn	MOVES.B	
moves %rn,<ea>	MOVES.W	
moves <ea>,%rn	MOVES.W	
movesl %rn,<ea>	MOVES.L	
movesl <ea>,%rn	MOVES.L	

Description:

Move the contents of %rn to the location <ea> in the address space

specified by %dfc or move the contents of the location <ea> in the address space specified by %sfc to %rn. If the destination is an address register, the source operand sign-extended to 32 bits before being moved.

Operands: %rn may be any address or data register.

<ea> may be any memory alterable addressing mode.

Note: For either of the two following examples where %an is the same register in the source and destination, and the moves instruction is any size, the value stored is undefined:

```
moves %an,(%an)+

moves %an,-(%an)
```

Instruction	*Motorola*	MC680xx Processor
muls <ea>,%dn	MULS.W	00/20/30/40/60

Description:

Multiplies two signed word operands yielding a long signed result. Only the low-order word of the register operand is used. All 32 bits of the product are saved in the destination register.

Operands: <ea> specifies the source operand. Only data addressing modes are allowed.

%dn specifies one of the data registers. The field always specifies the destination.

Instruction	*Motorola*	MC680xx Processor
mulsl <ea>,%dl	MULS.L	20/30/40/60*

Description:

Multiplies two signed long operands yielding a signed long result (%dl). Only the low-order 32 bits of the quad word product are saved.

Operands: <ea> specifies the source operand. Only data addressing modes are allowed.

%dl specifies one of the data registers.

Note: Overflow *will* occur if the high-order word of the possible quad word product is not a sign extension of the result %dl.

Programming Note:

This instruction is not implemented in hardware on the 060. Emulation support is required.

Instruction	*Motorola*	MC680xx Processor
mulsl <ea>,%dh:%dl	MULS.L	20/30/40

Description:

Multiplies two signed long operands yielding a signed quad result. The low-order 32 bits may be found in %dl and the high-order 32 bits in %dh.

Operands: <ea> specifies the source operand. Only data addressing modes are allowed.

%dh and %dl specify arbitrary data registers. If they are the same, the results are undefined.

Instruction	*Motorola*	MC680xx Processor
mulu <ea>,%dn	MULU.W	00/20/30/40/60

Description:

Multiplies two unsigned word operands yielding a long unsigned result. Only the low-order word of the register operand is used. All 32 bits of the product are saved in the destination register.

Operands: <ea> specifies the source operand. Only data addressing modes are allowed.

%dn specifies one of the data registers. This field always specifies the destination.

Instruction	*Motorola*	MC680xx Processor
mulul <ea>,%dl	MULU.L	20/30/40/60*

Description:

Multiplies two unsigned long operands yielding an unsigned long result (%dl). Only the low-order 32 bits of the quad word product are saved.

Operands: <ea> specifies the source operand. Only data addressing modes are allowed.

%dl specifies one of the data registers.

Note: Overflow will occur if the high-order word of the possible quad word product is not 0.

Programming Note:

This instruction is not implemented in hardware on the 060. Emulation support is required.

Instruction	*Motorola*	MC680xx Processor
mulul <ea>,%dh:%dl	MULU.L	20/30/40

Description:

Multiplies two unsigned long operands yielding an unsigned quad result. The low-order 32 bits may be found in %dl and the high order 32 bits in %dh.

Operands: <ea> specifies the source operand. Only data addressing modes are allowed.

%dh and %dl specify arbitrary data registers. If they are the same, the results are undefined.

Instruction	*Motorola*	MC680xx Processor
mvccr %ccr,<ea>	MOVE from CCR	12/20/30/40/60

Description:

Moves the condition code bits to the destination location, <ea>. Results are zero extended to a word (16 bits). Unimplemented bits are read as zeroes.

Operands: <ea> may be any data alterable addressing mode.

Instruction	*Motorola*	MC680xx Processor
mvsr %sr,<ea>	MOVE from SR	00/20/30/40/60

Description:

Moves the contents of the status register to the destination operand. The operand size is a word, and all bits of the status register are moved. Privileged on *Motorola* MC68012, MC68020, and MC68030 processors.

Operands: <ea> specifies the location of the source operand. Only data alterable addressing modes are allowed.

%sr specifies the status register.

Instruction	*Motorola*	MC680xx Processor
mvusp %usp,%an	MOVE USP	00/20/30/40/60
mvusp %an,%usp	MOVE USP	

Description:

Transfers the contents of the user stack pointer to or from the specified address register. This instruction is useful for supervisor state programs. Privileged.

Operands: %usp specifies the user stack pointer.

%an specifies the address register to or from which the user stack pointer is to be transferred.

Instruction	*Motorola*	MC680xx Processor
nbcd <ea>	NBCD	00/20/30/40/60

Description:

Subtracts the operand addressed as the destination and the extend bit from zero. The operation is performed using decimal arithmetic. The result is saved in the destination location. This instruction produces ten's complement of the destination if the extend bit is clear; it produces nine's complement if the extend bit is set. This is a byte operation only.

Operands: <ea> specifies the destination operand. Only data alterable addressing modes are allowed.

Instruction	*Motorola*	MC680xx Processor
negb <ea>	NEG.B	00/20/30/40/60
neg <ea>	NEG.W	
negl <ea>	NEG.L	

Description:

Subtracts from zero the operand addressed as the destination. Stores the result in the destination location.

Operands: <ea> specifies the destination operand. Only data alterable addressing modes are allowed.

Instruction	*Motorola*	MC680xx Processor
negxb <ea>	NEGX.B	00/20/30/40/60
negx <ea>	NEGX.W	
negxl <ea>	NEGX.L	

Description:

Subtracts the operand addressed as the destination and extend bit from zero. Stores the result in the destination location.

Operands: <ea> specifies the destination operand. Only data alterable addressing modes are allowed.

Instruction	*Motorola*	MC680xx Processor
nop	NOP	00/20/30/40/60

Description:

No operation occurs. The processor, except for the program counter, is unaffected. Execution continues with the instruction following the nop instruction.

Instruction	*Motorola*	MC680xx Processor
notb <ea>	NOT.B	00/20/30/40/60
not <ea>	NOT.W	
notl <ea>	NOT.L	

Description:

Converts the destination operand to 1's complement and stores the result in the destination location.

Operands: <ea> specifies the destination operand. Only data alterable addressing modes are allowed.

Instruction	*Motorola*	MC680xx Processor
orb $expr,<da>	ORI.B	00/20/30/40/60
orb %dn,<am>	OR.B	
orb <d>,%dn	OR.B	
orb $expr,%ccr	ORI to CCR	
or $expr,<da>	ORI.W	
or %dn,<am>	OR.W	
or <d>,%dn	OR.W	
or $expr,%sr	ORI to SR	
orl $expr,<da>	ORI.L	
orl %dn,<am>	OR.L	
orl <d>,%dn	OR.L	

Description:

Bitwise ORs the source operand to the destination operand and stores the result in the destination location.

Operands:

<am>	Alterable memory addressing modes.
<d>	Data addressing modes.
<da>	Data alterable addressing modes.
%ccr	Condition code register.
%dn	Data register.
%sr	Status register.
$expr	Constant expression.

Programming Note:

When the assembly instruction `or $expr,%dn` is given to the assembler, the `ORI` encoding is used instead of the `OR` encoding.

Instruction	*Motorola*	MC680xx Processor
pack –(%ax),–(%ay),$adj	PACK	20/30/40/60
pack %dx,%dy,$adj	PACK	

Description:

Adjust and pack the low four bits of each of two bytes into a single byte. When both arguments are data registers, the adjustment is added to the value contained in the source register. When the predecrement addressing mode is used, two bytes from the source are fetched and concatenated. The result is written to the destination.

Instruction	*Motorola*	MC680xx Processor
pea <ea>	PEA	00/20/30/40/60

Description:

Computes the effective address and pushes it onto the stack.

Operands: <ea> specifies the address to be pushed onto the stack. Only control addressing modes are allowed.

Instruction	*Motorola*	MC680xx Processor
pflusha	PFLUSHA	30/40
pflushan (%an)	PFLUSHAN	40/60

Description:

Invalidate all address translation cache entries. The `pflushan` instruction affects only non-global entries (i.e., pages whose global bit is not set).

Instruction	*Motorola*	MC680xx Processor
pflush <fc>,$<mask>	PFLUSH	30

Description:

Invalidate all address translation cache entries whose function code matches the <fc> operand modified by $<mask>. The 1 bits in $mask indicate which bits of <fc> must be matched, exactly. 0 bits indicate "don't cares." Privileged.

Operands: <fc> may be either:

1. A 3-bit immediate value.
2. A data register of which only the least significant 3 bits will be used.
3. %sfc or %dfc.

Instruction	*Motorola*	MC680xx Processor
pflush <fc>,$<mask>,<ea>	PFLUSH	30

Description:

Invalidate all address translation cache entries for the specified effective address, <ea>, whose function code matches the <fc> operand modified by $<mask>. The 1 bits in $mask indicate which bits of <fc> must be matched exactly. 0 bits indicate "don't cares." Privileged.

Operands: $<mask> is a 3-bit immediate value.

<fc> may be either:

1. A 3-bit immediate value.
2. A data register of which only the least significant 3 bits will be used.
3. %sfc or %dfc.

<ea> specifies the address to match. Only control alterable addressing modes are allowed.

Instruction	*Motorola*	MC680xx Processor
pflush (%an)	PFLUSH	40/60
pflushn (%an)	PFLUSHN	

Description:

Invalidates those address translation cache entries that match the logical address in %an and the function code specified in %dfc. Both instruction and data ATCs are affected. The pflushn instruction affects only non-global entries (i.e., pages whose global bit is not set).

Operands:

%an	The address register containing the logical address to match.
%dfc	Implied operand. If the value is 1 or 2, only user entries are flushed. If it is 5 or 6, only supervisor entries are flushed. Other values are undefined.

Instruction	*Motorola*	MC680xx Processor
plpar (%an)	PLPAR	60

Description:

Translate the logical address defined by the contents of the destination

function code register and the address register using full PPMU functionality and generate a full 32-bit physical address saving the results in %an.

Operands:

%an the address register.

Instruction	*Motorola*	MC680xx Processor
plpaw (%an)	PLPAW	60

Description:

Translate the logical address defined by the contents of the destination function code register and the address register using full PPMU functionality and generate a full 32-bit physical address writing the results in %an.

Operands:

%an the address register.

Instruction	*Motorola*	MC680xx Processor
ploadr <fc>,<ea>	PLOADR	30

Description:

Searches the address translation cache and translation table for the descriptor corresponding to the given function code, <fc>, and effective address, <ea>. Updates the address translation cache translation table and translation table U bits as though a read had occurred.

Operands: <fc> may be either:

1. A 3-bit immediate value.
2. A data register of which only the least significant 3 bits will be used.
3. %sfc or %dfc.

<ea> specifies the effective address to search for. Only control alterable addressing modes are allowed.

Instruction	*Motorola*	MC680xx Processor
ploadw <fc>,<ea>	PLOADW	30

Description:

Searches the address translation cache and translation table for the descriptor corresponding to the given function code, <fc>, and effective address, <ea>. Updates the address translation cache translation table and translation table U and M bits as though a write had occurred. Privileged.

Operands: <fc> may be either:

1. A 3-bit immediate value.
2. A data register of which only the least significant 3 bits will be used.
3. %sfc or %dfc.

<ea> specifies the effective address to search for. Only control alterable addressing modes are allowed.

Instruction	*Motorola*	MC680xx Processor
pmove %<pmmureg>,<ea>	PMOVE	30
pmove <ea>,%<pmmureg>	PMOVE	

Description:

Moves the contents of the pmmu register, %<pmmureg>, to the effective address, <ea> (or vice-verse). Flushes the address translation cache when %<pmmureg> is any pmmu register *except* %mmusr. Privileged.

Operands: %<pmmureg> may be either:

%crp

%srp

%tc

%mmusr

%tt0

%tt1

<ea> may be any control alterable addressing mode.

Instruction	*Motorola*	MC680xx Processor
pmovefd <ea>,%<pmmureg>	PMOVEFD	30

Description:

Moves the contents of the effective address, <ea>, to the specified pmmu register, %<pmmureg>. Flushing of the address translation cache is temporarily disabled. Privileged.

Operands: %<pmmureg> may be either:

%crp

%srp

%tc

%mmusr

%tt0

%tt1

<ea> may be any control alterable addressing mode.

Instruction	*Motorola*	MC680xx Processor
ptestr <fc>,<ea>,$<level>	PTESTR	30
ptestr <fc>,<ea>,$<level>,%an		

Description:

Search the address translation cache or translation tables to a specified level by simulating a read instruction. The function code, <fc>, and effective address arguments, <ea>, indicate what descriptor to search for.

If <level> is 0, the address translation cache is searched. Otherwise, the translation tables are searched down to level <level>. The %mmusr register is updated to reflect the results of the search. The bits that are set have different meanings depending upon whether the address translation cache or translation tables are searched. Privileged.

Operands: <fc> may be either:

1. A 3-bit immediate value.
2. A data register of which only the least significant 3 bits will be used.
3. %sfc or %dfc.

<ea> may be any control alterable addressing mode.

$<level> may be an immediate value between 0 and 7 inclusive.

%an will contain the physical address of the last descriptor successfully fetched if a translation table search is indicated.

Instruction	*Motorola*	MC680xx Processor
ptestr (%an)	PTESTR	40

Description:

Searches the translation tables for the page descriptor corresponding to the address in %an and sets the bits of the MMU status register, %mmusr, accordingly. The upper bits of the translated physical address are also stored in the %mmusr. A read access is simulated by setting the U bit in each descriptor. A matching entry in the ATCs will be flushed and a new entry created. Privileged.

Operands: %an is the address register containing the address corresponding to the page descriptor searched for.

%dfc is an implicit argument which indicates the function code for the address in %an.

Instruction	*Motorola*	MC680xx Processor
ptestw <fc>,<ea>,$<level> ptestw <fc>,<ea>,$<level>,%an	PTESTW	30

Description:

Search the address translation cache or translation tables to a specified level by simulating a write instruction. The function code, <fc>, and effective address arguments, <ea>, indicate what descriptor to search for. If <level> is 0, the address translation cache is searched. Otherwise, the translation tables are searched down to level <level>. The %mmusr register is updated to reflect the results of the search. The bits that are set have different meanings depending upon whether the address translation cache or translation tables are searched. Privileged.

Operands: <fc> may be either:

1. A 3-bit immediate value.
2. A data register of which only the least significant 3 bits will be used.
3. %sfc or %dfc.

<ea> may be any control alterable addressing mode.

$<level> may be an immediate value between 0 and 7 inclusive.

%an will contain the physical address of the last descriptor successfully fetched if a translation table search is indicated.

Instruction	*Motorola*	MC680xx Processor
ptestw (%an)	PTESTW	40

Description:

Searches the translation tables for the page descriptor corresponding to the address in %an and sets the bits of the MMU status register, %mmusr, accordingly. The upper bits of the translated physical address are also stored in the %mmusr. A write access is simulated by setting the U and M bits in each descriptor, the ATC entry, and %mmusr. A matching entry in the ATCs will be flushed and a new entry created. Privileged.

Operands: %an is the address register containing the address corresponding to the page descriptor searched for.

%dfc is an implicit argument which indicates the function code for the address in %an.

Instruction	*Motorola*	MC680xx Processor
reset	RESET	00/20/30/40/60

Description:

Asserts the reset line causing all external devices to be reset. The processor state (except the program counter) is unaffected and execution continues with the next instruction. Privileged.

Instruction	*Motorola*	MC680xx Processor
rolb %dx,%dy	ROL.B	00/20/30/40/60
rolb $expr,%dy	ROL.B	
rol %dx,%dy	ROL.W	
rol $expr,%dy	ROL.W	
rol <ea>	ROL.W	
roll %dx,%dy	ROL.L	
roll $expr,%dy	ROL.L	
rorb %dx,%dy	ROR.B	00/20/30/40/60
rorb $expr,%dy	ROR.B	
ror %dx,%dy	ROR.W	
ror $expr,%dy	ROR.W	
ror <ea>	ROR.W	
rorl %dx,%dy	ROR.L	
rorl $expr,%dy	ROR.L	

Description:

Rotates the bits of the operand in the direction specified. The extend bit

is not included in the rotation. The shift count for the rotation of a register can be specified in two ways:

Immediate - the shift count is specified in the instruction (shift range 1 through 8).

Register - the shift count is contained in a data register specified in the instruction.

The content of memory can be rotated 1 bit only where the operand size is restricted to a word.

ROL - the operand is rotated left. The number of positions shifted is the shift count. Bits shifted out of the low-order bit go to both the carry bit and back into the high-order bit. The extend bit is not modified or used.

ROR - the operand is rotated right. The number of positions shifted is the shift count. Bits shifted out of the low-order bit go to both carry bit and back into the high-order bit. The extend bit is not modified or used.

Operands: `<ea>` specifies the operand to be rotated. Only alterable memory addressing modes are allowed.

`%dy` specifies the data register whose content is to be shifted.

`$expr` or `%dx` specifies the rotate count or register where rotate count is located.

Instruction	*Motorola*	MC680xx Processor
roxlb %dx,%dy	ROXL.B	00/20/30/40/60
roxlb $expr,%dy	ROXL.B	
roxl %dx,%dy	ROXL.W	
roxl $expr,%dy	ROXL.W	
roxl <ea>	ROXL.W	
roxll %dx,%dy	ROXL.L	
roxll $expr,%dy	ROXL.L	
roxrb %dx,%dy	ROXR.B	00/20/30/40/60
roxrb $expr,%dy	ROXR.B	
roxr %dx,%dy	ROXR.W	
roxr $expr,%dy	ROXR.W	
roxr <ea>	ROXR.W	
roxrl %dx,%dy	ROXR.L	
roxrl $expr,%dy	ROXR.L	

Description:

Rotates the bits of the destination operand in the direction specified. The extend bit is included in the rotation. The shift count for the rotation of a register can be specified in two ways:

Immediate - the shift count is specified in the instruction (shift range 1 through 8).

Register - the shift count is contained in a data register specified in the instruction. The content of memory can be rotated 1 bit only where the operand size is restricted to a word.

ROXL - operand is rotated left. The number of positions shifted is the shift count. Bits shifted out of the high-order bit go to both the carry and extend bits; the previous value of the extend bit is shifted to the low-order bit.

ROXR - operand is rotated right. The number of positions shifted is the shift count. Bits shifted out of the low-order bit go to both the carry and extend bits; the previous value of the extend bit is shifted into the high-order bit.

Operands: <ea> specifies the operand to be rotated. Only alterable memory addressing modes are allowed.

%dy specifies the data register whose content is to be shifted.

$expr or %dx specifies immediate rotate count or register where rotate count is located.

Instruction	*Motorola*	MC680xx Processor
rtd $expr	RTD	12/20/30/40/60

Description:

Pulls the program counter from the stack and adds the sign-extended 16-bit value $expr to the stack pointer. The previous program counter is lost.

Instruction	*Motorola*	MC680xx Processor
rte	RTE	00/20/30/40/60

Description:

Pulls the status register and program counter from the system stack. The previous status register and program counter are lost. All bits in the status register are affected. Privileged.

Instruction	*Motorola*	MC680xx Processor
rtr	RTR	00/20/30/40/60

Description:

Pulls the condition codes and program counter from the stack. The previous condition codes and the program counter are lost. The status register's supervisor portion is unaffected.

Instruction	*Motorola*	MC680xx Processor
rts	RTS	00/20/30/40/60

Description:

Pulls the program counter from the stack. The previous program counter is lost.

Instruction	*Motorola*	MC680xx Processor
sbcd %dy,%dx	SBCD	00/20/30/40/60
sbcd –(%ay),–(%ax)	SBCD	

Description:

Subtracts the source operand from the destination operand (along with the extend bit), and stores the result in the destination location. The subtraction is performed in binary-coded decimal arithmetic. The operands can be addressed in two ways:

Data register to data register - the operands are contained in the data registers specified in the instructions.

Memory to memory - the operands are addressed with the predecrement addressing mode using the address registers specified in the instruction.

Operands: `%dy` specifies that the source register is a data register.

`%dx` specifies that the destination register is a data register.

`%ay` used to derive the memory address of the source operand using the predecrement addressing mode.

`%ax` used to derive the memory address of the destination operand using the predecrement addressing mode.

Instruction	*Motorola*	MC680xx Processor
scc <ea>	Scc	00/20/30/40/60

Description:

Tests the specified condition. If the condition is true, the byte specified by the effective address is set to TRUE (0xff); otherwise, this byte is set to FALSE (0x00).

The *cc* part of this instruction represents one of the following conditions:

CC - Carry clear (unsigned greater than or equal)
CS - Carry set (unsigned less than)
EQ - Equal
F - Always false
GE - Greater than or equal
GT - Greater than
HI - High (unsigned greater than)
LE - Less than or equal
LS - Low or same (unsigned less than or equal)
LT - Less than
MI - Minus
NE - Not equal
PL - Plus
T - Always true

VC - No overflow
VS - Overflow

Operands: <ea> specifies the location into which the TRUE/FALSE byte is stored. Only alterable data addressing modes are allowed.

Instruction	*Motorola*	MC680xx Processor
stop $expr	STOP	00/20/30/40/60

Description:

Moves the immediate operand into the entire status register; advances the program counter to point to the next instruction, and the processor stops fetching and executing instructions. Execution of instructions resumes when a trace, interrupt, or reset exception occurs. If an interrupt request arrives whose priority is higher than current processor priority, an interrupt exception occurs; otherwise, the interrupt request has no effect.

If the bit of the immediate data corresponding with the S bit is off, execution of the instruction causes a privilege violation. External reset will always initiate reset exception processing. Privileged.

Operands: $expr specifies the data to be loaded into the status register.

Instruction	*Motorola*	MC680xx Processor
subb $expr,<da>	SUBI.B	00/20/30/40/60
subb $expr,<a>	SUBQ.B	
subb %dn,<am>	SUB.B	
subb <d>,%dn	SUB.B	
sub $expr,<da>	SUBI.W	
sub $expr,<a>	SUBQ.W	
sub %dn,<<am>	SUB.W	
sub <ea>,%an	SUBA.W	
sub $<1-8>,%an	SUBQ.W	
sub <ea>,%dn	SUB.W	
subl $expr,<da>	SUBI.L	
subl $<1-8>,<a>	SUBQ.L	
subl %dn,<a>	SUB.L	
subl <ea>,%an	SUBA.L	
subl $<1-8>,%an	SUBQ.L	
subl <ea>,%dn	SUB.L	

Description:

Subtracts the source operand from the destination operand, and stores the result in the destination location.

Operands:

<a> Alterable addressing modes.

<am> Alterable memory addressing modes.

<d>	Data addressing modes.
<da>	Data alterable addressing modes.
<ea>	All addressing modes.
$<1-8>	Immediate data in the range of 1 through 8.
%dn	Specifies any of the eight data registers.

Programming Note:

When subtracting a constant from an address register, the `lea` instruction is better than `sub` if the constant is in the range of –32767 through 0 or 9 through 32768. For constants of 1 through 8 the `sub` instruction generates a `SUBQ` instruction which is better than an `lea`.

When the assembly instruction `sub $expr,%dn` is given to the assembler, the `SUBI` encoding is used instead of the `SUB` encoding.

Instruction	*Motorola*	MC680xx Processor
subxb %dy,%dx	SUBX.B	00/20/30/40/60
subxb –(%ay),–(%ax)	SUBX.B	
subx %dy,%dx	SUBX.W	
subx –(%ay),–(%ax)	SUBX.W	
subxl %dy,%dx	SUBX.L	
subxl –(%ay),–(%ax)	SUBX.L	

Description:

Subtracts the source operand from the destination operand (along with the extend bit), and stores the result in the destination location. Operands can be addressed in two ways:

Data register to data register - operands are contained in data registers specified in the instruction.

Memory to memory - operands are contained in memory and addressed with the predecrement addressing mode using the addressed registers specified in the instruction.

Operands: `%dx` specifies the destination register as a data register.

`%dy` specifies the source register as a data register.

`%ax` used to derive the memory address of the destination operand using the predecrement addressing mode.

`%ay` used to derive the memory address of the source operand using the predecrement addressing mode.

Instruction	*Motorola*	MC680xx Processor
swap %dn	SWAP	00/20/30/40/60

Description:

Exchanges the 16-bit halves of a data register.

Operands: `%dn` specifies the data register to swap.

Instruction	*Motorola*	MC680xx Processor
tas <ea>	TAS	00/20/30/40/60

Description:

Tests and sets the byte operand addressed by the effective address field. The current value of the operand is tested and N and Z are set accordingly. The operation is indivisible (using the read-modify-write memory cycle) to allow synchronization of several processors.

Operands: <ea> specifies the location of the tested operand. Only data alterable addressing modes are allowed.

Instruction	*Motorola*	MC680xx Processor
trap $vector	TRAP	00/20/30/40/60

Description:

Initiates exception processing. The vector number is generated to reference the trap instruction exception vector specified by the low-order 4 bits of the instruction. Sixteen trap instruction vectors are available. Privileged.

Operands: $vector specifies which trap vector contains the new program counter to be loaded.

