TABLE OF CONTENTS

PREFACE

Memory iButtons are high–capacity, general–purpose electronic data carriers, each with a registered serial number. Organized like a floppy disk, Memory iButtons can store data files for multiple applications within the same device.

This manual summarizes technical and mechanical specifications for the iButton$^{\text{TM}}$ family. It is intended as a guide to enable readers to integrate iButtons in their own identification systems.

The contents are as follows:

iButton^{TM} OVERVIEW

CHAPTER 1: iButton^{TM} OVERVIEW

I. Introduction

Although human–readable labels have been used for ages, it was the advent of computer–readable labels that quickly revolutionized the way grocery stores operate and made possible the overnight delivery of packages. When error–prone — and time consuming — key entry was replaced by bar codes, it became convenient to build large databases to help in making accurate and timely decisions.

In the next step in the evolution of labelling technology, ink–on–paper bar codes are surpassed by silicon media. With Dallas Semiconductor's automatic identification technology, a chip becomes the label that can serve as a standalone data base. Attached to an object or carried by a person, the chip identifies and carries relevant information available instantly with little or no human intervention. People access secure areas with convenience, health care professionals accurately create records, and workers efficiently track items as they travel along the assembly line.

Three distinct limitations of bar codes are overcome by chips:

1. They hold significantly more information.
2. Information on the chip can be changed; chips can be updated via computer while affixed to their object.
3. Cost of access points, that is the communication with computers, is drastically lower because of direct, chip–to–chip digital data transfers.

II. Identification by Touch

The lowest cost method of making a chip into a computer–readable label is to extend its internal connections out to electrical contacts suitable for probing. The simplest arrangement is a single data lead plus a ground contact. In this way, a two–piece stainless steel container called a MicroCan^{TM} serves both as protective housing and electrical contacts: surface (data) and rim (ground). Its circular shape guides a simple, cup–shaped probe over its rounded surfaces even if struck with significant misalignment. The 16 mm button shape serves all iButtons.

While iButtons share some of the characteristics of bar codes, these chip–based data carriers have many advantages over ink–on–paper technology:

- iButtons can be read without expensive electro–optical equipment.
- iButtons can hold up to 100 times the data of bar codes, with larger capacities in development.
- Each iButton proves its identity by its unique serial number.
- The serial number of an iButton acts as a globally unique node address to access the device as part of an unlimited network.
- The contents of the chip data carriers can be changed while attached to an object.
- iButtons can accommodate over one million changes.
- A clam–shell, steel container called a MicroCan is better suited to harsh operating environments.
- Hand–held equipment can be made smaller, lighter, and less expensive since virtually no energy is needed to read or write.

All communication with iButtons is reduced to a single signal plus ground for a simple, self–aligning contact. Long and short pulses encode the binary 1's and 0's. Because iButtons are digital circuits, they talk directly to other chips in a computer, resulting in minimal cost interface using one CMOS/TTL logic signal. A reader/writer for iButtons can be implemented with just one spare I/O line of a microcomputer, often a free resource in a system.

III. Alternate Identification Technologies

iButtons expand on existing Auto ID technologies. This section discusses some of the limitations of existing technologies and how iButtons overcome them.

A. Bar Codes

Bar code systems require electromechanical printers and complicated electro–optical readers that must cope with marginal signals as they occur with changing scanning speed, varying scanning angle, poor contrast or dirt. Sunlight impairs the readability of the bar code due to high ambient light. After the reflected light is converted to an electrical signal, the symbology must be decoded to obtain the desired character code.

By contrast, iButtons need no optics or decoding since the information can be stored as ASCII characters. It can deliver the ASCII characters directly, at a rate of 2,000 characters per second (16.3 kbps). This open information structure allows system integration which is both hardware and software independent. Furthermore, the "scanner", "printer", unique reference number and the computer interface are built into the chip.

B. Magnetic Stripes

Another method of identification is magnetic stripes on plastic carriers (e.g., credit cards) or paper ticket stock. Like bar codes, this method must overcome analog signals. Further, data can be altered easily with just a small magnet. Strong magnetic fields common to many environments can inadvertently erase data. Magnetic stripes are also sensitive to dirt that will scratch the reading coil of the card reader and damage the tape itself. Since the data density of magnetic stripes is significantly higher than that of bar codes, the readers need precise mechanics for correct alignment and smooth and continuous movement of the card. Magnetic stripes are unsuitable for labeling; they have to be removed from the object not only for writing but also for reading.

iButtons, by contrast, are self–centering. No alignment is required; a simple touch is all that is required to access digital information.

C. Chip Cards

Chip cards are credit card–size, multi–layer plastic cards that contain a complete microcontroller or memory and an 8–contact, gold–plated probe area for connection with the host computer and power supply. They are not designed for high resistance intermittent contacts. Since chip cards have eight contacts versus iButton's two, they are sensitive to alignment and the sequence in which contacts are made. For economic reasons, the contacts carry only a thin soft gold plating which may easily wear off, exposing the copper layer. Exposed copper forms a hard oxide which decreases the contact quality and leads to card malfunction. Another problem with cards is mechanical bending. The plastic material itself is flexible but the chip inside is as hard as glass. The chip can crack or the thin gold wires connected to the chip can rip off. Chip cards are also unsuitable for labeling, since they have to be removed from the object for both reading or writing. The whole system functions only if the card is inserted in the right way (four possibilities) and all eight contacts are made. Due to the limited lifetime of the contacts and the multi–layer structure, chip cards are throw–away products at prices that are not throw–away.

iButtons, on the other hand, are designed for poor, intermittent contacts and withstand large mechanical stresses. They need only two contacts, which are insensitive to angular orientation. By design it is not possible to probe iButtons incorrectly.

D. RF Tags

Although RF Tags are very convenient, they have some inherent problems. Depending on the desired range of reception, the energy consumption may be quite high. Wide variations in the minimum and maximum range make zoning difficult. RF Tags are prone to interference from intentional transmitters (radio stations) and unintentional transmitters (electronic equipment, engines, neon lamps, etc.). More serious problems are the availability of frequencies for the receive and transmit channels and the approval of national authorities. Every country has its own rules and frequencies, which prevents a common standard for world–wide use. Another issue – usually neglected – is the influence of electromagnetic fields on human bodies.

iButtons do not need radio frequencies, since data is transferred by electrical conductivity during the momentary contact. This allows their use without any license in every country. The metal package shields iButtons against electromagnetic fields and allows trouble–free operation even under intense electro–magnetic fields. Multiple iButtons sharing the same conductive surface can be individually read or written by the same contact. The specificity of the contact makes zoning precise and the digital communication gives contact ranges up to 300 meters.

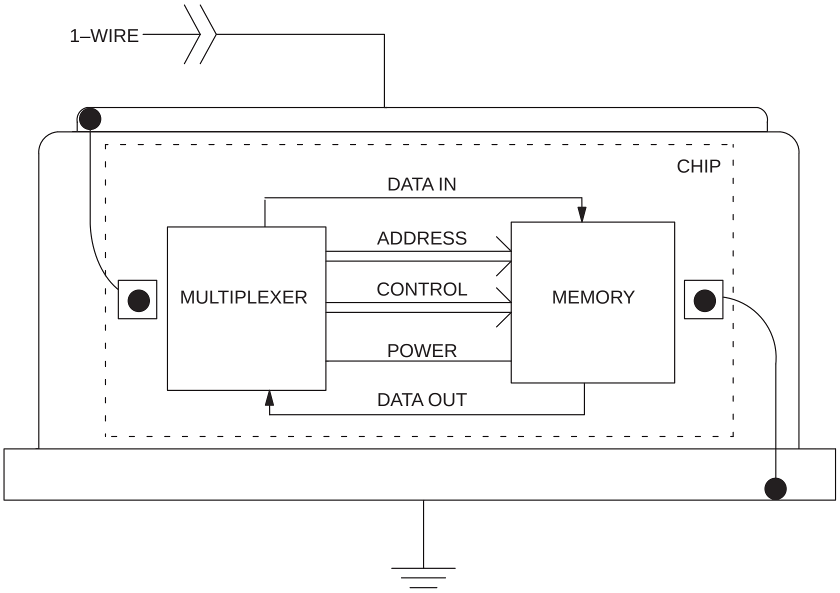
IV. Basics of iButton Operation

A. Technology

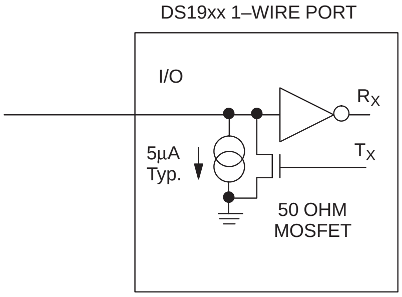
An iButton is a chip housed in a stainless steel enclosure. To keep the cost of access low, the electrical interface is reduced to an absolute minimum, i.e., one data line plus ground. The energy needed for communication is “stolen” from the data line (“parasitic power”) .

Figure 1–1 gives a general overview of an iButton. The chip inside is produced using CMOS technology and consumes only leakage current when in an idle state. To keep energy consumption as low as possible during active times, and to be compatible with existing logic families, an iButton's data line is designed as an open drain output (see Figure 1–2). The current source from the data line to ground returns the data line to ground if the iButton is removed from the probe. The open drain interface makes iButtons compatible with all microprocessors and standard logic systems. In a CMOS–environment, only a nominal 5 kΩ pull–up resistor to 5V V_{DD} is required to get normal operating conditions on an open–drain–type bidirectional port (see Figure 1–3). If input and output of the processor use separate pins, the wiring shown in Figure 1–4 will provide an appropriate interface.

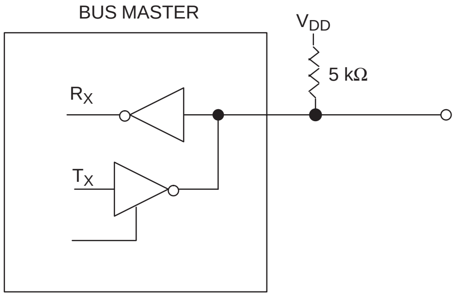
iButton BLOCK DIAGRAM Figure 1–1



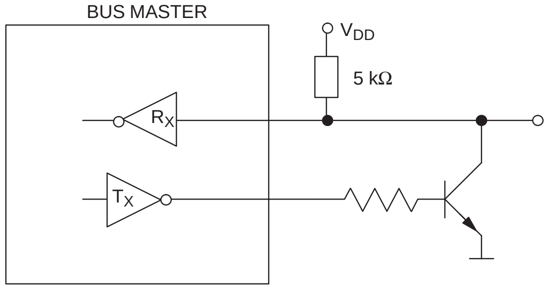
iButton INTERNAL DATA INTERFACE Figure 1–2



BUS MASTER CIRCUIT (OPEN DRAIN) Figure 1–3



BUS MASTER CIRCUIT SEPARATE I/O Figure 1–4



B. Protocol

In a simple environment as described above, an optimized approach for bidirectional communication, called the 1–Wire protocol, is used. The serial transfer is done half–duplex (i.e., either transmit or receive) within discretely defined time slots. In every case, the microcontroller (as the master, using a cup–shaped probe) initiates the transfer by sending a command word to the button–shaped slave iButton. Similar to electric plugs, where the male and female ends define sink and source, in the touch environment the cup–shaped probe defines the master and the button shape indicates the slave. This clean definition avoids conflicts like masters talking to each other.

Commands and data are sent bit by bit to make bytes, starting with the least significant bit. The synchronization of master and slave is based on the sharp slope that the master generates by pulling the data line low. A certain time after this slope, depending on data direction, either the master or the slave samples the voltage on the data line to get one bit of information. This method of operation is called data transfer in time slots. Each time slot is independently timed so that communication pauses can occur between bits if necessary, without causing errors. Figure 1–5 illustrates the general characteristics of this communication.

C. Synchronization

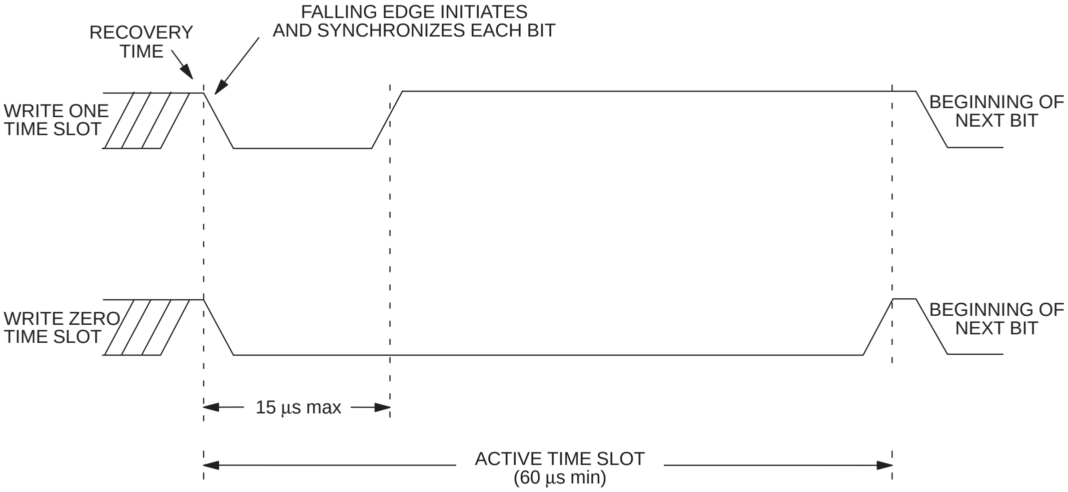
Data transfer cannot be done before the iButton and master are connected, i.e., before the memory touches the data and ground line of the microcontroller. Just a few microseconds after the connection is established (after touching), the iButton pulses the data line low to tell the master that it is on the line and is waiting to receive a command. This waveform is called a presence pulse. The master can also request an iButton to give a presence pulse by issuing a reset pulse. If the iButton receives a reset pulse or is disconnected, it will sense a low level on the data line and will generate a presence pulse just after the line reaches the high level again. A complete Reset/Presence Pulse sequence is shown in Figure 1–6.

D. Data Transfer

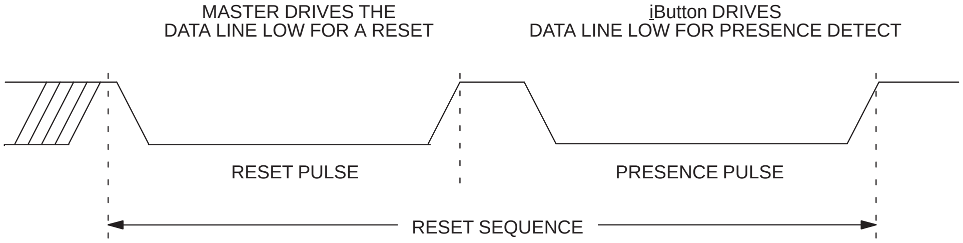
After the presence pulse, the iButton expects to receive a command. Any command is written to the iButton by concatenating write–one and write–zero time slots to create a complete command byte.

The data transfer in the opposite direction (reads from iButton) uses the same timing rules to represent a 1 or a 0, respectively. Since iButtons are designed to be slaves, they leave it to the master to define the beginning of each time slot. To do this, the master simply initiates a write–one time slot to read a data bit. If the iButton has to send a 1, all it has to do is wait for the next time slot. If it has to send a 0, the iButton will hold the data line low for a specified time, in spite of the release by the master. An example of a complete command sequence starting with a presence pulse and ending with data is shown in Figure 1–7. The activity of the master is drawn in bold lines. Gray lines mark the response of the iButton. Thin lines indicate that neither is active. The line is pulled high by a resistor.

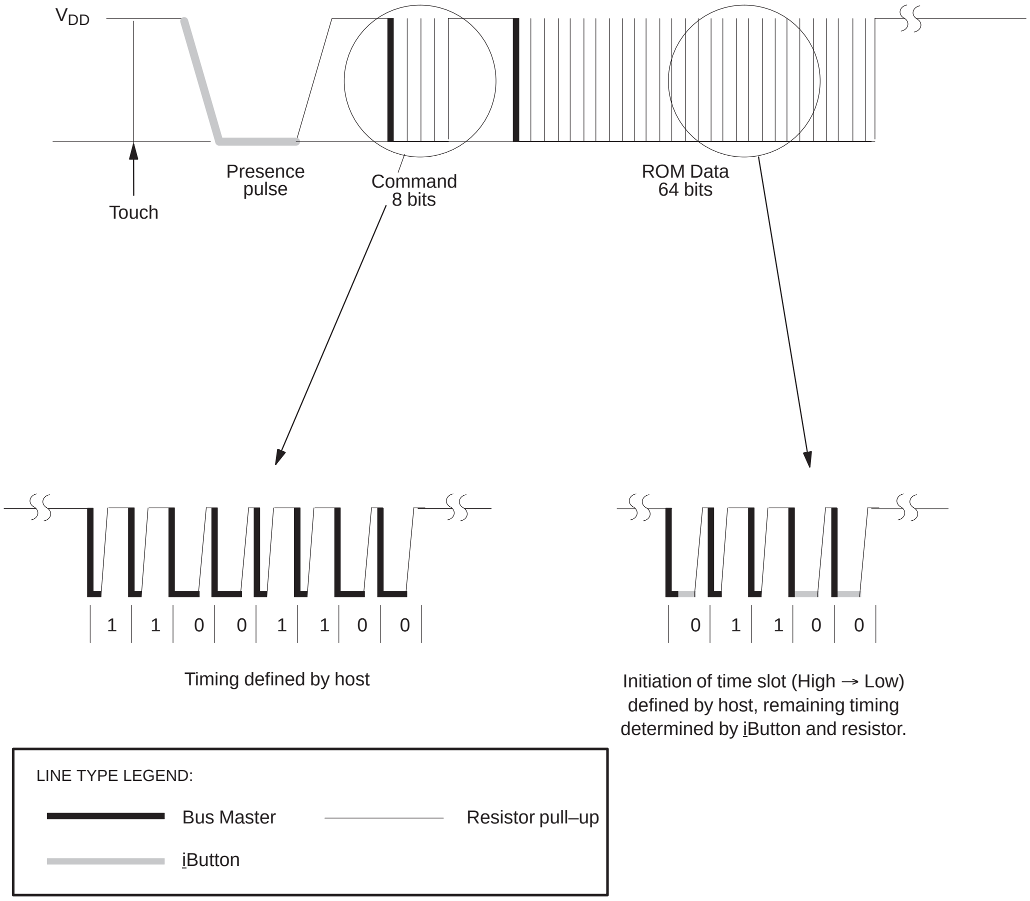
DATA TRANSFER IN TIME SLOTS Figure 1–5



RESET AND PRESENCE PULSE Figure 1–6



EXAMPLE READ ROM Figure 1–7



V. Memory iButton Application Example

A. Introduction

The attached Memory iButtons are mini databases for their associated object. For a minimal system, the user needs at least one personal computer to read and write Memory iButtons. For mobile workers, it is necessary to read and write Memory iButtons on the go. To allow this, many portable computers including iButton Recorder, iButton Editor, and Touch Transporter have been developed by independent companies. Figure 1–8 shows a general application example including all components and possible data paths.

B. iButton Recorder

The iButton Recorder is a pen–shaped mobile reader/writer for Memory iButtons. It can display (optional), rubber stamp, prompt the operator, read/write (designated A and G in Figure 1–8), store and time–stamp data from readings and dump that data either to a Transporter (D), another iButton Recorder (K), an iButton Editor (L), or to the system PC (H). The iButton Recorder loads its own application software via the serial port adaptor from the PC (H).

C. iButton Editor

iButton Recorders are especially relevant where many Memory iButtons need predefined data updates. The iButton Editor is a hand–held computer that provides all the functions of the iButton Recorder and additionally can accept data and commands via its keyboard. It can read and write data from/to Memory iButtons (C) and Transporters (F), exchange data with the system PC (I), read iButton Recorders, and supply data to be read by iButton Recorders (G).

D. Touch Transporter

For technical and economic reasons, networks cannot be made to link every point. Therefore other data carriers are needed, such as floppy disks. Like floppy disks, Memory iButtons are general data carriers. If a higher capacity Memory iButton is needed than is currently produced, Memory iButtons can be ganged together to form a larger capacity Memory iButton, referred to as a transporter. This high–capacity Memory iButton can act as a data dump for iButton Recorders (D) and iButton Editors (F). The dumped data can then be read by the system PC (E) or by an iButton Editor (F). The system PC can also write data to a Transporter (E) to be dumped later to an iButton Editor (F).