Instruction	*Motorola*	MC680xx Processor
trapv	TRAPV	00/20/30/40/60

Description:

If the overflow condition is on, the processor initiates exception processing. The vector number is generated in reference to the trapv exception vector. If the overflow condition is off, no operation is performed, and execution continues with the next instruction in sequence. Privileged.

Instruction	*Motorola*	MC680xx Processor
tstb <ea>	TST.B	00/20/30/40/60
tst <ea>	TST.W	
tstl <ea>	TST.L	

Description:

Compares the operand with zero. No results are saved; however, the condition codes are set according to the results of the test.

Operands: <ea> specifies the destination operand. For the *Motorola* MC68000 and MC68012 processors, only data alterable addressing modes are allowed. The *Motorola* MC68020 and later processors allow all modes with the restriction that the address register direct mode is only allowed for word and long operands.

Instruction	*Motorola*	MC680xx Processor
unlk %an	UNLK	00/20/30/40/60

Description:

Loads the stack pointer from the specified address register. The address register is then loaded with the value from the top of the stack.

Operands: %an specifies the address register through which the unlinking is to be performed.

Instruction	*Motorola*	MC680xx Processor
unpk –(%ax),–(%ay),$adj unpk %dx,%dy,%adj	UNPK	20/30/40/60

Description:

Places the two BCD digits in the source operand byte into the lower nibbles of two bytes, and places zero bits in the upper nibbles of both bytes.

A4. *Intel*[1] 8086 AND 80186 PROCESSOR INSTRUCTION SET

This appendix is divided into two sections. The first section describes the *Intel* 8086 and *Intel* 80186 processor instructions and the operands accepted by each in the *5ESS*® switch. The second section lists the opcode instruction set. For more details see the *iAPX 86, 88 User's Manual* for Intel's documentation on how the instructions function. In the first section of this appendix, the following notation is used:

c(operand) The contents of *operand*. If the operand is a register operand, referral is to the contents of the register. If the operand is a memory operand, referral is to the contents of the effective address. The contents of an immediate operand, by convention, is the value of the operand.

a(operand) The address of *operand*. This is only meaningful with memory operands and it means the effective address of the operand.

The *Intel* 80186 processor instructions are grouped together in a separate table for ease and clarity.

CONTROL TRANSFER INSTRUCTIONS			
iAPX Instruction	***Intel* Instruction**	**Description**	**Operands**
(a) call *dest* (b) call **dest*	CALL	Intrasegment call.	1. In (a), *dest* must be a 16-bit offset. 2. In (b), *dest* must be a memory operand or *reg16*.
int int *$type*	INT	Signal an interrupt to the *Intel* 8086 processor. All interrupts cause an indirect, long transfer through the interrupt vector located at physical address 0 × 0 through 0 × 3ff. See Note 1. A type 3 interrupt is a special, 1-byte instruction.	*Type* must be an 8-bit value.
into	INTO	Interrupt on overflow. Perform an interrupt of type 4 if the OF is set.	None
iret	IRET	Interrupt return.	None
(a) jmp *dest* (b) jmp **dest*	JMP	Intrasegment jump.	1. In (a), *dest* must be a 16-bit or an 8-bit offset. The 8-bit offsets are sign extended. The offset is relative to the current %ip. 2. In (b), *dest* must be a memory operand or *reg16*. The destination is relative to %cs.
1call *segment*, *offset* 1call **dest*	CALL	Intersegment call.	1. *Segment* and *offset* must be 16-bit offsets. 2. *Dest* must be a memory operand or *reg16*.
1jmp *segment offset* 1jmp **dest*	JMP	Intersegment jump.	1. *Segment* and *offset* must be 16-bit offsets. 2. *Dest* must be a memory operand or *reg16*.

1. Registered trademark of Intel Corporation.

CONTROL TRANSFER INSTRUCTIONS			
iAPX Instruction	***Intel* Instruction**	**Description**	**Operands**
1ret 1ret incr	RET	Intersegment return.	*Incr* must be a 16-bit offset.
ret ret incr	RET	Intrasegment return.	*Incr* must be a 16-bit offset.
Note 1: Refer to *iAPX 86 88 Users Manual*, Intel Corporation, August 1981.			

ARITHMETIC INSTRUCTIONS			
iAPX Instruction	***Intel* Instruction**	**Description**	**Operands**
aaa	AAA	Unpacked BCD (ASCII) adjust for addition.	None
aad	AAD	Unpacked BCD (ASCII) adjust for division.	None
aam	AAM	Unpacked BCD (ASCII) adjust for multiplication.	None
aas	AAS	Unpacked BCD (ASCII) adjust for subtraction.	None
adc *source,dest* adcb *source,dest*	ADC	Add with carry c(*source*) to c(*dest*).	1. Operands are words for adc and bytes for adcb. 2. *Dest* may not be an immediate oprtsnf. 3. *Source* and *dest* may not both be memory operands. 4. *Sreg* is not allowed.

add *source, dest*
 	
addb *source, dest*
	ADC	Add c(*source*) to c(*dest*).	1. Operands are words for add
			
			and bytes for addb.
			2. *Dest* may not be an immediate operand.
			3. *Source* and *dest* may not both be memory operands.
			4. *Sreg* is not allowed.

cbw	CBW	Convert byte to word in accumulator.	None

cmp *source,dest*
 	
cmpb *source, dest*
	CMP	Compare c(*source*) with c(*dest*). The comparison is done by subtracting c(*source*) from c(*dest*), throwing the result away, and setting flags.	1. Operands are words for cmp and bytes for cmpb.
			2. *Dest* may not be an immediate operand.
			3. *Source* and *dest* may not both be memory operands.
			4. *Sreg* is not allowed.

cwd	CWD	Convert accumulator to double word.	None
daa	DAA	Decimal adjust for addition.	None
das	DAS	Decimal adjust for subtraction.	None

ARITHMETIC INSTRUCTIONS			
iAPX Instruction	***Intel* Instruction**	**Description**	**Operands**
dec *dest* decb *dest*	DEC	Decrement c(*dest*).	1. c(*dest*) is a word for dec and a byte for decb. 2. *Dest* may not be an immediate operand. 3. *Sreg* is not allowed.
div *source* [,%ax] divb *source* [,%al]	DIV	Unsigned division of extended accumulator by c(*source*).	1. c(*source*) is a word for div and a byte for divb. 2. *Source* may not be an immediate operand. 3. *Sreg* is not allowed.
idiv *source* [,%ax] idivb *source* [,%al]	IDIV	Signed division of extended accumulator by c(*source*).	1. c(*source*) is a word for idiv and a byte for idivb. 2. *Source* may not be an immediate operand. 3. *Sreg* is not allowed.
imul *source* [,%ax] imulb *source* [,%al]	IMUL	Signed multiplication of accumulator by c(*source*).	1. Operands are words for imul and bytes for imulb. 2. *Source* may not be an immediate operand. 3. *Sreg* is not allowed.
inc *dest* incb *dest*	INC	Increment c(*dest*).	1. c(*dest*) is a word for inc and a byte for incb. 2. *Dest* may not be an immediate operand. 3. *Sreg* is not allowed.
mu1 *source* [,%ax] mulb *source* [,%a1]	MUL	Unsigned multiplication of accumulator by c(*source*).	1. Operands are words for mul and bytes for mulb. 2. *Source* may not be an immediate operand. 3. *Sreg* is not allowed.
neg *dest* negb *dest*	NEG	Negate c(*dest*).	1. c(*dest*) is a word for neg and a byte for negb. 2. *Dest* may not be an immediate operand or *sreg*.
sbb *source,* dest sbbb *source,* dest	SBB	Subtract with borrow c(*source*) from c(*dest*).	1. Operands are words for sbb and bytes for sbbb. 2. *Dest* may not be an immediate operand. 3. *Source* and *dest* may not both be memory operands. 4. *Sreg* is not allowed.
sub *source,* dest subb *source,* dest	SUB	Subtract c(*source*) from c(*dest*).	1. Operands are words for sub and bytes for subb. 2. *Dest* may not be an immediate operand. 3. *Source* and *dest* may not both be memory operands. 4. *Sreg* is not allowed.

CONDITIONAL JUMP INSTRUCTIONS			
Sixteen conditional short jump instructions test and branch on various flag conditions. The destination operand for all of these instructions is an 8-bit signed offset. If the destination will not fit in 8 bits, the assembler reverses the sense of the comparison and generates a long transfer to the desired location. For example, je OxOabc will have two instructions generated for it: jne tag jmp OxOabc tag: The assembler invents the artificial label *tag*.			
iAPX Instruction	***Intel* Instruction**	**Description**	**Operands**
ja *dest* jnbe *dest*	JA JNBE	Jump on above (not below or equal).	*Dest* is an 8-bit offset, sign-extended to 16 bits.
jae *dest* jnb *dest*	JAE JNB JNC	Jump on above or equal (not below). Jump on not carry.	*Dest* is an 8-bit offset, sign-extended to 16 bits.
jb *dest* jnae *dest*	JB JNAE JC	Jump on below (not above or equal). Jump on carry.	*Dest* is an 8-bit offset, sign-extended to 16 bits.
jbe *dest* jna *dest*	JBE JNA	Jump on below or equal (not above).	*Dest* is an 8-bit offset, sign-extended to 16 bits.
je *dest* jz *dest*	JE JZ	Jump on equal (zero).	*Dest* is an 8-bit offset, sign-extended to 16 bits.
jg *dest* jnle *dest*	JG JNLE	Jump on greater (not less or equal).	*Dest* is an 8-bit offset, sign-extended to 16 bits.
jge *dest* jnl *dest*	JGE JNL	Jump on greater or equal (not less).	*Dest* is an 8-bit offset, sign-extended to 16 bits.
jl *dest* jnge *dest*	JL JNGE	Jump on less (not greater or equal).	*Dest* is an 8-bit offset, sign-extended to 16 bits.
jle *dest* jng *dest*	JLE JNG	Jump on less or equal (not greater).	*Dest* is an 8-bit offset, sign-extended to 16 bits.
jne *dest* jnz *dest*	JNE JNZ	Jump on not equal (not zero).	*Dest* is an 8-bit offset, sign-extended to 16 bits.
jno *dest*	JNO	Jump on not overflow.	*Dest* is an 8-bit offset, sign-extended to 16 bits.
jns *dest*	JNS	Jump on not sign.	*Dest* is an 8-bit offset, sign-extended to 16 bits.
jo *dest*	JO	Jump on overflow.	*Dest* is an 8-bit offset, sign-extended to 16 bits.
jpe *dest* jp *dest*	JPE JP	Jump on parity even (parity).	*Dest* is an 8-bit offset, sign-extended to 16 bits.
jpo *dest* jnp *dest*	JPO JS	Jump on parity odd (not parity).	*Dest* is an 8-bit offset, sign-extended to 16 bits.
js *dest*	JS	Jump on sign.	*Dest* is an 8-bit offset, sign-extended to 16 bits.

DATA MOVEMENT INSTRUCTIONS			
iAPX Instruction	***Intel* Instruction**	**Description**	**Operands**

clr *dest*
clrb *dest*
	MOV	Clear c(*dest*). These instructions are abbreviations for mov and movb.	1. *Dest* must be either a register or a memory operand.
			2. If *dest* is a register operand, the following code is generated for cir and cirb, respectively:
			xor *dest, dest*
			xorb *dest, dest*
			3. Otherwise, clr and clrb are translated into the following:
			mov $0, *dest*
			movb $0, *dest*

in *port* in (%dx) inb *dest* inb (%dx)	INW IN	Input to the accumulator.	1. A word is transferred to %ax for in; a byte is transferred to %al for inb. 2. *Port* is an unsigned, 8-bit offset. It is not an immediate operand, and it is not sign-extended.
lahf	LAHF segment value, and *dest*	Load %ah with flags.	None
lds *source, dest*	LDS	Load a data segment pointer. The %ds register is set to a segment value, and *dest* is set to an offset.	1. *Source* must be a memory operand. 2. *Dest* must be *reg16*.
lea *source, dest*	LEA	Load effective address of *source*.	1. *Source* must be a memory operand. 2. *Dest* must be *reg16*.
les *source, dest*	LES	Load an extra segment pointer. The %es register is set to a segment value, and *dest* is set to an offset.	1. *Source* must be a memory operand. 2. *Dest* must be *reg16*.
mov *source, dest* movb *source, dest*	MOV	Move c(*source*) to c(*dest*).	1. Operands are words for mov and bytes for movb. 2. *Dest* may not be an immediate operand or %cs. 3. *Source* and *dest* may not both be memory operands.
out *port* out (%dx) outb *port* outb (%dx)	OUTW OUT	Output from the accumulator.	1. A word is transferred from %ax for out; a byte is transferred from %al for outb. 2. *Port* is an unsigned, 8-bit offset. It is not an immediate operand, and it is not sign-extended.
pop *dest*	POP	Pop a word off the stack into c(*dest*).	1. *Dest* may not be %cs. Use 1call and 1jmp instead. 2. *Dest* may not be an immediate operand.
popf	POPF	Pop a word off the stack into the flags.	None
push *source*	PUSH	Push c(*source*) into a word on the stack.	Immediate operands are not allowed (see *Intel* 80106 processor instruction).

DATA MOVEMENT INSTRUCTIONS			
iAPX Instruction	***Intel* Instruction**	**Description**	**Operands**
pushf	PUSHF	Push flags onto the stack.	None
sahf	SAHF	Store %ah into flags.	None
xchg *source, dest* xchgb *source, dest*	XCHG	Exchange c(*source*) and c(*dest*).	1. Operands are words for xchg and bytes for xchgb. 2. *Dest* may not be an immediate operand. 3. *Source* and *dest* may not both be memory operands. 4. *Sreg* is not allowed.
xlat	XLAT	Translate %a1 with the table addressed by %bx.	None

LOGICAL INSTRUCTIONS			
iAPX Instruction	***Intel* Instruction**	**Description**	**Operands**
and *source, dest* andb *source, dest*	AND	Bit-wise conjunction of c(*source*) to c(*dest*).	1. Operands are words for and bytes for andb. 2. *Dest* may not be an immediate operand. 3. *Source* and *dest* may not both be memory operands. 4. *Sreg* is not allowed.
not *dest* notb *dest*	NOT	Convert c(*dest*) to its one's complement.	1. c(*dest*) is a word to not and a byte to notb. 2. *Dest* may not be in immediate operand or *sreg*.
or *source, dest* orb *source, dest*	OR	Bit-wise disjunction of c(*source*) to c(*dest*).	1. Operands are words for or and bytes for orb. 2. *Dest* may not be an immediate operand. 3. *Source* and *dest* may not both be memory operands. 4. *Sreg* is not allowed.
rcl *dest* rcl %cl, *dest* rclb *dest* rclb %cl, *dest*	RCL	Rotate through carry left.	1. c(*dest*) is a word for rel and a byte for rclb. 2. *Dest* may not be an immediate operand or *sreg* (see *Intel* 80186 processor instructions).
rcr *dest* rcr %cl, *dest* rcrb *dest* rcrb %cl, *dest*	RCR	Rotate through carry right.	1. c(*dest*) is a word for rcr and a byte for rcrb. 2. *Dest* may not be an immediate operand or *sreg* (see *Intel* 80186 processor instructions).
rol *dest* rol %cl, *dest* rolb *dest* rolb %cl, *dest*	ROL	Rotate left.	1. c(*dest*) is a word for rol and a byte for rolb. 2. *Dest* may not be an immediate operand or *sreg* (see *Intel* 80186 processor instructions).

LOGICAL INSTRUCTIONS			
iAPX Instruction	***Intel* Instruction**	**Description**	**Operands**
ror *dest* ror %cl, *dest* rorb *dest* rorb %cl, *dest*	ROR	Rotate right.	1. c(*dest*) is a word for ror and a byte for rorb. 2. *Dest* may not be an immediate operand or *sreg* (see *Intel* 80186 processor instructions).
sar *dest* sar %cl, *dest* sarb *dest* sarb %cl, *dest*	SAR	Shift arithmetic right.	1. c(*dest*) is a word for shr and a byte for shrb. 2. *Dest* may not be an immediate operand or *sreg* (see *Intel* 80186 processor instructions).
shl *dest* shl %cl, *dest* shlb *dest* shlb %cl, *dest*	SHL SAL	Shift left. Note: sal and salb are equivalent to shl and shlb, respectively.	1. c(*dest*) is a word for shl and a byte for shlb. 2. *Dest* may not be an immediate operand or *sreg* (see *Intel* 80186 processor instructions).
shr *dest* shr %cl, *dest* shrb *dest* shrb %cl, *dest*	SHR	Shift right logical.	1. c(*dest*) is a word for shr and a byte for shrb. 2. *Dest* may not be an immediate operand or *sreg* (see *Intel* 80186 processor instructions).
test *source, dest* testb *source, dest*	TEST	Bit-wise conjunction between c(*source*) and c(*dest*). The conjunction is performed, the flags are set, but returned. See Note 1.	1. Operands are a words for test and bytes for testb. 2. *Dest* may not be an immediate operand. 3. *Source* and *dest* may not both be memory operands. 4. *Sreg* is not allowed.
xor *source, dest* xorb *source, dest*		Bit-wise exclusive or of c(*source*) to c(*dest*).	1. Operands are words for xor and bytes for xorb. 2. *Dest* may not be an immediate operand. 3. *Source* and *dest* may not both be memory operands. 4. *Sreg* is not allowed.

Note 1: The assembler allows several abbreviations for the operands of test and testb:

Short Form		**Long Form**	
test	reg16	test	reg16, reg16
test	ea	test	$0xff ff, ea
test	ea8	test	$0xff ff, ea8
test	ea16	test	$0xff ff, ea16
testb	reg8	testb	reg8, reg8
testb	ea	testb	$0xff, ea
testb	ea8	testb	$0xff, ea8
testb	ea16	test	$0xff, ea16

LOOPING INSTRUCTIONS

Loop instructions provide iteration control with %cx register. All of these instructions take an 8-bit offset as an operand and perform a short jump if certain conditions are satisfied. As in the conditional short jumps, the assembler will generate correct code, even if the supply operand will not fit into 8 bits. For example,

loop 0x0abc

will be expanded into the following:

loop tag1
jmp tag2 tag1:
jmp 0x0abc tag2:

The assembler invents the artificial labels tag1 and tag2.

iAPX Instruction	*Intel* Instruction	Description	Operands
jcxz *dest*	JCXZ	Jump on %cx zero.	*Dest* is an 8-bit offset, sign-extended to 16 bits.
loop *dest*	LOOP	Loop on %cx.	*Dest* is an 8-bit offset, sign-extended to 16 bits.
loope *dest* loopz *dest*	LOOPE LOOPZ	Loop while equal (zero).	*Dest* is an 8-bit offset, sign-extended to 16 bits.
loopne *dest* loopnz *dest*	LOOPNE LOOPNZ	Loop while not equal (not zero).	*Dest* is an 8-bit offset, sign-extended to 16 bits.

PSEUDO OPERATIONS

Name	Description	Operands
.bss	Change the current section to .bss.	None
.bss *tag, bytes*	Define symbol *tag* in the .bss section and add *bytes* to the value of *dot* for .bss. This does *not* change the current section to .bss.	*Tag* is a symbol name. *Bytes* must be an *absolute* value.
.byte *val* [, *val*]...	Generate initialized bytes into the current section; this is not valid for .bss.	Each *val* must be an 8-bit value.
.data	Change the current section to .data.	None
.def *name*	Start of symbolic description for symbol name. See .endef.	*Name* is a symbol name.
.dim *expr* [, *expr*]...	If *name* of .def is an array, the expressions give the dimensions. Up to four dimensions are accepted.	The type of each *expr* should be *absolute*.
.endef	Ending bracket for .def.	None
.even	Align the current *dot* to an even boundary.	None
.file "*name*"	C source filename. Only one allowed per assembly file.	One to 14 characters allowed.
.globl *name*	Equivalent to extern in C.	Name is treated as a global symbol.
ident "*string*"	Create an entry in the *.comment* section containing *string*.	*String* is any sequence of characters not including ".
.line *expr*	Defines the source line number of the definition of block symbol *name* in .def.	*Expr* should yield an *absolute* value.

PSEUDO OPERATIONS		
Name	**Description**	**Operands**
.ln *line* [, *addr*]	Create an entry in the line number table for a section. The current *dot* is the default value for *addr*.	*Line* should be an *absolute* value for the source line number. The type of *addr* tells which section owns the line number.
.scl *expr*	Within .def give *name* storage class *expr*.	The type of *expr* should be *absolute*.
.set *name, expr*	Set the value of symbol *name* to *expr*.	As described.
.size *expr*	If *name* of .def is an object such as a structure or an array, give it size *expr*.	The type of *expr* should be *absolute*.
.tag *str*	If *name* of .def is a structure or union, *str* should be the name of that structure or union tag, defined in a previous .def - .endef pair.	As described.
.text	Change the current section to .text.	None
.tv *name*	Declare *name* as a transfer vector symbol. If *name* will be used outside the defining file, .globl is needed. This should be used only for 20-bit addressing.	In general, *name* is a function name.
.type *expr*	Within .def, give name the C compiler-type representation *expr*.	The type of *expr* should be *absolute*.
.val *expr*	Within .def, give *name* the value *expr*.	The type of *expr* determines the section for *name*.
.value *expr* [, *expr*]...	Generate initialized words into the current section; this is not valid for .bss.	Each *expr* is a 16-bit value.

STRING PRIMITIVE INSTRUCTIONS			
iAPX Instruction	***Intel* Instruction**	**Description**	**Operands**
rep repz repnz	REP	Repeat prefixes for the string primitives. See Chapter 8, Bibliography, Reference 11, MCS-86 User's Manual, paragraph 4.4.6 for exact details. Instructions rep and repz are equivalent and cause ZF to be tested with zero; repnz uses a test on ZF with one.	
sca [%ax, (%di)] scab [%al, (%di)]	SCAW SCAB	String primitive scan. Note: A repeat prefix is allowed.	1. Operands are words for sca and bytes for scab. 2. (%di) always uses the extra segment register to compute the address of its operand.

STRING PRIMITIVE INSTRUCTIONS			
iAPX Instruction	***Intel* Instruction**	**Description**	**Operands**
scmp [(%si), (%di)] scmpb [(%si), (%di)]	CMPW CMPB	String primitive compare. Note: A repeat prefix is allowed.	1. Operands are words for scmp and bytes for scmpb. 2. (%di) always uses the extra segment register to compute the address of the destination. 3. (%si) uses the data segment by default, but a segment override prefix may be used.
slod [(%si), %ax] slodb [(%si), (%al)]	LODW LODB	String primitive load accumulator. A repeat prefix is possible but not normally useful.	1. c(c(%si)) is a word for slod and a byte for slodb. 2. (%si) uses the data segment by default, but a segment override prefix may be used.
smov [(%si), (%di)] smovb [(%si), (%di)]	MOVW MOVB	String primitive move. Note: A repeat prefix is allowed.	1. Operands are words for smov and bytes for smovb. 2. (%di) always uses the extra segment register to compute the address of the destination. 3. (%si) uses the data segment by default, but a segment override prefix may be used.
ssto [%ax, (%di)] ssstob [%al, (%di)]	STOW STOB	String primitive store accumulator. Note: A repeat prefix is allowed.	1. c(c(%di)) is a word for ssto and a byte for sstob. 2. (%di) always uses the extra segment register to compute the address of its operand.

***INTEL* 80186 PROCESSOR INSTRUCTIONS**			
iAPX Instruction	***Intel* Instruction**	**Description**	**Operands**
push *source*	PUSH	Push c(*source*) into a word on the stack.	1. Immediate operands are not allowed. 2. Source will be sign-extended to 16 bits.
pusha	PUSHA	Push all general registers onto the stack. Registers pushed are all reg 16 registers.	None
popa	POPA	Pop all general registers (see push description) off the stack. Registers popped are all reg 16 registers.	None

INTEL 80186 PROCESSOR INSTRUCTIONS			
iAPX Instruction	***Intel* Instruction**	**Description**	**Operands**
iimul *dest*, *source, immed*	IMUL	Signed multiply of source by *immed*.	1. *Dest* must be reg 16. 2. *Immed* is an immediate operand which will be sign-extended to 16 bits.
bound *register, memory*	BOUND	This instruction checks array index bounds. If c(*register*) is less than c(*memory*) or greater than c(*memory + 2*), then an interrupt is generated. The interrupt is trap type 5.	1. Register must be reg16. 2. The c(*memory*) is expected to contain the lower bound of an array and c(*memory + 2*) is expected to contain the upper bound.
enter locals, level	ENTER	Enter executes a calling sequence. It saves, %bp (the frame pointer) copies previous procedure frame pointers, and allocates c(*locals*) amount of space on the stack for local variable.	1. *Locals* in an immediate 16-bit operand represents amount of space to alllocate. 2. *Level* is an immediate 8-bit operand and represents the current procedure level.
leave	LEAVE	Restores the stack. Used upon exiting a routine.	None
all shift and rotate instruction		Now can shift or rotate the destination by immed amount, whereas before it could only shift or rotate by 1 or C(%cl).	First operand can now be an immediate operand.