E. Archive Computer

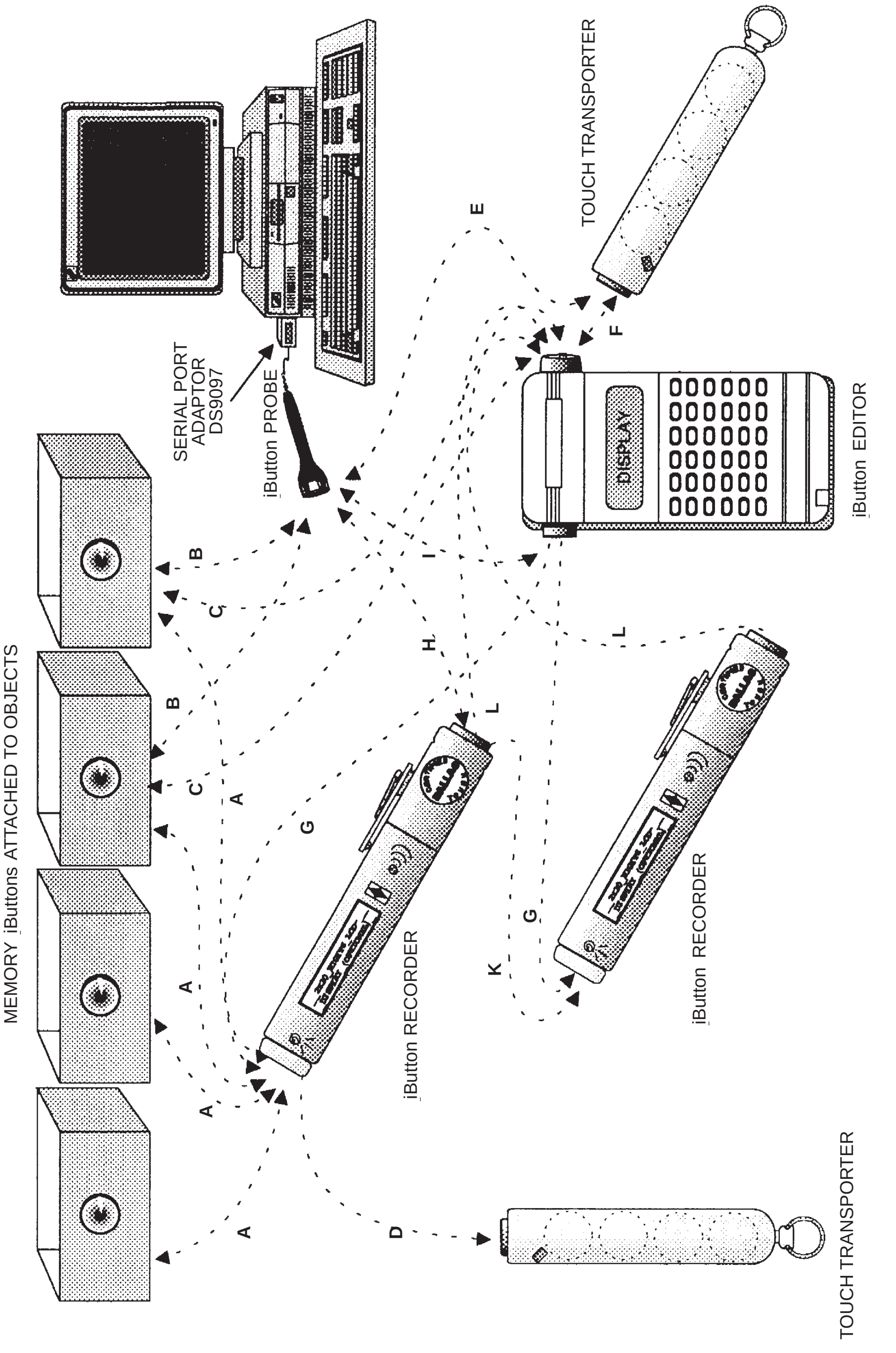
This computer can hold an inventory of all objects carrying iButtons. It receives new data about the objects, their contents and location via iButton Recorders (H), Transporters (E), iButton Editors (I) or directly (B). It can write to Memory iButtons either via iButton Recorder (H,A), iButton Editor (I,C) or directly (B). It also can load new application software to iButton Recorders (H).

VI. Chapter Summary

iButtons let users convert manual information gathering, data transport and identification into a completely electronic system. Equivalent to a document number, the unique serial number of each iButton acts as node address within an unlimited network. The memory acts as buffer storage, collecting information while insulated from the network. Data is then deposited to the network with a simple touch. In contrast to paper labels, Memory iButtons can be read and written, making them reusable for a virtually unlimited number of cycles. A kind of re–writable silicon label, the Memory iButton replaces paper documents that are difficult to attach to objects and are prone to damage or illegibility. Data stored in Memory iButtons is directly available as a digital signal, which is the native language of all computers.

iButton provide a very high immunity to electro–magnetic fields, mechanical stress and dirt. They can be reprogrammed with the same probe that reads them. No additional equipment is required to keep information up–to–date, permitting Memory iButtons to be recycled for thousands of uses. The flexibility and the excellent price/performance ratio of silicon auto ID technology is based on standard mass–produced iButtons and customer–specific software. To realize a specific application, first a data flow chart including type and quantity of data must be detailed.

iButton ENVIRONMENT AND DATA FLOW CHART Figure 1–8



PRODUCT OVERVIEW

CHAPTER 2: PRODUCT OVERVIEW

I. Common Features

A. Mechanics

The iButton MicroCan is 16.3 mm in diameter. There are two standard thicknesses: 3.1 mm and 5.9 mm. Devices that are powered by the master via the data line (parasite–powered) are available in both package types. All other devices are available in the 5.9 mm MicroCan only. Figure 2–1 shows the mechanical outlines of both versions. Since the lid of either MicroCan is the same, both can use the same probe. The flange at the bottom of the MicroCan allows for flexible mounting. Further details of iButton mechanics are found in Chapter 3.

B. Electrical Behavior

Other features common to all iButtons are the serial 1–Wire protocol, presence detect, and communication in discrete time slots. These electrical details are discussed in Chapter 4.

C. ROM Registration Number

A laser–programmed ROM–section, containing a 6–byte device–unique serial number, a one–byte family code, and a CRC verification byte, is also common to all iButtons. Details about the CRC are found in Chapter 5.

C.1. Family Code

The family code is a type–specific value that references the device's functionality and capacity. The lower seven bits of the family code indicate the device type; the most significant bit of the family code is used to flag customer–specific versions. Thus 128 different standard devices can be coded.

C.2. Serial Number

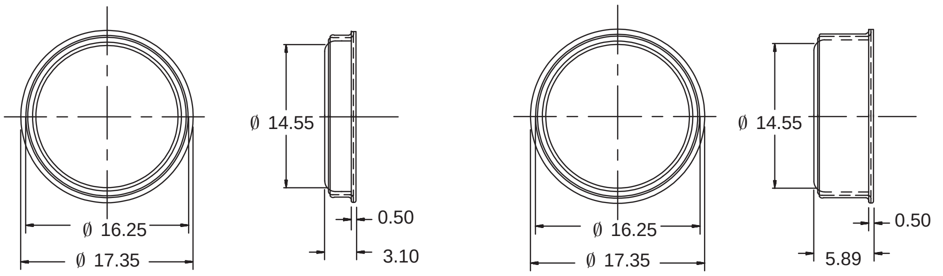
The 48–bit (6–byte) serial number can represent any decimal number up to 2.81*10^{14}. If 1000 billion (1.0 * 10^{12}) devices of the same family code were produced per year, this number range would be sufficient for 281 years. In addition there are 128 family codes available. If the most significant bit of the family code is set, the device's functionality is still the same as that of the standard device, but the serial number follows special rules.

C.3. Special Rules for Customer Codes

If the custom flag of the family code is set, a part of the number pool is reserved to designate specific customers. That is, the 12 most significant bits of the serial number allow 4096 different customers each to have their own special device. The code for these 12 bits is assigned by Dallas Semiconductor with the first customer order. Since the ROM section is 64 bits, and 8 bits are taken each for family code and CRC, there remain 36 bits to store customer–defined data together with unique serial numbers. Customer–specific devices require special registration and ordering procedures to control access to only one customer. Customer–specific devices can be made public if officially authorized by the originator.

Depending on their requirements, customers have four options for using the remaining 36 ROM bits. Option A allows the eight most significant bits of this range to be programmed with customer–defined data, leaving 28 bits for unique serial numbers (268.4 million combinations). Option B allows the 12 most significant bits to be customer–defined, still allowing 16.8 million unique serial numbers. With Option C, the customer can specify the 16 most significant bits; the pool of unique serial numbers, however, diminishes to 1.05 million. Option D allows the 20 most significant bits to be defined by the customer, but the total number of unique serial numbers reduces to just 65,536. A more complete description of customer–specific devices is available on request.

Figure 2–1



All dimensions are in millimeters

C.4. Example of a Private–to–Public Code Conversion

One of the components inside the DS1994 Memory Pluse Time iButton is the DS2404, also available separately. Depending on the application, this chip may be connected to a microprocessor using its 3–wire interface, while the 1–Wire interface operates as iButton. One such application is the Touch Pen chip set, where dual–porting the DS2404 is required. Dallas Semiconductor has customized the chip so that it can be distinguished from a single–ported DS2404. In order to make the customized chip generally available, the private–to–public conversion has been authorized. Instead of 04H, this customized version carries the family code 84H to mark it as a custom part. The 12 most significant bits of the serial number are coded 001H to indicate dual–port operation. Using Option A, the customer field is programmed 00H, leaving 28 bits for serialization. This chip is available as part number DS2404S–C01 (SOIC–package).

II. Devices

A. MicroCans

The sections below explain the different versions of iButton MicroCans. Table 2–1 gives a complete overview of the product family.

A.1. DS1990A Serial Number iButton

The simplest iButton is the DS1990A, a factory–programmed ROM. Since the information is stored in laser–cut polysilicon links (not as charge on gates or as states of flip–flops), the DS1990A needs no energy to retain data. Furthermore, almost no energy is required for operation. The DS1990A uses the voltage of the data line and stores a minimum of charge internally to maintain operation during the presence pulse and the low time of any time slot during a read operation. Figure 2–2 shows how data is organized within the DS1990A.

The first byte to be transmitted out of the ROM is the family code. After this, the guaranteed unique serial number follows, least significant byte first. The last byte is a cyclic redundancy check (CRC). The CRC is a kind o f signature of the first 7 bytes. It allows fast checking of the complete communication sequence. If the CRC calculated by the reading master matches the CRC read from the device, the reading was completely correct. This is one of the reasons why iButtons don't require stable electrical contacts.

Because of its design and the strict control of the manufacturing process, the DS1990A is a unique electronic identifier that is impractical to be counterfeited. It is appropriate for applications where absolute identification is required.

A.2. DS1991 MultiKey iButton

Like the DS1990A, the DS1991 incorporates a serial number with family code and CRC. To this it adds a 64–byte nonvolatile scratchpad RAM, and three independent password–secured nonvolatile RAM areas of 48 bytes each, called subkeys. For every secured RAM area there is also a public identification field of eight bytes. Figure 2–3 illustrates the internal organization of the device.

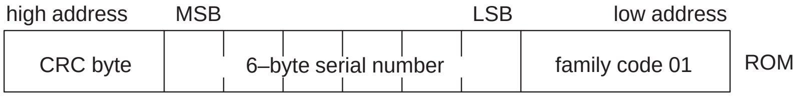
The DS1991 is designed as a high security electronic key that allows access to different applications with only one device. In fact, each of the three keys can be regarded as a protected application file. The ID field contains the file name, and the secured RAM houses the access code. Thus several persons can use the same access code although they carry different samples of the DS1991.

The DS1991 is tamper–proof. If the wrong password is used to read data, the device will output random numbers. If a new password is programmed, all data in the sub–key data field is automatically erased. Although direct write access is possible, the scratchpad should be used as intermediate storage to verify data before it is copied to its final place. This ensures that garbled data is not accepted, even if the contact should break during communication. Depending on the application, the scratchpad alternatively can be used as unprotected, general–purpose read/write memory.

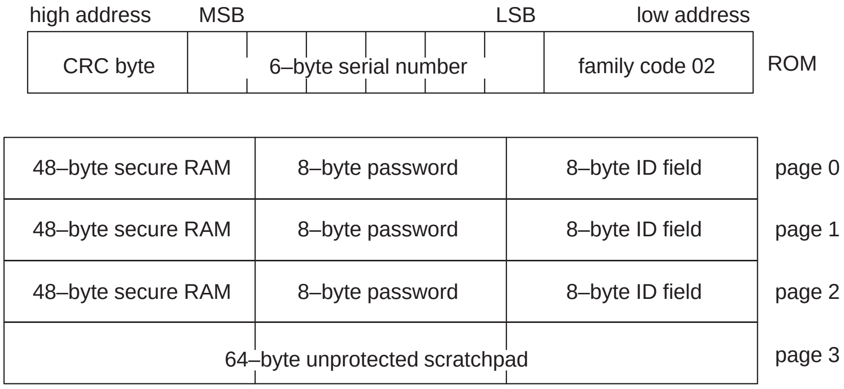
iButton DEVICES Table 2–1

Device Type	**Family Code**	**Serial Number**	**Memory Bits Type**	**Protected NV RAM bits**	**Real Time Clock**	**Interval Timer**	**Cycle Counter**
DS1990A	01H	yes	—–	—–	—–	—–	—–
DS1991	02H	yes	512, NVRAM	3 * 384	—–	—–	—–
DS1992	08H	yes	1K, NVRAM	—–	—–	—–	—–
DS1993	06H	yes	4K, NVRAM	—–	—–	—–	—–
DS1994	04H	yes	4K, NVRAM	—–	yes	yes	yes
DS1995	0AH	yes	16K, NVRAM	—–	—–	—–	—–
DS1996	0CH	yes	64K, NVRAM	—–	—–	—–	—–
DS1982	09H	yes	1K, EPROM	—–	—–	—–	—–
DS1985	0BH	yes	16K, EPROM	—–	—–	—–	—–
DS1986	0FH	yes	64K, EPROM	—–	—–	—–	—–
DS1920	10H	yes	16, EEPROM	TEMPERATURE iButton			

DATA STRUCTURE DS1990A Figure 2–2



DATA STRUCTURE DS1991 Figure 2–3



A.3. DS1992 Memory iButton: 1K–Bit NV RAM

As with all iButtons, the DS1992 contains a unique serial number. The internal 128 bytes of nonvolatile RAM are organized as four final storage areas of 32 bytes each and a scratchpad of 32 bytes (see Figure 2–4). The RAM can be read starting at any byte position of any page. Writing is only possible via the scratchpad. After the data is verified against the original by reading the scratchpad, the copy scratchpad command copies it to the final position. This way of writing data guaranties that even if the contact should break during communication with the device, garbled data will not reach the final destination; it will stay in the scratchpad.

A.4. DS1993 Memory iButton: 4K–Bit NV RAM

The DS1993 is a larger version of the DS1992. As Figure 2–5 shows, the DS1993 has four times the storage capacity of the DS1992. Of course, it has its own family code within the ROM.

The DS1992 and DS1993 are each designed as identification device and mobile data carrier in one unit. Using a special data structure, these devices can store multiple independent application files. Moreover, for secured access, the public serial number can be used as a seed together with a secret keyword to encrypt non–public data files. Although encrypted data can be read, it is impractical to duplicate since no two serial numbers are the same.

A.5. DS1994 Memory Plus Time iButton: 4K–Bit NV RAM

The DS1994 adds to the DS1993 a real–time clock, interval timer and cycle counter, plus alarm features for these counters. With the exception of the family code, the DS1994 is completely compatible with the DS1993. The extra registers for clock, etc. are located in another page at the high end of the memory. Figure 2–6 shows details.

With respect to time representation, the clock has features different from common real time clocks on the market. The clock in the DS1994 is a binary counter with a resolution of 1/256 second. Minute, hour, day, month and year are recalculated from the number of seconds that have elapsed since an arbitrarily chosen "zero date" (usually January 1st, 1970, 00:00:00 hours). Thus any variance with country–dependent daylight savings rules become a matter of application software and can be handled as required. Furthermore, this representation simplifies calculations of time intervals between events and allows a simple algorithm to improve the accuracy by calibration.

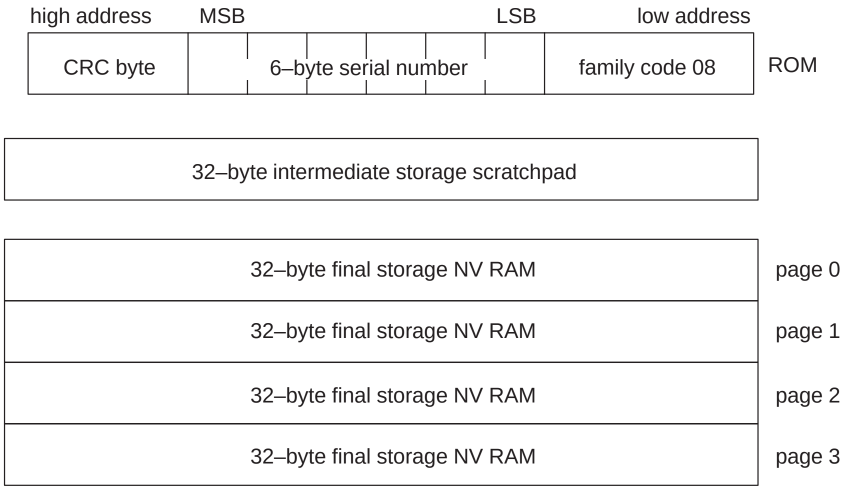
The interval timer can be used as a stopwatch to count the time between certain events of the application environment or as a tool to time–control a machine, since the DS1994 includes a feature to generate interrupts. To obtain operation statistics, the cycle counter keeps track of how often the application equipment has been switched on; the interval timer adds up the operation time. This application, however, requires that the DS1994 be mounted within the equipment. Also when the DS1994 is used in a touch environment, it gives useful information about the frequency of its use and the average time of each touch. The RTC together with the RTC alarm register provides a time–limited access function. As soon as a certain time point is reached, access to a secured building for example is denied by the controlling computer. Alarms or interrupts can even be indicated without using a computer.

The ability to write–protect the counters and lock the alarm registers within the DS1994 converts the device into an non–resettable expiration controller. All these extra features and their related registers and control flags are located in page 16. The access method is exactly the same as for the RAM. Although the scratchpad is involved for writing, the command structure allows writing single or multiple bytes.

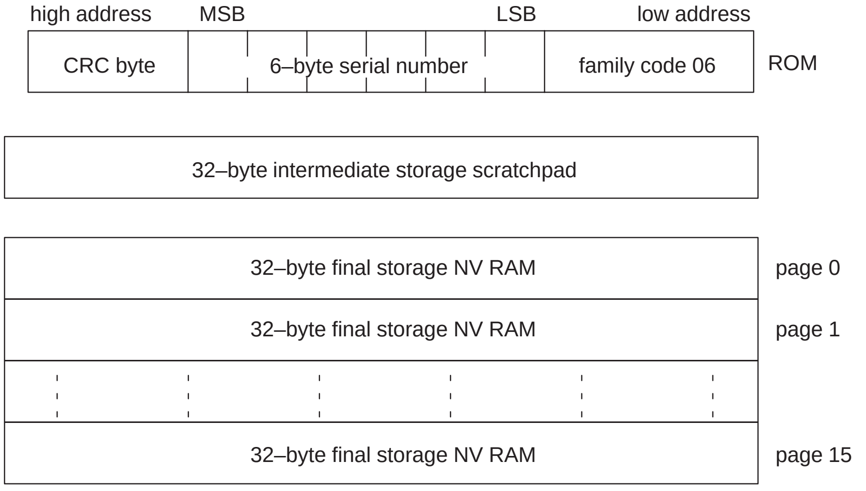
A.6. DS1995 Memory iButton: 16K–Bit NV RAM

For applications that require storing several files of different size, the capacity of the DS1993 may be insufficient. The DS1995 quadruples the available storage capacity of earlier Memory iButtons to 16K bits or 64 pages of 32 bytes. (See Figure 2–7.) Since the DS1995 has the same logical structure and understands the same set of commands as other NV RAM iButtons, it is completely compatible with existing application software. The unique family code indicates the extended capacity.