INTEL 80286 PROCESSOR INSTRUCTIONS PROTECTED VIRTUAL ADDRESS MODE and PROTECTION PARAMETER VERIFICATION			
iAPX Instruction	***Intel* Instruction**	**Description**	**Operands**
arpl *source*, *dest*	ARPL	Adjust requested privilege level field of selector to not less than that of the source.	1. *Dest* is a 16-bit memory or register value containing the value of a selector. 2. *Source* is a register operand.
clts	CLTS	Clear the Task Switched Flag (TS) which the hardware sets every time it performs a task switching operation.	None
lar *source*, *dest*	LAR	Load access rights from descriptor to register.	1. *Source* is 16-bit memory or register operand containing a selector. 2. *Dest* is a word register of which the upper byte gets set and the lower byte gets cleared.
lgdt *source*	LGDT	Load the Global Descriptor Table Register (GDT) with base and limit information for the system's global descriptor table.	Operand is 6 bytes in memory starting at *source*.

INTEL 80286 PROCESSOR INSTRUCTIONS PROTECTED VIRTUAL ADDRESS MODE and PROTECTION PARAMETER VERIFICATION			
iAPX Instruction	***Intel* Instruction**	**Description**	**Operands**
lidt *source*	LIDT	Load Interrupt Descriptor Table Register (IDT) with six bytes of base and limit information for the system's interrupt descriptor table.	Operand is 6 bytes in memory starting at *source*.
lldt *source*	LLDT	Load Local Descriptor Table Register (LDT) with a 16-bit selector value indicating one of the local descriptor table listed within the GDT.	Operand is a 16-bit memory or register operand.
lmsw *source*	LMSW	Load Machines Status Word Register.	*Source* is a 16-bit memory or register operand.
lsl *source*, *dest*	LSL	Load segment limit from descriptor.	1. *Source* is 16-bit memory or register containing a selector. 2. *Dest* is a word register operand.
ltr *source*	LTR	Load Task Register.	*Source* is a 16-bit memory or register operand.
sgdt *dest*	SGDT	Stores the contents of the Global Descriptor Table (GDT).	Operand is 6 bytes in memory starting at *dest*.
sidt *dest*	SIDT	Store the contents of the Interrupt Descriptor Table (IDT) at six bytes starting at operand.	Operand is 6 bytes in memory starting at *dest*.
sldt *dest*	SLDT	Store Local Descriptor Table Register.	*Dest* is 16-bit register or memory operand.
smsw *dest*	SMSW	Store machine status word	*Dest* is 16-bit register or memory operand.
str *dest*	STR	Store task register	*Dest* is 16-bit mem or reg selector value.
verr *source*	VERR	Verify Read Access	*Source* is a 16-bit register or memory operand containing a selector.
verw *source*	VERW	Verify Write Access	*Source* is a 16-bit register or memory operand containing a selector.

Table A4-1 — iAPX *INTEL* 80186 Processor Instructions in Hex Order: 00 - 3F

Opcode			Operands	Function
Hex/Binary		Instruction		
00 00000000	MOD REG R/M	ADD	EA,REG	BYTE ADD (REG) TO EA
01 00000001	MOD REG R/M	ADD	EA,REG	WORD ADD (REG) TO EA
02 00000010	MOD REG R/M	ADD	REG,EA	BYTE ADD (EA) TO REG
03 00000011	MOD REG R/M	ADD	REG,EA	WORD ADD (EA) TO REG
04 00000100		ADD	AL,DATA8	BYTE ADD DATA TO REG AL
05 00000101		ADD	AX,DATA16	WORD ADD DATA TO REG AX
06 00000110		PUSH	ES	PUSH (ES) ON STACK
07 00000111		POP	ES	POP STACK TO REG ES
08 00001000	MOD REG R/M	OR	EA,REG	BYTE OR (REG) TO EA
09 00001001	MOD REG R/M	OR	EA,REG	WORD OR (REG) TO EA
0A 00001010	MOD REG R/M	OR	REG,EA	BYTE OR (EA) TO REG
0B 00001011	MOD REG R/M	OR	REG,EA	WORD OR (EA) TO REG
0C 00001100		OR	AL,DATA8	BYTE OR DATA TO REG AL
0D 00001101		OR	AX,DATA16	WORD OR DATA TO REG AX
0E 00001110		PUSH	CS	PUSH (CS) ON STACK
0F 00001111		(not used)		
10 00010000	MOD REG R/M	ADC	EA,REG	BYTE ADD (REG) W/ CARRY TO EA
11 00010001	MOD REG R/M	ADC	EA,REG	WORD ADD (REG) W/ CARRY TO EA
12 00010010	MOD REG R/M	ADC	REG,EA	BYTE ADD (EA) W/ CARRY TO REG
13 00010011	MOD REG R/M	ADC	REG,EA	WORD ADD (EA) W/ CARRY TO REG
14 00010100		ADC	AL,DATA8	BYTE ADD DATA W/ CARRY TO REG AL
15 00010101		ADC	AX,DATA16	WORD ADD DATA W/ CARRY TO REG AX
16 00010110		PUSH	SS	PUSH (SS) ON STACK
17 00010111		POP	SS	POP STACK TO REG SS
18 00011000	MOD REG R/M	SBB	EA,REG	BYTE SUB (REG) W/ BORROW FROM EA
19 00011001	MOD REG R/M	SBB	EA,REG	WORD SUB (REG) W/ BORROW FROM EA
1A 00011010	MOD REG R/M	SBB	REG,EA	BYTE SUB (EA) W/ BORROW FROM REG
1B 00011011	MOD REG R/M	SBB	REG,EA	WORD SUB (EA) W/ BORROW FROM REG
1C 00011100		SBB	AL,DATA8	BYTE SUB DATA W/ BORROW FROM REG AL
1D 00011101		SBB	AX,DATA16	WORD SUB DATA W/ BORROW FROM REG AX
1E 00011110		PUSH	DS	PUSH (DS) ON STACK
1F 00011111		POP	DS	POP STACK TO REG DS
20 00100000	MOD REG R/M	AND	EA,REG	BYTE AND (REG) TO EA
21 00100001	MOD REG R/M	AND	EA,REG	WORD AND (REG) TO EA
22 00100010	MOD REG R/M	AND	REG,EA	BYTE AND (EA) TO REG
23 00100011	MOD REG R/M	AND	REG,EA	WORD AND (EA) TO REG
24 00100100		AND	AL,DATA8	BYTE AND DATA TO REG AL
25 00100101		AND	AX,DATA16	WORD AND DATA TO REG AX

Table A4-1 — iAPX *INTEL* 80186 Processor Instructions in Hex Order: 00 - 3F (Contd)

Opcode			Operands	Function
Hex/Binary		Instruction		
26 00100110		ES:		SEGMENT OVERRIDE W/ SEGMENT REGS ES
27 00100111		DAA		DECIMAL ADJUST FOR ADD
28 00101000	MOD REG R/M	SUB	EA,REG	BYTE SUBTRACT (REG) FROM EA
29 00101001	MOD REG R/M	SUB	EA,REG	WORD SUBTRACT (REG) FROM EA
2A 00101010	MOD REG R/M	SUB	REG,EA	BYTE SUBTRACT (EA) FROM REG
2B 00101011	MOD REG R/M	SUB	REG,EA	WORD SUBTRACT (EA) FROM REG
2C 00101100		SUB	AL,DATA8	BYTE SUBTRACT DATA FROM REG AL
2D 00101101		SUB	AX,DATA16	WORD SUBTRACT DATA FROM REG AX
2E 00101110		CS:		SEGMENT OVERRIDE W/ SEGMENT REGS CS
2F 00101111		DAS		DECIMAL ADJUST FOR SUBTRACT
30 00110000	MOD REG R/M	XOR	EA,REG	BYTE XOR (REG) TO EA
31 00110001	MOD REG R/M	XOR	EA,REG	WORD XOR (REG) TO EA
32 00110010	MOD REG R/M	XOR	REG,EA	BYTE XOR (EA) TO REG
33 00110011	MOD REG R/M	XOR	REG,EA	WORD XOR (EA) TO REG
34 00110100		XOR	AL,DATA8	BYTE XOR DATA TO REG AL
35 00110101		XOR	AX,DATA16	WORD XOR DATA TO REG AX
36 00110110		SS:		SEGMENT OVERRIDE W/ SEGMENT REGS SS
37 00110111		AAA		ASCII ADJUST FOR ADD
38 00111000	MOD REG R/M	CMP	EA,REG	BYTE COMPARE (EA) WITH (REG)
39 00111001	MOD REG R/M	CMP	EA,REG	WORD COMPARE (EA) WITH (REG)
3A 00111010	MOD REG R/M	CMP	REG,EA	BYTE COMPARE (REG) WITH (EA)
3B 00111011	MOD REG R/M	CMP	REG,EA	WORD COMPARE (REG) WITH (EA)
3C 00111100		CMP	AL,DATA8	BYTE COMPARE DATA WITH (AL)
3D 00111101		CMP	AX,DATA16	WORD COMPARE DATA WITH (AX)
3E 00111110		DS:		SEGMENT OVERRIDE W/ SEGMENT REGS DS
3F 00111111		AAS		ASCII ADJUST FOR SUBTRACT

Table A4-2 — iAPX *INTEL* 80186 Processor Instructions in Hex Order: 40 - 7F

Opcode			Operands	Function
Hex/Binary		Instruction		
40 01000000		INC	AX	INCREMENT (AX)
41 01000001		INC	AX	INCREMENT (CX)
42 01000010		INC	DX	INCREMENT (DX)
43 01000011		INC	DX	INCREMENT (BX)
44 01000100		INC	SP	INCREMENT (SP)
45 01000101		INC	BP	INCREMENT (BP)
46 01000110		INC	SI	INCREMENT (SI)
47 01000111		INC	DI	INCREMENT (DI)
48 01001000		DEC	AX	DECREMENT (AX)
49 01001001		DEC	CX	DECREMENT (CX)
4A 01001010		DEC	DX	DECREMENT (DX)
4B 01001011		DEC	BX	DECREMENT (BX)
4C 01001100		DEC	SP	DECREMENT (SP)
4D 01001101		DEC	BP	DECREMENT (BP)
4E 01001110		DEC	SI	DECREMENT (SI)
4F 01001111		DEC	DI	DECREMENT (DI)
50 01010000		PUSH	AX	PUSH (AX) ON STACK
51 01010001		PUSH	CX	PUSH (CX) ON STACK
52 01010010		PUSH	DX	PUSH (DX) ON STACK
53 01010011		PUSH	BX	PUSH (BX) ON STACK
54 01010100		PUSH	SP	PUSH (SP) ON STACK
55 01010101		PUSH	BP	PUSH (BP) ON STACK
56 01010110		PUSH	SI	PUSH (SI) ON STACK
57 01010111		PUSH	DI	PUSH (DI) ON STACK
58 01011000		POP	AX	POP STACK TO REG AX
59 01011001		POP	CX	POP STACK TO REG CX
5A 01011010		POP	DX	POP STACK TO REG DX
5B 01011011		POP	BX	POP STACK TO REG BX
5C 01011100		POP	SP	POP STACK TO REG SP
5D 01011101		POP	BP	POP STACK TO REG BP
5E 01011110		POP	SI	POP STACK TO REG SI
5F 01011111		POP	DI	POP STACK TO REG DI
60 01100000		PUSHA		PUSH ALL DATA, INDEX, AND POINTER REGISTERS
61 01100001		POPA		POP ALL DATA, INDEX, AND POINTER REGISTERS
62 01100010	MOD REG R/M	BOUND	REG,EA	CHECK INDEX IN REG AGAINST BOUNDS AT EA
63 01100011		(not used)		
64 01100100		(not used)		
65 01100101		(not used)		
66 01100110		(not used)		
67 01100111		(not used)		
68 01101000		PUSH	DATA16	PUSH WORD DATA ON STACK
69 01101001	MOD REG R/M	IMUL	REG,EA,DATA16	MULTIPLY (EA) BY WORD DATA; SIGNED

Table A4-2 — iAPX *INTEL* 80186 Processor Instructions in Hex Order: 40 - 7F (Contd)

Opcode			Operands	Function
Hex/Binary		Instruction		
6A 01101010		PUSH	DATA8	PUSH BYTE DATA ON STACK, SIGN-EXTEND
6B 01101011	MOD REG R/M	IMUL	REG,EA,DATA8	MULTIPLY (EA) BY BYTE DATA; SIGNED
6C 01101100		INS	DST8	BYTE INPUT, STRING OP
6D 01101101		INS	DST16	WORD INPUT, STRING OP
6E 01101110		OUTS	DST8	BYTE OUTPUT, STRING OP
6F 01101111		OUTS	DST16	WORD OUTPUT, STRING OP
70 01110000		JO	DISP8	JUMP ON OVERFLOW
71 01110001		JNO	DISP8	JUMP ON NOT OVERFLOW
72 01110010		JC/JB/JNAE	DISP8	JUMP ON BELOW/NOT ABOVE OR EQUAL
73 01110011		JNC/JNB/JAE	DISP8	JUMP ON NOT BELOW/ABOVE OR EQUAL
74 01110100		JE/JZ	DISP8	JUMP ON EQUAL//ZERO
75 01110101		JNE/JNZ	DISP8	JUMP ON NOT EQUAL/NOT ZERO
76 01110110		JBE/JNA	DISP8	JUMP ON BELOW OR EQUAL/NOT ABOVE
77 01110111		JNBE/JA	DISP8	JUMP ON NOT BELOW OR EQUAL/ABOVE
78 01111000		JS	DISP8	JUMP ON SIGN
79 01111001		JNS	DISP8	JUMP ON NOT SIGN
7A 01111010		JP/JPE	DISP8	JUMP ON PARITY/PARITY EVEN
7B 01111011		JNP/JPO	DISP8	JUMP ON NOT PARITY/PARITY ODD
7C 01111100		JL/JNGE	DISP8	JUMP ON LESS/NOT GREATER OR EQUAL
7D 01111101		JNL/JGE	DISP8	JUMP ON NOT LESS/GREATER OR EQUAL
7E 01111110		JLE/JNG	DISP8	JUMP ON LESS OR EQUAL/NOT GREATER
7F 01111111		JNLE/JG	DISP8	JUMP ON NOT LESS OR EQUAL/GREATER

Table A4-3 — iAPX *INTEL* 80186 Processor Instructions in Hex Order: 80 - BF

Opcode			Operands	Function
Hex/Binary		Instruction		
80 10000000	MOD 000 R/M	ADD	EA,DATA8	BYTE ADD DATA TO EA
80 10000000	MOD 001 R/M	OR	EA,DATA8	BYTE OR DATA TO EA
80 10000000	MOD 010 R/M	ADC	EA,DATA8	BYTE ADD DATA W/CARRY TO EA
80 10000000	MOD 011 R/M	SBB	EA,DATA8	BYTE SUB DATA W/BORROW FROM EA
80 10000000	MOD 100 R/M	AND	EA,DATA8	BYTE AND DATA TO EA
80 10000000	MOD 101 R/M	SUB	EA,DATA8	BYTE SUBTRACT DATA FROM EA
80 10000000	MOD 110 R/M	XOR	EA,DATA8	BYTE XOR DATA TO EA
80 10000000	MOD 111 R/M	CMP	EA,DATA8	BYTE COMPARE DATA WITH (EA)
81 10000001	MOD 000 R/M	ADD	EA,DATA16	WORD ADD DATA TO EA
81 10000001	MOD 001 R/M	OR	EA,DATA16	WORD OR DATA TO EA
81 10000001	MOD 010 R/M	ADC	EA,DATA16	WORD ADD DATA W/CARRY TO EA
81 10000001	MOD 011 R/M	SBB	EA,DATA16	WORD SUB DATA W/BORROW FROM EA
81 10000001	MOD 100 R/M	AND	EA,DATA16	WORD AND DATA TO EA
81 10000001	MOD 101 R/M	SUB	EA,DATA16	WORD SUBTRACT DATA FROM EA
81 10000001	MOD 110 R/M	XOR	EA,DATA16	WORD XOR DATA TO EA
81 10000001	MOD 111 R/M	CMP	EA,DATA16	WORD COMPARE DATA WITH (EA)
82 10000010	MOD 000 R/M	ADD	EA,DATA8	BYTE ADD DATA TO EA
82 10000010	MOD 001 R/M	(not used)		
82 10000010	MOD 010 R/M	ADC	EA,DATA8	BYTE ADD DATA W/CARRY TO EA
82 10000010	MOD 011 R/M	SBB	EA,DATA8	BYTE SUB DATA W/BORROW FROM EA
82 10000010	MOD 100 R/M	(not used)		
82 10000010	MOD 101 R/M	SUB	EA,DATA8	BYTE SUBTRACT DATA FROM EA
82 10000010	MOD 110 R/M	(not used)		
82 10000010	MOD 111 R/M	CMP	EA,DATA8	BYTE COMPARE DATA WITH (EA)
83 10000011	MOD 000 R/M	ADD	EA,DATA8	WORD ADD DATA TO EA
83 10000011	MOD 001 R/M	(not used)		
83 10000011	MOD 010 R/M	ADC	EA,DATA8	WORD ADD DATA W/CARRY TO EA
83 10000011	MOD 011 R/M	SBB	EA,DATA8	WORD SUB DATA W/BORROW FROM EA
83 10000011	MOD 100 R/M	(not used)		
83 10000011	MOD 101 R/M	SUB	EA,DATA8	WORD SUBTRACT DATA FROM EA
83 10000011	MOD 110 R/M	(not used)		
83 10000011	MOD 111 R/M	CMP	EA,DATA8	WORD COMPARE DATA WITH (EA)
84 10000100	MOD REG R/M	TEST	EA,REG	BYTE TEST (EA) WITH (REG)
85 10000101	MOD REG R/M	TEST	EA,REG	WORD TEST (EA) WITH (REG)
86 10000110	MOD REG R/M	XCHG	REG,EA	BYTE EXCHANGE (REG) WITH (EA)
87 10000111	MOD REG R/M	XCHG	REG,EA	WORD EXCHANGE (REG) WITH (EA)
88 10001000	MOD REG R/M	MOV	EA,REG	BYTE MOVE (REG) TO EA
89 10001001	MOD REG R/M	MOV	EA,REG	WORD MOVE (REG) TO EA
8A 10001010	MOD REG R/M	MOV	REG,EA	BYTE MOVE (EA) TO (REG)
8B 10001011	MOD REG R/M	MOV	REG,EA	WORD MOVE (EA) TO (REG)

Table A4-3 — iAPX *INTEL* 80186 Processor Instructions in Hex Order: 80 - BF (Contd)

Opcode			Operands	Function
Hex/Binary		Instruction		
8C 10001100	MOD 0SR R/M	MOV	EA,SR	WORD MOVE (SEGMENT REG SR) TO EA
8C 10001100	MOD 1— R/M	(not used)		
8D 10001101	MOD REG R/M	LEA	REG,EA	LOAD EFFECTIVE ADDRESS OF EA TO REG
8E 10001110	MOD 0SB R/M	MOV	SR,EA	WORK MOVE (EA) TO SEGMENT REG SR
8E 10001110	MOD —- R/M	(not used)		
8F 10001111	MOD 000 R/M	POP	EA	POP STACK TO EA
8F 10001111	MOD 001 R/M	(not used)		
8F 10001111	MOD 010 R/M	(not used)		
8F 10001111	MOD 011 R/M	(not used)		
8F 10001111	MOD 100 R/M	(not used)		
8F 10001111	MOD 101 R/M	(not used)		
8F 10001111	MOD 110 R/M	(not used)		
8F 10001111	MOD 111 R/M	(not used)		
90 10010000		XCHG	AX,AX	EXCHANGE (AX) WITH (AX), (NOP)
91 10010001		XCHG	AX,CX	EXCHANGE (AX) WITH (CX)
92 10010010		XCHG	AX,DX	EXCHANGE (AX) WITH (DX)
93 10010011		XCHG	AX,BX	EXCHANGE (AX) WITH (BX)
94 10010100		XCHG	AX,SP	EXCHANGE (AX) WITH (SP)
95 10010101		XCHG	AX,BP	EXCHANGE (AX) WITH (BP)
96 10010110		XCHG	AX,SI	EXCHANGE (AX) WITH (SI)
97 10010111		XCHG	AX,DI	EXCHANGE (AX) WITH (DI)
98 10011000		CBW		BYTE CONVERT (AL) TO WORD (AX)
99 10011001		CWD		WORD CONVERT (AX) TO DOUBLE WORD
9A 10011010		CALL	DISP16,SEG16	DIRECT INTER SEGMENT CALL
9B 10011011		WAIT		WAIT FOR TEST SIGNAL
9C 10011100		PUSHF		PUSH FLAGS ON STACK
9D 10011101		POPF		POP STACK TO FLAGS
9E 10011110		SAHF		STORE (AH) INTO FLAGS
9F 10011111		LAHF		LOAD REG AH WITH FLAGS
A0 10100000		MOV	AL,ADDR16	BYTE MOVE (ADDR) TO REG AL
A1 10100001		MOV	AX,ADDR16	WORD MOVE (ADDR) TO REG AX
A2 10100010		MOV	ADDR16,AX	BYTE MOVE (AL) TO ADDR
A3 10100011		MOV	ADDR16,AX	WORD MOVE (AX) TO ADDR
A4 10100100		MOVS	DST8,SRC8	BYTE MOVE, STRING OP
A5 10100101		MOVS	DST16,SRC16	WORD MOVE, STRING OP
A6 10100110		CMPS	SIPTR,DIPTR	COMPARE BYTE, STRING OP
A7 10100111		CMPS	SIPTR,DIPTR	COMPARE WORD, STRING OP
A8 10101000		TEST	AL,DATA8	BYTE TEST (AL) WITH DATA
A9 10101001		TEST	AX,DATA16	WORD TEST (AX) WITH DATA

Table A4-3 — iAPX *INTEL* 80186 Processor Instructions in Hex Order: 80 - BF (Contd)

Opcode		Operands	Function
Hex/Binary	Instruction		
AA 10101010	STOS	DST8	BYTE STORE, STRING OP
AB 10101011	STOS	DST16	WORD STORE, STRING OP
AC 10101100	LODS	SRC8	BYTE LOAD, STRING OP
AD 10101101	LODS	SRC16	WORD LOAD, STRING OP
AE 10101110	SCAS	DIPTR8	BYTE SCAN, STRING OP
AF 10101111	SCAS	DIPTR16	WORD SCAN, STRING OP
B0 10110000	MOV	AL,DATA8	BYTE MOVE DATA TO REG AL
B1 10110001	MOV	CL,DATA8	BYTE MOVE DATA TO REG CL
B2 10110010	MOV	DL,DATA8	BYTE MOVE DATA TO REG DL
B3 10110011	MOV	BL,DATA8	BYTE MOVE DATA TO REG BL
B4 10110100	MOV	AH,DATA8	BYTE MOVE DATA TO REG AH
B5 10110101	MOV	CH,DATA8	BYTE MOVE DATA TO REG CH
B6 10110110	MOV	DH,DATA8	BYTE MOVE DATA TO REG DH
B7 10110111	MOV	BH,DATA8	BYTE MOVE DATA TO REG BH
B8 10111000	MOV	AX,DATA16	WORD MOVE DATA TO REG AX
B9 10111001	MOV	CX,DATA16	WORD MOVE DATA TO REG CX
BA 10111010	MOV	DX,DATA16	WORD MOVE DATA TO REG DX
BB 10111011	MOV	BX,DATA16	WORD MOVE DATA TO REG BX
BC 10111100	MOV	SP,DATA16	WORD MOVE DATA TO REG SP
BD 10111101	MOV	BP,DATA16	WORD MOVE DATA TO REG BP
BE 10111110	MOV	SI,DATA16	WORD MOVE DATA TO REG SI
BF 10111111	MOV	DI,DATA16	WORD MOVE DATA TO REG DI