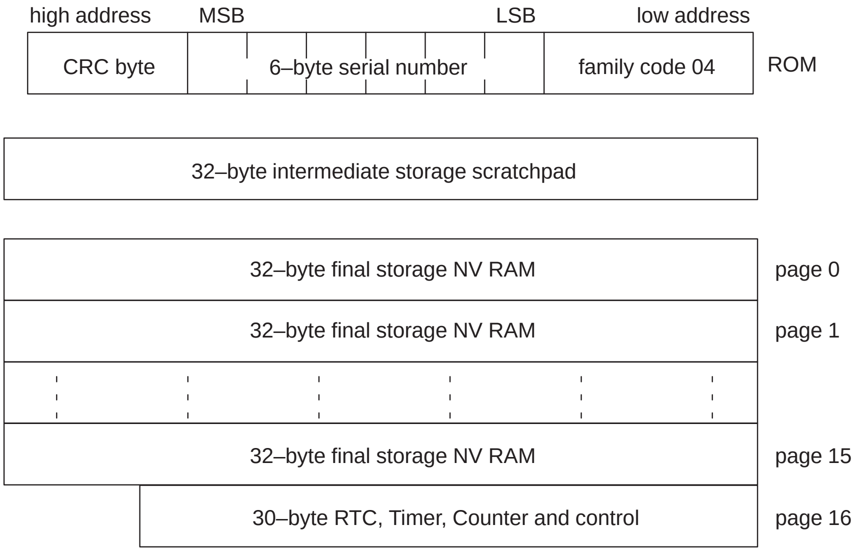
DATA STRUCTURE DS1992 Figure 2–4


high address
MSB
LSB
low address
CRC byte
6–byte serial number
family code 08
ROM
32–byte intermediate storage scratchpad
32–byte final storage NV RAM
page 0
32–byte final storage NV RAM
page 1
32–byte final storage NV RAM
page 2
32–byte final storage NV RAM
page 3

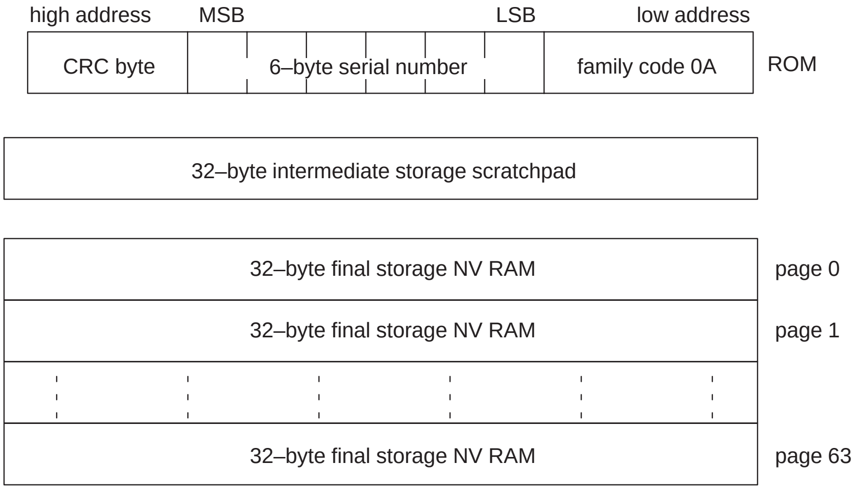
DATA STRUCTURE DS1993 Figure 2–5


high address
MSB
LSB
low address
CRC byte
6–byte serial number
family code 06
ROM
32–byte intermediate storage scratchpad
32–byte final storage NV RAM
page 0
32–byte final storage NV RAM
page 1
32–byte final storage NV RAM
page 15

DATA STRUCTURE DS1994 Figure 2–6


high address
MSB
LSB
low address
CRC byte
6–byte serial number
family code 04
ROM
32–byte intermediate storage scratchpad
32–byte final storage NV RAM
page 0
32–byte final storage NV RAM
page 1
32–byte final storage NV RAM
page 15
30–byte RTC, Timer, Counter and control
page 16

DATA STRUCTURE DS1995 Figure 2–7


high address
MSB
LSB
low address
CRC byte
6–byte serial number
family code 0A
ROM
32–byte intermediate storage scratchpad
32–byte final storage NV RAM
page 0
32–byte final storage NV RAM
page 1
32–byte final storage NV RAM
page 63

A.7. DS1996 Memory iButton: 64K–Bit NV RAM

The DS1996 quadruples the capacity of the DS1995 to 64K bits or 256 pages of 32 bytes (see Figure 2–8). Using the same commands as other NV RAM iButtons, the DS1996 allows easy upgrading of existing systems. As with all iButtons, this device has a unique family code.

Both the DS1995 and DS1996 substantially surpass the capacity of existing mobile read/write data carriers, such as serial memory cards or magnetic stripes. Using the serial number as a seed together with a secret keyword allows storage of public and encrypted data files in the same device. Chapter 7 shows further ways to use the large capacities of these devices.

A.8. DS1982 Add–Only iButton: 1K–Bit OTP EPROM

The DS198x series of iButtons uses EPROM that does not require an embedded energy source to maintain data. Like the DS1990A, the energy for operation is taken directly from the data line. As a standard feature, the DS1982 contains a ROM section with a serial number and family code. The memory is organized as four pages of 32 bytes each (see Figure 2–9).

The DS1982 is read in the same way as other Memory iButtons; however, writing is done differently. Before a data byte arrives at the final memory location, it first is written to a one–byte scratchpad. The subsequent verification involves checking the write command itself, the destination address, and the data using an 8–bit CRC. If the verification is positive, a pulse of 1 ms at 12V will copy the data from the scratchpad to the memory. This procedure prevents writing incorrect data even if the contact should break during communication with the device.

A sophisticated verification is essential for EPROM devices since once data is written incorrectly, it cannot be changed. When data needs to be updated, the old data is “redirected” and a new set of data is added. This mode of operation explains the name “Add–Only iButton” for this group of iButton products. It is not possible to erase Add–Only iButtons. Each page can be individually hardware–protected against subsequent write attempts. Thus every update will leave a permanent audit trail.

Flags indicating whether a page of data is write–protected or redirected are stored in the eight bytes of status memory of the device. Writing to the status memory employs the same integrity procedures as for the data pages. When reading data or status information, an on–chip CRC generator protects the data stream against potential transmission errors.

A.9. DS1985 Add–Only iButton: 16K–Bit OTP EPROM

With 16 times the memory capacity of the DS1982, the DS1985 is the smallest Add–Only iButton that completely supports storage and update of multiple application files. Details on how this is accomplished are discussed in Chapter 7. The memory is organized as 64 pages of 32 bytes. Figure 2–10 shows details. In addition to the application memory, there are 88 bytes of status memory dedicated as redirection bytes, flags and write protect bits. A special read command is implemented to signal redirection before time is wasted by reading invalid data. The other functions of the DS1985 are exactly the same as the DS1982.

A.10. DS1986 Add–Only iButton: 64K–Bit OTP EPROM

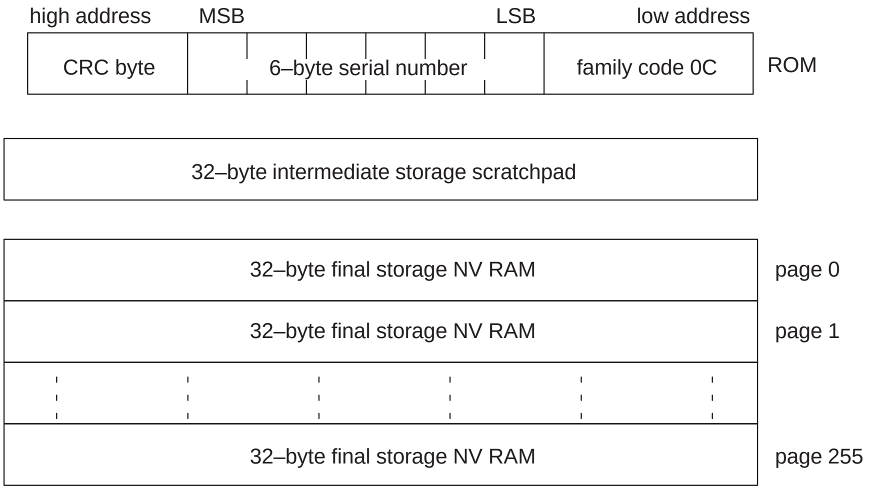
The DS1986 is the 64K bit upgrade of the DS1985. As shown in Figure 2–11, the memory is organized as 256 pages of 32 bytes. The extended memory capacity requires that the status memory be expanded to 352 bytes. All other features of the DS1986 are identical to the DS1985.

The outstanding feature of Add–Only iButtons is the impossibility of deleting data. If data needs to be updated this is done by patching it with another page, thus leaving a permanent trail of changes. It is possible to reconstruct the original and intermediate versions of data. Due to a hardware write–protect feature, the devices are tamper–proof. If the write–protect bits are programmed, there is no chance to falsify a single bit of the corresponding page or the redirection byte.

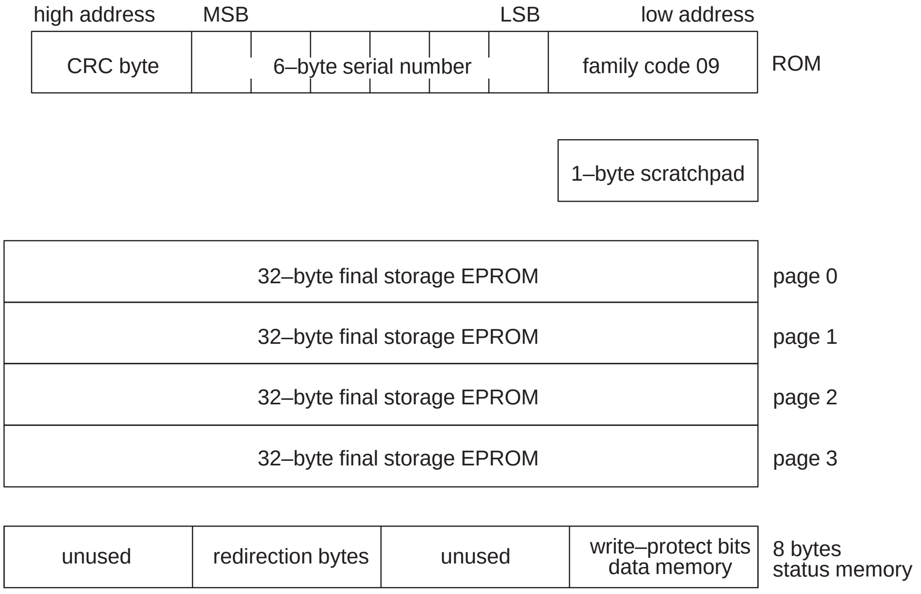
A.11. DS1920 Temperature iButton

As the name states, this device is a memory plus thermometer in a MicroCan. Instead of a memory, the user has access to a 9–bit converter as if it were memory, giving a resolution of 0.5°C to a control register. A unique ROM section is also standard with these devices, allowing one to build a chain of thermometers and to read all of them from one location. The accuracy of temperature measurement is 0.5°C within the range of 0°C to +70°C. In the ranges of –40°C to 0°C and +70°C to +85°C, the accuracy decreases to 1°C. The temperature conversion time is about one second. This device is discussed in greater detail in Chapter 6.

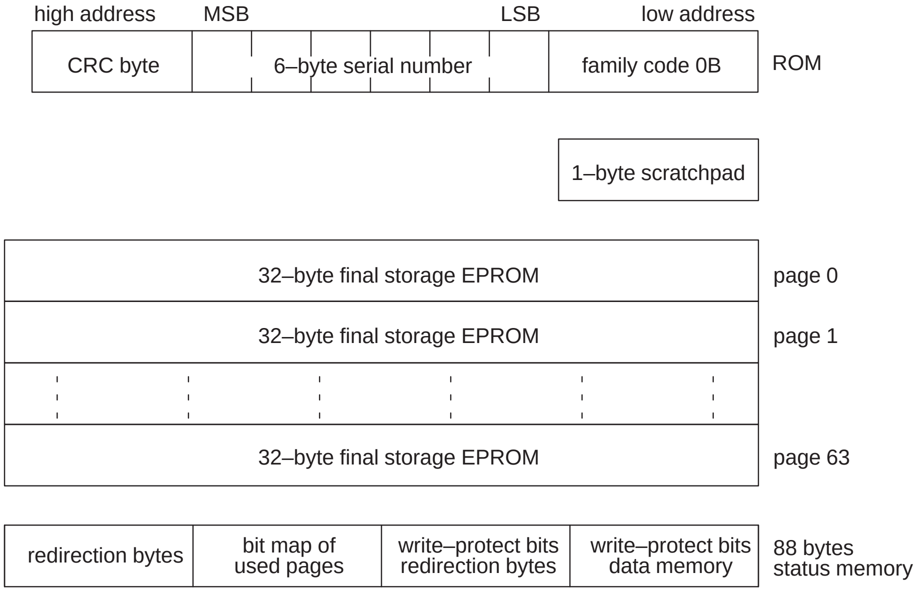
DATA STRUCTURE DS1996 Figure 2–8


high address
MSB
LSB
low address
CRC byte
6–byte serial number
family code 0C
ROM
32–byte intermediate storage scratchpad
32–byte final storage NV RAM
page 0
32–byte final storage NV RAM
page 1
32–byte final storage NV RAM
page 255

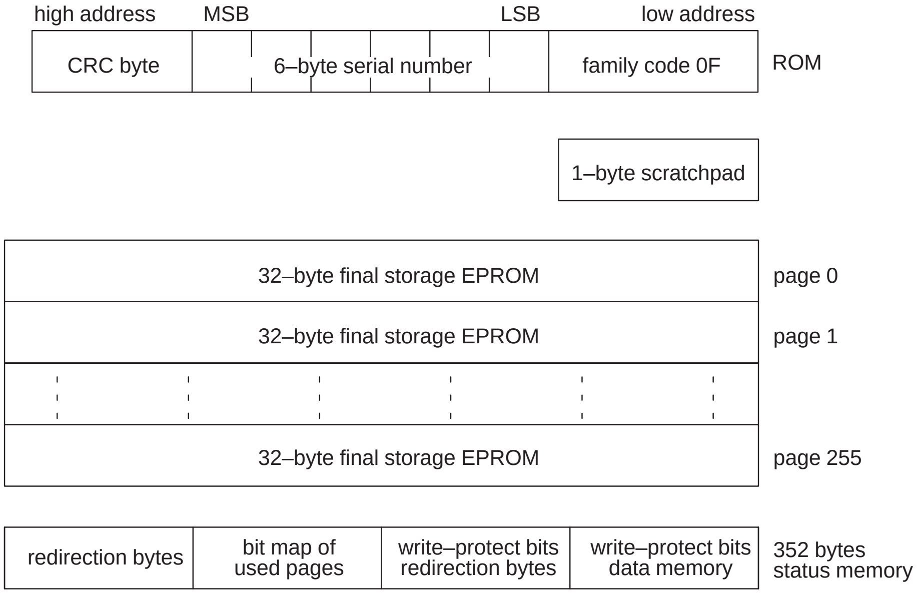
DATA STRUCTURE DS1982 Figure 2–9



DATA STRUCTURE DS1985 Figure 2–10



DATA STRUCTURE DS1986 Figure 2–11



B. Solder Mount Products

This section contains products that share the same electrical and logical characteristics as iButtons, but cannot be made available as MicroCans since they have communication ports in addition to the 1–Wire bus. They are normally used in the wiring of MicroLANs.

B.1. DS2407 Addressable Switch

The DS2407 (formerly referenced as DS2405A) is a combination of two open drain transistors (”switches”) with associated digital sensors and 1K–bit of EPROM. It can be employed to remotely sense the state of mechanical switches, or together with power transistors, to control a solenoid or DC motor. Since the DS2407 is completely in compliance with the 1–Wire standards and also includes a unique family code and serial number, many of these devices can be connected in parallel to form a 1–Wire bus. This allows, for example, monitoring of all sensors of a burglar alarm system with the absolute minimum of wiring, only two wires. The DS2407 can also be used for diagnostics of digital circuits, e.g., by sensing the logical state of a node or forcing a node to 0 emulating a malfunction or to gate a signal. Only one additional wire needs to be routed through the printed circuit board to implement this feature. For more details, see Chapter 6.

B.2. DS2404S–C01 Dual Port Memory Plus Time

The 1–Wire MicroLAN is a general–purpose single–master network for digital communication. All iButtons have a built–in MicroLAN interface as a standard feature. Another MicroLAN device is the Addressable Switch, mentioned above.

In order to provide universal access to the MicroLAN, the DS2404S–C01 Dual Port RAM Plus Time has been developed. This device has both 1–Wire and a 3–Wire serial microcontroller interface. It provides 512 bytes of memory plus a real time clock. A special family code distinguishes this device from other 1–Wire products. The DS2404S–C01 can be used to make complex functions involving microcontrollers behave as if they were iButtons. The DS2404S–C01 is discussed in more detail in Chapter 6 of this book.

III. Commands

All iButton devices support the ROM commands to read the family code, serial number and CRC and to search for ROM contents. The Skip ROM and Match ROM are not applicable with the DS1990A since there is no other memory accessible. Common to many devices are the scratchpad commands Read, Write and Copy. Also widely used is the Read Memory command. Because of its very special application, the DS1991 also requires a command to write passwords.

For applications where a stable contact is available, the DS1991 supports a Write Memory command. Table 2–2 summarizes iButton commands. The Skip ROM command allows getting to the data faster if the registration number is not of interest.

Although some devices share the same commands, it does not necessarily mean that the binary command word is the same. This holds for the DS1991 because of its special application areas. The DS1992 to DS1996 have the same command words. Due to their different memory technology, the DS1982 to DS1986 require a special set of commands. Although they use the same command codes as the NV RAM devices, the effects of the commands will be different. For compatibility reasons, the effect of the Read Memory command is identical to the other Memory iButtons excluding the DS1991 and DS1982

IV. Chapter Summary

The iButton family has consistent operating characteristics. The logical functions range from simple serial number and password–protected memory, 64K bits of nonvolatile RAM or EPROM and beyond, to a real time clock plus 4K bits of nonvolatile RAM. Common to all of them is an individual serial number and the 1–Wire protocol electrical interface. If any writable memory is included, writing is done first to a scratchpad. The size of the scratchpad may vary: one byte (EPROM devices), 32 bytes (SRAM devices) or 64 bytes (password–protected memory). After writing, data is verified before it is transferred to its final destination to insure data integrity. Reading is always done directly.

iButton COMMANDS Table 2–2

Device Type	ROM Commands	Scratchpad Commands	Memory Commands		Password Commands	Status Commands
	Read Skip Match Search	Read Write Copy	Read	Write	Write	Read Write
DS1990A	yes	——	——	——	——	——
DS1991	yes	yes	yes	yes	yes	——
DS1992	yes	yes	yes	——	——	——
DS1993	yes	yes	yes	——	——	——
DS1994	yes	yes	yes	——	——	——
DS1995	yes	yes	yes	——	——	——
DS1996	yes	yes	yes	——	——	——
DS1982	yes	——	yes	yes	——	yes
DS1985	yes	——	yes	yes	——	yes
DS1986	yes	——	yes	yes	——	yes
DS1920	yes	yes	recall	——	——	——

MECHANICAL STANDARDS

CHAPTER 3: MECHANICAL STANDARDS

I. Introduction

iButtons identify the objects they are affixed to. Since these objects have an abundance of different shapes and sizes, there are many ways to mount an iButton. This chapter provides details on MicroCans, iButton probes and mounting techniques.

II. MicroCans

A. Package Types

An iButton is packaged in the F5 flanged MicroCan as shown in Figure 3–1a. This package is the standard for all devices that contain an internal energy source. Devices that are powered by the master (parasite–powered) can be made available in this package as well as the thinner F3 package (Figure 3–1b).

B. Stability

MicroCans are made from solid stainless steel with a thickness of 0.254 mm. The insulating material between the bottom part and the top contact is UV–inhibited black polypropylene. This design gives excellent mechanical stability and is corrosion–resistant. All MicroCans withstand mechanical shocks of 500 g (1 g = 9.81 m/s^2) in all directions. A drop from a height of 1.5 m to a concrete floor does not damage the can or its contents. The same holds for a constant force of 110 Newton on either side of the can. Repeated probing does not degrade the contact since there is no plating; iButtons will withstand a minimum of one million probing cycles.

C. Temperature Range

iButtons (DS1990A, DS1982 to DS1986) have been specified for the extended temperature range of –40°C to +85°C (–40°F to 185°F). The operating temperature range of the DS1920 is even larger. Devices containing lithium cells (DS1991 to DS1996) should not be stored or operated below –40°C or above +70°C (–40°F or 158°F). At –55°C the electrolyte of the lithium cell freezes; above 85°C the vapor pressure of the electrolyte increases, causing diffusion through the seal, which dries out the cell after some time.

D. Human Readable Engraving

All MicroCans are laser–branded to provide all important information about the device. (See Figure 3–2.) The part number is at the bottom. The extension XXX of the part number may indicate the package type (F3 or F5) or a customer–specific version. Customer–specific devices can be branded 001 through FFF (hexadecimal) in combination with a special family code. Independently of this, a customer specific name can also be put in place of the name “DALLAS”.

III. iButton Probes

iButton are read or written with a probe. The basic probe is shown in Figure 3–3a. It is also available with tactile feedback (Figure 3–3b). The standard probe is usually preferred when the probe is fixed and the iButton is mobile.

If the iButton is read by a mobile reader like the iButton Recorder, or in general if long contact dwells are required, the probe with tactile feedback performs better.

iButton Probes are usually mounted on panels. A retaining ring pressed on from the back side provides sufficient mechanical hold (see Figure 3–3c). The cross–shaped cross section of the iButton Probe’s back end prevents twisting the connected wires after mounting. For some applications, a hand–held wand for iButtons is convenient. This wand consists of a hand–grip mount and an iButton probe with tactile feedback. Figure 3–4 shows details.

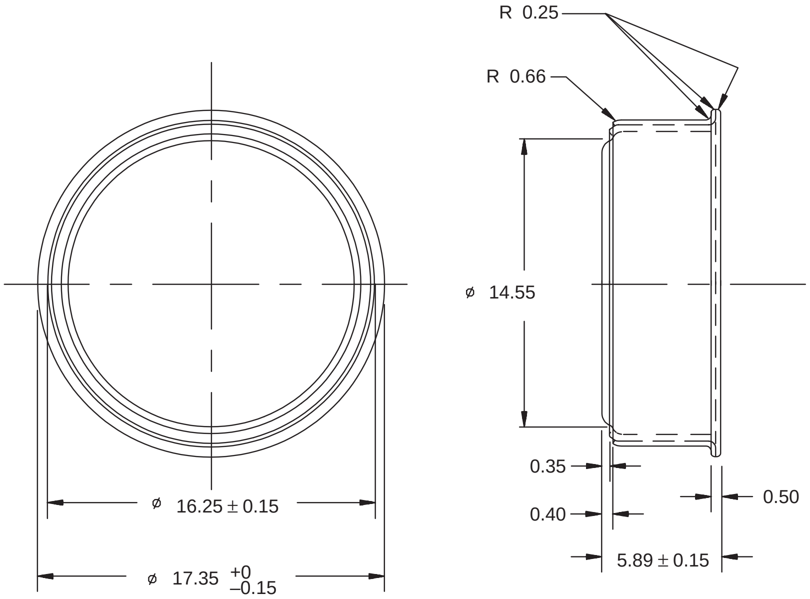
IV. iButton Mounts

A. Through–Mount

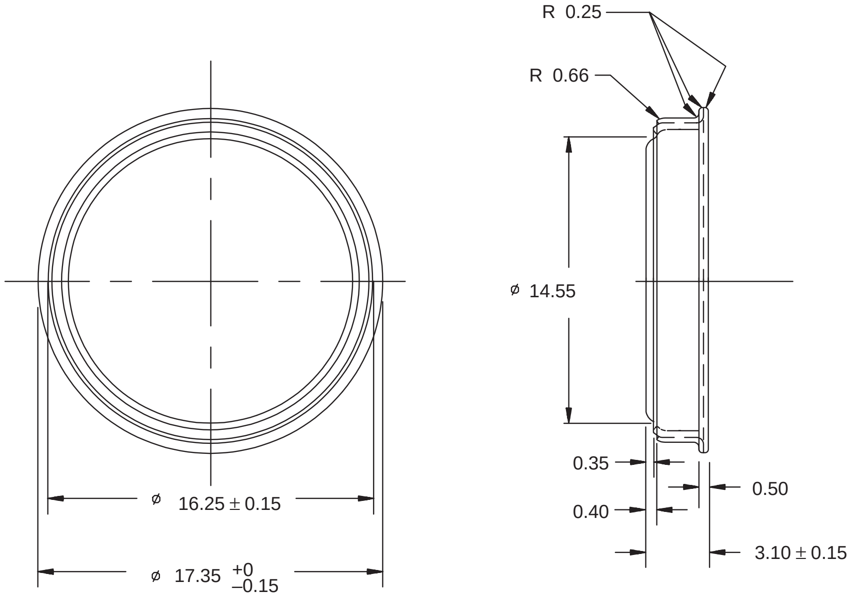
The flange at the bottom of the MicroCan makes the part ideally suited for through–mounting. This technique can be used to mount iButtons as personal identification labels for access control and time stamping for accounting purposes. Figure 3–5 shows a practical example. The retaining ring DS9093RA is press fit on the iButton from the front side of the label.

The through–mount system can also be used to mount iButtons on large containers. An iButton retainer as shown in Figure 3–6 can easily be screwed on metal or plastic containers. This plastic part is designed for the F5 MicroCan. Since the bottom of the MicroCan is pressed against the container and because of the guarding rim of the retainer, this mounting is well suited for harsh environments. The retainer in Figure 3–6 can be screwed or pop–riveted. A special version of the retainer with a pin instead of one hole allows the use of only one screw or rivet to reduce mounting time.

MICROCAN DIMENSIONS Figure 3–1



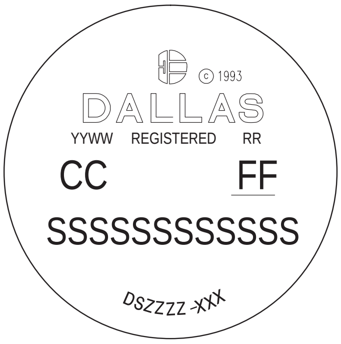
a) F5 MicroCan



b) F3 MicroCan

All dimensions are in millimeters.

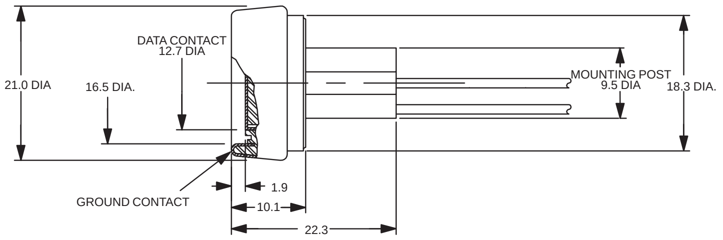
iButton ENGRAVING ON STAINLESS STEEL LID Figure 3–2

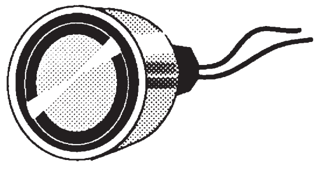


YYWW	=	YEAR, WORK WEEK
CC	=	CRC
SSSSSSSSSSSS	=	12 DIGIT HEX SERIAL #
RR	=	PACKAGE REV.
FF	=	FAMILY CODE
ZZZZ	=	GENERIC PART NUMBER

XXX:		
□ NNN	–	001 through FFF for Custom Code
□ F3	–	For F3 Package
□ F5	–	For F5 Package

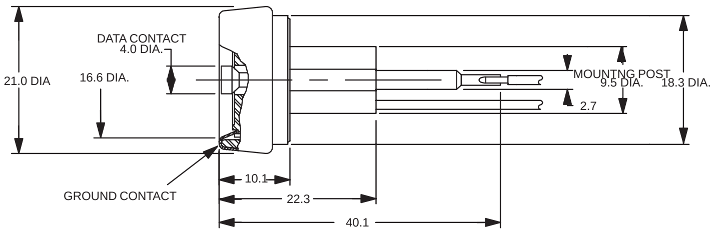
iButton PROBE Figure 3–3a

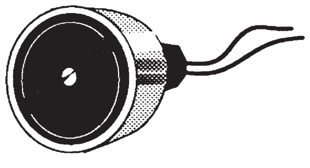




DS9092

iButton PROBE WITH TACTILE FEEDBACK Figure 3–3b

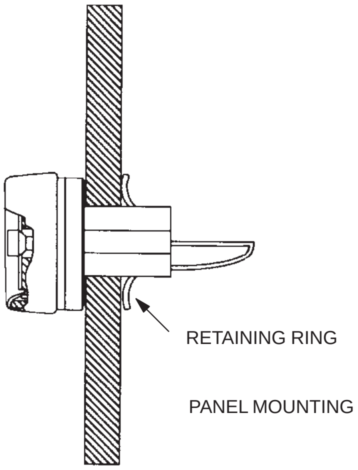




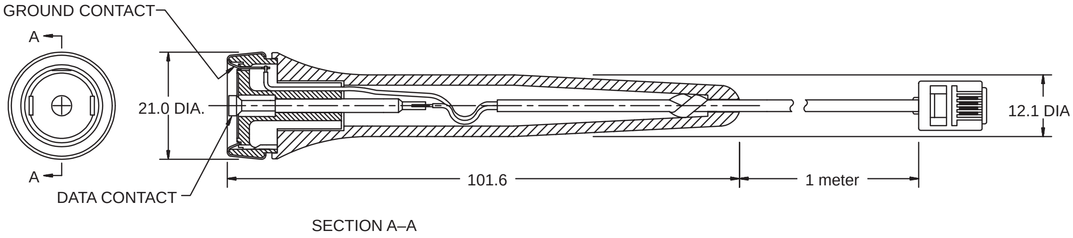
DS9092T

All dimensions are in millimeters.

MOUNTING AN iButton PROBE Figure 3–3c

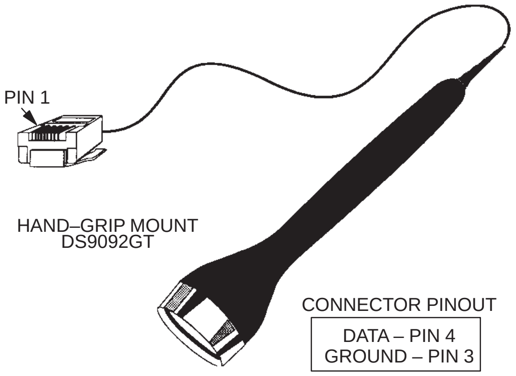


HAND GRIP PROBE Figure 3–4



DS9092GT

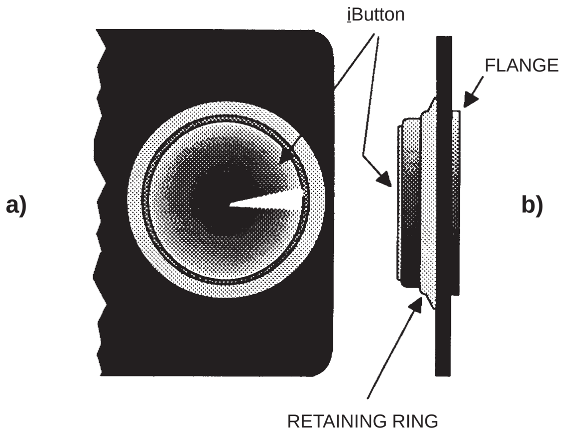
a) Hand Grip Probe, dimensions



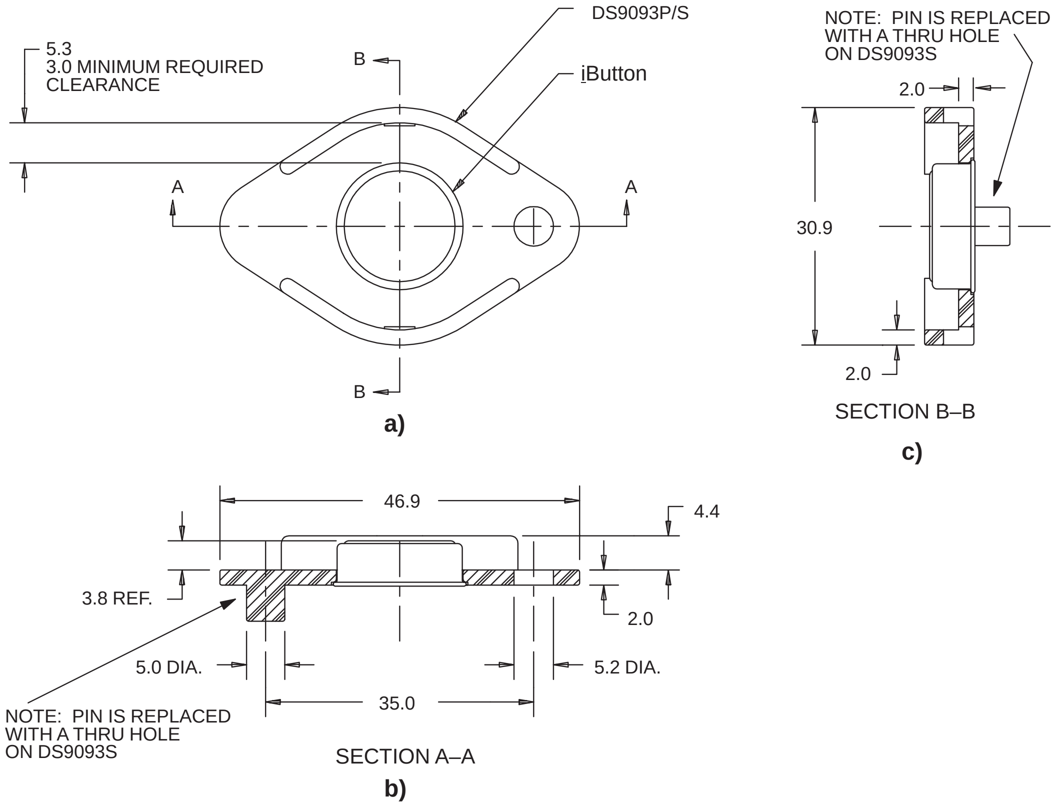
All dimensions are in millimeters.

b) Hand Grip Probe

THROUGH–MOUNT IDENTIFICATION LABEL Figure 3–5

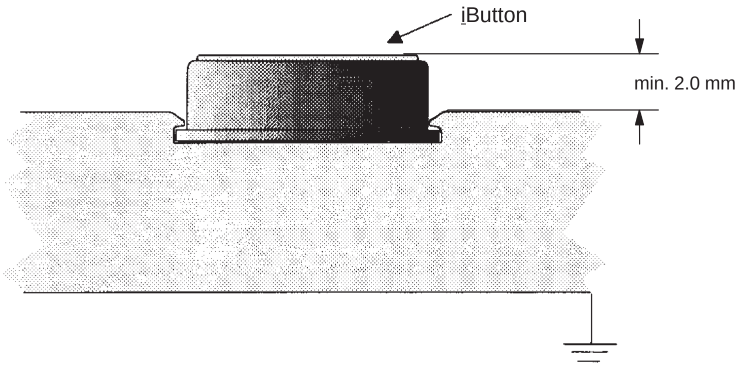


THROUGH–MOUNT FOR CONTAINERS Figure 3–6



All dimensions are in millimeters.

PRESS FIT MOUNT Figure 3–7



B. Press Fit

MicroCans are very stable. For that reason, they can be directly mounted on metal casings. To do this, first a cavity must be milled into the metal. Then the iButton is inserted and the rim of the cavity is pressed down so that it tightly holds the MicroCan in its position. Figure 3–7 shows this technique. In order to contact iButtons with a probe, a minimum clearance of 3.0 mm around the iButton is required (Figure 3–6a). The protrusion of the iButton above the surface must be at least 2.0 mm.

A similar mounting technique is used with the plastic keyring shown in Figure 3–8. The slot in this keyring provides the elasticity to insert or withdraw flanged MicroCans. Mechanical dimensions allow its use with both the F5 and F3 MicroCan and standard iButton Probes.

C. Spring Hold

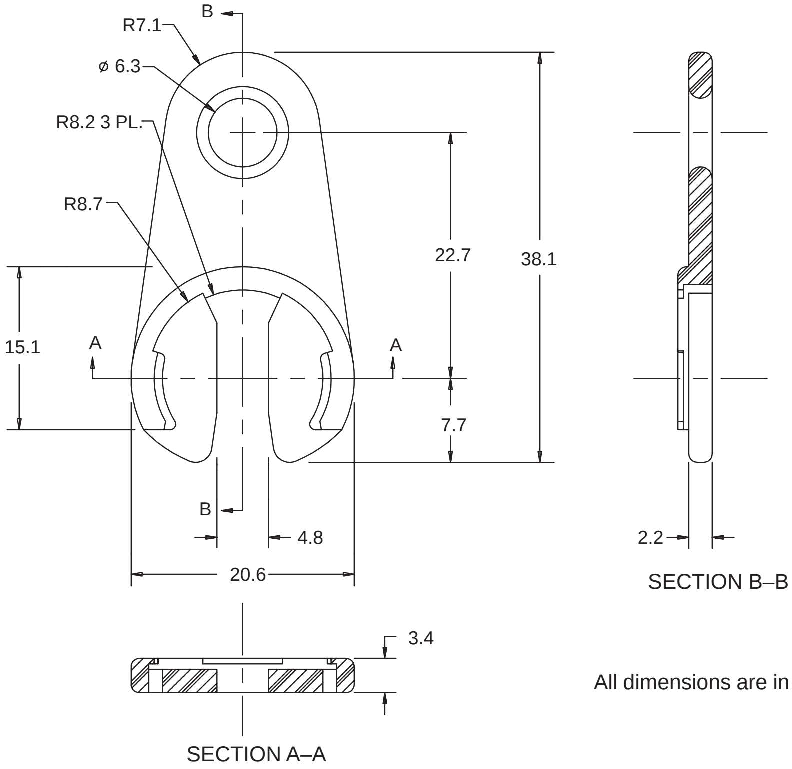
Certain applications may require mounting iButtons on printed circuit boards. The most advanced single–piece retainer for F5 MicroCans is the DS9098 (see Figure 3–9). Like the MicroCan, it is made from stainless steel. Selective tin–lead plating provides optimal solderability. By design, this retainer is compatible with standard pick–and–place and cleaning equipment. At the first insertion of an iButton, the inner contact breaks away from the outer ring and acts as a spring to hold the MicroCan with a force of a minimum of two Newtons. The cross section view A–A in Figure 3–9a explains how the iButton is held and how it can be easily removed again.

Another way to mount iButtons on printed circuit boards are MicroCan Clips. Figure 3–10 shows the standard clip DS9094F and a surface mount version DS9094FS to be used with the F5 MicroCan. These clips are similar to common battery clips but prevent contact if one tries to insert a MicroCan in the reverse direction. In contrast to the DS9098, the DS9094F accepts the MicroCan horizontally not vertically.

In principle, it is possible to mount iButtons on rotating parts and to use sliding spring pressed contacts, as shown in Figure 3–11, though mounting hardware for this application is not yet available as a standard product.

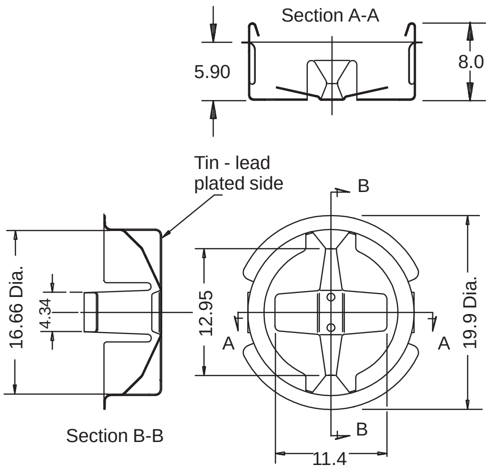
The DS9100 Touch and Hold Probe is shown in Figure 3–12. This probe is similar to the DS9098 MicroCan Retainer. An F5 MicroCan will fit completely into the DS9098 retainer, but the flange and about 1/3 of the can will stick out if pressed into the DS9100. As a probe, the DS9100 allows reading iButtons on contact. At further pressure the stiff springs of the DS9100's outer ring will hold the MicroCan sufficiently to give a good contact to both the ground and the data lid. To increase the strength of holding, future versions of F5 MicroCans may have a tiny indentation at the outer rim. This will have no impact on probe contact or mounting techniques explained in this chapter.

SNAP–IN KEY RING Figure 3–8



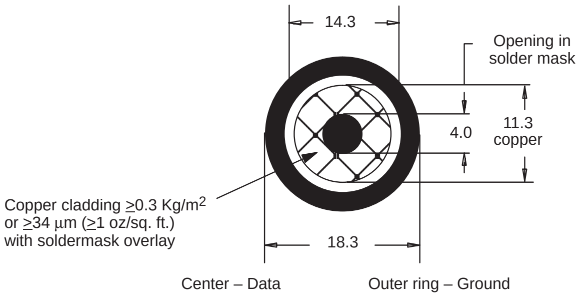
All dimensions are in millimeters.

DS9098 iButton RETAINER Figure 3–9a



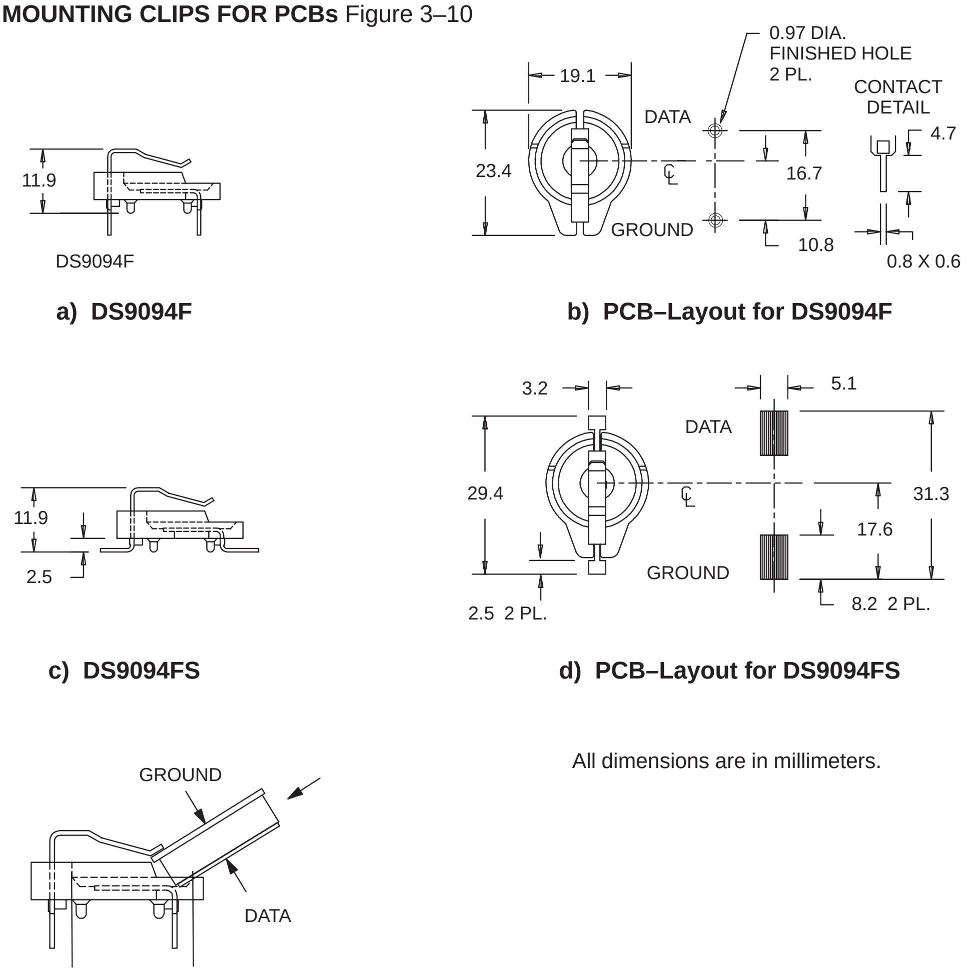
All dimensions are in millimeters.