Table A4-4 — iAPX *INTEL* 80186 Processor Instructions in Hex Order: C0 - FF

Opcode			Operands	Function
Hex/Binary		Instruction		
C0 11000000	MOD 000 R/M	ROL	EA,DATA8	BYTE ROTATE EA LEFT DATA8 BITS
C0 11000000	MOD 001 R/M	ROR	EA,DATA8	BYTE ROTATE EA RIGHT DATA8 BITS
C0 11000000	MOD 010 R/M	RCL	EA,DATA8	BYTE ROTATE EA LEFT THRU CARRY DATA8 BITS
C0 11000000	MOD 011 R/M	RCR	EA,DATA8	BYTE ROTATE EA RIGHT THRU CARRY DATA8 BITS
C0 11000000	MOD 100 R/M	SHL/SAL	EA,DATA8	BYTE SHIFT EA LEFT DATA8 BITS
C0 11000000	MOD 101 R/M	SHR	EA,DATA8	BYTE SHIFT EA RIGHT DATA8 BITS
C0 11000000	MOD 110 R/M	(not used)		
C0 11000000	MOD 111 R/M	SAR	EA,DATA8	BYTE SHIFT SIGNED EA RIGHT DATA8 BITS
C1 11000001	MOD 000 R/M	ROL	EA,DATA8	WORD ROTATE EA LEFT DATA8 BITS
C1 11000001	MOD 001 R/M	ROR	EA,DATA8	WORD ROTATE EA RIGHT DATA8 BITS
C1 11000001	MOD 010 R/M	RCL	EA,DATA8	WORD ROTATE EA LEFT THRU CARRY DATA8 BITS
C1 11000001	MOD 011 R/M	RCR	EA,DATA8	WORD ROTATE EA RIGHT THRU CARRY DATA8 BITS
C1 11000001	MOD 100 R/M	SHL/SAL	EA,DATA8	WORD SHIFT EA LEFT DATA8 BITS
C1 11000001	MOD 101 R/M	SHR	EA,DATA8	WORD SHIFT EA RIGHT DATA8 BITS
C1 11000001	MOD 110 R/M	(not used)		
C1 11000001	MOD 111 R/M	SAR	EA,DATA8	WORD SHIFT SIGNED EA RIGHT DATA8 BITS
C2 11000010		RET	DATA16	INTRA SEGMENT RETURN, ADD DATA TO REG SP
C3 11000011		RET		INTRA SEGMENT RETURN
C4 11000100	MOD REG R/M	LES	REG,EA	WORD LOAD REG AND SEGMENT REG ES
C5 11000101	MOD REG R/M	LDS	REG,EA	WORD LOAD REG AND SEGMENT REG DS
C6 11000110	MOD 000 R/M	MOV	EA,DATA8	BYTE MOVE DATA TO EA
C6 11000110	MOD 001 R/M	(not used)		
C6 11000110	MOD 010 R/M	(not used)		
C6 11000110	MOD 011 R/M	(not used)		
C6 11000110	MOD 100 R/M	(not used)		
C6 11000110	MOD 101 R/M	(not used)		
C6 11000110	MOD 110 R/M	(not used)		
C6 11000110	MOD 111 R/M	(not used)		
C7 11000111	MOD 000 R/M	MOV	EA,DATA16	WORD MOVE DATA TO EA
C7 11000111	MOD 001 R/M	(not used)		
C7 11000111	MOD 010 R/M	(not used)		
C7 11000111	MOD 011 R/M	(not used)		
C7 11000111	MOD 100 R/M	(not used)		
C7 11000111	MOD 101 R/M	(not used)		

Table A4-4 — iAPX *INTEL* 80186 Processor Instructions in Hex Order: C0 - FF (Contd)

Opcode			Operands	Function
Hex/Binary		Instruction		
C7 11000111	MOD 110 R/M	(not used)		
C7 11000111	MOD 111 R/M	(not used)		
C8 11001000		ENTER	DATA16,DATA8	PERFORM ENTER SEQUENCE
C9 11001001		LEAVE		PERFORM LEAVE SEQUENCE
CA 11001010		RET	DATA16	INTER SEGMENT RETURN, ADD DATA TO REG SP
CB 11001011		RET		INTER SEGMENT RETURN
CC 11001100		INT	3	TYPE 3 INTERRUPT
CD 11001101		INT	TYPE	TYPED INTERRUPT
CE 11001110		INTO		INTERRUPT ON OVERFLOW
CF 11001111		IRET		RETURN FROM INTERRUPT
D0 11010000	MOD 000 R/M	ROL	EA,1	BYTE ROTATE EA LEFT 1 BIT
D0 11010000	MOD 001 R/M	ROR	EA,1	BYTE ROTATE EA RIGHT 1 BIT
D0 11010000	MOD 010 R/M	RCL	EA,1	BYTE ROTATE EA LEFT 1 BIT
D0 11010000	MOD 011 R/M	RCR	EA,1	BYTE ROTATE EA RIGHT THRU CARRY 1 BIT
D0 11010000	MOD 100 R/M	SHL	EA,1	BYTE SHIFT EA LEFT 1 BIT
D0 11010000	MOD 101 R/M	SHR	EA,1	BYTE SHIFT EA RIGHT 1 BIT
D0 11010000	MOD 110 R/M	(not used)		
D0 11010000	MOD 111 R/M	SAR	EA,1	BYTE SHIFT SIGNED EA RIGHT 1 BIT
D1 11010001	MOD 000 R/M	ROL	EA,1	WORD ROTATE EA LEFT 1 BIT
D1 11010001	MOD 001 R/M	ROR	EA,1	WORD ROTATE EA RIGHT 1 BIT
D1 11010001	MOD 010 R/M	RCL	EA,1	WORD ROTATE EA LEFT THRU CARRY 1 BIT
D1 11010001	MOD 011 R/M	RCR	EA,1	WORD ROTATE EA RIGHT THRU CARRY 1 BIT
D1 11010001	MOD 100 R/M	SHL	EA,1	WORD SHIFT EA LEFT 1 BIT
D1 11010001	MOD 101 R/M	SHR	EA,1	WORD SHIFT EA RIGHT 1 BIT
D1 11010001	MOD 110 R/M	(not used)		
D1 11010001	MOD 111 R/M	SAR	EA,1	WORD SHIFT SIGNED EA RIGHT 1 BIT
D2 11010010	MOD 000 R/M	ROL	EA,CL	BYTE ROTATE EA LEFT (CL) BITS
D2 11010010	MOD 001 R/M	ROR	EA,CL	BYTE ROTATE EA RIGHT (CL) BITS
D2 11010010	MOD 010 R/M	RCL	EA,CL	BYTE ROTATE EA LEFT THRU CARRY (CL) BITS
D2 11010010	MOD 011 R/M	RCR	EA,CL	BYTE ROTATE EA RIGHT THRU CARRY (CL) BITS
D2 11010010	MOD 100 R/M	SHL	EA,CL	BYTE SHIFT EA LEFT (CL) BITS
D2 11010010	MOD 101 R/M	SHR	EA,CL	BYTE SHIFT EA RIGHT (CL) BITS
D2 11010010	MOD 110 R/M	(not used)		
D2 11010010	MOD 111 R/M	SAR	EA,CL	BYTE SHIFT SIGNED EA RIGHT (CL) BITS
D3 11010011	MOD 000 R/M	ROL	EA,CL	WORD ROTATE EA LEFT (CL) BITS
D3 11010011	MOD 001 R/M	ROR	EA,CL	WORD ROTATE EA RIGHT (CL) BITS

Table A4-4 — iAPX *INTEL* 80186 Processor Instructions in Hex Order: C0 - FF (Contd)

Opcode			Operands	Function
Hex/Binary		Instruction		
D3 11010011	MOD 010 R/M	RCL	EA,CL	WORD ROTATE EA LEFT THRU CARRY (CL) BITS
D3 11010011	MOD 011 R/M	RCR	EA,CL	WORD ROTATE EA RIGHT THRU CARRY (CL) BITS
D3 11010011	MOD 100 R/M	SHL	EA,CL	WORD SHIFT EA LEFT (CL) BITS
D3 11010011	MOD 101 R/M	SHR	EA,CL	WORD SHIFT EA RIGHT (CL) BITS
D3 11010011	MOD 110 R/M	(not used)		
D3 11010011	MOD 111 R/M	SAR	EA,CL	WORD SHIFT SIGNED EA RIGHT (CL) BITS
D4 11010100	00001010	AAM		ASCII ADJUST FOR MULTIPLY
D5 11010101	00001010 R/M	AAD		ASCII ADJUST FOR DIVIDE
D6 11010110		(not used)		
D7 11010111		XLAT	TABLE	TRANSLATE USING (BX)
D8 11011—	MOD— R/M	ESC	EA	ESCAPE TO EXTERNAL DEVICE
D8 11011000	MOD 000 R/M	FADD	Short-real	ADD 4-BYTE EA TO ST
D8 11011000	MOD 001 R/M	FMUL	Short-real	MULTIPLY ST BY 4-BYTE EA
D8 11011000	MOD 010 R/M	FCOM	Short-real	COMPARE 4-BYTE EA WITH ST
D8 11011000	MOD 011 R/M	FCOMP	Short-real	COMPARE 4-BYTE EA WITH ST AND POP
D8 11011000	MOD 100 R/M	FSUB	Short-real	SUBTRACT 4-BYTE EA FROM ST
D8 11011000	MOD 101 R/M	FSUBR	Short-real	SUBTRACT ST FROM 4-BYTE EA
D8 11011000	MOD 110 R/M	FDIV	Short-real	DIVIDE ST BY 4-BYTE EA
D8 11011000	MOD 111 R/M	FDIVR	Short-real	DIVIDE 4-BYTE EA BY ST
D8 11011000	1 1 000 (i)	FADD	ST,ST(i)	ADD ELEMENT TO ST
D8 11011000	1 1 001 (i)	FMUL	ST,ST(i)	MULTIPLY ST BY ELEMENT
D8 11011000	1 1 010 (i)	FCOM	ST(i)	COMPARE ST(i) WITH ST
D8 11011000	1 1 011 (i)	FCOMP	ST(i)	COMPARE ST(i) WITH ST AND POP
D8 11011000	1 1 100 (i)	FSUB	ST,ST(i)	SUBTRACT ELEMENT FROM ST
D8 11011000	1 1 101 (i)	FSUBR	ST,ST(i)	SUBTRACT ST FROM STACK ELEMENT
D8 11011000	1 1 110 (i)	FDIV	ST,ST(i)	DIVIDE ST BY ELEMENT
D8 11011000	1 1 111 (i)	FDIVR	ST,ST(i)	DIVIDE ST(i) BY ST
D9 11011001	MOD 000 R/M	FLD	Short-real	PUSH 4-BYTE EA TO ST
D9 11011001	MOD 001 R/M	(not used)		
D9 11011001	MOD 010 R/M	FST	Short-real	STORE 4-BYTE REAL TO EA
D9 11011001	MOD 011 R/M	FSTP	Short-real	STORE 4-BYTE REAL TO EA AND POP
D9 11011001	MOD 100 R/M	FLDENV	14-BYTES	LOAD 8087 ENVIRONMENT FROM EA
D9 11011001	MOD 101 R/M	FLDCW	2-BYTES	LOAD CONTROL WORK FROM EA
D9 11011001	MOD 110 R/M	FSTENV	14-BYTES	STORE 8087 ENVIRONMENT INTO EA
D9 11011001	MOD 111 R/M	FSTCW	2-BYTES	STORE CONTROL WORD INTO EA
D9 11011001	1 1 000 (i)	FLD	ST(i)	PUSH ST(i) ONTO ST
D9 11011001	1 1 001 (i)	FXCH	ST(i)	EXCHANGE ST AND ST(i)
D9 11011001	1 1 010 000	FNOP		STORE ST IN ST

Table A4-4 — iAPX *INTEL* 80186 Processor Instructions in Hex Order: C0 - FF (Contd)

Opcode		Instruction	Operands	Function
Hex/Binary				
D9 11011001	1 1 010 001	(not used)		
D9 11011001	1 1 010 01-	(not used)		
D9 11011001	1 1 010 1—	(not used)		
D9 11011001	1 1 011 (i)	*(1)		
D9 11011001	1 1 100 000	FCHS		CHANGE SIGN OF ST
D9 11011001	1 1 100 001	FABS		TAKE ABSOLUTE VALUE OF ST
D9 11011001	1 1 100 01-	(not used)		
D9 11011001	1 1 100 100	FTST		TEST ST AGAINST 0.0
D9 11011001	1 1 100 101	FXAM		EXAMINE ST AND REPORT CONDITION CODE
D9 11011001	1 1 100 11-	(not used)		
D9 11011001	1 1 101 000	FLD1		PUSH +1.0 TO ST
D9 11011001	1 1 101 001	FLDL2T		PUSH log 10 TO ST
D9 11011001	1 1 101 010	FLDL2E		PUSH loge TO ST
D9 11011001	1 1 101 011	FLDP1		PUSH Pi TO ST
D9 11011001	1 1 101 100	FLDLG2		PUSH log TO ST
D9 11011001	1 1 101 101	FLDLN2		PUSH loge TO ST
D9 11011001	1 1 101 110	FLDZ		PUST ZERO TO ST
D9 11011001	1 1 101 111	(not used)		
D9 11011001	1 1 110 000	F2XM1		CALCULATE 2x – 1
D9 11011001	1 1 110 001	FYL2X		CALCULATE FUNCTION Y*logX
D9 11011001	1 1 110 010	FPTAN		CALCULATE TAN OF ANGLE AS A RATIO
D9 11011001	1 1 110 011	FPATAN		CALCULATE ARCTAN OF ANGLE
D9 11011001	1 1 110 100	FXTRACT		EXTRACT EXPONENT AND SIGNIFICANT FROM ST VALUE
D9 11011001	1 1 110 101	(not used)		
D9 11011001	1 1 110 110	FDECSTP		DECREMENT STACK POINTER IN STATUS WORD
D9 11011001	1 1 110 111	FINCSTP		INCREMENT STACK POINTER IN STATUS WORD
D9 11011001	1 1 111 000	FPREM		MODULO DIVISION OF ST BY ST(1)
D9 11011001	1 1 110 001	FYL2XP1		CALCULATE VALUE OF Y*log(X+1)
D9 11011001	1 1 111 010	FSORT		CALCULATE SQUARE ROOT OF ST
D9 11011001	1 1 111 011	(not used)		
D9 11011001	1 1 111 100	FRNDINT		ROUND ST TO INTEGER
D9 11011001	1 1 111 101	FSCALE		ADD ST(1) TO EXPONENT OF ST
D9 11011001	1 1 111 11-	(not used)		
DA 11011010	MOD 000 R/M	FIADD	Short-integer	ADD 4-BYTE INTEGER EA TO ST
DA 11011010	MOD 001 R/M	FIMUL	Short-integer	MULTIPLY ST BY 4-BYTE INTEGER EA
DA 11011010	MOD 010 R/M	FICOM	Short-integer	CONVERT 4-BYTE INTEGER EA, AND COMPARE WITH ST
DA 11011010	MOD 011 R/M	FICOMP	Short-integer	CONVERT 4-BYTE INTEGER EA, COMPARE WITH ST, POP

Table A4-4 — iAPX *INTEL* 80186 Processor Instructions in Hex Order: C0 - FF (Contd)

Opcode			Operands	Function
Hex/Binary		Instruction		
DA 11011010	MOD 100 R/M	FISUB	Short-integer	SUBTRACT 4-BYTE INTEGER EA FROM ST
DA 11011010	MOD 101 R/M	FISUBR	Short-integer	SUBTRACT ST FROM 4-BYTE INTEGER EA
DA 11011010	MOD 110 R/M	FIDIV	Short-integer	DIVIDE ST BY 4-BYTE INTEGER EA
DA 11011010	MOD 111 R/M	FIDIVR	Short-integer	DIVIDE 4-BYTE INTEGER EA BY ST
DA 11011010	1 1—	(not used)		
DB 11011011	MOD 000 R/M	FILD	Short-integer	PUSH 4-BYTE INTEGER EA ONTO ST
DB 11011011	MOD 001 R/M	(not used)		
DB 11011011	MOD 010 R/M	FIST	Short-integer	STORE ROUNDED ST IN 4-BYTE INTEGER EA
DB 11011011	MOD 011 R/M	FISTP	Short-integer	STORE ROUNDED ST IN 4-BYTE INTEGER EA, POP
DB 11011011	MOD 100 R/M	(not used)		
DB 11011011	MOD 101 R/M	FLD	Temp-real	PUSH 10-BYTE EA ONTO ST
DB 11011011	MOD 110 R/M	Reserved		
DB 11011011	MOD 111 R/M	FSTP	Temp-real	STORE ST INTO 10-BYTE EA, POP
DB 11011011	1 1 0—	Reserved		
DB 11011011	1 1 100 000	FENI		ENABLE INTERRUPT
DB 11011011	1 1 100 001	FDISI		DISABLE INTERRUPT
DB 11011011	1 1 100 010	FCLEX		CLEAR EXCEPTIONS
DB 11011011	1 1 100 011	FINIT		INITIALIZE PROCESSOR
DB 11011011	1 1 100 1—	Reserved		
DB 11011011	1 1 101 —	Reserved		
DB 11011011	1 1 11—	Reserved		
DC 11011100	MOD 000 R/M	FADD	Long-real	ADD 8-BYTE EA TO ST
DC 11011100	MOD 001 R/M	FMUL	Long-real	MULTIPLY ST BY 8-BYTE EA
DC 11011100	MOD 010 R/M	FCOM	Long-real	COMPARE ST WITH 8-BYTE EA
DC 11011100	MOD 011 R/M	FCOMP	Long-real	COMPARE ST WITH 8-BYTE EA, POP STACK
DC 11011100	MOD 100 R/M	FSUB	Long-real	SUBTRACT 8-BYTE EA FROM ST
DC 11011100	MOD 101 R/M	FSUBR	Long-real	SUBTRACT ST FROM 8-BYTE EA
DC 11011100	MOD 110 R/M	FDIV	Long-real	DIVIDE ST BY 8-BYTE EA
DC 11011100	MOD 111 R/M	FDIVR	Long-real	DIVIDE 8-BYTE EA BY ST
DC 11011100	1 1 000 (i)	FADD	ST(i),ST	ADD ST TO ELEMENT
DC 11011100	1 1 001 (i)	FMUL	ST(i),ST	MULTIPLY ELEMENT BY ST
DC 11011100	1 1 010 (i)	*(2)		
DC 11011100	1 1 011 (i)	*(3)		
DC 11011100	1 1 100 (i)	FSUBR	ST(i),ST	SUBTRACT ST FROM ELEMENT
DC 11011100	1 1 101 (i)	FSUB	ST(i),ST	SUBTRACT ELEMENT FROM ST
DC 11011100	1 1 110 (i)	FDIVR	ST(i),ST	DIVIDE ST(i) BY ST
DC 11011100	1 1 111 (i)	FDIV	ST(i),ST	DIVIDE ST BY ST(i)
DD 11011101	MOD 000 R/M	FLD	Long-real	PUSH 8-BYTE EA ONTO ST

Table A4-4 — iAPX *INTEL* 80186 Processor Instructions in Hex Order: C0 - FF (Contd)

Opcode			Operands	Function
Hex/Binary		Instruction		
DD 11011101	MOD 001 R/M	Reserved		
DD 11011101	MOD 010 R/M	FST	Long-real	STORE ST INTO 8-BYTE EA
DD 11011101	MOD 011 R/M	FSTP	Long-real	STORE ST INTO 8-BYTE EA, POP
DD 11011101	MOD 100 R/M	FRSTOR	94-BYTES	RESTORE 8087 STATE FROM EA
DD 11011101	MOD 101 R/M	Reserved		
DD 11011101	MOD 110 R/M	FSAVE	94-BYES	SAVE 8087 STATE TO EA
DD 11011101	MOD 111 R/M	FSTSW	2-BYTES	STORE 8087 STATUS WORD TO 2-BYTE EA
DD 11011101	1 1 000 (i)	FFREE	ST(i)	SET STACK TAG TO "EMPTY"
DD 11011101	1 1 001 (i)	*(4)		
DD 11011101	1 1 010 (i)	FST	ST(i)	STORE ST INTO ST(i)
DD 11011101	1 1 011 (i)	FSTP	ST(i)	STORE ST INTO ST(i), POP
DD 11011101	1 1 11—	Reserved		
DE 11011110	MOD 000 R/M	FIADD	Word-integer	ADD 2-BYTE INTEGER EA TO ST
DE 11011110	MOD 001 R/M	FIMUL	Word-integer	MULTIPLY ST BY 2-BYTE INTEGER EA
DE 11011110	MOD 010 R/M	FICOM	Word-integer	COMPARE 2-BYTE EA INTEGER WITH ST
DE 11011110	MOD 011 R/M	FICOMP	Word-integer	COMPARE 2-BYTE INTEGER EA WITH ST, POP
DE 11011110	MOD 100 R/M	FISUB	Word-integer	SUBTRACT 2-BYTE INTEGER EA FROM ST
DE 11011110	MOD 101 R/M	FISUBR	Word-integer	SUBTRACT ST FROM 2-BYTE INTEGER EA
DE 11011110	MOD 110 R/M	FIDIV	Word-integer	DIVIDE ST BY 2-BYTE INTEGER EA
DE 11011110	MOD 111 R/M	FIDIVR	Word-integer	DIVIDE 2-BYTE INTEGER EA BY ST
DE 11011110	1 1 000 (i)	FADDP	ST(i),ST	ADD ST TO ELEMENT, POP
DE 11011110	1 1 001 (i)	FMULP	ST(i),ST	MULTIPLY ST BY ELEMENT, POP
DE 11011110	1 1 010 —-	*(5)		
DE 11011110	1 1 011 000	Reserved		
DE 11011110	1 1 011	FCOMPP		COMPARE ST WITH ST(1), POP TWICE
DE 11011110	1 1 011 01-	Reserved		
DE 11011110	1 1 011 1—	Reserved		
DE 11011110	1 1 100 (i)	FSUBRP	ST(i),ST	SUBTRACT ST FROM ELEMENT, POP
DE 11011110	1 1 101 (i)	FSUBP	ST(i),ST	SUBTRACT ST(i) FROM ST, POP
DE 11011110	1 1 110 (i)	FDIVRP	ST(i),ST	DIVIDE STACK ELEMENT BY ST, POP
DE 11011110	1 1 111 (i)	FDIVP	ST(i),ST	DIVIDE ST BY STACK ELEMENT, POP
DF 11011111	MOD 000 R/M	FILD	Word-integer	CONVERT 2-BYTE EA AND PUSH ONTO STACK
DF 11011111	MOD 001 R/M	Reserved		
DF 11011111	MOD 010 R/M	FIST	Word-integer	ROUND ST AND STORE IN 2-BYTE INTEGER EA

Table A4-4 — iAPX *INTEL* 80186 Processor Instructions in Hex Order: C0 - FF (Contd)

Opcode			Operands	Function
Hex/Binary		Instruction		
DF 11011111	MOD 011 R/M	FISTP	Word-integer	ROUND ST, STORE IN 2-BYTE INTEGER EA, POP
DF 11011111	MOD 100 R/M	FBLD	Packed decimal	LOAD BCD TO ST
DF 11011111	MOD 101 R/M	FILD	Long-integer	CONVERT 8-BYTE INTEGER EA AND PUSH ONTO STACK
DF 11011111	MOD 110 R/M	FBSTP	Packed decimal	CONVERT ST, STORE IN 10-BYTE BCD EA, POP
DF 11011111	MOD 111 R/M	FISTP	Long-integer	ROUND ST, STORE IN 8-BYTE INTEGER EA, POP
DF 11011111	1 1 000 (i)	*(6)		
DF 11011111	1 1 001 (i)	*(7)		
DF 11011111	1 1 010 (i)	*(8)		
DF 11011111	1 1 011 (i)	*(9)		
DF 11011111	1 1 —- —-	Reserved		
E0 11100000		LOOPNZ/LOOPNE	DISP8	LOOP (CX) TIMES WHILE NOT ZERO/NOT EQUAL
E1 11100001		LOOPZ/LOOPE	DISP8	LOOP (CX) TIMES WHILE ZERO/EQUAL
E2 11100010		LOOP	DISP8	LOOP (CX) TIMES
E3 11100011		JCXZ	DISP8	JUMP ON (CX)=0
E4 11100100		IN	AL,PORT	BYTE INPUT FROM PORT TO REG AL
E5 11100101		IN	AX,PORT	WORD INPUT FROM PORT TO REG AX
E6 11100110		OUT	PORT,AL	BYTE OUTPUT (AL) TO PORT
E7 11100111		OUT	PORT,AX	WORD OUTPUT (AX) TO PORT
E8 11101000		CALL	DISP16	DIRECT INTRA SEGMENT CALL
E9 11101001		JMP	DISP16	DIRECT INTRA SEGMENT JUMP
EA 11101010		JMP	DISP16,SEG16	DIRECT INTER SEGMENT JUMP
EB 11101010		JMP	DISP8	DIRECT INTRA SEGMENT JUMP
EC 11101010		IN	AL,DX	BYTE INPUT FROM PORT (DX) TO REG AL
ED 11101010		IN	AX,DX	WORD INPUT FROM PORT (DX) TO REG AX
EE 11101010		OUT	DX,AL	BYTE OUTPUT (AL) TO PORT (DX)
EF 11101010		OUT	DX,AX	WORD OUTPUT (AX) TO PORT (DX)
F0 11110000		LOCK		BUS LOCK PREFIX
F1 11110001		(not used)		
F2 11110010		REPNZ/REPNE		REPEAT WHILE (CX)0 AND (ZF)=0
F3 11110011		REPZ/REPE/REP		REPEAT WHILE (CX)0 AND (ZF)=1
F4 11110100		HLT		HALT
F5 11110101		CMC		COMPLEMENT CARRY FLAG
F6 11110110	MOD 000 R/M	TEST	EA,DATA8	BYTE TEST (EA) WITH DATA
F6 11110110	MOD 001 R/M	(not used)		
F6 11110110	MOD 010 R/M	NOT	EA	BYTE INVERT EA
F6 11110110	MOD 011 R/M	NEG	EA	BYTE NEGATE EA

Table A4-4 — iAPX *INTEL* 80186 Processor Instructions in Hex Order: C0 - FF (Contd)

Opcode			Operands	Function
Hex/Binary		Instruction		
F6 11110110	MOD 100 R/M	MUL	EA	BYTE MULTIPLY BY (EA), UNSIGNED
F6 11110110	MOD 101 R/M	IMUL	EA	BYTE MULTIPLY BY (EA), SIGNED
F6 11110110	MOD 110 R/M	DIV	EA	BYTE DIVIDE BY (EA), UNSIGNED
F6 11110110	MOD 111 R/M	IDIV	EA	BYTE DIVIDE BY (EA), SIGNED
F7 11110111	MOD 000 R/M	TEST	EA,DATA16	WORD TEST (EA) WITH DATA
F7 11110111	MOD 001 R/M	(not used)		
F7 11110111	MOD 010 R/M	NOT	EA	WORD INVERT EA
F7 11110111	MOD 011 R/M	NEG	EA	WORD NEGATE EA
F7 11110111	MOD 100 R/M	MUL	EA	WORD MULTIPLY BY (EA), UNSIGNED
F7 11110111	MOD 101 R/M	IMUL	EA	WORD MULTIPLY BY (EA), SIGNED
F7 11110111	MOD 110 R/M	DIV	EA	WORD DIVIDE BY (EA), UNSIGNED
F7 11110111	MOD 111 R/M	IDIV	EA	WORD DIVIDE BY (EA), SIGNED
F8 11111000		CLC		CLEAR CARRY FLAG
F9 11111001		STC		SET CARRY FLAG
FA 11111010		CLI		CLEAR INTERRUPT FLAG
FB 11111011		STI		SET INTERRUPT FLAG
FC 11111100		CLD		CLEAR DIRECTION FLAG
FD 11111101		STD		SET DIRECTION FLAG
FE 11111110	MOD 000 R/M	INC	EA	BYTE INCREMENT EA
FE 11111110	MOD 001 R/M	DEC	EA	BYTE DECREMENT EA
FE 11111110	MOD 010 R/M	(not used)		
FE 11111110	MOD 011 R/M	(not used)		
FE 11111110	MOD 100 R/M	(not used)		
FE 11111110	MOD 101 R/M	(not used)		
FE 11111110	MOD 110 R/M	(not used)		
FE 11111110	MOD 111 R/M	(not used)		
FF 11111111	MOD 000 R/M	INC	EA	WORD INCREMENT EA
FF 11111111	MOD 001 R/M	DEC	EA	WORD DECREMENT EA
FF 11111111	MOD 010 R/M	CALL	EA	INDIRECT INTRA SEGMENT CALL
FF 11111111	MOD 011 R/M	CALL	EA	INDIRECT INTRA SEGMENT CALL
FF 11111111	MOD 100 R/M	JMP	EA	INDIRECT INTRA SEGMENT JUMP
FF 11111111	MOD 101 R/M	JMP	EA	INDIRECT INTRA SEGMENT JUMP
FF 11111111	MOD 110 R/M	PUSH	EA	PUSH (EA) ON STACK
FF 11111111	MOD 111 R/M	(not used)		

REG is assigned according to the following table:

16–BIT (W=1)		8–BIT (W=0)		SEGMENT REG	
000	AX	000	AL	00	ES
001	CX	001	CL	01	ES
010	DX	010	DL	10	SS
011	BX	011	BL	11	DS
100	SP	100	AH		
101	BP	101	CH		
110	SI	110	DH		
111	DI	111	BH		

EA is computed as follows: (DISP8 sign extended to 16 bits)

00 000	(BX)+(SI)	DS
00 001	(BX)+(DI)	DS
00 010	(BP+(SI)	SS
00 011	(BP)+(DI)	SS
00 100	(SI)	DS
00 101	(DI)	DS
00 110	DISP16 (DIRECT ADDRESS)	DS
00 111	(BX)	DS
01 000	(BX)+(SI)+DISP8	DS
01 001	(BS)+(DI)+DISP8	DS
01 010	(BP)+(SI)+DISP8	SS
01 011	(BP)+(DL)+DISP8	SS
01 100	(SI)+DISP8	DS
01 101	(DI)+DISP8	DS
01 110	(BP)+DISP8	SS
01 111	(BX)+DISP8	DS
10 000	(BX)+(SI)+DISP16	DS
10 001	(BX)+(DI)+DISP16	DS
10 010	(BP)+(SI)+DISP16	SS
10 011	(BP)+(DI)+DISP16	SS
10 100	(SI)+DISP16	DS
10 101	(DI)+DISP16	DS
10 110	(BP)+DISP16	SS
10 111	(BX)+DISP16	DS
11 000	REG AX/AL	
11 001	REG CX/CL	
11 010	REG DX/DL	
11 011	REG BX/BL	
11 100	REG SP/AH	
11 101	REG BP/CH	
11 110	REG SI/DH	

11 111	REG DI/BH	

FLAGS register contains:

X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)

A5. *PowerPC*[1] PROCESSOR FAMILY INSTRUCTION LIST (BY MNEMONIC)

The information in this appendix was reproduced from documentation at Motorola's `www.mot-sps.com/products/index.html` web site.

The following sections include an instruction field summary, a list of split-field notation and conventions, and the entire *PowerPC* instruction set sorted by mnemonic.

A5.1 Instruction Field Summary

Table A5-1 describes the instruction fields used in the various instruction formats.

Table A5-1 — Instruction Syntax Conventions

Field	Description
AA (30)	Absolute address bit. 0 The immediate field represents an address relative to the current instruction address (CIA). The effective (logical) address of the branch is either the sum of the LI field sign-extended to 64 bits and the address of the branch instruction or the sum of the BD field sign-extended to 64 bits and the address of the branch instruction. 1 The immediate field represents an absolute address. The effective address (EA) of the branch is the LI field sign-extended to 64 bits or the BD field sign-extended to 64 bits. ***Note:*** The LI and BD fields are sign-extended to 32 bits in 32–bit implementations.
BD (16–29)	Immediate field specifying a 14–bit signed two's complement branch displacement that is concatenated on the right with 0b00 and sign-extended to 64 bits (32 bits in 32–bit implementations).
BI (11–15)	Field used to specify a bit in the CR to be used as the condition of a branch conditional instruction.
BO (6–10)	Field used to specify options for the branch conditional instructions.
crbA (11–15)	Field used to specify a bit in the CR to be used as a source.
crbB (16–20)	Field used to specify a bit in the CR to be used as a source.
crbD (6–10)	Field used to specify a bit in the CR, or in the FPSCR, as the destination of the result of an instruction.
crfD (6–8)	Field used to specify one of the CR fields, or one of the FPSCR fields, as a destination.
crfS (11–13)	Field used to specify one of the CR fields, or one of the FPSCR fields, as a source.
CRM (12–19)	Field mask used to identify the CR fields that are to be updated by the **mtcrf** instruction.
d (16–31)	Immediate field specifying a 16–bit signed two's complement integer that is sign-extended to 64 bits (32 bits in 32–bit implementations).

1. Trademark of International Business Machines Corporation.

Table A5-1 — Instruction Syntax Conventions (Contd)

Field	Description
ds (16–29)	Immediate field specifying a 14–bit signed two's complement integer which is concatenated on the right with 0b00 and sign-extended to 64 bits. This field is defined in 64–bit implementations only.
FM (7–14)	Field mask used to identify the FPSCR fields that are to be updated by the **mtfsf** instruction.
frA (11–15)	Field used to specify an FPR as a source.
frB (16–20)	Field used to specify an FPR as a source.
frC (21–25)	Field used to specify an FPR as a source.
frD (6–10)	Field used to specify an FPR as the destination.
frS (6–10)	Field used to specify an FPR as a source.
IMM (16–19)	Immediate field used as the data to be placed into a field in the FPSCR.
L (10)	Field used to specify whether an integer compare instruction is to compare 64–bit numbers or 32–bit numbers. This field is defined in 64–bit implementations only.
LI (6–29)	Immediate field specifying a 24–bit signed two's complement integer that is concatenated on the right with 0b00 and sign-extended to 64 bits (32 bits in 32–bit implementations).
LK (31)	Link bit. 0 Does not update the link register (LR). 1 Updates the LR. If the instruction is a branch instruction, the address of the instruction following the branch instruction is placed into the LR.
MB (21–25) and ME (26–30)	Fields used in rotate instructions to specify a 64–bit mask (32 bits in 32–bit implementations) consisting of 1 bits from bit MB + 32 through bit ME + 32 inclusive, and 0 bits elsewhere.
NB (16–20)	Field used to specify the number of bytes to move in an immediate string load or store.
OE (21)	Used for extended arithmetic to enable setting OV and SO in the XER.
OPCD (0–5)	Primary opcode field.
rA (11–15)	Field used to specify a GPR to be used as a source or destination.
rB (16–20)	Field used to specify a GPR to be used as a source.

Table A5-1 — Instruction Syntax Conventions (Contd)

Field	Description
Rc (31)	Record bit. 0 Does not update the condition register (CR). 1 Updates the CR to reflect the result of the operation. For integer instructions, CR bits 0–2 are set to reflect the result as a signed quantity and CR bit 3 receives a copy of the summary overflow bit, XER[SO]. The result as an unsigned quantity or a bit string can be deduced from the EQ bit. For floating-point instructions, CR bits 4–7 are set to reflect floating-point exception, floating-point enabled exception, floating-point invalid operation exception, and floating-point overflow exception. (Note that the architecture specification refers to exceptions also as interrupts.)
rD (6–10)	Field used to specify a GPR to be used as a destination.
rS (6–10)	Field used to specify a GPR to be used as a source.
SH (16–20)	Field used to specify a shift amount.
SIMM (16–31)	Immediate field used to specify a 16–bit signed integer.
SR (12–15)	Field used to specify one of the 16 segment registers (32–bit implementation only).
TO (6–10)	Field used to specify the conditions on which to trap.
UIMM (16–31)	Immediate field used to specify a 16–bit unsigned integer.
XO (21–29, 21–30, 22–30, 26–30, 27–29, 27–30, or 30–31)	Extended opcode field. Bits 21–29, 27–29, 27–30, 30–31 pertain to 64–bit implementations only.

Split fields — mb, me, sh, spr, and tbr — are described in Table A5-2.

Table A5-2 — Split-Field Notation and Conventions

Field	Description
mb (21–26)	Field used in rotate instructions to specify the first 1 bit of a 64–bit mask (32 bits in 32–bit implementations). This field is defined in 64–bit implementations only.
me (21–26)	Field used in rotate instructions to specify the last 1 bit of a 64–bit mask (32 bits in 32–bit implementations). This field is defined in 64–bit implementations only.
sh (16–20) and sh (30)	Fields used to specify a shift amount (64–bit implementations only).
spr (11–20)	Field used to specify a special purpose register for the **mtspr** and **mfspr** instructions.
tbr (11–20)	Field used to specify either the time base lower (TBL) or time base uppper (TBU).

A5.2 *PowerPC* Instruction Set Listings

This section lists the *PowerPC* architecture's instruction set. Instructions are sorted by mnemonic. Split fields, that represent the concatenation of sequences from left to right, are shown in lowercase. Note that the MPC750 is a 32–bit microprocessor, and doesn't implement any 64–bit instructions.

Table A5-3 lists the instructions implemented in the *PowerPC* architecture in alphabetical order by mnemonic.

Key: **0 0 0 0 0*** denotes reserved bits.

Table A5-3 — Complete Instruction List Sorted by Mnemonic

Name	0–5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
add*x*	31	D					A					B					OE	266								Rc
addc*x*	31	D					A					B					OE	10								Rc
adde*x*	31	D					A					B					OE	138								Rc
addi	14	D					A					SIMM															
addic	12	D					A					SIMM															
addic.	13	D					A					SIMM															
addis	15	D					A					SIMM															
addme*x*	31	D					A					**0 0 0 0 0***					OE	234								Rc
addze*x*	31	D					A					**0 0 0 0 0***					OE	202								Rc
and*x*	31	S					A					B					28									Rc
andc*x*	31	S					A					B					60									Rc
andi.	28	S					A					UIMM															
andis.	29	S					A					UIMM															
b*x*	18	LI																							AA	LK	
bc*x*	16	BO					BI					BD													AA	LK	
bcctr*x*	19	BO					BI					**0 0 0 0 0***					528									LK	
bclr*x*	19	BO					BI					**0 0 0 0 0***					16									LK	
cmp	31	crfD			**0***	L	A					B					0										**0***
cmpi	11	crfD			**0***	L	A					SIMM															
cmpl	31	crfD			**0***	L	A					B					32										**0***
cmpli	10	crfD			**0***	L	A					UIMM															
cntlzd*x*[d]	31	S					A					**0 0 0 0 0***					58									Rc
cntlzw*x*	31	S					A					**0 0 0 0 0***					26									Rc
crand	19	crbD					crbA					crbB					257										**0***
crandc	19	crbD					crbA					crbB					129										**0***
creqv	19	crbD					crbA					crbB					289										**0***
crnand	19	crbD					crbA					crbB					225										**0***
crnor	19	crbD					crbA					crbB					33										**0***
cror	19	crbD					crbA					crbB					449										**0***
crorc	19	crbD					crbA					crbB					417										**0***
crxor	19	crbD					crbA					crbB					193										**0***
dcba[e,h]	31	**0 0 0 0 0***					A					B					758										**0***
dcbf	31	**0 0 0 0 0***					A					B					86										**0***
dcbi[a]	31	**0 0 0 0 0***					A					B					470										**0***
dcbst	31	**0 0 0 0 0***					A					B					54										**0***
dcbt	31	**0 0 0 0 0***					A					B					278										**0***
dcbtst	31	**0 0 0 0 0***					A					B					246										**0***
dcbz	31	**0 0 0 0 0***					A					B					1014										**0***
divd*x*[d]	31	D					A					B					OE	489								Rc
divdu*x*[d]	31	D					A					B					OE	457								Rc
divw*x*	31	D					A					B					OE	491								Rc
divwu*x*	31	D					A					B					OE	459								Rc
eciwx	31	D					A					B					310										**0***
ecowx	31	S					A					B					438										**0***
eieio	31	**0 0 0 0 0***					**0 0 0 0 0***					**0 0 0 0 0***					854										**0***

See note(s) at end of table.

Table A5-3 — Complete Instruction List Sorted by Mnemonic (Contd)

Name	0–5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
eqv*x*	31	S					A					B					284										Rc
extsb*x*	31	S					A					**0 0 0 0 0***					954										Rc
extsh*x*	31	S					A					**0 0 0 0 0***					922										Rc
extsw*x*[d]	31	S					A					**0 0 0 0 0***					986										Rc
fabs*x*	63	D					**0 0 0 0 0***					B					264										Rc
fadd*x*	63	D					A					B					**0 0 0 0 0***					21				Rc
fadds*x*	59	D					A					B					**0 0 0 0 0***					21				Rc
fcfid*x*[d]	63	D					**0 0 0 0 0***					B					846										Rc
fcmpo	63	crfD			**0 0***		A					B					32										**0***
fcmpu	63	crfD			**0 0***		A					B					0										**0***
fctid*x*[d]	63	D					**0 0 0 0 0***					B					814										Rc
fctidz*x*[d]	63	D					**0 0 0 0 0***					B					815										Rc
fctiw*x*	63	D					**0 0 0 0 0***					B					14										Rc
fctiwz*x*	63	D					**0 0 0 0 0***					B					15										Rc
fdiv*x*	63	D					A					B					**0 0 0 0 0***					18				Rc
fdivs*x*	59	D					A					B					**0 0 0 0 0***					18				Rc
fmadd*x*	63	D					A					B					C					29				Rc
fmadds*x*	59	D					A					B					C					29				Rc
fmr*x*	63	D					**0 0 0 0 0***					B					72										Rc
fmsub*x*	63	D					A					B					C					28				Rc
fmsubs*x*	59	D					A					B					C					28				Rc
fmul*x*	63	D					A					**0 0 0 0 0***					C					25				Rc
fmuls*x*	59	D					A					**0 0 0 0 0***					C					25				Rc
fnabs*x*	63	D					**0 0 0 0 0***					B					136										Rc
fneg*x*	63	D					**0 0 0 0 0***					B					40										Rc
fnmadd*x*	63	D					A					B					C					31				Rc
fnmadds*x*	59	D					A					B					C					31				Rc
fnmsub*x*	63	D					A					B					C					30				Rc
fnmsubs*x*	59	D					A					B					C					30				Rc
fres*x*[e]	59	D					**0 0 0 0 0***					B					**0 0 0 0 0***					24				Rc
frsp*x*	63	D					**0 0 0 0 0***					B					12										Rc
frsqrte*x*[e]	63	D					**0 0 0 0 0***					B					**0 0 0 0 0***					26				Rc
fsel*x*[e]	63	D					A					B					C					23				Rc
fsqrt*x*[e,h]	63	D					**0 0 0 0 0***					B					**0 0 0 0 0***					22				Rc
fsqrts*x*[e,h]	59	D					**0 0 0 0 0***					B					**0 0 0 0 0***					22				Rc
fsub*x*	63	D					A					B					**0 0 0 0 0***					20				Rc
fsubs*x*	59	D					A					B					**0 0 0 0 0***					20				Rc
icbi	31	**0 0 0 0 0***					A					B					982										**0***
isync	19	**0 0 0 0 0***					**0 0 0 0 0***					**0 0 0 0 0***					150										**0***
lbz	34	D					A					d															
lbzu	35	D					A					d															
lbzux	31	D					A					B					119										**0***
lbzx	31	D					A					B					87										**0***
ld[d]	58	D					A					ds														0	

See note(s) at end of table.

Table A5-3 — Complete Instruction List Sorted by Mnemonic (Contd)

Name	0–5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ldarx[d]	31	D					A					B					84										**0***
ldu[d]	58	D					A					ds														1	
ldux[d]	31	D					A					B					53										**0***
ldx[d]	31	D					A					B					21										**0***
lfd	50	D					A					d															
lfdu	51	D					A					d															
lfdux	31	D					A					B					631										**0***
lfdx	31	D					A					B					599										**0***
lfs	48	D					A					d															
lfsu	49	D					A					d															
lfsux	31	D					A					B					567										**0***
lfsx	31	D					A					B					535										**0***
lha	42	D					A					d															
lhau	43	D					A					d															
lhaux	31	D					A					B					375										**0***
lhax	31	D					A					B					343										**0***
lhbrx	31	D					A					B					790										**0***
lhz	40	D					A					d															
lhzu	41	D					A					d															
lhzux	31	D					A					B					311										**0***
lhzx	31	D					A					B					279										**0***
lmw[c]	46	D					A					d															
lswi[c]	31	D					A					NB					597										**0***
lswx[c]	31	D					A					B					533										**0***
lwa[d]	58	D					A					ds														2	
lwarx	31	D					A					B					20										**0***
lwaux[d]	31	D					A					B					373										**0***
lwax[d]	31	D					A					B					341										**0***
lwbrx	31	D					A					B					534										**0***
lwz	32	D					A					d															
lwzu	33	D					A					d															
lwzux	31	D					A					B					55										**0***
lwzx	31	D					A					B					23										**0***
mcrf	19	crfD			**0 0***		crfS			**0 0***		**0 0 0 0 0***					0										**0***
mcrfs	63	crfD			**0 0***		crfS			**0 0***		**0 0 0 0 0***					64										**0***
mcrxr	31	crfD			**0 0***		**0 0 0 0 0***					**0 0 0 0 0***					512										**0***
mfcr	31	D					**0 0 0 0 0***					**0 0 0 0 0***					19										**0***
mffs$x$	63	D					**0 0 0 0 0***					**0 0 0 0 0***					583										Rc
mfmsr[a]	31	D					**0 0 0 0 0***					**0 0 0 0 0***					83										**0***
mfspr[b]	31	D					spr										339										**0***
mfsr[a,f,g]	31	D					**0***	SR				**0 0 0 0 0***					595										**0***
mfsrin[a,f,g]	31	D					**0 0 0 0 0***					B					659										**0***
mftb	31	D					tbr										371										**0***
mtcrf	31	S					**0***	CRM								**0***	144										**0***

See note(s) at end of table.

Table A5-3 — Complete Instruction List Sorted by Mnemonic (Contd)

Name	0–5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
mtfsb0*x*	63	crbD					**0**	**0**	**0**	**0**	**0***	**0**	**0**	**0**	**0**	**0***	70										Rc
mtfsb1*x*	63	crbD					**0**	**0**	**0**	**0**	**0***	**0**	**0**	**0**	**0**	**0***	38										Rc
mtfsf*x*	31	**0***	FM								**0***	B					711										Rc
mtfsfi*x*	63	crfD			**0**	**0***	**0**	**0**	**0**	**0**	**0***	IMM				**0***	134										Rc
mtmsra,g	31	S					**0**	**0**	**0**	**0**	**0***	**0**	**0**	**0**	**0**	**0***	146										**0***
mtmsrda,d	31	S					**0**	**0**	**0**	**0**	**0***	**0**	**0**	**0**	**0**	**0***	178										**0***
mtsprb	31	S					spr										467										**0***
mtsra,f,g	31	S					**0***	SR				**0**	**0**	**0**	**0**	**0***	210										**0***
mtsrda,f	31	S					**0***	SR				**0**	**0**	**0**	**0**	**0***	82										**0***
mtsrdina,f	31	S					**0**	**0**	**0**	**0**	**0***	B					114										**0***
mtsrina,f,g	31	S					**0**	**0**	**0**	**0**	**0***	B					242										**0***
mulhd*x*d	31	D					A					B					**0***	73									Rc
mulhdu*x*d	31	D					A					B					**0***	9									Rc
mulhw*x*	31	D					A					B					**0***	75									Rc
mulhwu*x*	31	D					A					B					**0***	11									Rc
mulld*x*d	31	D					A					B					OE	233									Rc
mulli	07	D					A					SIMM															
mullw*x*	31	D					A					B					OE	235									Rc
nand*x*	31	S					A					B					476										Rc
neg*x*	31	D					A					**0**	**0**	**0**	**0**	**0***	OE	104									Rc
nor*x*	31	S					A					B					124										Rc
or*x*	31	S					A					B					444										Rc
orc*x*	31	S					A					B					412										Rc
ori	24	S					A					UIMM															
oris	25	S					A					UIMM															
rfia,g	19	**0**	**0**	**0**	**0**	**0***	**0**	**0**	**0**	**0**	**0***	**0**	**0**	**0**	**0**	**0***	50										**0***
rfida,d	19	**0**	**0**	**0**	**0**	**0***	**0**	**0**	**0**	**0**	**0***	**0**	**0**	**0**	**0**	**0***	18										**0***
rldcl*x*d	30	S					A					B					mb						8				Rc
rldcr*x*d	30	S					A					B					me						9				Rc
rldic*x*d	30	S					A					sh					mb						2			sh	Rc
rldicl*x*d	30	S					A					sh					mb						0			sh	Rc
rldicr*x*d	30	S					A					sh					me						1			sh	Rc
rldimi*x*d	30	S					A					sh					mb						3			sh	Rc
rlwimi*x*	20	S					A					SH					MB					ME					Rc
rlwinm*x*	21	S					A					SH					MB					ME					Rc
rlwnm*x*	23	S					A					B					MB					ME					Rc
sc	17	**0***					**0***					**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1	**0***
slbiaa,d,e	31	**0**	**0**	**0**	**0**	**0***	**0**	**0**	**0**	**0**	**0***	**0**	**0**	**0**	**0**	**0***	498										**0***
slbiea,d,e	31	**0**	**0**	**0**	**0**	**0***	**0**	**0**	**0**	**0**	**0***	B					434										**0***
sld*x*d	31	S					A					B					27										Rc
slw*x*	31	S					A					B					24										Rc
srad*x*d	31	S					A					B					794										Rc
sradi*x*d	31	S					A					sh					413									sh	Rc
sraw*x*	31	S					A					B					792										Rc

See note(s) at end of table.

Table A5-3 — Complete Instruction List Sorted by Mnemonic (Contd)

Name	0–5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
srawi*x*	31	S					A					SH					824										Rc
srd*x*[d]	31	S					A					B					539										Rc
srw*x*	31	S					A					B					536										Rc
stb	38	S					A					d														
stbu	39	S					A					d														
stbux	31	S					A					B					247										**0***
stbx	31	S					A					B					215										**0***
std[d]	62	S					A					ds														0
stdcx.[d]	31	S					A					B					214										1
stdu[d]	62	S					A					ds														1
stdux[d]	31	S					A					B					181										**0***
stdx[d]	31	S					A					B					149										**0***
stfd	54	S					A					d														
stfdu	55	S					A					d														
stfdux	31	S					A					B					759										**0***
stfdx	31	S					A					B					727										**0***
stfiwx[e]	31	S					A					B					983										**0***
stfs	52	S					A					d														
stfsu	53	S					A					d														
stfsux	31	S					A					B					695										**0***
stfsx	31	S					A					B					663										**0***
sth	44	S					A					d														
sthbrx	31	S					A					B					918										**0***
sthu	45	S					A					d														
sthux	31	S					A					B					439										**0***
sthx	31	S					A					B					407										**0***
stmw[c]	47	S					A					d														
stswi[c]	31	S					A					NB					725										**0***
stswx[c]	31	S					A					B					661										**0***
stw	36	S					A					d														
stwbrx	31	S					A					B					662										**0***
stwcx.	31	S					A					B					150										1
stwu	37	S					A					d														
stwux	31	S					A					B					183										**0***
stwx	31	S					A					B					151										**0***
subf*x*	31	D					A					B					OE	40									Rc
subfc*x*	31	D					A					B					OE	8									Rc
subfe*x*	31	D					A					B					OE	136									Rc
subfic	08	D					A					SIMM														
subfme*x*	31	D					A					**0 0 0 0 0***					OE	232									Rc
subfze*x*	31	D					A					**0 0 0 0 0***					OE	200									Rc
sync	31	**0 0 0 0 0***					**0 0 0 0 0***					**0 0 0 0 0***					598										**0***
td[d]	31	TO					A					B					68										**0***
tdi[d]	02	TO					A					SIMM														

See note(s) at end of table.