RECOMMENDED PCB LAYOUT Figure 3–9b


14.3
Opening in solder mask
4.0
11.3 copper
Copper cladding ≥0.3 Kg/m2 or ≥34 µm (≥1 oz/sq. ft.) with soldermask overlay
18.3
Center – Data
Outer ring – Ground

All dimensions are in millimeters

iButton MOUNTING CLIPS FOR PCBs Figure 3–10



a) DS9094F

b) PCB–Layout for DS9094F

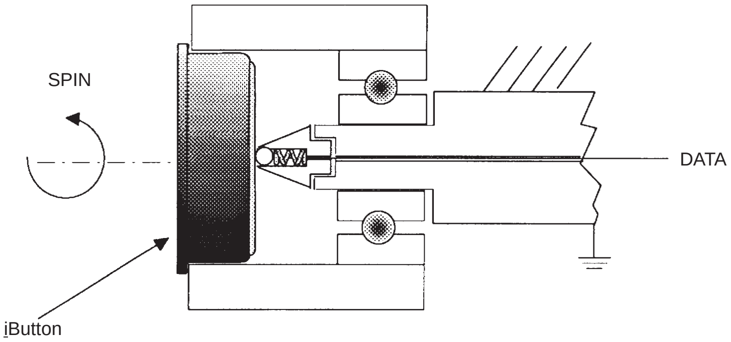
c) DS9094FS

d) PCB–Layout for DS9094FS

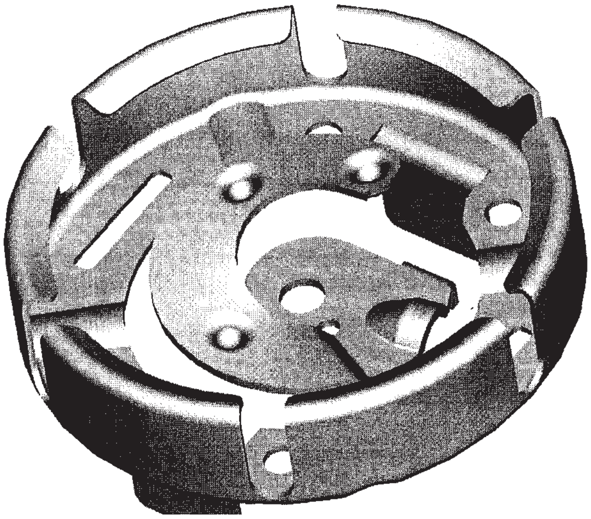
All dimensions are in millimeters.

e) Insertion of an iButton

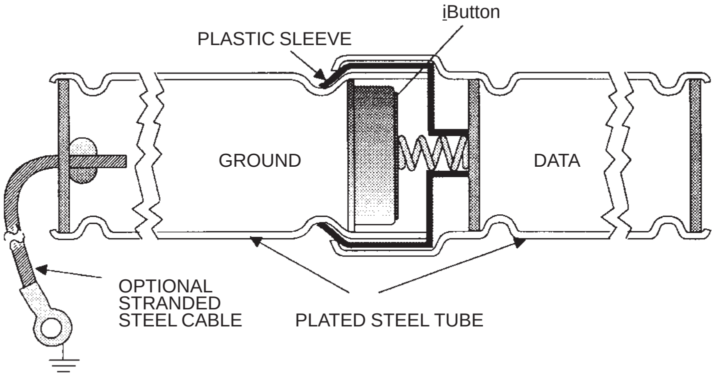
SPINNING WHEEL Figure 3–11



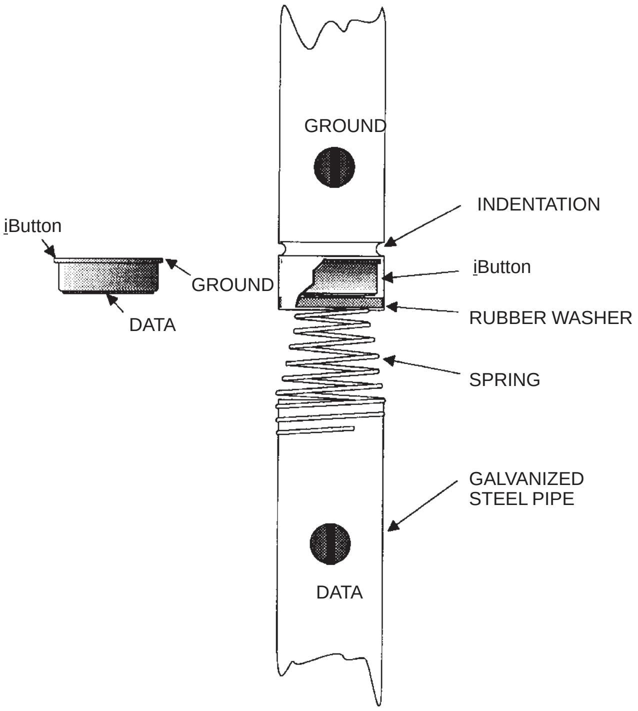
TOUCH AND HOLD PROBE Figure 3–12



TOUCH BAR Figure 3–13



SPLIT CONDUIT Figure 3–14



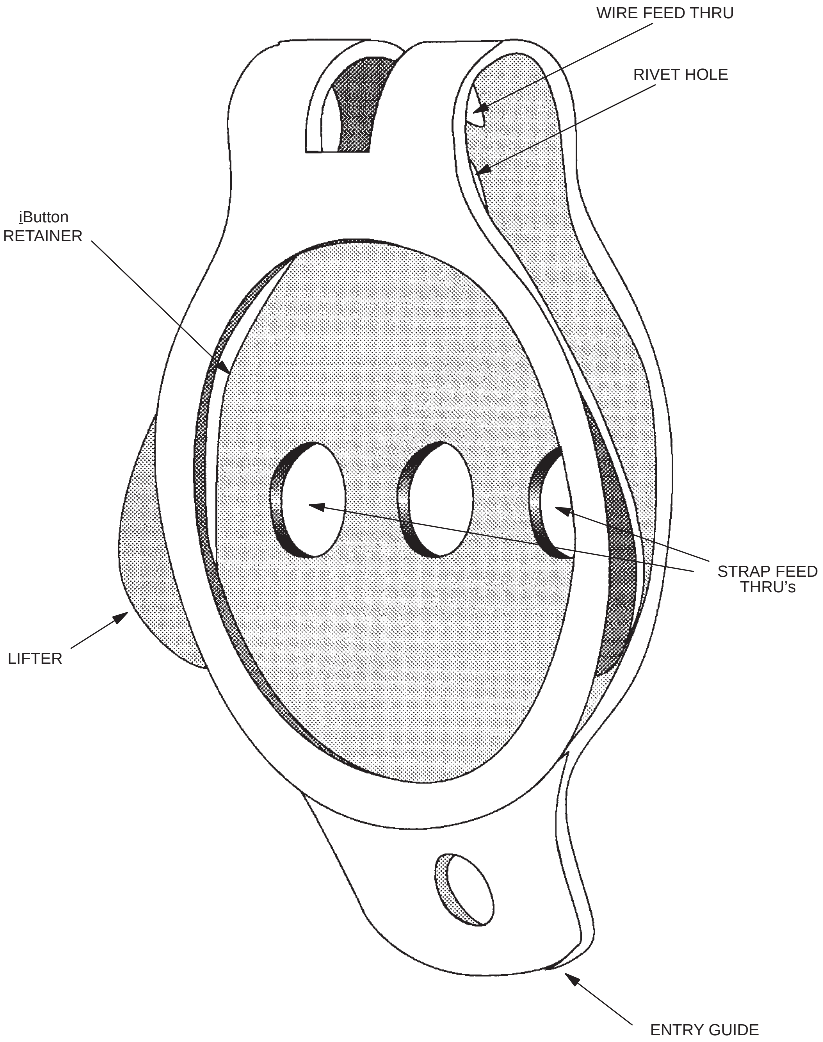
V. Special Mounting Examples

Some applications require mechanical tolerances that are large compared to the size of an iButton. In such cases, the surface of the iButton can be extended to obtain the required tolerances. One way to do this is to build a Touch Bar with the iButton inside (Figure 3–13). Another possibility is to cut the handling rod of a container, put the iButton into the middle, and mount the rod again keeping one part electrically insulated from the other (Figure 3–14).

VI. Summary

iButtons are electronic chips housed in a stainless steel package. Special probes, clips and retainers are available for reading and writing iButtons. iButtons can be mounted on objects using either special retainers, metal forming techniques or adhesives. Further, the MicroCan's surface can be extended for applications that require larger contact surfaces.

MULTI–PURPOSE CLIP DS9101 Figure 3–15



ELECTRICAL STANDARDS AND CHARACTERISTICS

CHAPTER 4: ELECTRICAL STANDARDS AND CHARACTERISTICS

I. 1–Wire Interface – Timing

A. Introduction

iButtons are self–timed silicon devices that require electrical contact for operation. The timing logic provides a means of measuring and generating digital pulses of various widths. Data transfers are bit–asynchronous and half–duplex. Data can be interpreted as commands (according to the prearranged format identified by the family code) that are compared to information already stored in the iButton to make a decision, or can simply be stored in the iButton for later retrieval. Because the falling slope is the lease sensitive to capacitive loading in an open drain environment, iButtons use this edge to synchronize their internal timing circuitry. iButtons are considered slaves, while the host reader/writer is considered a master.

B. Write Time Slots

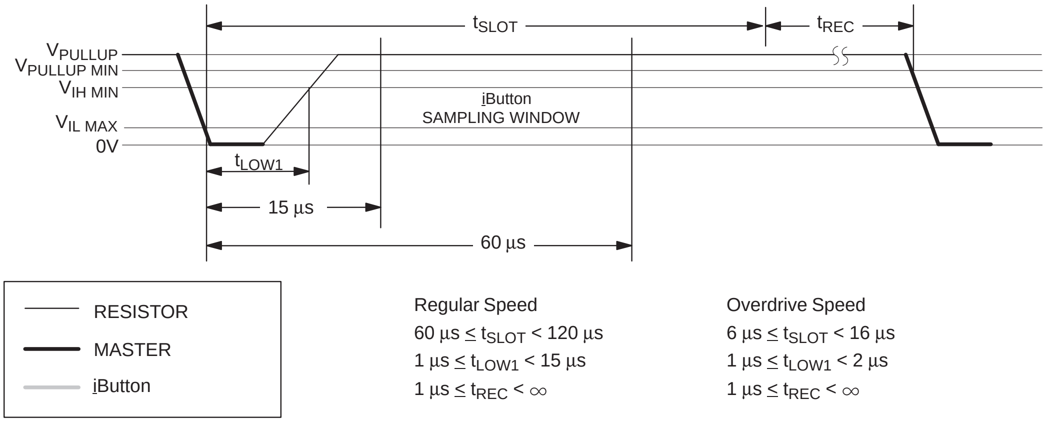
Timing relations in iButtons are defined with respect to time slots. To provide maximum margin for all types of tolerances, iButtons sample the status of the data line in the middle of a time slot. By definition the active part of a 1–Wire time slot (t_{SLOT}) is 60 µs. Under nominal conditions, an iButton will sample the line 30 µs after the falling edge of the start condition.

The internal time base of an iButton may deviate from its nominal value. The allowed tolerance band ranges from 15 µs to 60 µs. This means that the actual slave sampling may occur anywhere between 15 and 60 µs after the start condition, which is a ratio of 1 to 4. During this time frame the voltage on the data line must stay below V_{ILMAX} or above V_{IHMIN}. This explains the basic form of the write–1 and the write–0 time slots (Figures 4–1 and 4–2) as they are needed to write commands or data to iButtons. The duration of a low pulse to write a 1 (t_{LOW1}) must be shorter than 15 µs; to write a 0 the duration of the low pulse (t_{LOW0}) must be at least 60 µs to cope with worst–case conditions.

The duration of the active part of a time slot can be extended beyond 60 µs. The maximum extension is limited by the fact that a low pulse of a duration of at least eight active time slots (480 µs) is defined as a Reset Pulse. Allowing the same worst–case tolerance ratio, a low pulse of 120 µs might be sufficient for a reset. This limits the extension of the active part of a time slot to a maximum of 120 µs to prevent misinterpretation with reset.

At the end of the active part of each time slot, an iButton needs a recovery time t_{REC} of a minimum of 1 µs to prepare for the next bit. This recovery time may be regarded as the inactive part of a time slot, since it must be added to the duration of the active part to obtain the time it takes to transfer one bit. The wide tolerance of the time slots and the non–critical recovery time allow even slow microprocessors to meet the timing requirements for 1–Wire communication easily.

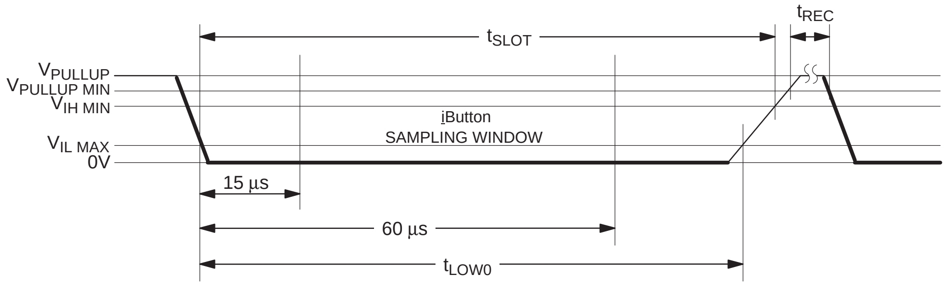
WRITE–ONE TIME SLOT Figure 4–1



Regular Speed
$60\ \mu s \le t_{SLOT} < 120\ \mu s$
$1\ \mu s \le t_{LOW1} < 15\ \mu s$
$1\ \mu s \le t_{REC} < \infty$

Overdrive Speed
$6\ \mu s \le t_{SLOT} < 16\ \mu s$
$1\ \mu s \le t_{LOW1} < 2\ \mu s$
$1\ \mu s \le t_{REC} < \infty$

WRITE–ZERO TIME SLOT Figure 4–2



─ RESISTOR

━ MASTER

▬ iButton

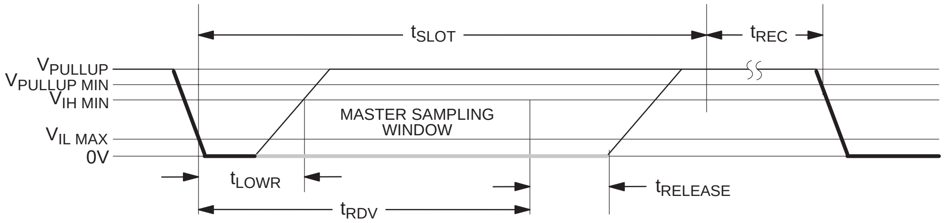
Regular Speed	Overdrive Speed
$60\ \mu s \le t_{LOW0} < t_{SLOT} < 120\ \mu s$	$6\ \mu s \le t_{LOW0} < t_{SLOT} < 16\ \mu s$
$1\ \mu s \le t_{REC} < \infty$	$1\ \mu s \le t_{REC} < \infty$

C. Read Time Slots

Commands and data are sent to iButtons by combining Write–Zero and Write–One time slots. To read data, the master has to generate Read–Data time slots to define the start condition of each bit. The Read–Data time slot looks essentially the same as the Write–One time slot from the master's point of view. Starting at the high–to–low transition, the iButton sends one bit of its addressed contents. If the data bit is a 1, the iButton leaves the pulse unchanged. If the data bit is a 0, the iButton will pull the data line low for t_{RDV} or 15 µs (see Figure 4–3). In this time frame data is valid for reading by the master.

The duration t_{LOWR} of the low pulse sent by the master should be a minimum of 1 µs with a maximum value as short as possible to maximize the master sampling window. In order to compensate for the cable capacitance of the 1–Wire line the master should sample as close to 15 µs after the synchronization edge as possible. Following t_{RDV} there is an additional time interval, $t_{RELEASE}$, after which the iButton releases the 1–Wire line so that its voltage can return to V_{PULLUP}. The duration of $t_{RELEASE}$ may vary from 0 to 45 µs; its nominal value is 15 µs.

READ–DATA TIME SLOT Figure 4–3



─ RESISTOR

━ MASTER

▬ iButton

Regular Speed	Overdrive Speed
$60\ \mu s \le t_{SLOT} < 120\ \mu s$	$6\ \mu s \le t_{SLOT} < 16\ \mu s$
$1\ \mu s \le t_{LOWR} < 15\ \mu s$	$1\ \mu s \le t_{LOWR} < 2\ \mu s$
$0 \le t_{RELEASE} < 45\ \mu s$	$0 \le t_{RELEASE} < 4\ \mu s$
$1\ \mu s \le t_{REC} < \infty$	$1\ \mu s \le t_{REC} < \infty$
$t_{RDV} = 15\ \mu s$	$t_{RDV} = 2\ \mu s$

D. Presence Detect

As mentioned above, 1–Wire timing also supports a Reset Pulse. This pulse is defined as a single low pulse of minimum duration of eight time slots or 480 µs followed by a Reset–high time t_{RSTH} of another 480 µs (see Figure 4–4). This high time is needed for the iButton to assert its Presence Pulse. During t_{RSTH}, no other communication on the 1–Wire line is allowed. The Reset Pulse is intended to provide a clear starting condition that supersedes any time slot synchronization. In an environment with uncertain contacts it is essential to have a means to start again if the contact is broken. If the master sends a Reset Pulse, the iButton will wait for the time t_{PDH} and then generate a Presence Pulse of the duration t_{PDL}. This allows the master to easily determine whether an iButton is on the line. Moreover, if several iButtons are connected in parallel (see Chapter 5, “Logical Standards and Characteristics”), the master can measure both times and thus gain knowledge about the actual worst–case timing tolerance of all devices on the line.

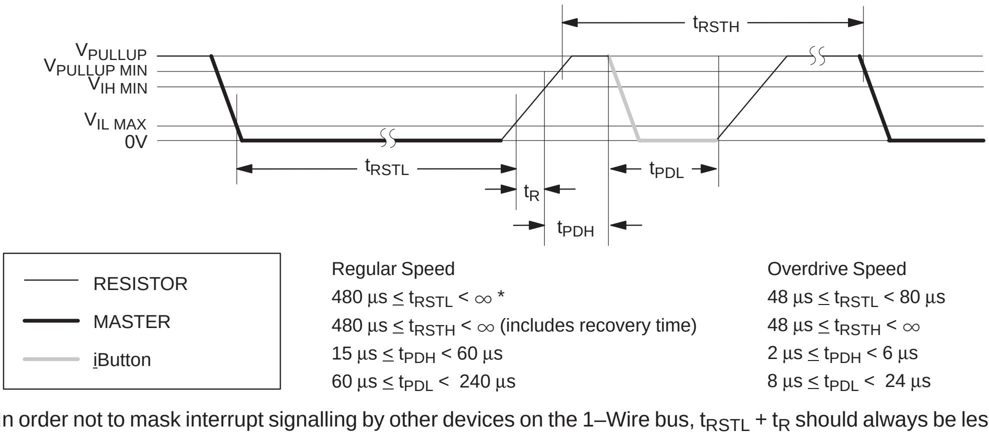
The nominal values are 30 µs for t_{PDH} and 120 µs for t_{PDL}. With the same worst–case tolerance band, the measured t_{PDH} value indicates the internal time base of the fastest device. The sum of the measured t_{PDH} and t_{PDL} values is five times the internal time base of the slowest device. If there is only one device on the line, both values will deviate in the same direction. This correlation can be used to build an adaptive system. Special care must be taken to recalibrate timing after every reset since the individual timing characteristics of the devices vary with temperature and load.

The accuracy of the time measurements required for adaptive timing is limited by the characteristics of the master’s input logic, the time constant of the 1–Wire line (pullup resistor x cable capacitance) and the applied sampling rate. If the observed rise time or fall time exceeds 1 µs or the highest possible sampling rate is less than 1 MHz, adaptive timing should not be attempted.

If an iButton is disconnected from the probe, it will pull its data line low via an internal current source of 5 µA. This simulates a Reset Pulse of unlimited duration. As soon as the iButton detects a high level on the data line, it will generate a Presence Pulse. This feature can be used to automatically power up the iButton Recorder, for example, to save energy between reads or writes. It is also useful to trigger a serial port hardware interrupt when using a PC and COM port adaptor.

iButtons are designed to operate with poor electrical connections. A certain contact dwell (minimum time of contact), however, is required to transfer commands and a packet of data. This time depends on the operation that has to be performed. As will be explained in Chapter 5, a ROM command and memory command form a command entity. A ROM command can be sent independently; a memory command must be preceded by a ROM command. Before an iButton can accept a ROM command, it needs to perform a Reset and Presence Detect cycle for a total of 16 time slots or a minimum of 0.96 ms. This time must be added to the results of the examples following.

RESET AND PRESENCE PULSE Figure 4–4



Regular Speed	Overdrive Speed
$480\ \mu s \le t_{RSTL} < \infty$ *	$48\ \mu s \le t_{RSTL} < 80\ \mu s$
$480\ \mu s \le t_{RSTH} < \infty$ (includes recovery time)	$48\ \mu s \le t_{RSTH} < \infty$
$15\ \mu s \le t_{PDH} < 60\ \mu s$	$2\ \mu s \le t_{PDH} < 6\ \mu s$
$60\ \mu s \le t_{PDL} < 240\ \mu s$	$8\ \mu s \le t_{PDL} < 24\ \mu s$

* In order not to mask interrupt signalling by other devices on the 1–Wire bus, $t_{RSTL} + t_R$ should always be less than 960 µs.

E. Examples

The minimum amount of time required to read the ROM–data is 8 + 64 complete time slots or 72 x 61 µs, which is 4.4 ms. Since data is written to Memory iButtons using the scratchpad, the minimum duration to transfer one 32–byte page of data is (8 + 8 + 16 + 256) x 61 µs or 17.6 ms. To read back the scratchpad for verification exceeds the writing time by eight time slots since a read–only status byte is transmitted in addition to base address and data. To write the complete scratchpad of the DS1991 takes about twice as long since it has a capacity of 64 bytes. Of course, it is possible to write less data than one complete scratchpad, but each new transfer requires another reset cycle, slowing down the effective data rate. If a sliding contact or another forced intermittence limits the duration of the contact time, the number of pages that can be read in a continuous sequence may be affected. In the worst case, where a certain device must be addressed, the total sequence to read one page takes (8 + 64 + 8 + 16 + 256) x 61 µs or 21.5 ms. If there is only one device on the data line, 17.6 ms is sufficient. These values are important to remember since the file structure to be used with Memory iButtons is based on reading and writing entire pages. The iButton file structure is discussed in Chapter 7.

F. Overdrive

As will be explained in Chapter 5, it is possible to put an iButton into an idle state from which it will no longer respond to anything on the 1–Wire bus until it sees a Reset pulse. With this feature, high–speed communication capability can coexist on the same 1–Wire bus without interfering with the existing protocol, provided that the 1–Wire bus never goes low for more than 120 µs.

The actual implementation of overdrive speeds up the internal time base of an iButton by the factor of 10. This applies to all communication waveforms including the Reset and Presence Pulse, but excluding programming pulses for EPROM devices. See Figures 4–1 to 4–4 for details. All iButtons capable of overdrive communication will communicate at regular speed if not explicitly set into overdrive mode. Overdrive–capable devices are identified by their family code or detected by using special command codes that are not understood by other devices. More details on the protocol are found in Chapter 5.

II. 1–Wire Interface – Electrical

A. Parasite Power

AC timing values in digital systems usually refer to voltages that represent maximum low and minimum high levels of the logic family. One–wire timing differs slightly from this well–known scheme due to the parasite power system that supplies the ROM and front–end logic of most iButtons. The DS1991 does not use a parasitic power supply since the device is of no use as a protected memory if the internal energy source is exhausted. The DS1990A, all Add–Only iButtons DS198x and the DS1920 Temperature iButton are designed for parasite power only. The ROM–logic of the DS1992 to DS1996 feeds either from the internal lithium voltage of 3V or from the 1–Wire voltage, whichever is higher. This feature allows access to the lasered ROM section of the DS1992 to DS1996 even if the lithium cell is exhausted after 10 years or more.

Parasite power systems need a capacitor to store energy and diodes to prevent unwanted discharge from the supply line. This is exactly the scheme of a half–wave rectifier. After the capacitor is charged to the normal operating level, there will be only some ripple voltage caused by recharging pulses restoring charge lost during 1–Wire low times. The size of an iButton's storage capacitor is about 800 pF. This capacity is seen for a short time when an iButton is contacted by a probe. After the capacitor is charged, only a very small fraction of this capacity is recognizable, according to the charge required to refill to full charge. The total time constant to charge the capacitor is defined by the capacitor itself, the internal resistances of about 1 kΩ, the resistance of the cable and contacts, the cable capacitance, and the resistor pulling up the data line. Adding the voltage drop of the diode and the minimum internal operating voltage of the chip gives the minimum required pull–up voltage Vpull–up on the 1–Wire bus. Table 4–1 shows the details, valid for all iButton products. The minimum pull–up voltage is important to define the recovery time t_{REC} and the reset high time t_{RSTH}. All other timing is based on the minimum high and maximum low logic levels (see Figures 4–1 to 4–4).

GENERIC ELECTRICAL SPECIFICATION OF iButtons Table 4–1

Value	DS1990A – DS1996	DS1982–DS1986	DS1920
Vpull–up min	2.8V		
V_{IH} min	2.2V		
V_{IL} max	0.8V		
V_{OL} max	0.4V at 4 mA[1]		
t_{SLOT} min	60 µs		
t_{SLOT} max	120 µs		
t_{RDV}	15 µs		
Special Power Supply	None	V_{PP} Programming Pulse	1 mA at 5V

NOTES:

1. Also see Figure 4–5 for worst case I_{OL} min at 50°C for DS1990A, DS1982, DS1991, and DS1992/3/4.

Due to their EPROM technology, Add–Only iButtons require pulses of a certain duration, voltage and current for programming. For the DS1982 the following values apply: pulse duration 500 µs minimum, voltage 11.5V nominal, peak current 10 mA maximum. To prevent damage to other iButtons on the MicroLAN, only DS198x type devices should be present during programming. The DS1920 Temperature iButton requires a strong active pull–up to 5V during temperature conversion and writing to the internal memory cells. This is not critical to other devices on the bus since iButtons will not start any communication by themselves. Even interrupts of DS1994 cannot occur in this situation since they are automatically disabled (see Chapter 6).

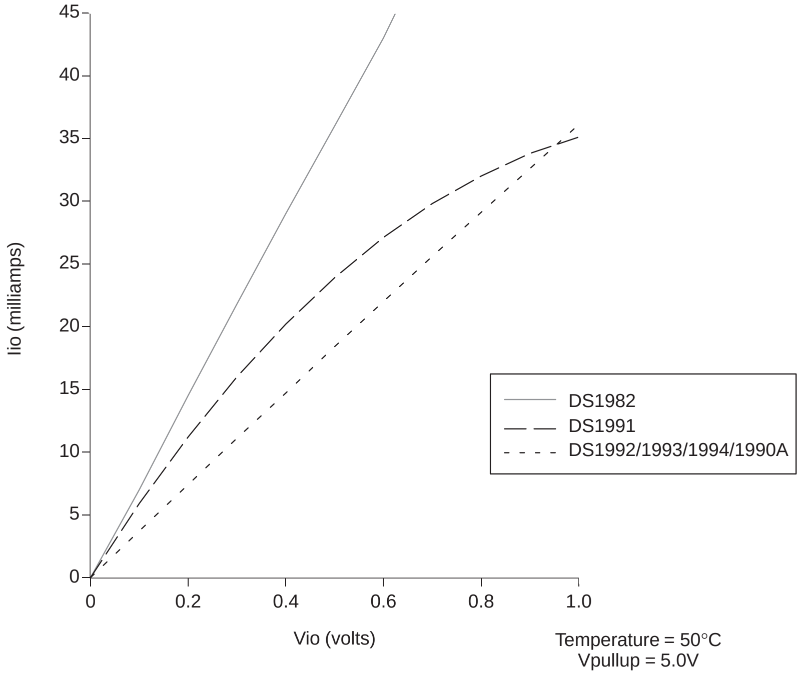
B. Pull–Up Resistor

The recommended pull–up resistor for the 1–Wire line is 5 kΩ for a short–length, single–contact probe. This value is chosen to allow tolerance for highly resistive contacts and to provide good logic levels at both ends of a short cable. Usually the reading circuitry of the master will accept a voltage of up to 0.8V as logic 0. Since the loading from the cable between master and iButton influences the time constant of the 1–Wire bus, it may be necessary to use pull–up resistors below 5 kΩ. Figure 4–5 shows the minimum sink current iButtons can supply to discharge the line at 50°C and the resulting output voltage. At lower temperatures, iButtons are able to sink even higher currents at lower voltage drops than shown in these graphs.

C. Margin Optimization

The ideal connection between an iButton and a computer is a short cable with low capacitance. As long as a iButton Recorder, iButton Editor or a host computer with a short cable are used, these requirements are fulfilled. The longer the lines, the more care must be taken about the DC and especially AC behavior. The recovery time and the conditions where a 1 is read or written become critical points. Recovery time (t_{REC}) becomes critical when two consecutive write–zero time slots are required for communication. If the data transfer is occurring at the maximum possible rate, a minimum t_{REC} pulse of 1 µs must be created between two 60 µs write–zero low times. The ability to propagate this small pulse down a long length of cable without severe degradation becomes increasingly difficult as lengths increase. Eventually the pulse is filtered completely and the driver becomes unable to communicate with the iButton at the end of the long cable. It may be possible to improve transmission distance by simply increasing the t_{REC} value. For example, if t_{REC} was increased from 1 µs to 15 µs, the maximum data rate would decrease from 16.3Kbits/s to 13.3Kbits/s, but the 15 µs t_{REC} pulse might allow operation to occur over much greater distances since there may be a sufficient recovery level presented to the iButton even with the filtering effect of the long cable because of the increased width of the recovery pulse. The write 1 condition can be improved by reducing the time t_{LOW1} of the write 1 time slot but not going below the minimum value. This also helps to increase the margin of the read 1 condition. As the cable becomes long, it will be necessary to reduce the pull–up resistor.

iButton OUTPUT VOLTAGE VS. OUTPUT CURRENT Figure 4–5



Tests have shown that pull–up resistors down to 1.0 kΩ allow operation over the maximum cable length. If the resistor is less than 1 kΩ, logic levels are degraded, and with higher values capacitive loading prevents the proper waveforms. Lower resistor values increase the sensitivity with respect to contact resistance. To maximize cable length, it is recommended to use low–capacitance cables, with a specific capacitance of about 15 pF/meter, and the largest resistor consistent with good operating margins.

Tests have also shown that twisted–pair cable gives better results than two wires in parallel. The simple pull–up resistor at the master has proven superior to strong pulsed active pull–up circuits since it matches the cable impedance better. For the same reason, it can be advantageous to limit the falling slope by introducing a driver with a soft turn–on at the master. Under extreme conditions, it may be necessary to add a comparator at the receive input of the master to optimize the voltage threshold for reading logic 1's and logic 0's on long lines.

As tests have proven, correct operation of iButtons can be achieved on lines up to 300 meters using common twisted–pair telephone cable. The pull–up resistor was reduced to 1.0 kΩ; 30 iButtons of any type were connected in parallel at the end of the cable. The 1–Wire bus was controlled by a port 0 pin of a DS5000 (Intel 8051 compatible) microcontroller. The COM port adaptor for PCs operates for distances up to 200 meters on most PCs. Further details are available in Dallas Semiconductor Application Note 55, "Extending the Contact Range of iButtons."

III. Summary

Communication with iButtons is done in time slots of nominal 60 µs duration. Each time slot transfers 1 bit. An extra long low pulse acts as a reset. In response to a reset, iButtons generate a Presence Pulse. To read one page of data (32 bytes), the contact dwell must be about 20 ms. Choosing low–capacitance, twisted–pair cable and a 1 kΩ pull–up resistor, iButtons can be operated over lines exceeding lengths of 200 meters. In order to create the maximum reading window, the master should sample the 1–Wire bus as close as possible to 15 µs after the synchronization edge. Increasing t_{REC} time to 15 µs will improve the recovery pulse received by the iButton.

LOGICAL STANDARDS AND CHARACTERISTICS

CHAPTER 5: LOGICAL STANDARDS AND CHARACTERISTICS

I. Overview

A. Protocol Architectural Model

The software or firmware that manages data transfer to and from iButtons can be related to the International Standards Organization (ISO) reference model of Open System Interconnection (OSI), which specifies a layered protocol having up to seven layers, denoted as Physical, Link, Network, Transport, Session, Presentation, and Application. According to this model, the electrical and timing requirements of iButtons and the characteristics of the 1–Wire bus comprise the Physical layer. The software functions TouchReset, TouchByte, and TouchBit correspond in this model with the Link layer, and the multidrop access system functions First, Next, Access, etc., that support selection of individual network nodes correspond with the Network layer. The software that transfers memory data other than ROM contents to and from the individual network nodes corresponds to the Transport layer. A Session layer may or may not be needed, depending on the environment in which the iButtons are used. The Presentation layer provides a file structure that allows Memory iButton data to be organized into independent files and randomly accessed (as with a diskette). The Application layer represents the final application designed by the customer. Each of the software layers relies on certain intrinsic commands within the iButton to accomplish its specific functions. In the following discussion of the operation of iButtons, the intrinsic commands that support the operation of a given software layer are described as belonging to that layer. Figure 5–1 shows the hierarchy of these layers. The following description follows the hierarchy of these protocol layers.

A.1. Physical Layer

This layer defines the electrical characteristics, logical voltage levels and general timing of iButton communication. Details are discussed in Chapter 4, “Electrical Standards and Characteristics.” How the Physical layer influences the electrical interface to computers and microcontrollers is detailed in Chapter 8, “Systems Integration Hardware.”

A.2. Link Layer

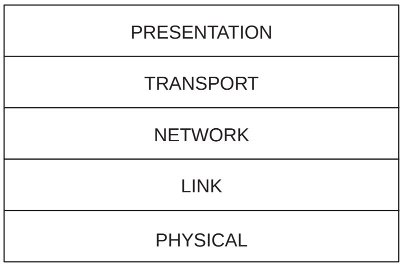
This layer defines the basic communication functions of iButtons. It provides the basic functions of Reset, Presence Detect and bit transfer. Details about the iButton hardware are presented in Chapter 4. The software issues are discussed in Chapter 9, “Systems Integration Software.” After issuing the Presence Pulse, iButton communication enters the Network layer.

A.3. Network Layer

This layer provides the identification of iButtons and the associated network capabilities. Each iButton has its own unique identification number that is lasered into a ROM section of the chip during the manufacturing process. Because of the ROM, all commands referring to the Network layer are also called ROM commands.

With the exception of the DS1990A, all iButtons support all commands of the Network layer. The DS1990A only supports Read ROM and Search ROM. The commands Skip ROM and Match ROM are not applicable since there is no other memory to access. However, if applied to the DS1990A, these commands will not cause any further activities on the 1–Wire bus. Software for the Network layer is discussed in Chapter 9. The following overview lists the commands, the command codes and some application details.

1–WIRE NETWORKING PROTOCOL LAYERED ARCHITECTURE Figure 5–1



READ ROM	code 33H	– to identify a device. – to determine if several devices are connected in parallel.
	code 0FH	– the DS1990A accepts this code as an alternative to 33H.
SKIP ROM	code CCH	– to omit addressing if only one device can be connected. – to broadcast data to all devices on the bus, e.g., to format many devices or to copy the contents of one device to many others. This application usually requires all iButtons to be of the same type and to be connected properly.
MATCH ROM	code 55H	– to address one specific iButton among several connected to the 1–Wire bus. If only one iButton can be connected, the Skip ROM command can be used instead.
SEARCH ROM	code F0H	– to get the registration numbers (= addresses) of all iButtons connected to the 1–Wire bus. – to get the serial number of one device and to simultaneously address it.
OVERDRIVE SKIP ROM	code 3CH	– to set all overdrive–capable devices into overdrive mode and to subsequently broadcast data at overdrive speed.
OVERDRIVE MATCH ROM	code 69H	– to address one specific iButton among several connected to the 1–Wire bus and set it into overdrive mode for subsequent communication at overdrive speed.

For reading the ROM the Search ROM command should be preferred over Read ROM. Search ROM is automatically compatible with the multidrop environment. After having read the ROM this way, the CRC can still be checked before communication is continued.

How to use the Search ROM command is detailed in subsection “Networking Capabilities” later in this chapter. After sending any ROM command and providing the required data or read time slots, one gets to the Transport layer. If this is not desired, a Reset Cycle returns to the Network layer.

A.4. Transport Layer

This layer is responsible for the data transfer between the non–ROM segments of the iButton and the master, and the data transfer from the scratchpad to the final storage areas and special registers of the Memory iButton.

Since the DS1990A contains no further memory, it cannot support the Transport layer. The DS1920 Temperature iButton is not a typical memory device and therefore differs on the Transport layer. All NV RAM–based iButtons support the memory commands Read Memory (called Read Subkey with the DS1991), Write Scratchpad, Read Scratchpad, and Copy Scratchpad. Because of its very special application area, the DS1991 also supports the commands Write Subkey and Write Password. The DS1991 uses different command words, addressing modes and page sizes than other Memory iButtons. For this reason it is not compatible on this layer. Due to their special technology, Add–Only iButtons require different command structures on the Transport layer. However, compatibility for the Read Memory command is provided. The following overview lists all the commands of iButtons, command codes, and application hints. For DS1920 and DS2407, refer to Chapter 6. The CRC16 is described in section II.A.3 of this chapter.

READ MEMORY	code F0H	– to read one or several consecutive bytes of one or consecutive pages starting at any valid address.
EXTENDED READ MEMORY	code A5H	– EPROM devices only: to read the redirection byte followed by an inverted CRC16, then read consecutive data bytes starting at any valid data address and obtain an inverted CRC16 of the previous data bytes at the end of the page; continued reading delivers the same sequence of information for the next pages (not available with DS1982).

READ SUBKEY	code 66H	– DS1991 only: to read one or several consecutive bytes of one password–protected page starting at any address from 10H to 3FH.
WRITE SCRATCHPAD	code 0FH	– NV RAM devices only: to supply the destination address and to write 1 to 32 consecutive bytes of data to the scratchpad.
	code 96H	– DS1991 only: to write one or several consecutive bytes to the scratchpad starting at any address from 0 to 3FH.
READ SCRATCHPAD	code AAH	– NV RAM devices only: to verify the destination address and the data previously written to the scratchpad. Three bits of a status byte read after the destination address indicate if scratchpad overflow or an incomplete byte occurred and if data has already been copied to memory.
	code 69H	– DS1991 only: to read one or several consecutive bytes of the scratchpad starting at any address from 0 to 3FH.
COPY SCRATCHPAD	code 55H	– NV RAM devices only: to copy data stored in the scratchpad to the destination address. This command requires that the scratchpad be read before getting the 3–byte authorization code that must be supplied with the copy scratchpad command.
	code 3CH	– DS1991 only: to copy either the entire scratchpad data or one specified 8–byte segment of the scratchpad to one password–protected page; the copied area of the scratchpad is automatically erased after the transaction is finished. This command requires the password of the destination page.
WRITE SUBKEY	code 99H	– DS1991 only: to write one or several consecutive bytes to one password–protected page starting at any address from 10H to 3FH. WARNING: if the iButton is removed from the probe too early, the secured data may be garbled. It is highly recommended to write data to the scratchpad, verify it, and then copy it to the password–protected page.
WRITE PASSWORD	code 5AH	– DS1991 only: to initialize the identifier and the password of a password– protected page. This command automatically erases the entire protected page. For subsequent changes of password or identifier, the new values should be written to the scratchpad, verified, and then copied to the final destination.
WRITE MEMORY	code 0FH	– EPROM devices only: to transfer, verify and program one or several consecutive bytes starting at any valid address within the data memory section.
WRITE STATUS	code 55H	– EPROM devices only: to transfer, verify and program one or several consecutive bytes starting at any valid address within the status memory section.
READ STATUS	code AAH	– EPROM devices only: to read one or several consecutive bytes starting at any valid address within the status memory section and to obtain an inverted CRC16 at the end of each page (different implementation with DS1982, see Chapter 6).

After these commands have been sent and executed, one must return to the Link layer to issue a new Reset Pulse. Additional readings after the end of data yield 1's. Extended writing is ignored. If applicable, the overflow bit is set.

A.5. Presentation Layer

The layers Link, Network, and Transport are the foundations of the Presentation layer. This layer provides a DOS–like file system supporting functions like Format, Directory, Type, Copy, Delete, Optimize, and integrity check. This makes Memory iButtons operate as conveniently as floppy disks. Details about the Presentation layer are given in Chapter 7, "iButtonFile Structure," and Chapter 9, "Systems Integration Software."

B. Section Summary

Because of different target applications iButtons differ in their logical behavior. The NV RAM iButtons DS1992 to DS1996 comprise a uniform group. With its password–protected memory, the DS1991 is also a group of its own, although it is compatible with the first three layers. The DS1990A contains only an electronic serial number; it has limited network capabilities but is completely compatible with the first three layers.

EPROM iButtons (DS1982 to DS1986) can replace NV RAM iButtons in applications where data is written once and read many times (“WORM”) or where adding data becomes more important than rewriting it. The semiconductor technology implies a different programming technique, resulting in limited compatibility on the Transport layer. Special commands are provided to support byte applications as well as file–oriented applications. The complete iButton compatibility matrix is shown in Table 5–1. Table 5–2 gives a summary of all command words used with iButtons.