Table A5-3 — Complete Instruction List Sorted by Mnemonic (Contd)

Name	0–5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
tlbia[a,e,h]	31	**0**	**0**	**0**	**0**	**0***	**0**	**0**	**0**	**0**	**0***	**0**	**0**	**0**	**0**	**0***	370										**0***
tlbie[a,e]	31	**0**	**0**	**0**	**0**	**0***	**0**	**0**	**0**	**0**	**0***	B					306										**0***
tlbsync[a,e]	31	**0**	**0**	**0**	**0**	**0***	**0**	**0**	**0**	**0**	**0***	**0**	**0**	**0**	**0**	**0***	566										**0***
tw	31	TO					A					B					4										**0***
twi	03	TO					A					SIMM															
xor*x*	31	S					A					B					316										Rc
xori	26	S					A					UIMM															
xoris	27	S					A					UIMM															

Note(s):
a. Supervisor-level instruction
b. Supervisor- and user-level instruction
c. Load and store string or multiple instruction
d. 64–bit instruction
e. Optional instruction
f. 32–bit instruction only
g. Optional 64–bit bridge instruction
h. 32–bit instruction not implemented by the MPC750

A6. SWITCHING MODULE ASSERT ANALYSIS EXAMPLE AUDIT TRAIL

This appendix is the audit trail used to construct the stack in the "Assert Analysis Example — SM," Section 5.2.2.

```
LOCAL DATA OF THE FUNCTION SMiclvco
 struct {
            OSMSGHEAD                       msghead;
            struct mgCDFH2MRA               text;
        }                       mramsg;     /* outgoing message buffer */
        struct rlSMEST          smest;      /* local SMEST tuple buffer */
        struct mgDFIHMSG        outmsg;     /* buffer for PERFR msg*/
        DMCIRCUIT               cfac_name;  /* CFAC internal name */
 register char                  *ddlptr;    /* DDL message pointer*/
 register short                 i;
GH:GHDR7 5D00725
/* PTC:
    DM: OSMSGHEAD FALSE TRUE TRUE
 */
 #include <hdr/smfw/FWospid.h>
 /* OSDS message header definition.
 */
 typedef struct {
        OSPID           from;       /* Sending process.  */
        unsigned short  type;       /* Message type.  */
        unsigned char   priority;   /* Message priority. */
        unsigned char   length;     /* Text length, in bytes. */
 } OSMSGHEAD;
GH:GHDR7 5D00725
/*    PTC:
       DM: OSPID FALSE TRUE TRUE
 */
 /* Definition of an OSDS process I.D.
 */
 typedef  struct {
         short          procno; /* Process number  */
         unsigned char  pcrid;  /* Processor id  */
         unsigned char  uniq;   /* Uniqueness field  */
 } OSPID;
SM:SMRSMCDFH 5D76400
 @LOCAL HEADER FILE: smim/hdr/MGcdfh2mra.h
 /*
 * File:  smim/hdr/MGcdfh2mra.h
 *
 * Description: This file defines the message sent between the CDFI
 *   handler system process, CDFH, and various MRA terminal
 *   processes to report the completion of action taken on
 *   RCLs. These actions include a connectivity exercise,
 *   or a request to remove or restore ddl communication
 *   over an ICL.
 */
 #include <hdr/db/DMcircuit.h>
 #include <hdr/db/DMmrartval.h>
 #include <hdr/db/DMmraverb.h>

 #define  MGCDFH2MRA 2049

 struct   mgCDFH2MRA {
          DMCIRCUIT      cfac_name;  /* CFAC internal circuit name */
          DMMRAVERB      verb;       /* response to action of this type */
          DMMRARTVAL     result;     /* result of attempting action */
          unsigned char  serial;     /* msg serial number (for  */
                                     /* connectivity exercise only) */
 };
GH:GHDR2 5D00600
 /*       PTC:
           DM: DMCIRCUIT TRUE TRUE FALSE
```

```
           SB: DMCIRCUIT 0 65535
 */

 /*****************************************************************
 *
 * dom_name domain description
 * -------- ------------------
 * circuit Internal names for hardware circuits
 *
 *****************************************************************/

 typedef   unsigned short   DMCIRCUIT; /* range = 0, 65535 */
GH:GHDR2 5D00600
 /*     PTC:
         DM: DMMRAVERB TRUE TRUE FALSE
         BF: BFMRAVERB 4
 */

 #define BFMRAVERB 4

 /*****************************************************************
 *
 * dom_name domain description
 * -------- ------------------
 * mraverb Craft command verbs
 *
 *****************************************************************/

 typedef   enum {
           SMRMV,   /* Remove */
           SMRST,   /* Restore */
           SMDGN,   /* Diagnose */
           SMEX,    /* Exercise */
           SMSW,    /* Switch */
           SMSTOP,  /* Stop */
           SMABT,   /* Abort */
           SMINH,   /* Inhibit */
           SMALW,   /* Allow */
           SMOP,    /* Output */
           SMRTN,   /* Routine Exercise */
           SMTEST,  /* Special Test */
           SMNULL,  /* Null Verb */
           SMCFG,   /* Configuration */
           SMCLR,   /* Clear */
           SMSET,   /* Set */
 } DMMRAVERB;
GH:GHDR2 5D00600
 /*
 * File:        DMmrartval.G
 *
 * Pathname:    [u]hdr/db (CMS instance = msg)
 *
 * Description: This file defines the structure used by SMIM-MRA
 *              to pass return codes around (some other areas use this
 *              structure when interfacing with MRA). The two fields
 *              info and resp provide low- and high-level
 *              information respectively, and may eventually
 *              produce craft output. Examples:
 *                resp - stopped, completed, not started, ...
 *                info - hardware init failure, previous
 *                       request in progress, ...
 */

 #define _DMMRARTVAL

 #ifndef _DMMRARESP
 #include <hdr/db/DMmraresp.h>
 #endif
```

```
 #ifndef _DMMRAINFO
 #include <hdr/db/DMmrainfo.h>
 #endif


 typedef struct {
                 DMMRAINFO  info;
                 DMMRARESP  resp;
 } DMMRARTVAL;
GH:GHDR2 5D00600
#define _DMMRAINFO

 typedef enum {
               SMNUL = 0,   SMDG0 = 1,   SMDG1 = 2,
               SMDG2 = 3,   SMDG3 = 4,   SMDG4 = 5,
               SMDG5 = 6,   SMDG6 = 7,   SMDG7 = 8,
               SMDG8 = 9,   SMDG9 = 10,  SMDG10 = 11,
               SMDG11 = 12, SMDG12 = 13, SMDG13 = 14,
               SMDG14 = 15, SMDG15 = 16, SMDG16 = 17,
               SMDG17 = 18, SMDG18 = 19, SMDG19 = 20,
               SMDG20 = 21, SMDG21 = 22, SMDG22 = 23,
               SMDG23 = 24, SMDG24 = 25, SMDG25 = 26,
               SMDG26 = 27, SMDG27 = 28, SMDG28 = 29,
               SMDG29 = 30, SMDG30 = 31, SMDG31 = 32,
               SMDG32 = 33, SMDG33 = 34, SMDG34 = 35,
               SMDG35 = 36, SMDG36 = 37, SMDG37 = 38,
               SMDG38 = 39, SMDG39 = 40, SMDG40 = 41,
               SMDG41 = 42, SMDG42 = 43, SMDG43 = 44,
               SMDG44 = 45, SMDG45 = 46, SMDG46 = 47,
               SMDG47 = 48, SMDG48 = 49, SMDG49 = 50,
               SMDG50 = 51, SMDG51 = 52, SMDG52 = 53,
               SMDG53 = 54, SMDG54 = 55, SMDG55 = 56,
               SMDG56 = 57, SMDG57 = 58, SMDG58 = 59,
               SMDG59 = 60, SMDG60 = 61, SMDG61 = 62,
               SMDG62 = 63, SMDG63 = 64, SMDG64 = 65,
               SMDG65 = 66, SMDG66 = 67, SMDG67 = 68,
               SMDG68 = 69, SMDG69 = 70, SMDG70 = 71,
               SMDG71 = 72, SMDG72 = 73, SMDG73 = 74,
               SMDG74 = 75, SMDG75 = 76, SMDG76 = 77,
               SMDG77 = 78, SMDG78 = 79, SMDG79 = 80,
               SMDG80 = 81, SMATP = 82,  SMSTF = 83,
               SMCATP = 84, SMNTR = 85,  SMUU = 86,
               SMOTE = 87,  SMPLE = 88,  SMRNA = 89,
               SMAOS = 90,  SMUAF = 91,  SMMOS = 92,
               SMMAF = 93,  SMNCR = 94,  SMURS = 95,
               SMPRA = 96,  SMOHP = 97,  SMFAE = 98,
               SMPOS = 99,  SMDAR = 100, SMNRQ = 101,
               SMUGS = 102, SMDLD = 103, SMDBE = 104,
               SMPCE = 105, SMRTAE = 106, SMHDWE = 107,
               SMTME = 108, SMSCE = 109, SMUTBA = 110,
               SMQF = 111,  SMFWN = 112, SMSLC01 = 113,
               SMSLC02 = 114, SMSLC03 = 115, SMSLC04 = 116,
               SMSLC05 = 117, SMSLC06 = 118, SMSLC07 = 119,
               SMSLC08 = 120, SMSLC09 = 121, SMSLC10 = 122,
               SMSLC11 = 123, SMSLC12 = 124, SMSLC13 = 125,
               SMSLC14 = 126, SMSLC15 = 127, SMSLC16 = 128,
               SMSLC17 = 129, SMSLC18 = 130, SMSLC19 = 131,
               SMSLC20 = 132, SMSLC21 = 133, SMSLC22 = 134,
               SMSLC23 = 135, SMSLC24 = 136, SMSLC25 = 137,
               SMSLC26 = 138, SMSLC27 = 139, SMSLC28 = 140,
               SMSLC29 = 141, SMSLC30 = 142, SMSLC31 = 143,
               SMSLC32 = 144, SMSLC33 = 145, SMSLC34 = 146,
               SMSLC35 = 147, SMSLC36 = 148, SMSLC37 = 149,
               SMSLC38 = 150, SMSLC39 = 151, SMSLC40 = 152,
               SMSLC41 = 153, SMSLC42 = 154, SMSLC43 = 155,
               SMSLC44 = 156, SMSLC45 = 157, SMSLC46 = 158,
               SMSLC47 = 159, SMSLC48 = 160, SMSLC49 = 161,
```

```
SMSLC50 = 162,
SMAMLE = 163, /* AML exceeded */
SMAMLNE = 164, /* AML not exceeded */
SMMRA00 = 165, SMMRA01 = 166, SMMRA02 = 167,
SMFLTC = 168, /* CPI fault */
SMSTUKC = 169, /* CPI status unknown */
SMELS00 = 170, SMELS01 = 171, SMELS02 = 172,
SMELS03 = 173, SMELS04 = 174, SMELS05 = 175,
SMELS06 = 176, SMELS07 = 177, SMELS08 = 178,
SMRSM00 = 179, SMRSM01 = 180, SMRSM02 = 181,
SMRSM03 = 182, SMRSM04 = 183, SMRSM05 = 184,
SMRSM06 = 185, SMRSM07 = 186, SMRSM08 = 187,
SMRSM09 = 188, SMRSM10 = 189, SMRSM11 = 190,
SMRSM12 = 191, SMRSM13 = 192, SMRSM14 = 193,
SMRSM15 = 194, SMRSM16 = 195, SMRSM17 = 196,
SMRSM18 = 197, SMRSM19 = 198, SMRSM20 = 199,
SMRSM21 = 200, SMRSM22 = 201, SMRSM23 = 202,
SMRSM24 = 203, SMRSM25 = 204, SMRSM26 = 205,
SMRSM27 = 206, SMRSM28 = 207, SMRSM29 = 208,
SMRSM30 = 209, SMRSM31 = 210, SMRSM32 = 211,
SMRSM33 = 212, SMRSM34 = 213, SMRSM35 = 214,
SMRSM36 = 215, SMRSM37 = 216, SMRSM38 = 217,
SMRSM39 = 218, SMRSM40 = 219, SMRSM41 = 220,
SMRSM42 = 221, SMRSM43 = 222, SMRSM44 = 223,
SMRSM45 = 224, SMRSM46 = 225, SMRSM47 = 226,
SMRSM48 = 227, SMRSM49 = 228,
SMDCE = 229, ** DCTU communication error **
SMDG81 = 230, SMDG82 = 231, SMDG83 = 232,
SMDG84 = 233, SMDG85 = 234, SMDG86 = 235,
SMDG87 = 236, SMDG88 = 237, SMDG89 = 238,
SMDG90 = 239, SMDG91 = 240, SMDG92 = 241,
SMDG93 = 242, SMDG94 = 243, SMDG95 = 244,
SMDG96 = 245, SMDG97 = 246, SMDG98 = 247,
SMDG99 = 248, SMDG100 = 249,
SMDMNULL = 250, SMDM00 = 251, SMDM01 = 252,
SMDM02 = 253, SMDM03 = 254, SMDM04 = 255,
SMDM05 = 256, SMDM06 = 257, SMDM07 = 258,
SMDM08 = 259, SMDM09 = 260, SMDM10 = 261,
SMDM11 = 262, SMDM12 = 263, SMDM13 = 264,
SMDM14 = 265, SMDM15 = 266, SMDM16 = 267,
SMDM17 = 268, SMDM18 = 269, SMDM19 = 270,
SMDM20 = 271, SMDM21 = 272, SMDM22 = 273,
SMDM23 = 274, SMDM24 = 275, SMDM25 = 276,
SMDM26 = 277, SMDM27 = 278, SMDM28 = 279,
SMDM29 = 280, SMDM30 = 281, SMDM31 = 282,
SMDM32 = 283, SMDM33 = 284, SMDM34 = 285,
SMDM35 = 286, SMDM36 = 287, SMDM37 = 288,
SMDM38 = 289, SMDM39 = 290, SMDM40 = 291,
SMDM41 = 292, SMDM42 = 293, SMDM43 = 294,
SMDM44 = 295, SMDM45 = 296, SMDM46 = 297,
SMDM47 = 298, SMDM48 = 299, SMDM49 = 300,
SMCAMPON = 301,
SMELS09 = 302, SMELS10 = 303, SMELS11 = 304,
SMELS12 = 305, SMELS13 = 306,
SMMRA03 = 307, SMMRA04 = 308, SMMRA05 = 309,
SMMRA06 = 310, SMMRA07 = 311, SMMRA08 = 312,
SMMPO = 313, /* mate powered off */
SMCHE1 = 314,
SMCHE2 = 315,
SMCHE3 = 316,

/* Following return values are used by the SMckptckt()
function which is called by TM to determine equipment
status when checking the PSA tables. The convention
for LEN status given below is :
[base state,qualifier,op restriction,suppl info] */

SMPMFE = 317, /* PPMCKT OOS with
                  [OOS, MTCE, PPMFE, NULL] line status */
```

```
SMMFE = 318, /* PPMCKT OOS with
                 [OOS, MTCE, FE, PPM] line status */
SMFEPMFE = 319, /* PPMCKT OOS with both line status */

SMLPMFE = 320, /* LEN OOS and PPMCKT OOS with
                  [OOS, MTCE, PPMFE, NULL] line status */
SMLMFE = 321, /* LEN OOS and PPMCKT OOS with
                 [OOS, MTCE, FE, PPM] line status */
SMLFEPMFE = 322, /* LEN OOS and PPMCKT OOS with
                    both line status */
SMAUTSTP = 323, /* Diagnostic Stopped by PFR */
SMMANSTP = 324, /* Diagnostic Stopped by CFT/MRA */
SML16LB = 325, /* Campon Timeout, < 16 Lines busy */
SML32LB = 326, /* Campon Timeout, < 32 Lines busy */
SMFNL = 327, /* Feature Package Not Loaded */
SMNSPAR = 328, /* No Spare Available */
SMSWF = 329, /* Switch Failed */
SMDFD = 330, /* Duplex Failure Denied, Mate Being */
             /* Diagnosed */
SMDFDG = 331, /* Duplex Failure, Diagnostic in */
              /* progress */
SMDFM = 332, /* Duplex Failure, Manual Action */
             /* Required */
SM0LB = 333, /* Campon Successful, All Lines Removed */
SM1LB = 334, /* Campon Timeout, 1 Line Busy */
SM2LB = 335, /* Campon Timeout, 2 Lines Busy */
SM3LB = 336, /* Campon Timeout, 3 Lines Busy */
SM4LB = 337, /* Campon Timeout, 4 Lines Busy */
SM5LB = 338, /* Campon Timeout, 5 Lines Busy */
SM6LB = 339, /* Campon Timeout, 6 Lines Busy */
SM7LB = 340, /* Campon Timeout, 7 Lines Busy */
SM8LB = 341, /* Campon Timeout, 8 Lines Busy */
SM9LB = 342, /* Campon Timeout, 9 Lines Busy */
SM10LB = 343, /* Campon Timeout, 10 Lines Busy */
SM15LB = 344, /* Campon Timeout, 15 Lines Busy */
SM20LB = 345, /* Campon Timeout, 20 Lines Busy */
SM30LB = 346, /* Campon Timeout, 30 Lines Busy */
SM32LB = 347, /* Campon Timeout, 32 Lines Busy */
SMDFR = 348, /* Restored From Duplex Failure */
SMMBSY = 349, /* Mate circuit is MRA Busy */
SMRQDN = 350, /* Request Denied */
SMRACT = 351, /* Restore to Active */
SMRSTBY = 352, /* Restore to Standby */
SMDSLWS = 353, /* DSL Group Left without Service */
SMPHNR = 354, /* Not all PSIU PH Units Restored */
SMDRS = 355, /* Deferred Restoral Scheduled */
SMPTOOS = 356, /* Warning Ports may still be OOS */
SMRACTMIN = 357, /* Restored to Active Minor */
SMSWSHS = 358, /* Soft SW Escalated to Hard SW,
                  Rst Scheduled */
SMSWFHS = 359, /* Soft and Hard Switch Failed,
                  Rst Scheduled */
SMM32LB = 360, /* Campon Timeout,32 or More Lines Busy */

/* New DGerrcode.h values added for ISLU operation */

SMPCFAIL = 361, /* Failure to seize resource */
SMSEIZE = 362, /* SRA Failure */
SMCKTDF = 363, /* CKTDATA Read Failure */
SMDCBF = 364, /* Diagnostic Control Block Corrupted */
SMSMESTF = 365, /* SMEST Read Failure */
SMOPCMDF = 366, /* Bad argument in operate call */
SMPATHF = 367, /* Path hunt failure in PC */
SMUCIWF = 368, /* Cannot write to ISLU */
SMUCIRF = 369, /* Cannot read from ISLU */
SMCCIF = 370, /* Failure to initialize ISLUCC */
SMUCIQF = 371, /* SMIM primitive failure in ISLU */

SM_BAD_FID = 372, /* Unable to Match on DG Feature ID */
```

```
SM_FNL = 373, /* DG Feature Package Not Loaded */
SM_PSERMSG = 374, /* Port Processor Send Message
                     Error Return */
SM_PSNRMSG = 375, /* No Resources Available for Port */
                  /* Processor Send Message */

/* Spare code for ISDN DG use */
SMDGIS00 = 376,
SM_PSCCERR = 377, /* Port Processor Completion Code */
                  /* Error in Message Returned */
SM_PSTOERR = 378, /* Port Processor Message Verify */
                  /* Timeout Error */
SM_PSSNERR = 379, /* Port Processor Sequence Number */
                  /* Error in Message Returned */
/* Spare codes for ISDN DG use */
SMDGIS04 = 380,
SMDGIS05 = 381,
SMDGIS06 = 382,
SMDGIS07 = 383,
SMDGIS08 = 384,
SMDGIS09 = 385,
SMMRA09 = 386, /* Mate Failed While Initializing a
                  subunit */
SMSPRACT = 387, /* Circuit involved in Line Card
                   Sparing */
SMSPRRS = 388, /* Resource unavail due tp sparing
                  conflict */
SMSPRCMPL = 389, /* Spare completed */
SMSPRFL1 = 390, /* spare fail message 1 */
SMSPRFL2 = 391, /* Spare failure message 2 */
SMSPRFL3 = 392, /* Spare failure message 3 */
SMPTIS = 393, /* Warning Ports May Still Be In Service */
SMOSPSBUSY = 394, /* Request Denied OSPS Ports Busy */
SMCCDF = 395, /* CC Duplex Failure */
SMRQDDIP = 396, /* Request Denied, Diagnostic In */
                /* Progress */
SMCCDFRMF = 397, /* CC Duplex Failure; Remove Mate First */
SMPSATDP = 398, /* Ports Still Assigned To DPIDB */
SMNAILUP = 399, /* Port is nailed up */
SMRGOLST = 400, /* RG confg attempt lost some of orig
                   relays */
SMRGBUSSM = 401, /* RG init passed but no relays able
                    to close */
SMRGNOBSAV = 402, /* No RG buses avail to connect to */
SMRSUNAV = 403, /* Resource was unavailable from SRA */
SMSPRFAIL = 404, /* spare of zcard failed */
SMSPRTSK = 405, /* zcard sparing task failed */
SMSWDEGR = 406, /* Escalated to hard switch
                   - act unit was degraded */
SMSWPROS = 407, /* Escalated to hard switch
                   - act unit was pre-OOS */
SMSSCBUSY = 408, /* Some Subtending Circuits are Busy */
SMRQDPORTBSY = 409, /* Request Denied - Some Ports Are
                       Busy */
SMSWUDGINP = 410, /* Switch Unsuccessful - DGN in
                     Progress on Assoc Ckts */
SMSWUMCB = 411, /* Switch Unsuccessful - Mate Circuit
                   is MRA Busy */
SMSPRRLSD = 412, /* line card spare released */
SMSPRPOS = 413, /* Sparing possible for card */
SMSPR1 = 414, /* extra for future Ucard sparing */
SMSPR2 = 415, /* extra for future Zcard sparing */
SMSPR3 = 416, /* extra for ?    */
SMHSERR = 417, /* hashsum error */
SMRGRLY0 = 418, /* RG relay config failure */
SMRGRLY1 = 419, /* RG relay config failure */
SMRGRLY2 = 420, /* RG relay config failure */
SMRGRLY3 = 421, /* RG relay config failure */
SMRGRLY4 = 422, /* RG relay config failure */
```

```
                SMRGRLY5 = 423, /* RG relay config failure */
                SMRGRLY6 = 424, /* RG relay config failure */
                SMRGRLY7 = 425, /* RG relay config failure */
                SMRGRLY8 = 426, /* RG relay config failure */
                SMRGRLY9 = 427, /* RG relay config failure */
                SMNSAPHD = 428, /* No Spare Available, PH Degraded */
                SMPRFDRR = 429, /* Port Restoral failed, Deferred
                  * Restoral WITH DGN Requested.
                  */
                SMSTPMP = 430, /* Pump received stop request */
                SMTOPI = 431, /* SM timed out waiting for PI */
                SMTOPH = 432, /* SM timed out waiting for PH */
                SMPHSWFL = 433, /* PH Switch Failed */
                SMGRPOOS = 434, /* Cannot assign CG, another group OOS */
                SMGRPDGR = 435, /* Cannot assign CG, another PH DGR */
                SMGRPBY = 436, /* Requested group is MRA Busy */
                SMGRPUNEQ = 437, /* Requested group is unequipped */
                SMPRF = 438, /* Port Restoral failed */
                SM_SPR_1 = 439, /* Spare Code for MRA ISLU use */
                SM_SPR_2 = 440, /* Spare Code for MRA ISLU use */
                SM_SPR_3 = 441, /* Spare Code for MRA ISLU use */
                SM_SPR_4 = 442, /* Spare Code for MRA ISLU use */

                SMCPHPMPF = 443, /* No Phs fully passed pump and init */
                SMPHDWINC = 444, /* PH-DSLG Hardware Inconsistency */
                SMPHMPI1 = 445, /* Spare Code for MRA MPI usage */
                SMPHMPI2 = 446, /* Spare Code for MRA MPI usage */
                SMPHMPI3 = 447, /* Spare Code for MRA MPI usage */
                SMPHMPI4 = 448, /* Spare Code for MRA MPI usage */
                SMPHMPI5 = 449, /* Spare Code for MRA MPI usage */
                SMINHF = 450, /* INHIBIT FAILURE */
                SMISM = 452, /* Insufficient Memory */
                SMINCDATA = 453, /* Inconsistent Data found during */
                                 /* switch attempt    */
 } DMMRAINFO;
GH:GHDR2 5D00600
#define _DMMRARESP

typedef enum {
              SMCMPL = 0,  SMCMPL_CERT = 1,  SMSTPD = 2,
              SMABTD = 3,  SMNS = 4,         SMSKIP = 5
} DMMRARESP;
GH:GHDR3 5D00625
 * msgtype  message type
 * atpcount number of times diagnostics ran ATP
 * qual  type of inhibit
 * state  inhibit
 * errorblk link to SMPERBLK tuple
 * dgstat diagnostic return code
 * transient flag to indicate the tuple is in a transient state
 * progflag audit progress flag
 * quarstate audit quarantine field
 * rexinh inhibit status for REX for each unit that REX schedules,
 * iatouch  indicates when IMIIA touches the tuple
 * pinhtype type of peripheral circuit inhibit
 * red_alm  indicates that a RED CARRIER GROUP ALARM is active
 *          on a DFI-2 T1 facility
 * ylw_alm  indicates that a YELLOW CARRIER GROUP ALARM is active
 *          on a DFI-2 T1 facility
 * ais_alm  indicates that an ALARM INDICATION SIGNAL is active
 *          on a DFI-2 T1 facility
 ***************************************************************************
#define RLSMEST 65

struct rlSMEST {
   OSPID orig_pid;
   OSPID tp_pid;
   DMCIRCUIT unit_id;
```

```
   DMUNS16 msgtype;
   DMUCHAR atpcount;
   DMUCHAR errorblk;
   DMACTIVITY stat_act   :BFACTIVITY;
   DMMRASTAT bas_stat   :BFMRASTAT;
   DMINHTYPE qual    :BFINHTYPE;
   DMMEASTAT qual_2   :BFMRASTAT;
   DMBOOL transient   :BFBOOL;
   DMDGSTAT dgstat    :BFDGSTAT;
   DMMRASRC qual_1    :BDMRASRC;
   DMBOOL mccupd    :BFBOOL;
   DMBOOL rexinh    :BFBOOL;
   DMBOOL progflag    :BFBOOL;
   DMSTATUS state    :BFSTATUS;
   DMAUQUAR quarstate   :BFAUQUAR;
   DMPINHTYPE pinhtype   :BFPINHTYPE;
   DMPBOOL ais_alm    :BFBOOL;
   DMBOOL ylw_alm    :BFBOOL;
   DMBOOL red_alm    :BFBOOL;
   DMBOOL iatouch    :BFBOOL;
};
GH:GHDR3 5D00625
 /*  PTC:
  DM: DMUNS16 TRUE TRUE FALSE
  SB: DMUNS16 0 65535
 */
 /*****************************************************************
 * dom_name domain description
 * -------- ------------------
 * uns16  Unsigned 16 bits
 *****************************************************************
 typedef unsigned short DMUNS16; /* range = 0, 65535 */
GH: GHDR3 5D00625
 /*  PTC:
  DM: DMUCHAR TRUE TRUE FALSE
 */
 #define _DMUCHAR
 /*****************************************************************
 /* The #define _DM... or _RL... symbol or #ifndef _DM /#endif
 /* construct has been left in
 /* this domain or relation because it is used in or is a
 /* DBM dictionary relation. The dictionaries (via a constructed
 /* .c file DBddat.c) are compiled by ODA and the Data Population
 /* Environment Group using cc and cc370 respectively. This
 /* compile requires that the domains and relations to be included
 /* only once. Therefore the #ifndef _DM or #ifndef _RL constructs
 /* must remain for all DBM dictionary relations and the domains
 /* used in them. The DBM dictionary relations involved are:
 /*  RLdb_accdict.h
 /*  RLdb_attdict.h
 /*  RLdb_domdict.h
 /*  RLdb_drldict.h
 /*  RLdb_dsc.h
 /*  RLdb_enudict.h
 /*  RLdb_inddict.h
 /*  RLdb_reldict.h
 /*  RLdb_satdict.h
 /*  RLdb_cmdct.h
 /*  RLdbincdct.h
 *****************************************************************/
 /*****************************************************************
 *
 * dom_name  domain description
 * --------  ------------------
 * uchar  character
 *****************************************************************
 typedef unsigned char DMUCHAR;
 GH:GHDR2 5D00600
```

```
/* PTC:
 DM: DMACTIVITY TRUE TRUE FALSE
 BF: BFACTIVITY 2
*/

#define BFACTIVITY 2

/*****************************************************************
*
* dom_name domain description
* -------- ------------------
* activity The activity status
*  *
*****************************************************************/

typedef enum {
 DBBUSY,
 DBIDLE,
 TM_ACTV
} DMACTIVITY;


GH:GHDR2 5D00600
/* PTC:
 DM: DMMRASTAT TRUE TRUE FALSE
 BF: BFMRASTAT 6
*/


#define BFMRASTAT 6
/*****************************************************************
*
* dom_name domain description
* -------- ------------------
* mrastat Equipment status states
*
*****************************************************************/


typedef enum {
        SMSTNUL, /* Null value */
        SMACT,  /* active */
        SMSTBY, /* standby (not in PDS) */
        SMUNAV, /* unavailable (not in PDS) */
        SMOOS,  /* out of service */
        SMRMVD, /* removed */
        SMUPDT, /* update */
        SMDGNS, /* diagnose */
        SMFLT,  /* fault */
        SMTBLA, /* trouble analysis */
        SMRTN_EX, /* routine exercise */
        SMFE, /* family of equipment */
        SMPROOS, /* pre-OOS */
        SMAMJ,  /* active major */
        SMAMI,  /* active minor */
        SMLOOP, /* looped */
        SMSPR,  /* line card that is spared */
        SMDSPR, /* designated spare line card actively spared */
        SMDG_EX,  /* diagnostic interactive exercise */
        SMACTR, /* active restricted */
        SMRATST,  /* in service test */
        SMCMPON,  /* camped on */
        SMST1,  /* state 1 */
        SMST2,  /* state 2 */
        SMST3,  /* state 3 */
        SMHELPER, /* Diagnose Temp (Helper) */
        SMRCNORM, /* MMRSM remote clock in normal mode */
        SMRCFAST, /* MMRSM remote clock in fast mode */
        SMRCHOLD, /* MMRSM remote clock in holdover mode */
```

```
        SMRCFREE, /* MMRSM remote clock in free run mode */
        SMRCWARM, /* MMRSM remote clock oscillator not warmed */
                  /* up after power up */
        SMCDNY, /* Indicates unit campon timed out */
        SMSWTCH,  /* Indicates unit is being switched */
        SMDEGRADED, /* Indicates some component is defective unit */
        SMINIT_ACT, /* Indicates unit to be initialized to active */
        SMINIT_STBY, /* Indicates unit to be initialized to standby */
        SMAUOOS,  /* Taken OOS by Audits */
        SMDEFRD,  /* Deferred */
        SMPWROFF, /* Power off */
        SMPWRALM, /* Power alarm */
} DMMRASTAT;
GH:GHDR2 5D00600
 /* PTC:
  DM: DMBOOL TRUE TRUE FALSE
  BF: BFBOOL 1
 */
 #define _DMBOOL
 /*************************************************************
 * The above #define _DM... symbol has been left in this domain
 * because the domain is used by the ACP subsystem which is a
 * pre-modularity subsystem. The _DM... symbol is still used
 * in pre-modularity environments.
 *************************************************************/
 /*************************************************************
 /* The #define _DM... or _RL... symbol or #ifndef _DM /#endif
 /* construct has been left in
 /* this domain or relation because it is used in or is a
 /* DBM dictionary relation. The dictionaries (via a constructed
 /* .c file DBddat.c) are compiled by ODA and the Data Population
 /* Environment Group using cc and cc370 respectively. This
 /* compile requires that the domains and relations to be included
 /* only once. Therefore the #ifndef _DM or #ifndef _RL constructs
 /* must remain for all DBM dictionary relations and the domains
 /* used in them. The DBM dictionary relations involved are :
 /* RLdb_accdict.h
 /* RLdb_attdict.h
 /* RLdb_domdict.h
 /* RLdb_drldict.h
 /* RLdb_dsc.h
 /* RLdb_enudict.h
 /* RLdb_inddict.h
 /* RLdb_reldict.h
 /* RLdb_satdict.h
 /* RLdb_cmdct.h
 /* RLdbincdct.h
 *************************************************************/

 #define BFBOOL 1

 /*********************************************************************
 * dom_name domain description
 * -------- ------------------
 * bool  yes or no
 *********************************************************************/

 typedef enum {
        DBNO,
        DBYES
 } DMBOOL;


GH:GHDR2 5D00600
 /* PTC:
    DM: DMDGSTAT TRUE TRUE FALSE
    BF: BFDGSTAT 3
 */
```

```
 #define BFDGSTAT 3

 /*****************************************************
 *
 * dom_name  domain description
 * --------  ------------------------------
 * dgstat  diagnostic return codes
 *
 *****************************************************/


 typedef enum {
         DGNTR,  /* No Tests Run */
         DGATP,  /* All Tests Passed */
         DGCATP,  /* Conditional All Tests Passed */
         DGSTF,  /* Some Tests Failed */
         DGNULL,  /* Null Code */
         DGCMPABT,  /* Test Aborted */
         DGCMPSTP,  /* Test Stopped */
         DGCMPSPR,  /* Spare  */
 } DMDGSTAT;
GH:GHDR2 5D00600
 /*       PTC:
         DM: DMMRASRC TRUE TRUE FALSE
         BF: BFMRASRC 4
 */

 #define BFMRASRC 4

 /*****************************************************************
 *
 * dom_name domain description
 * -------- ------------------
 * mrasrc  Source of maintenance request
 *
 *****************************************************************/

 typedef enum {
          SMSNUL,  /* Null value */
          SMAUTO,  /* Automatic */
          SMAN,  /* Manual */
          SMQTEST, /* Test request */
          SMUN,  /* Unequipped */
          SMGR,  /* Growth */
          SMSGR, /* Special Growth */
          SMREX, /* Routine Exercise */
          SMRESTORE, /* Comm. Link Restore */
          SMRESTART, /* Comm. Link Restart */
          SMSTABLE, /* Comm. Link Stable */
          SMFRCD,  /* Forced */
          SMAUTOSUB, /* Automatic LGC SUB option - transient state */
          SMANSUB, /* Manual LGC SUB option - transient state */
          SMRISLUCGA, /* Terminal Maintenance, RISLU DFI conditioning */
          SMPSURCV, /* SMIM/RCV - transient state. */
 } DMMRASRC;
GH:GHDR2 5D00600
 /*      PTC:
        DM: DMAUQUAR TRUE TRUE FALSE
        BF: BFAUQUAR 3
 */

 #define BFAUQUAR 3

 /*****************************************************************
 *
 * dom_name domain description
 * -------- ------------------
```

```
 * auquar  quarantine states for data blocks
 *
 ****************************************************************/


 typedef enum {
         AUNOTSET = 0,
         AUSISET = 1,
         AUAUSET = 2,
         AUDEFCHECK = 3,
         AU1TRANTIME = 4,
         AU2TRANTIME = 5,
         AU1RACECON = 6,
         AU2RACECON = 7
 } DMAUQUAR;

GH:GHDR2 5D00600
 /*    PTC:
       DM: DMINHTYPE TRUE TRUE FALSE
       BF: BFINHTYPE 2
 */

 #define BFINHTYPE 2

 /****************************************************************
 *
 * dom_name domain description
 * -------- ------------------
 * inhtype Inhibit type
 *
 ****************************************************************/


 typedef enum {
         SMIA_MAN,
         SMIA_AUTO,
         SMIATEMP
 } DMINHTYPE;
```