TOUCH MEMORY COMPATIBILITY Table 5–1

Device Type	Physical Layer	Link Layer	Network Layer	Transport Layer	Presentation Layer
DS1990A	yes	yes	yes*	N/A	N/A
DS1991	yes	yes	yes	no****	no
DS1992	yes	yes	yes	yes	yes
DS1993	yes	yes	yes	yes	yes
DS1994	yes	yes	yes	yes	yes
DS1995	yes	yes	yes	yes	yes
DS1996	yes	yes	yes	yes	yes
DS1982	yes	yes	yes	yes***	yes
DS1985	yes	yes	yes	yes**	yes
DS1986	yes	yes	yes	yes**	yes
DS1920	yes	yes	yes	no****	N/A

* not all ROM functions applicable

** read compatible to NVRAM iButtons, different implementation of write commands

***The Read Memory command operates almost identically to NVRAM iButtons, different implementation of write commands.

**** application–specific command set, see Chapter 6 for details

iButton COMMANDS Table 5–2

Device Type	ROM Commands		Scratchpad Commands	Memory Commands		Password Commands	Status Commands
	Read	Skip Match Search	Read Write Copy	Read	Write	Write	Read Write
DS1990A	33H (0FH)	N/A N/A F0H	—	—	—	—	—
DS1991	33H	CCH 55H F0H	69H 96H 3CH	66H	99h	5AH	—
DS1992	33H	CCH 55H F0H	AAH 0FH 55H	F0H	—	—	—
DS1993	33H	CCH 55H F0H	AAH 0FH 55H	F0H	—	—	—
DS1994	33H	CCH 55H F0H	AAH 0FH 55H	F0H	—	—	—
DS1995	33H	CCH 55H F0H	AAH 0FH 55H	F0H	—	—	—
DS1996	33H	CCH 55H F0H	AAH 0FH 55H	F0H	—	—	—
DS1982	33H	CCH 55H F0H	—	F0H/C3H	0FH	—	AAH 55H
DS1985	33H	CCH 55H F0H	—	F0H/A5H	0FH	—	AAH 55H
DS1986	33H	CCH 55H F0H	—	F0H/A5H	0FH	—	AAH 55H
DS1920	33H	CCH 55H F0H	BEH 4EH 48H	Recall: B8H	—	—	Convert: 44H

II. Details

A. Fault–Tolerant Data Transfer

A.1. Introduction

iButtons are designed to operate under uncertain electrical conditions. The contact between master and iButton may break at any time and resume shortly afterwards. Despite these conditions, no data will be lost. If data is garbled, it must be discovered immediately before the data is used elsewhere.

Similar bad operating conditions are found with magnetic storage media like floppy disks. Floppies therefore don’t store data bit by bit along the whole track. Instead the track (i.e., a concentric cylinder) is divided into many sectors. Each sector is preceded by a small identification sector containing data about the sector itself. Between identification sector and data sector, and vice versa, are gaps containing constant bit patterns, but no stored data. To detect reading errors, both the identification sector and the data sector are extended by a 16–bit Cyclic Redundancy Check (CRC) pattern. This pattern is generated from the preceding data using a standardized algorithm. While reading, the disk controller recalculates the pattern. If the recalculated pattern and the pattern read from the disk match, the complete data transfer is assumed to be correct. Otherwise, another trial is started to read again. If data has to be written, the same calculation is done and the CRC pattern is written following the last data bit. Verification is possible by immediate reading after a write. This basic principle to cope with a difficult environment has proven so successful that it is used with every floppy system and most other magnetic storage media worldwide.

A.2. ROM–Section

Although Memory iButtons are silicon chips packaged in a MicroCan, they show amazing similarities to floppy disks. Both transfer data in a bit–sequential manner, have to cope with a difficult environment, and are organized into pages or sectors, respectively. Each iButton has one identification page, called ROM, containing device data. Unlike floppy disks, however, Memory iButtons need no gaps between data areas; every storage location is available for data. Because of the common demands of data transfer and error detection, it is no surprise that iButtons apply similar procedures to secure the data transfer. Unlike floppy systems that use the same CRC algorithm to protect the short identifier and the long data sector, iButtons apply different algorithms for each type of storage area.

The identifier, which is the ROM in iButton terminology, uses a one–byte CRC to protect the family code and the serial number. The CRC is generated using the polynomial $x^8 + x^5 + x^4 + 1$ and is written in its true form to each individual iButton chip by laser during manufacture. The identifiers of floppy sectors are written during the process of formatting the disk; unlike the identifier of an iButton, they can be erased by reformatting the disk.

To check if the ROM data is read correctly, the master reading the iButton has to recalculate the CRC and compare it with the value read from the device. (Detailed documentation on CRCs is found in the appendix of this book.) Non–matching values occur if the contact was bad or if several iButtons are connected at the same time. Repeated readings with the same data pattern (1’s and 0’s) but a mismatched CRC indicate several iButtons in parallel. How to obtain the ROM data of all of these iButtons is discussed in the subsection “Networking Capabilities” later in this chapter.

A.3. Memory Section

Despite of the fact that the logic of NV RAM iButtons supports writing single or multiple bytes, it is recommended to always write complete pages as is done with sectors of floppy disks. The iButton file system is also based on pages. It defines each page to start with the length byte indicating the number of bytes to follow, not counting the CRC, which takes two bytes instead of one as with the ROM. This structure is close to the definition of data sectors of floppies and allows easy checking of single pages or data packets. A two–byte CRC, the standardized CRC16, is used here since a one–byte CRC would not provide an adequate security level for the larger amount of data. The CRC16 is defined by the polynomial $x^{16} + x^{15} + x^2 + 1$. It is calculated from the data including the length byte and appended in its inverted form (1’s complement) to the data as it is written to the scratchpad. Before starting the CRC calculation, the CRC16 accumulator must be initialized by setting it equal to the Memory iButton page number where data is to be read or written. This allows the physical page address to be included as a factor in the CRC16 calculation for the data at a particular location. After verification of the scratchpad, the total page is copied to the final location in the Memory iButton. (For more details on CRCs of iButtons please refer to the appendix.) To check data integrity while reading the master will proceed as with ROM data, but use the CRC16 algorithm. The fact that every page of data has its own CRC is one of the fundamentals of the iButton file system. A mismatched CRC indicates a page addressing error, bad contacts, or the command Skip ROM was used in a situation where several iButtons are connected in parallel. Reading 0's permanently will indicate a short or a missing pull–up resistor at the 1–Wire bus; reading only 1's without receiving a Presence Pulse indicates a broken data line.

B. Command Processing

B.1. Introduction

iButtons are ready to accept commands as soon as they have reached the Network layer or the Transport layer. In each layer, an iButton can accept exactly one of the applicable commands. After execution of a Network command, the iButton automatically reaches the Transport layer. If no data transfer is requested except another Network command, e.g., Search ROM, the iButton needs to receive a Reset Pulse.

Usually the normal sequence of communication is

1. Master: sends Reset Pulse.
   T.M.: sends Presence Pulse.
2. Master: sends ROM Command, possibly followed by data or read time slots.
   T.M.: listens or sends data.
3. Master: sends Memory Function Command, possibly followed by data or read time slots.
   T.M.: listens or sends data.

After the memory command is executed, the master must send a Reset Pulse to begin another communication session with iButtons on the 1–Wire bus.

This method of operation requires no special addressing of a command register. Simply the fact that bits are sent after reaching a certain layer qualifies them as commands. Invalid command codes set iButtons into an idle state; the next Reset Pulse will synchronize them again.

B.2. Bit Sequence

Commands and data can be understood as binary numbers. The least significant bit of a byte or character is always the first to be sent or received. Multiple precision numbers are usually stored with the least significant byte at the lower address. The first character of a text string also has a lower address than the last one. Therefore, to write data to an iButton, the master has to start at the lowest address, load a byte, generate the time slot that corresponds to the least significant bit, shift the byte to the right, generate the next time slot, etc. The iButton will assemble the bits into bytes and store them at ascending addresses. Reading iButtons works generally in the same way. Instead of Write time slots, the master generates Read time slots, enters received data bits into the most significant position of a shift register, shifts to the right and stores the data in the same sequence as it had been loaded for writing. Again, the iButton will act as the master did while writing the data and advance the memory address after every transferred byte. This structured method of operation is fundamental to iButton communication.

C. MicroLAN – Networking Capabilities

C.1. Introduction

All iButton Products are designed to operate in a networking environment. This extends the field of applications to higher storage capacity and to distributed data storage using only one common data line to the master. Networks always require identification numbers of all nodes within the network. iButtons have their individual ROM, which is well suited as a node identifier. The user needn't worry about conflicting node identifiers. The open drain interface of the 1–Wire data bus avoids potential problems if bus conflicts occur. In fact, the 1–Wire data interface actually is a 1–Wire LAN (MicroLAN^{TM}) with all features required for single master multidrop bus operation.

C.2. Command Overview

To operate standalone as well as on a bus, iButtons support the following ROM–based Networking Commands: Read ROM, Skip ROM, Match ROM and Search ROM. After execution of any ROM command, the Transport layer is reached. The command Read ROM, code 33H, is used to identify a device on the 1–Wire bus or to find out if several devices are connected at the same time. After sending this command the master has to generate 64 read time slots. The iButton will send its ROM contents least significant bit first, starting with family code, followed by the serial number and the CRC byte. If several iButtons are connected, no reading will provide a matching CRC–byte. In this case, the command Search ROM F0H must be used to determine the ROM contents of the devices before they can be addressed. If the ROM contents are not of interest because there can be only one iButton on the data bus, the search can be skipped by sending the Skip ROM command, code CCH. Immediately after this command, the device reaches the Transport layer.

The command Match ROM can be used to address one device if one of several iButtons are connected in parallel. The ROM contents act as a device address to activate exactly one device. The same ROM contents in more than one device are impossible due to strict manufacturing control. If two iButtons have the same serial number, their family codes will be different. In this manner, any confusion or contention is avoided. The Match ROM command, code 55H, requires the ROM contents of the desired device to be sent by the master during the 64 time slots following the command. The sequence of the bits must be the same as they were delivered by reading the ROM, i.e., least significant bit first, starting with family code, followed by the serial number and the CRC. All iButtons whose ROM contents do not match the requested code will stay idle until they receive another Reset Pulse.

C.3. Search ROM Command

Even if the master does not know the serial numbers of the devices connected to the 1–Wire bus, it is possible to address one single device at a time. This is done by using the command Search ROM, code F0H. This command acts like Read ROM combined with Match ROM. All iButtons will sequentially send the true and the false value of the actual ROM bit during the two Read time slots following the Search ROM command. If all devices have a 0 in this bit position, the reading will be 01; if the bit position contains a 1, the result will be 10. If both, a 1 and a 0 occur in this bit position, reading will result in two 0 bits, indicating a conflict. The master now has to send the bit value 1 or 0 to select the devices that will remain in the process of selection. All deselected devices will be idle until they receive a Reset Pulse. After the first stage of selection, 63 reading/selecting cycles will follow, until finally the master has learned one device's ROM code and simultaneously has addressed it. Each stage of selection consists of two Read time slots and one Write time slot. The complete process of learning and simultaneous addressing is about three times the length of the Match ROM command, but it allows selection of all the connected devices sequentially without knowing the ROM values beforehand. In an application where the iButtons are fixed in position on the 1–Wire bus, it is most efficient for the master to evaluate all ROM contents with the Search ROM command and then to use the Match ROM command to address specific devices. If the application requires constant identification and communication with new devices as they come and go, it will be necessary to use the Search ROM command to identify and address each new part.

A flowchart of all ROM Commands is shown in Figure 5–2. Since the logic of the Search ROM command is the most complex, the following example is used to illustrate it step by step.

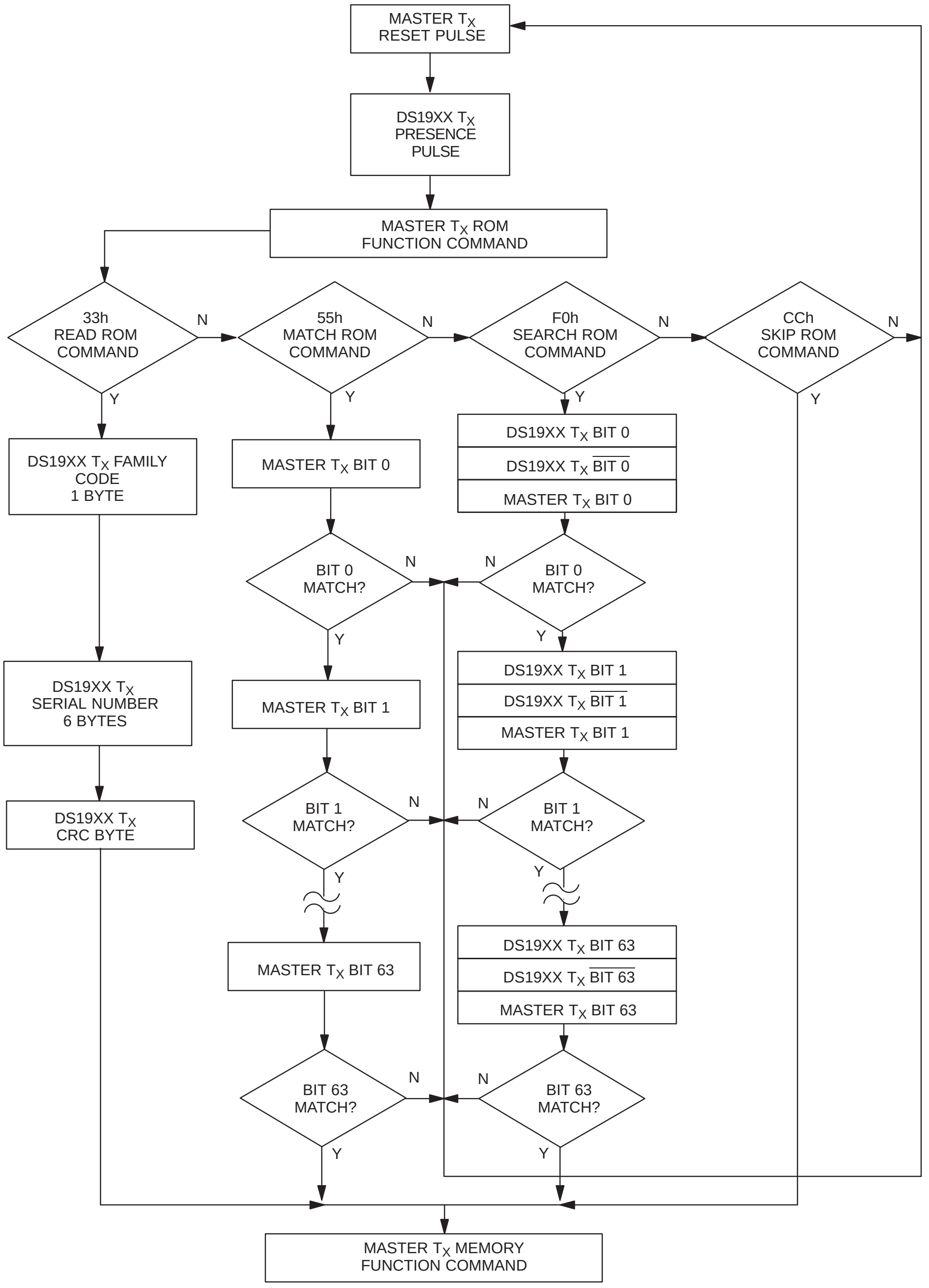
Four devices are connected to the 1–Wire bus. Their binary ROM contents are:

device 1: xxxxxx10101100
device 2: xxxxxx01010101
device 3: xxxxxx10101111
device 4: xxxxxx10001000

The x's represent the higher remaining bits. Shown are the lowest eight bits of the ROM contents. The least significant bit is rightmost in this representation. The search process runs as follows:

1. The master begins the initialization sequence by issuing a Reset Pulse. The iButtons respond by issuing Presence pulses.
2. The master will then issue the Search ROM command on the 1–Wire Bus.
3. The master reads one bit from the 1–Wire bus. Each device will respond by placing the value of the first bit of its respective ROM data onto the 1–Wire bus. Devices 1 and 4 will place a 0 onto the 1–Wire bus; that is, they pull it low. Devices 2 and 3 will send a 1 by allowing the line to stay high. The result is the logical AND of all devices on the line; therefore the master reads a 0. The master will read another bit. Since the ROM Search command is being executed, all devices respond to this second read by placing the complement of the first bit of their respective ROM data onto the 1–Wire Bus. Devices 1 and 4 will send a 1; devices 2 and 3 will send a 0. Thus the 1–Wire bus will be pulled low. The master again reads a 0 for the complement of the first ROM data bit. This tells the master that there are devices on the bus that have a 0 in the first position and others that have a 1. If all devices had a 0 in this bit position, the reading would be 01; if the bit position contained a 1, the result would be 10.

ROM FUNCTIONS FLOW CHART   Figure 5–2



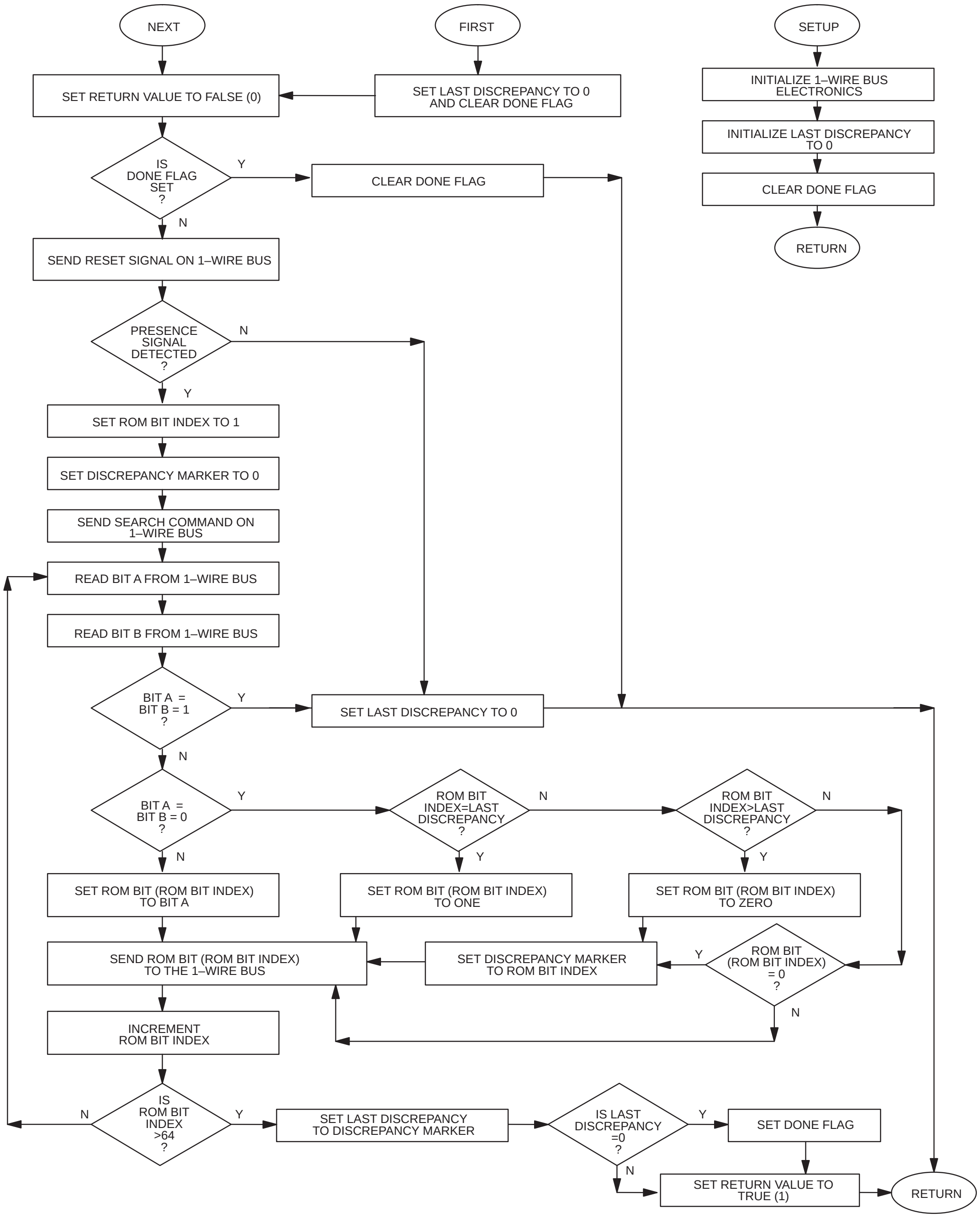
4. The master now decides to write a 0 on the 1–Wire bus. This deselects Devices 2 and 3 for the remainder of the search pass, leaving only devices 1 and 4 participating in the search process.
5. The master performs two more reads and receives a 0 followed by a 1 bit. This indicates that all active devices have a 0 in this bit position of their ROM.
6. The master then writes a 0 to keep devices 1 and 4 selected.
7. The master executes two reads and receives two 0 bits. This again indicates that both 1 and 0 exist as the third bit of the ROM of the active devices.
8. The master again writes a 0. This deselects device 1, leaving device 4 as the only active device.
9. The following reads to the end of the ROM will not show bit conflicts. Therefore, they directly tell the master the ROM contents of the active device. After having learned any new ROM bit, the master has to resend this bit to keep the device selected. As soon as all ROM bits of the device are known and the last bit is resent by the master, the device is ready to accept a command of the Transport layer.
10. The master must learn the other devices' ROM data. Therefore, it starts another ROM Search sequence by repeating steps 1 through 7.
11. At the highest bit position, where the master wrote a 0 at the first pass (step 8), it now writes a 1. This deselects device 4, leaving device 1 active.
12. As in step 9, the following reads to the end of the ROM will not show bit conflicts. This completes the second ROM Search pass where the master has learned another ROM's contents.
13. The master must learn the other devices' ROM data. Therefore, it starts another ROM Search sequence by repeating steps 1 to 3.
14. At the second highest bit position where the master wrote a 0 at the first pass (step 4), it now writes a 1. This deselects devices 1 and 4, leaving devices 2 and 3 active.
15. The master sends two read time slots and receives two 0 bits, indicating a bit conflict.
16. The master again decides to write a 0. This deselects device 3, leaving device 2 as the only active device.
17. As in step 9, the following reads to the end of the ROM will not show bit conflicts. This completes the third ROM Search pass where the master has learned another ROM's contents.
18. The master must learn the other devices' ROM data. Therefore it starts another ROM Search sequence by repeating steps 13 to 15.
19. At the highest bit position where the master wrote a 0 at the previous pass (step 16), it now writes a 1. This deselects device 2, leaving device 3 active.
20. As in step 17, the following reads to the end of the ROM will not show bit conflicts. This completes the fourth ROM Search pass where the master has learned another ROM's contents.