GLOSSARY

3B20D	3B20 Duplex computer
3B31D	3B21 Duplex computer
Address	The number assigned to a particular memory or storage location.
Addressing	The means of assigning program test, data, or stack to storage locations, and subsequently retrieving them.
AIM	Application Integrity Monitor. The subsystem environment in which the detection and correction of insanity in application software is overseen.
ALCB	Access Link Control Block
AM	Administrative Module. The central processing unit of the *5ESS*® switch. Previously, the AM was called the central processor (CP).
Analog Signal	A signal that varies in a continuous manner as the voice varies (frequency and amplitude).
APDL	Application Processor Data Link
Array	A number of elements of the same type and size arranged in rows and columns.
AS	Administrative Services. The subsystem that collects and processes information about switching activities.
Assembly Language	A programming language that lets programmers write their programs at the machine language level.
Assert	A small segment of code within an application program that checks the ranges, redundancy, linkage, and/or consistency of program variables. A common way to perform defensive checks, record failures, and invoke recovery actions.
Assert Handler	The software invoked when a defensive check fails. It reports and recovers from the assert.
ASW	All Seems Well
ATB	Address Translation Buffer
Attributes	Columns in a data table having unique names.
Audit	An autonomous subroutine that verifies and/or re-establishes the consistency of data, data relations, and the linkages between relations.
Audit Control	The program unit that schedules and controls all application audits.
Audit System	The software system that detects, isolates, and corrects errors in the *5ESS* switch data structures.
AUTISS	Automatic Time Slot Switching
BGB	Bidirectional Gating Bus
BIC	Bus Interface Controller. A circuit that provides a 32-bit to 16-bit data interface between the duplex dual serial bus selector (DDSBS) and the peripheral interface controller (PIC).

Bit The smallest unit of information represented in a computer. The information is represented by a positive or negative charge, or an on or off state. Short for binary digit.

Bookstrap An initialization or re-initialization action that results in a memory reload.

Breakpoint A programming entity that enables users to halt a process running in the *5ESS* switch, execute a predefined set of utility functions, and resume execution of the process.

Brevity The feature that limits the number of audit output messages logged and printed.

BRI Basic Rate Interface. The customer's interface to the integrated services line unit (ISLU) that combines two B channels and one D channel.

BSTR Bootstrapper. A circuit in the module controller/time slot interchange (MCTSI) that performs initialization actions that result in memory reloads.

Buffer An area of memory that temporarily holds data that is being received, transmitted, read, or written. It is often used to compensate for differences in speed or timing of devices.

BUS One or more conductors (communications path) over which information is transmitted from any of several sources to any of several destinations.

Bus Seizure Circuit The circuit that interfaces the necessary test set leads on the update bus to gain control of the bus. Also buffers a reset signal from the module processor (MP) to the bootstrapper (BTSR).

BWM Broadcast Warning Message

Byte A contiguous group of binary digits (8 bits) usually operated on as a unit.

C The high-level programming language in which most of the *5ESS* switch software is written.

C Language Structure A collection of variables, possibly of different types, grouped together under a single name for convenient handling.

Call Processing The activities involved in transmitting voice and data calls from inception, through the network, to their final destination. Three subsystems handle call processing: feature control (FC), peripheral control (PC), and routing and terminal allocation (RTA).

CCBCOM Channel Control Block Common Area

CCS Common Channel Signaling

CDAL Control and Diagnostic Access Link

CDR Channel Data Register

CF	Control Fanout. The circuitry in the packet switch unit (PSU) that interfaces the peripheral interface control buses (PICBs) from the module controller/time slot interchange (MCTSI). It also terminates the control interface buses (CIBs) from the data fanout (DF) and packet fanout (PF) units.
CFDB	Call Flow Data Block
Channel	A peripheral time slot.
CI	Control Interface. Circuitry in the module controller/time slot interchange (MCTSI) that communicates control information between the switching module processor (SMP) and the peripheral units.
CIB	Control Interface Bus. The bus that carries control messages between the control fanout (CF) and the data fanout (DF).
CM	Communication Module. The *5ESS* switch hardware that provides the interface between the administrative module (AM) and the switching modules (SMs). In a multimodule office, the CM consists of the message switch (MSGS) and the time multiplexed switch (TMS). The CM has a maximum capacity of 30 SMs.
CM2	Communications Module - Model 2. Combines the MSGS and TMS into a single frame with a maximum capacity of 190 SMs or 192 SMs/RSMs.
CMKP	Communications Kernel Process. The principal function of the CMKP is to maintain the process identifier (PID) translation tables. The information in these tables is responsible for *UNIX*[1]-RTR message communication between the OSDS and *UNIX*-RTR operating system environments.
CMP	Communication Module Processor. A processor connected to the communication module (CM) that provides call processing, routing, and recent change and verify (RC/V) functions. These functions were previously provided by the administrative module (AM).
CMPPUMP	Communication Module Processor Pump (Supervisor Process)
CNI	Common Network Interface
Compiler	A computer program that translates a set of program code written in a higher level language into machine language.
Control Register	The register used for initializing the bootstrapper.
CPI	Central Processor Intervention
CSD	Circuit Switched Data
CSV	Circuit Switched Voice
CTS	Control Time Slot
CTSP	Control Time Slot Pump
CTSPUMP	Control Time Slot Pump (Supervisor Process)

1. Registered trademark of The Open Group.

DA	Directed Audit
DA	Discontinued Availability
DALB	D-Channel Application Linkage Block
DBM	Database Monitor
DCDB	Digital Collection Data Block
DCF	Defensive Check Failure. A synonym for assert.
DCLU	Digital Carrier Line Unit. The *5ESS* switch hardware that consists of two service groups which concentrate six *SLC*® 96 carrier system mode I and II remote terminals.
DCTU	Directly Connected Test Unit
DF	Data Fanout. The circuitry that switches any directly connected peripheral interface data bus (DPIDB) time slot to any protocol handler (PH).
DI	Data Interface. A hardware circuit located in the module controller/time slot interchange (MCTSI) that terminates peripheral interface data buses (PIDBs). (See PIDB.)
Diagnostics	The programs that locate faults and verify repairs.
DLI	Dual Link Interface. Transmits time slot data between the time multiplexed switch (TMS) and the time slot interchange (TSI) via the network control and timing (NCT) links, transmits control data between the message switch (MSGS) and the switching module processor (SMP) via the control time slot on each NCT link, and delivers timing information to the TSI for distribution to the SM peripherals.
DLTU	Digital Line and Trunk Unit. The *5ESS* switch hardware that terminates digital trunks from remote switching modules (RSMs) or other offices via T1 facilities.
DMA	Direct Memory Access
DMERT	Duplex Multi-Environment Real-Time. An old term for the *UNIX* real time reliable (RTR) operating system. An operating system for the 3B20D computer. In the *5ESS* switch, it controls the operating system for distributed switching in the administrative module (OSDS-C). (See OSDS-C.)
DMI	Duplex Message Interface. The circuitry that terminates the message interface buses (MIBs) from the module message processors (MMPs).
DMU	Data Manipulation Unit
DN	Directory Number
Domain	A specific set of values that an attribute can have.
Double Indexing	A way of accessing data in a database such that the relation key is divided to produce two indexes. The first index is used to find an offset in the first data page that gives the block number of the pointer to a page at the second level. The second index number is used with the tuple size to find the offset in the second page where tuples are being stored.

DPB D-Channel Port Block

DPIDB Directly Connected Peripheral Interface Data Bus. The bus that carries user and control messages between the integrated services line unit (ISLU) and the packet switch unit (PSU).

DSI Digital Signal Interface

DSL Digital Subscriber Line

DSN Digital Switching Network

DSU Digital Service Unit

Dynamic Data Data that changes relatively frequently. Equipment status, call status, and work queues are examples of dynamic data.

EA Effective Address

EAI Emergency Action Interface

ECD Equipment Configuration Database. The database that describes the physical and logical configuration of the computer and the peripherals.

EGRASP Enhanced Generic Access Package

EIH Error Interrupt Handler

Element A component of an array, structure, or union.

Elevated Segmented Mode The mode of operation in which audits run at priority 1 and take real-time breaks.

Environment A collection of resources used to support a function.

ER Error Register. The register that logs errors from various points in the system. The ERs with the least significant bit have the highest priority.

Escalation A recovery action taken when a critical resource is in error and when taking segment breaks before the error is corrected would further degrade processing.

ESR Error Source Register. A register used to store error indications.

Event 1. A system fault. Events are assigned event numbers, which are printed in the output messages.

2. A two-bit piece of information sent by a process to notify another process about an impending message.

FC Feature Control. A set of procedures that sequence call processing actions. This call processing subsystem receives customer inputs (dialed digits), determines the actions to be performed, and calls on peripheral control (PC) to initiate hardware actions that advance the call.

FEX Feature Execution

Fiber Optics Cables made of glass fibers that are used to transmit data encoded into light pulses at very high speeds.

FI Full Initialization

FIDB Facility Interface Data Bus. The bus that transmits 32 time slots (TSs) from the digital line and trunk unit (DLTU) to the facilities interface unit (FIU) in a remote switching module (RSM).

Finite State Machine An abstraction that is useful for modeling event-driven sequential processes. A finite state machine consists of a finite set of states, a finite set of inputs, an output function,and a next state function.

FIU Facility Interface Unit

FP Fast Pump

FPC Foundation Peripheral Controller. A component of the message switch (MSGS) in the CM. Handles message traffic between the 3B processor and the module controller and also provides maintenance and control access to the network clock.

FPI Full Process Initialization

FPUMP Fast Pump

Function A section of code from a software design unit. A well-defined operation performed during development.

GIDB Group Interface Data Bus. The bus that carries user and control information between the basic rate interface (BRI) card and the line group controller (LGC).

Global Data The data in memory that is not allocated on a software releaess basis or at compile time to individual subsystems.

GRASP Generic Access Program

GRASP Generic Access Package

Halfword A halfword consists of 2 bytes or 16 bits.

Hardware Registers Registers that are used to give information about various hardware functions.

Hashing A more sophisticated way to access data in a database. It uses a relation key in an algorithm. The algorithm produces a number that is used to find the data.

HM Human Machine Interface. The maintenance software subsystem that provides interaction between maintenance personnel and the switching system.

HSM Host Switching Module

HSR Hardware Status Register. The bits that set the priorities for interface to the bidirectional gating bus (BGB).

I/O Input/Output

IC Integrity Control

IDCU Integrated Digital Carrier Unit

IDCULSI Integrated Digital Carrier Unit Loop Side Interface

IM Interface Module (now SM)

IM Interrupt Mask Register

IMR	Interrupt Mask Register
Index	A number that specifies an element in an array.
INIT	Initialization
Initialization Mode	The mode of operation in which audits are run during system initialization.
Insanity	The condition of the system in which abnormal processing occurs.
Interrupt	The method of notifying a hardware unit of certain asynchronous conditions.
IPL	Interrupt Priority Level
IRR	Interrupt Request Register
IS	In Service
IS	Interrupt Set Register
ISDN	Integrated Services Digital Network. A service of international scope that features simultaneous voice, interactive data, telemetry, bulk data, and video.
ISLU	Integrated Services Line Unit. The unit that houses the basic rate interface (BRI) cards.
ISR	Interrupt Source Register. The register that logs interrupts from 32 sources. The interrupts with the least significant bits have the highest priority.
ITIME	Operating system for distributed switching (OSDS) time relative to OSDS initialization.
JSR	Jump to Subroutine
Kernel	The lowest level of virtual machine in *UNIX* RTR. It controls the hardware directly and does not depend on other operating system services.
Kernel Process	A logical collection of software running at the level closest to the machine.
LBPUMP	Little Boot Pump (Supervisor Process)
LCCB	Logical Channel Control Block
LDSU	Local Digital Service Unit. The service unit that provides low-level signaling and measurement functions within the SM.
LDSUB	Local Digital Service Unit Bus. Carries data channels from the local digital service unit (LDSU) to the time slot interchange (TSI).
LGC	Line Group Controller
LIDB	Local Interface Data Bus
LIFO	Last In First Out
LLCB	Logical Link Control Block
LSI	Loop Side Interface
LSM	Local Switching Module

LU	Line Unit
Macro	A predefined shorthand. Arguments that can be part of a substitution process.
Mapping	A way of establishing a relationship between the elements of one set and the elements of another. The verification that is smaller address refers to the address in main memory. Performed by the storage address translator (SAT) and cache.
MCB	Message Control Block. A data structure linked to a process's process control block (PCB) that is used by the operating system for distributed switching (OSDS) to store messages to be retrieved by the receiving process.
MCC	Maintenance (Master) Control Center
MCC/ROP	Master Control Center/Receive Only Printer
MCTSI	Module Controller/Time Slot Interchanger. The switching module (SM) hardware unit that provides the control, time division switching, call processing, call supervision, and maintenance functions for the SM. Also called the module control and time slot interchange unit (MCTU).
MCTU	Module Controller and TSI Unit. The switching module (SM) hardware unit that provides the control, time division switching, call processing, call supervision, and maintenance functions for the SM. Also called the module controller/time slot interchange (MCTSI).
MIB	Message Interface Bus
MICO	Module Integrity Control
MICU	Message Interface Clock Unit
MIR	Micro-Instruction Register
MLHG	Multi-Line Hunt Group
MMP	Module Message Processor. Circuitry in the message switch (MSGS) that handles control message protocol on the link to the SM.
MMU	Memory Management Unit
Modularity	A design in which the hardware and software consist of many small units.
MOTD	Message Of The Day
MP	Module Processor
MPU	Microprogram Unit
MRF	Maintenance Reset Function
MSCU	Message Switch Control Unit
MSGS	Message Switch. The hardware in the communication module (CM) that is the center of all communication between the many processors in a *5ESS* switch.
MSKP	Message Switch Kernel Process
MSPU	Message Switch Peripheral Unit

Multiplexing	A process that enables a digital transmission system to transmit many telephone conversations over a single pair of wires.
NARTAC	North American Regional Technical Assistance Center
NCT	Network Control and Timing Links. Internal fiber optic links that connect the switching modules (SMs) with the communication module (CM) to provide time slot paths for network connections, carry control messages (time slots) to the modules, and distribute timing to the module.
NOC	Normalized Office Code
NOG	Network Operations Group
Nonreturning Assert	An assert that does not return to program control and also purges the current process.
Nonsegmented Mode	The mode of operation in which audits run to completion without real-time breaks.
ODBE	Office Data Base Editor
ODD	Office Dependent Data
ODP	On-Demand Packet
OKP	Operational Kernel Process. An environment in the administrative module (AM) in which application-oriented call processing occurs. The application software that is controlled by OSDS-C and combines with OSDS-C.
OM	Output Message
ONTC	Office Network Timing Complex
OOS	Out of Service
OP Code	Operation Code. The portion of an instruction that identifies the particular operation to take place.
Operating System	The software that organizes the storage of information, compiles programs, oversees input/output (I/O) operations, reports system status, provides the maintenance personnel interface facilities, provides program modification and debugging routines, and, in general controls system operations.
OSDS	Operating System for Distributed Switching. The *5ESS* switch real time operating system that provides identical interfaces to application software under its control in the AM, CMP, and SMs. Provides process management, memory management, interprocess communications, and timing for the application programs running under it.
OSDS Monitor	Input messages and flags that are used to gather performance data and investigate performance problems without the need for specialized tools such as logic analyzers. Developed to collect data about the operating system for distributed switching (OSDS) in the AM, CMP, and SMs.

OSDS-C	Operating System for Distributed Switching in the Administrative Module (AM). Software controlled by the *UNIX* RTR operating system, OSDS-C provides the interface with the operating system, the input/output, and the software in other subsystems in the AM.
OSDS-M	Operating System for Distributed Switching in the Switching Module (SM). Provides interfaces with other software subsystems in the SM and communications software used to send messages to either the AM or another SM.
OSDS-P	Operating System for Distributed Switching - Protocol Handler
OSDS-X	Operating System for Distributed Switching on Axp.
OSPID	A unique number assigned to each process. Includes processor ID, uniqueness, and process number. Maintained by OSDS.
OSPS	Operator Services Position System
PA	Program Address Register. The register that holds the address of the last address retrieved from main store.
PAB	Peripheral Address Bit
Packet	A grouping of data into a relatively small block (for example, 1000 bits) with an identifying header.
Parity Bit	The bit that is used to make the sum of the "1" bits in a word odd or even. In odd parity, the sum of the bits is odd, for even parity the sum of the bits is even. Parity is used to help verify that the information is valid.
PAS	Protected Application Segment
PB	Packet Bus. The buss that carries packet information between the packet switch unit (PSU) and the switching module processor (SMP).
PBX	Private Branch Exchange. A customer premises switching system.
PC	Peripheral Control. A software subsystem that manages and controls the switching peripherals. The call processing subsystem acts on a customer request collected by the feature control (FC) subsystem and insulates the FC and terminal maintenance (TM) subsystems from changes in hardware configuration.
PC	Program Change
PCB	Process Control Block. A block of relevant information about a process's environment. Stores all of the data that is required to represent and administer the process.
PCBI	Process Control Block Index
PCBLA	Process Control Block Link Area. A data structure assigned to a terminal process that links other data blocks to the process. Contains all pertinent data about a call.

PCM Pulse Code Modulation. A technique for coding analog signals for transmission on a digital circuit, by sampling the analog signal at regular intervals and converting each sample into a digital codeword.

PCRID Processor ID

PERAD Peripheral Address Register

PF Packet Fanout. The circuitry that sends control messages to or from the protocol handlers (PHs) and the control fanout (CF).

PH Protocol Handler. The circuitry that switches control messages between the basic rate interface (BRI) and the module controller/time slot interchange (MCTSI).

PHDB Protocol Handler Data Bus. The bus that transmits control messages between the data fanout (DF) and the protocol handler (PH).

PI Packet Interface. Circuitry that is only present in SMs that offer integrated services digital network (ISDN) services. Routes packet information between the switching module processor (SMP) and the protocol handlers (PHs) of the packet switch unit (PSU).

PIB Packet Interconnect Bus

PIC Peripheral Interface Controller. A 16-bit microprocessor. In the administrative module (AM), the PIC controls the I/O transfer between the main store (MAS) and the peripheral controllers. In the message switch (MSGS), the PIC handles the interface between the AM and the MSGS peripheral units (MSPUs).

PIC Programmable Interrupt Controller

PICB Peripheral Interface Control Bus. A duplex bus that carries clock, data, and control messages from the control interface (CI) to a switching module (SM) peripheral unit, and returns data and service requests to the CI.

PICO Protocol Handler Integrity Control

PID Process Identifier

PIDB Peripheral Interface Data Bus. A duplex bus providing 32 time slots of voice data between the data interface (DI) and a peripheral unit in the SM.

PIR Programmed Interrupt Request

PLL Phase Locked Loop

PLOD Process for Loading the ODD onto Disk

PLOP Process for Loading the ODD in the PAS

PMDB Process Message Data Block. A data structure assigned to a terminal process in an SM for receiving messages (PMDB-IN) or constructing and sending messages (PMDB-OUT).

PMKP Partition Mounting Kernel Process. The kernel process that mounts new *5ESS* switch test and office dependent data (ODD) partitions and creates the application integrity monitor (AIM) on full configuration CP boots.

Point-To/Point-Back Check
A verification process that determines whether a relation pointed to from a source relation points back to the source relation.

Pointer A variable used to store the address of another variable.

Popping The operation of removing a node from a stack.

PORTLA Port Linkage Area

Postinitialization Mode
The mode in which audits restore noncritical data structures after system initialization.

PPB Permanent Packet B-channel

PPC Pump Peripheral Controller. Circuitry located in the message switch (MSGS) that is responsible for the rapid reinitialization (fast pump) of the switching module (SM) in case of total failure.

PPD Permanent Packet D-channel

PPR Pulse Point Register

Process A programming entity that implements an activity. An instance of programming execution plus the data necessary for the execution.

PSBR Primary Segment Base Register

PSU Packet Switch Unit. Switches packet data and signaling information. A central processing unit including mainframe and peripheral equipment which, when connected via access lines to a customer node, will pass information between customer nodes.

PSUPH Packet Switch Unit Protocol Handler

PSW Program Status Word. The bits used by the system software to sent and maintain the status of the currently executing program.

PTSB Peripheral Time Slot Block

PUCR Pump Control

PUSG Peripheral Unit Service Group

Pushing The operation of adding a node to a stack.

RAM Random Access Memory

RBOC Regional Bell Operating Company

RC/V Recent Change and Verify

RCKP Recent Change Kernel Process

Relation An array of tuples.

Relational Database A logical view of data items organized into a table or tables.

RISLU Remote Integrated Services Line Unit

ROP Receive/Read Only Printer

Routine Segmented Mode
The mode of operation in which audits run at priority 0 and take real-time breaks.

RPI Return to Point of Interrupt

RSM Remote Switching Module

RSTSR Reset Error Source Register

RTA Routing and Terminal Allocation

RTA DCF Routing and Terminal Allocation Defensive Check Failure

RTC Real Time Clock

RTIME The operating system for distributed switching (OSDS) time relative to switch initialization

RTR Real Time Reliable. Designation of the *UNIX* operating system used in the 3B20D computers.

RWPAB Read/Write Peripheral Address Bit

Sanity The normal processing state of the system.

Sanity Timer The system integrity measure that detects malfunctions by checking the time required to perform system processes.

SAR Store Address Register

SAS Stop and Switch

SBT Stack Back Trace

SCB Stack Control Block. The representation of a stack in the operating system for distributed switching (OSDS), including the stack location and the process that owns the stack.

SCC Switching Control Center. A centralized system that controls the switching operations of many electronic switching systems (ESSs).

SDFI Subscriber Digital Facility Interface

SDL State Definition Language

SDLC Synchronous Data Link Controller

SDR Store Data Register

Segment 1. An interval of time during which an audit runs.

2. The basic memory unit in RTR, composed of from 1 to 64 pages. Each segment is 512 32-bit words in length.

Segmented Programs that may be interrupted at certain points to let other processes execute.

SG Service Group

SI Selective Initialization. The form of initialization in which transient calls (those note in a stable talking state) are disconnected.

SI System Integrity. The maintenance software subsystem that provides treatment of software errors and handling of overload conditions.

SICO System Integrity Control

Signaling The switching function that transmits information about lines, trunks, or calls to control switching equipment, station equipment, and other aspects of call handling.

SIM System Integrity Monitor

SIR Store Instruction Register

SM Switching Module. A module controller/time slot interchange (MCTSI) along with a number of peripherals. The general name for remote SMs (RSMs), host SMs (HSMs), and local SMs (LSMs). Previously, the SM was called the interface module (IM).

SMKP Switch Maintenance Kernel Process. Switch maintenance software that is controlled by OSDS-M that works with OSDS-M and OSDS-C to provide the interface for maintenance software and switch maintenance processes in the message switch (MSGS) and switching modue (SM).

SMP Switching Module Processor. Microprocessor in the module controller/time slot interchange (MCTSI) that performs call processing and maintenance functions, controls the peripheral units, and communicates with other SMs, the CMP, and the AM.

SODD Static Office Dependent Data

SP Signal Processor. Circuitry in the module controller/time slot interchange (MCTSI) that checks the time slot signaling bits for changes in their state and reports the changes to the switching module processor (SMP).

SPC Stored Program Control

SPP Single Process Purge. The form of initialization in which a single process is terminated.

SRC Source

SSR System Status Register. The register that controls the status of the system configuration.

Stack A section of memory used for short-term storage. Stores local variables and return points for function calls. A linked list in which additions and deletions can only be made at one end.

Standby In this state, the equipment is not active but available to be switched to the active state.

Static Data Data that is permanently stored in the *5ESS* switch. It represents, for example, equipment, lines, and trunks.

Structure See C Language structure.

System Process A process that provides services on a system-wide basis, within or outside of call processing. It provides one service (such as routing) and manages requests from more than one terminal process.

TCB Timer Control Block. The representation of a timer in the operating system for distributed switching (OSDS); linked to the process's process control block (PCB) when the process requests a timer.

TEDB Terminal Equipment Data Block

TEILA Terminal Endpoint Identifier Linkage Area

Terminal Process A process that is created on demand to provide customer services (such as calls) and terminal maintenance, then the process is terminated.

Time Slot The time interval to which sequential representations of a given signal can be assigned.

TM Terminal Maintenance

TMS Time Multiplexed Switch. In the *5ESS* switch communication module (DM), the TMS provides the physical path for the digital signals carrying data and control information between SMs and between the SMs and the AM.

TSI Time Slot Interchange. Circuitry that performs the time division switching between the peripheral units of the SM and the time multiplexed switch (TMS); rearranges the order of the time slots.

TU Trunk Unit

Tuple The content of any row in a relation.

TV Transfer Vector

Two-Dimensional Array An array having more than one row. A two-dimensional array is implemented as an array of arrays.

Union A type of structure that saves memory space by overwriting portions of it. A device that allows storage of different data types in the same memory space. Unions are set up much the same way as structures.

UNIX RTR Operating System *UNIX* Real Time Reliable (RTR). The main operating system in the 3B administrative module (AM). Previously, the *UNIX* RTR operating system was called the DMERT operating system.

UP Program Update Subsystem

VBD Voice Band Data

Virtual Address The addresses, starting at zero, that a programmer reserves in the program for the program. Basically, it tells how much space the program needs to run.

Virtual Machine Software that performs lower-level functions. In other words, all of the code and data needed to do a specific function contained in one set of programs.

VLMM Very Large Main Memory

VLSI Very Large Scale Integration

Word A discrete area of memory, usually four bytes long. The size is machine dependent. In the AM, the size is four bytes; in the SM, the size is 2 bytes.

A

Address modes
  3B20D and 3B21D computer addressing modes, 4.1-4
  Intel 80186 processor addressing modes, 4.5-5
  MC68000 processor family addressing modes, 4.3-4
Array size limitations - *5ESS* switch, 3.2-1
Assert reports
  Assert messages, 5.1-1
Asserts
  CMP assert analysis example, 5.2-9
  Handler - assert, 5.1-1, 5.1-8, 5.1-9
  Macro - AUCFTASRT, 5.1-4
  Macro - AUCFTREFASRT, 5.1-4
  Macro - RTDUMPDATA, 5.1-5
  Macro - RTRTGERR, 5.1-5
  Macros, 5.1-2
  Memory stack - AM, 5.1-8
  ROP output - assert, 5.1-5
  SM assert analysis example, 5.2-1
  SM/CMP memory stack, 5.1-11
  Stack frame variations - AM and SM/CMP, 5.1-7
Assistance, Technical, 1-5
Auxiliary Status Register, 8.2-47

C

C language
  5ESS switch array size limitations, 3.2-1
  5ESS switch sizeof distinctions, 3.2-1
  Arithmetic operators, 3.1-4
  Arrays, 3.1-15
  Bitfields, 3.1-3
  Cast, 3.1-20
  Control statements, 3.1-22
  Data types, 3.1-1
  Functions, 3.1-21
  Logical operators, 3.1-5
  Membership operator, 3.1-19
  Multiple structure assignments - *5ESS* switch, 3.2-1
  Operator precedence, 3.1-6
  Pointers, 3.1-15
  Preprocessor, 3.1-14
  Relational operators, 3.1-5
  Storage class specifiers, 3.1-12
  Structures and unions, 3.1-11
  Unsigned data types - *5ESS* switch, 3.2-1
  Variables (local and global), 3.1-10
C programming language, 3-1
Comment Form, 1-4
CTAM, 1-5
Customer Information Center, 1-5

Customer Technical Assistance Management (CTAM), 1-5

D

Data conversion
3B20D and 3B21D processor data conversion rules, 4.2-4
Intel 80186 processor data conversion rules, 4.6-3
MC68000 processor data conversion rules, 4.4-3
Data dump layout
MCB, 13-3
PCB, 13-2
SCB, 13-2
TCB, 13-3
Data sizes and alignment
3B20D and 3B21D processor data sizes and alignment, 4.2-2
AM, SM, and CMP data sizes and alignment, 5.1-10
Intel 80186 processor data sizes and alignment, 4.6-1
MC68000 processor data sizes and alignment, 4.4-1
Document Distribution, 1-5
Documentation Hotline, 1-5

E

Event history
SM/CMP/PI/PH event history, 11-1
EXAMPLES
GRASP/EGRASP Analysis Example, 6-16

F

Feature execution (FEX), 12.1-4
Messages, 12.1-4

G

Generic Utilities
Breakpoint usage - SM, CMP, and peripherals, 7-1
GRASP/EGRASP
ALW command - AM, 6-7
Breakpoint definition - AM, 6-7
COPY command - AM, 6-1
DUMP command - AM, 6-3
IN:DTIME command - AM, 6-10
LOAD command - AM, 6-5
Message Acknowledgements - AM, 6-14
Overriding default time limits - AM, 6-10
Trace and matching messages - AM, 6-11
Transfer trace function - AM, 6-10
WHEN command - AM, 6-7

H

Hotline, Documentation, 1-5

I

Information Product Hotline, 1-5
Interrupt
  Analysis, 8-1
  Definition, 8-1
  Types, 8-1
Interrupt Enable Register, 8.2-45
Interrupt Hierarchy, 8.2-35
Interrupts
  3B20D processor error register, 8.1-6
  3B20D processor error register bit configuration, 8.1-8
  3B20D processor interrupt source register, 8.1-11
  3B20D processor interrupts, 8.1-1
  AM interrupt ROP output, 8.3-1
  Autovectored interrupts, 8.2-17
  Bootstrapper, 8.2-4
  CI error source register, 8.2-27
  Control interface (CI), 8.2-4
  Control interface (CI) error sources, 8.2-14
  Data interface (DI), 8.2-4
  DLI error source register, 8.2-23
  DLI error source register 1, 8.2-32
  DLI error source register 2, 8.2-33
  Dual link interface (DLI), 8.2-4
  Dual link interface (DLI) error sources, 8.2-16
  Error interrupt handler (EIH), 8.1-2, 8.1-5
  Error sources of interrupts by category, 8.2-14
  Execution levels, 8.1-3
  Hardware error source register, 8.2-22
  Interrupt stack, 8.1-1
  Interrupts analysis example - hardware, 8.4-1
  Interrupts analysis example - software, 8.5-1
  ISDN peripheral units, 8.2-13
  Levels of interrupts - *Motorola* MC68000 processor family, 8.2-17
  Local digital service unit buses (LDSUBs), 8.2-12
  Masking - 3B20D processor, 8.1-12
  Masking - module processor, 8.2-50
  Masking - *Motorola*MC68000 processor, 8.2-50
  MCTU interrupt registers, 8.2-18
  Memory error source register, 8.2-23
  Module controller/time slot interchange (MCTSI) functions, 8.2-1
  Module controller/time slot interchange (MCTSI) interfaces, 8.2-9
  Module controller/time slot interchange (MCTSI) subunits, 8.2-3
  MOTOROLA MC68000 Processor Family, 8.2-1
  Motorola MC68000 processor family level 4 interrupts, 8.2-24
  Motorola MC68000 processor family level 7 interrupts, 8.2-18
  Network control and timing (NCT) links, 8.2-12

Non-operational interrupt: error - 3B20D processor, 8.1-3
Non-operational interrupt: maintenance - 3B20D processor, 8.1-6
Packet bus (PB), 8.2-12
Packet interface (PI), 8.2-4
Peripheral interface control buses (PICBs), 8.2-11
Peripheral interface data buses (PIDBs), 8.2-10
PI error source register, 8.2-26
PIC B register, 8.2-25
PIC C register, 8.2-26
PICB circuitry related error sources, 8.2-14
Reset source register, 8.2-19
Serial control messages (distribute/scan orders), 8.2-2
Service requests, 8.2-14
Signal processor (SP), 8.2-4
Signal processor (SP) error sources, 8.2-15
SM, 8.2-1
SM interrupt ROP output, 8.3-1
Software error source register, 8.2-21
SP error source register, 8.2-28
Status register and interrupt masking structure - *Motorola*MC68000 processor, 8.2-51
Switching module processor (SMP), 8.2-4
Switching module processor (SMP) error sources, 8.2-16
Time slot interchange (TSI), 8.2-4
Time slot interchange (TSI) error sources, 8.2-15
TSI error source register 1, 8.2-29
TSI error source register 2, 8.2-30
TSI error source register 3, 8.2-32
Vectored interrupts, 8.2-17
ISDN
Packet interface (PI), 8.2-4
Switching types in ISDN, 8.2-13

L

Listings menu, 2-2
Listings menu options, 2-3
Log file
Postmortem log file (PMLOG), 8.1-6
Log files
Error interrupt handler log file (ERLOG), 8.1-4
Memory history log file (MEMLOG), 8.1-4
Lucent Technologies Customer Information Center, 1-5

M

MCB data dump layout, 13-3
MCB overloads, 13-1
MCTSI
Bootstrapper, 8.2-4
Control interface (CI), 8.2-4
Data interface (DI), 8.2-4

Dual link interface (DLI), 8.2-4
Module controller/time slot interchange (MCTSI) functions, 8.2-1
Module controller/time slot interchange (MCTSI) interfaces, 8.2-9
Module controller/time slot interchange (MCTSI) subunits, 8.2-3
Packet interface (PI), 8.2-4
Signal processor (SP), 8.2-4
Switching module processor (SMP), 8.2-4
Time slot interchange (TSI), 8.2-4
Memory layout
3B20D and 3B21D processor instruction example, 4.1-8
3B20D and 3B21D processor instruction template, 4.1-8
Intel 80186 processor instruction example, 4.5-6
Intel 80186 processor instruction template, 4.5-5
MC68000 processor instruction example, 4.3-9
MC68000 processor instruction template, 4.3-5
Memory management
3B20D and 3B21D processor memory segment, 4.2-4
Intel 80186 processor memory management, 4.6-3
Interrupt stack, 8.1-1
MC68000 processor optimization, 4.4-4

N

NARTAC, 1-5
North American Regional Technical Assistance Center (NARTAC), 1-5

O

Online access procedure, 2-1
Online program listings, 2-1
Operands
3B20D and 3B21D computer operands, 4.1-4
Intel 80186 processor operands, 4.5-4
MC68000 processor family operands, 4.3-4
OSDS
Message control block (MCB), 12.1-3
Messages, 12.1-4
Process control block link area (PCBLA), 12.1-2
Process control block (PCB), 12.1-2
Process message data block (PMDB), 12.1-2
Processing levels, 12.1-3
Resources, 12.1-1
Segment breaks, 12.1-3
Stack control block (SCB), 12.1-2
System processes, 12.1-1
Terminal processes, 12.1-1
Timer control block (TCB), 12.1-3
OSDS monitor
Buffer, 12.2-1
Buffer layout for the AM, 12.7-1
Buffer layout for the CMP, 12.7-13
Buffer layout for the SM, 12.7-9

Client data dump control flags, 12.4-4
Data control flags for per-event data, 12.4-2
Data dump control flags, 12.4-4
Event control flags for per-event data, 12.4-3
Functions, 12.2-1
Input commands to clear and dump the buffer, 12.5-1
Input message control flags, 12.4-1
Input messages, 12.3-1
Messaging control flags, 12.4-1
MSKP data dump description, 12.4-31
OKP data dump description, 12.4-29
OSDS usage control flags, 12.4-3
Output messages, 12.5-2
SM and CMP data dump description, 12.4-29
SMKP data dump description, 12.4-30
Snapped data dump layouts, 12.8-1
Start and stop control flags, 12.4-1
up-loading contention, 13-4
Useful input message sequences, 12.6-1
What to do flags, 12.4-5
What to dump flags, 12.4-27
OSDS overload monitor
output file, 13-1
overview, 13-1
Output file location and layout
OSDS overload monitor, 13-1
Overloads
MCB, 13-1
PCB, 13-1
SCB, 13-1
TCB, 13-1
overview
OSDS overload monitor, 13-1

P

PCB data dump layout, 13-2
PCB overloads, 13-1
PERAD register, 8.4-2
PIC-A Register, 8.2-38
PIC-B Register, 8.2-41
PIC-C Register, 8.2-43

R

Read Interrupt Status Register, 8.2-46
References, 1-5
Register
PERAD, 8.4-2
Registers
3B20D and 3B21D processor error register, 4.1-3
3B20D and 3B21D processor hardware registers, 4.1-2

3B20D and 3B21D processor hardware status register, 4.1-4
3B20D and 3B21D processor interrupt mask register, 4.1-3
3B20D and 3B21D processor interrupt set register, 4.1-3
3B20D and 3B21D processor register notation, 4.1-2
3B20D and 3B21D processor registers, 4.2-6
3B20D and 3B21D processor system status register, 4.1-3
3B20D processor error register, 8.1-6
3B20D processor error register bit configuration, 8.1-8
3B20D processor interrupt source register, 8.1-11
3B20D processor primary segment base register, 8.1-2
3B20D processor program address register, 8.1-2
3B20D processor secondary segment base register, 8.1-2
3B20D processor status word register, 8.1-2
3B2XD processor argument pointer, 5.1-8
CI error source register, 8.2-27
DLI error source register, 8.2-23
DLI error source register 1, 8.2-32
DLI error source register 2, 8.2-33
Hardware error source register, 8.2-22
iAPX 8 byte registers, 4.5-4
iAPX word registers, 4.5-3
Intel 80186 processor 8 byte registers, 4.5-4
Intel 80186 processor data registers, 4.6-5
Intel 80186 processor pointer and index registers, 4.6-5
Intel 80186 processor segment registers, 4.6-5
Intel 80186 processor status and control registers, 4.6-6
Intel 80186 processor word registers, 4.5-3
MC68000 processor address registers, 4.4-6
MC68000 processor control registers, 4.3-3
MC68000 processor register notation, 4.3-3
Memory error source register, 8.2-23
Motorola MC68XXX processor family distinctions, 8.2-33
PI error source register, 8.2-26
PIC B register, 8.2-25
PIC C register, 8.2-26
Program status word register, 4.1-3
Reset source register, 8.2-19
Software error source register, 8.2-21
SP error source register, 8.2-28
Status register, 8.2-51
TSI error source register 1, 8.2-29
TSI error source register 2, 8.2-30
TSI error source register 3, 8.2-32

S

SCB data dump layout, 13-2
SCB overloads, 13-1
Single process purge (SPP)
  Single process purge (SPP) analysis example, 9-2
SMP40 Interrupt Hierarchy, 8.2-37
SMP60 Interrupt Hierarchy, 8.2-49

Stacks
3B20D and 3B21D processor stack usage, 4.2-7
Intel 80186 processor stack usage, 4.6-7
MC68000 processor stack usage, 4.4-7
Status and Control Register, 8.2-50

T

TCB data dump layout, 13-3
TCB overloads, 13-1
Technical Assistance, 1-5
Test Utility Bus Status Register, 8.2-47
Transfer vector, 4.2-5, 4.4-6, 4.6-7

U

Up-loading contention
OSDS monitor, 13-4
User Feedback, 1-4
User Feedback Form, 1-4
Using program listings, 2-4
Using the program listings, 2-1