The general principle of this search process is to deselect one device after another at every conflicting bit position. At the end of each ROM Search process, the master has learned another ROM's contents. The next pass is the same as the previous pass up to the point of the last decision. At this point the master goes in the opposite direction and continues. If another conflict is found, again 0 is written, and so on. After both ways at the highest conflicting bit position are followed to the end, the master goes the same way as before but deciding oppositely at a lower conflicting bit position, and so on, until all ROM data are identified.

An optimized flowchart of the Search ROM algorithm is shown in Figure 5–3. This figure explains how to perform a general ROM search. For the purpose of this flowchart, the ROM data is accumulated into a bit array named ROM Bit, with bits numbered 1 to 64. Setup must be called before any other function to initialize the 1–Wire system. A call to "First" resets the search to the beginning and identifies the first ROM code, and calls to "Next" identify successive ROM codes. A false value returned indicates no more ROM codes to be found.

The time required to learn one ROM's contents (not counting the master's CPU time) is 960 μs+(8+3*64)*61 μs =13.16 ms. Thus it is possible to identify up to 75 different iButtons per second.

ROM SEARCH Figure 5–3


NEXT
FIRST
SETUP
SET RETURN VALUE TO FALSE (0)
SET LAST DISCREPANCY TO 0 AND CLEAR DONE FLAG
INITIALIZE 1–WIRE BUS ELECTRONICS
INITIALIZE LAST DISCREPANCY TO 0
CLEAR DONE FLAG
RETURN
IS DONE FLAG SET ?
Y
N
CLEAR DONE FLAG
SEND RESET SIGNAL ON 1–WIRE BUS
PRESENCE SIGNAL DETECTED ?
N
Y
SET ROM BIT INDEX TO 1
SET DISCREPANCY MARKER TO 0
SEND SEARCH COMMAND ON 1–WIRE BUS
READ BIT A FROM 1–WIRE BUS
READ BIT B FROM 1–WIRE BUS
BIT A = BIT B = 1 ?
Y
N
SET LAST DISCREPANCY TO 0
BIT A = BIT B = 0 ?
Y
N
ROM BIT INDEX=LAST DISCREPANCY ?
N
Y
ROM BIT INDEX>LAST DISCREPANCY ?
N
Y
SET ROM BIT (ROM BIT INDEX) TO BIT A
SET ROM BIT (ROM BIT INDEX) TO ONE
SET ROM BIT (ROM BIT INDEX) TO ZERO
SEND ROM BIT (ROM BIT INDEX) TO THE 1–WIRE BUS
SET DISCREPANCY MARKER TO ROM BIT INDEX
Y
ROM BIT (ROM BIT INDEX) = 0 ?
N
INCREMENT ROM BIT INDEX
N
IS ROM BIT INDEX >64 ?
Y
SET LAST DISCREPANCY TO DISCREPANCY MARKER
IS LAST DISCREPANCY =0 ?
Y
N
SET DONE FLAG
SET RETURN VALUE TO TRUE (1)
RETURN

C.4. Overdrive

As has been explained above, the ROM contents play an important role in addressing and selecting devices on the 1–Wire bus. All devices but one will be in an “idle” state after the Match ROM command or the Search ROM command has been executed. They will return to the normal state only by receiving a Reset pulse.

Since devices with overdrive capability will be distinguished from others by their family code and the two additional ROM commands Overdrive Skip ROM and Overdrive Match ROM, they can easily be identified and addressed. The first transmission of the ROM command itself takes place at the ”normal” speed that is understood by all 1–Wire devices. After a device with overdrive capability has been addressed and set into overdrive mode (i.e., after the appropriate ROM command has been received) further communication to that device has to occur at overdrive speed. Since all deselected devices remain in the idle state as long as no Reset pulse of regular duration is detected, even multiple overdrive components can reside on the same 1–Wire bus. A Reset pulse of regular duration will reset all 1–Wire devices on the bus and simultaneously set all overdrive–capable devices to regular speed. The first overdrive–capable devices are the DS1995, DS1996 and DS1986. For a full description of the overdrive protocol please refer to one of these product data sheets.

D. Data Transfer

At the Transport layer, iButtons are split into three different groups: a) NV RAM devices, b) EPROM devices, and c) the DS1991 MultiKey. The following sections explain data transfer of NV RAM devices and EPROM devices. For the DS1991 MultiKey as well as other iButtons (no memories), see Chapter 6.

D.1. Memory iButtons (NVRAM)

All iButtons in this group share the same command structure at the Transport layer. The available commands are Read Memory, Write Scratchpad, Read Scratchpad, and Copy Scratchpad.

D.1.a. Transfer Status

Any memory access within the Transport layer requires addressing of data. Because of the serial data transfer, NV RAM iButtons employ three address registers, called TA1, TA2 and E/S (see Figure 5–4). Registers TA1 and TA2 must be loaded with the target address to which the data will be written or from which data will be sent to the master upon a Read command. Register E/S acts like a byte counter and Transfer Status register. It is used to verify data integrity with Write commands. Therefore, the master only has read access to this register. The lower five bits of the E/S register indicate the address of the last byte that has been written to the scratchpad . This address is called Ending Offset. Bit 5 of the E/S register, called PF or “partial byte flag,” is set if the number of data bits sent by the master is not an integer multiple of 8. Bit 6, OF or “Overflow,” is set if more bits are sent by the master than can be stored in the scratchpad. Note that the lowest five bits of the target address also determine the address within the scratchpad, where intermediate storage of data will begin. This address is called byte offset. If the target address for a Write command is 13CH for example, then the scratchpad will store incoming data beginning at the byte offset 1CH and will be full after only four bytes. The corresponding ending offset in this example is 1FH. For best economy of speed and efficiency, the target address for writing should point to the beginning of a new page, i.e., the byte offset will be 0. Thus the full 32–byte capacity of the scratchpad is available, resulting also in the ending offset of 1FH. The iButton File System also organizes the data packets of files exactly this way to achieve the best match to the hardware characteristics of the iButton device.

For special purposes, however, such as accessing the clock and associated registers of the DS1994, it is necessary to write one or several contiguous bytes somewhere within a page. This is the reason for the implementation of the byte offset mechanism. The ending offset together with the Partial and Overflow Flag is mainly a means to support the master checking the data integrity after a Write command. The highest valued bit of the E/S register, called AA or Authorization Accepted, acts as a flag to indicate that the data stored in the scratchpad has already been copied to the target memory address. Writing data to the scratchpad clears this flag.

MEMORY iButton ADDRESS REGISTERS TA1, TA2 AND E/S Figure 5–4

T7	T6	T5	T4	T3	T2	T1	T0	TA1
T15	T14	T13	T12	T11	T10	T9	T8	TA2
AA	OF	PF	E4	E3	E2	E1	E0	E/S

D.1.b. Reading

To read data from a Memory iButton the master first sets the internal logic of the device to the Transport layer. This can be done by sending a Reset Pulse followed by a Read ROM command. After the ROM contents are verified with the CRC byte, the master sends the READ Memory command, code F0H, followed by the low byte and the high byte of the target address. The least significant address bit is sent first. Now with every Read Data time slot, the master will get one more of the addressed data bits; again the least significant bit of a byte is sent first. After a complete byte is transmitted, the Memory iButton will increment the address pointer to transmit the next byte. Thus the entire memory can be read. Page boundaries have no impact on reading. After the data of the highest available address is sent out, the master will read 1's during all subsequent Read time slots.

D.1.c. Writing with Verification

To write data to the Memory iButton, the scratchpad has to be used as intermediate storage. First the master issues the Write Scratchpad command to specify the desired target address, followed by the data to be written to the scratchpad. In the next step, the master sends the Read Scratchpad command to read the scratchpad and to verify data integrity. As preamble to the scratchpad data, the Memory iButton sends the requested target address TA1 and TA2 and the contents of the E/S register. This speeds up checking the data integrity. If one of the flags OF or PF is set, data did not arrive correctly in the scratchpad. The master does not need to continue reading; it can start a new trial to write data to the scratchpad. Similarly, a set AA flag indicates that no write scratchpad command has been received by the Memory iButton since the last copy scratchpad command. If everything went correctly, all three flags are cleared and the ending offset indicates the address of the last byte written to the scratchpad. Now the master can continue verifying every data bit. After the master has verified the data, it has to send the Copy Scratchpad command. This command must be followed exactly by the data of the three address registers TA1, TA2 and E/S as the master has read them verifying the scratchpad. As soon as the Memory iButton has received these bytes, it will copy the data to the requested location beginning at the target address. Copying takes about 30 µs. If after copying the scratchpad is read again, the set AA flag indicates that the data in the scratchpad is now obsolete and the device can accept new data without losing anything.

This complex mechanism for writing is implemented to provide the highest possible level of data integrity without losing too much speed. To verify correct reading, it is highly recommended to initialize the CRC16 with the iButton page number, to put a length byte at the beginning of each data packet, and to append an inverted CRC16 double byte at the end of the data, as is practiced with the iButton File System (see Chapter 7, "File Structure"). Details of writing data to Memory iButtons are shown in Figure 5–5.

Directly after the Write Scratchpad command, code 0FH, the master must send the target address and data. The maximum number of bytes the scratchpad accepts without overflow is 20H minus data offset, which is the five least significant bits of the target address. After the data is transferred, the master must send a Reset Pulse and again set the Memory iButton to the Transport layer. The Read Scratchpad command, code AAH, activates the Memory iButton to reply with the target address, ending offset, with transfer status and the received data. In the third step, the master must again issue a Reset Pulse, set the iButton to the Transport layer, and send the Copy Scratchpad command, code 55H, followed by the actual target address and the E/S byte. Only if these three bytes match those stored in the Memory iButton registers is the command accepted. As long as copying is in progress, the device will ignore any Reset pulse.

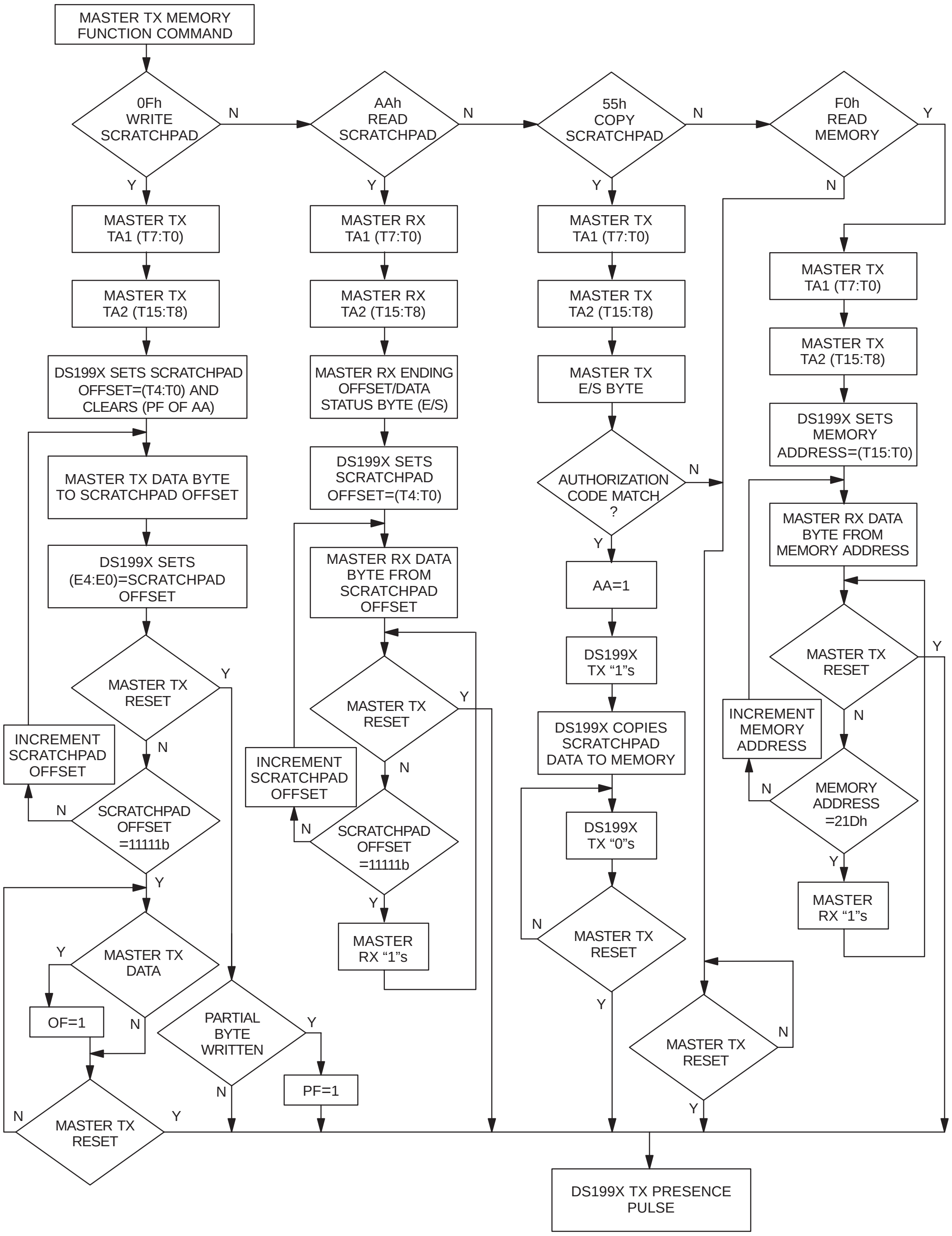
D.2. Add–Only iButtons (OTP EPROM)

The high–capacity devices in this group share the same command structure at the Transport layer. The available commands are Read Memory, Extended Read Memory (not available with the DS1982), Write Memory, Read Status and Write Status. Since the scratchpad is only one byte long, it needs no special addressing. The write command implies writing to the scratchpad first, before data is copied to memory by the program pulse.

D.2.a. Transfer Status

The scratchpad of EPROM iButtons is only one byte. Thus special scratchpad commands and a transfer status register are not required. In the same way as for NV RAM–based iButtons, the bus master must supply the target address TA1 and TA2. EPROM iButtons have a CRC generator onboard to enhance data integrity for both reading and writing. For the high–capacity devices the CRC16 algorithm is used; the CRC is sent out in bit–inverted form (one's complement) least significant byte first.

MEMORY FUNCTION FLOW CHART NVRAM DEVICES Figure 5–5


MASTER TX MEMORY FUNCTION COMMAND
0Fh WRITE SCRATCHPAD
AAh READ SCRATCHPAD
55h COPY SCRATCHPAD
F0h READ MEMORY
N
Y
MASTER TX TA1 (T7:T0)
MASTER TX TA2 (T15:T8)
DS199X SETS SCRATCHPAD OFFSET=(T4:T0) AND CLEARS (PF OF AA)
MASTER TX DATA BYTE TO SCRATCHPAD OFFSET
DS199X SETS (E4:E0)=SCRATCHPAD OFFSET
MASTER TX RESET
INCREMENT SCRATCHPAD OFFSET
SCRATCHPAD OFFSET =11111b
MASTER TX DATA
OF=1
PARTIAL BYTE WRITTEN
PF=1
MASTER RX TA1 (T7:T0)
MASTER RX TA2 (T15:T8)
MASTER RX ENDING OFFSET/DATA STATUS BYTE (E/S)
DS199X SETS SCRATCHPAD OFFSET=(T4:T0)
MASTER RX DATA BYTE FROM SCRATCHPAD OFFSET
MASTER RX "1"s
MASTER TX E/S BYTE
AUTHORIZATION CODE MATCH ?
AA=1
DS199X TX "1"s
DS199X COPIES SCRATCHPAD DATA TO MEMORY
DS199X TX "0"s
DS199X SETS MEMORY ADDRESS=(T15:T0)
MASTER RX DATA BYTE FROM MEMORY ADDRESS
INCREMENT MEMORY ADDRESS
MEMORY ADDRESS =21Dh
DS199X TX PRESENCE PULSE

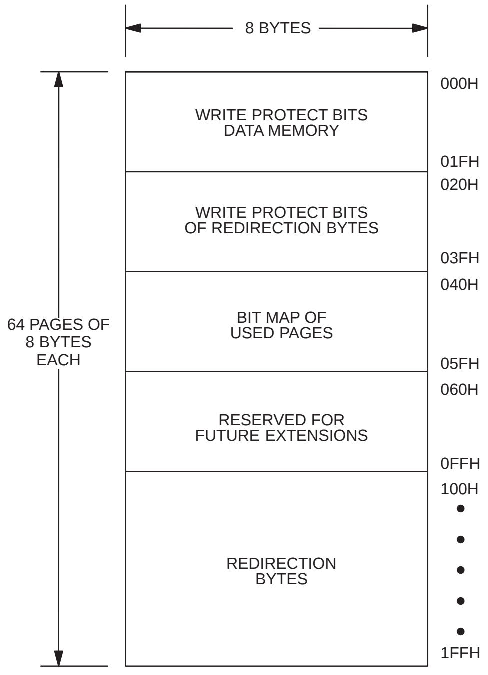
D.2.b. Status Memory

Due to their special features, EPROM iButtons or Add–Only iButtons as they are called, also require additional memory to store write protect flags, redirection bytes and a bitmap indicating used and empty pages. The bitmap supports the operating system, while the other status information interacts directly with the logic on–chip, either by preventing an action (write protect) or by being sent to the master as part of a data stream (redirection byte). All of these pieces of status and management information are stored in the status memory of the device. With the exception of the DS1982, the status memory has the same predetermined structure for all devices (see Figure 5–6).

The status address range contains 512 bytes (000H to 1FFH). This is sufficient for up to 256 pages, equal to the upper limit of the extended file structure (for details see Chapter 7 of this book). The first 32 bytes (address 000H to 01FH) represent all write protect bits for the data memory. Bit 0 of the first byte of the status memory is associated with page 0 and so on. The next 32 bytes (address 020H to 03FH) are used to write protect the redirection byte of each page. Bit 0 of the first byte of this group is associated with page 0. The following bytes (address 040H to 05FH) store the bitmap of used pages. (With EPROM devices, this bitmap is not stored as part of the device directory or as a separate file. Keeping the same scheme as is used with NV RAM iButtons is theoretically possible but very uneconomical, since an update of a single bit would require rewriting and redirecting at least one total page of memory.) Starting at address 100H, the memory area for redirection bytes begins. Page numbers range from 0 to 255 (0 to FFH). The physical address of the redirection byte is identical to the page number plus 256 (or page number in hex plus 100H). The address range of 60H to 0FFH is reserved for future extensions.

If certain areas of status memory are not used since the device has less than 256 pages, these areas are not implemented in silicon. Reading these gaps produces 1's on the data bus; writing is ignored. The status memory can be programmed in the same way as the data memory. To allow fast and integrity–checked access, the page size of status memory is limited to 8 bytes (instead of 32). After a page boundary is encountered (xx7H), the device will send an inverted CRC16 to the master.

STATUS MEMORY MAP OF ADD–ONLY iButtons Figure 5–6



The DS1982, as a 4–page device, does not support the full functionality of Add–Only iButtons. Therefore its total status memory is organized as one page of 8 bytes, avoiding big gaps in the memory map. The first 5 bytes contain status information. The contents of the other bytes can be ignored by the application software. Details on the DS1982 are found in Chapter 6.

D.2.c Reading

Reading Add–Only iButtons is very similar to reading NV RAM–based iButtons. Figure 5–7 shows the flowchart of the available read commands. Compared to the path READ MEMORY of Figure 5–5, there is one minor difference:

If the master continues reading beyond the end of memory, an Add–Only iButton will send out another CRC16 after the last memory byte is transmitted. Further Read Time Slots will transmit 1's only. An NV RAM device will send 1's as soon as the end of memory is reached. The command code F0H to read an Add–Only iButton remains exactly the same as for NV RAM iButtons. If software already exists using the file structure as described in Chapter 7, Add–Only iButtons can be integrated into the system by discarding the additional CRC information.

The Read Memory command allows NVRAM–compatible reads of Add–Only iButtons in an environment where the device is programmed once according to the rules of the extended file structure. In an environment where Add–Only iButtons act as data carriers that store valid and invalid information at the same time, it is clumsy to first access the status memory to find out if the page you want to read is still up–to–date. For this reason and to support files as well as byte applications, there is an additional read function with Add–Only iButtons called Extended Read Memory, command code A5H.

The major difference between Read Memory and Extended Read Memory is that in an Extended Read the addressed data is preceded by the redirection byte of the addressed page and the inverted CRC16 of the command, address and redirection byte. As the end of a memory page is encountered, the master will receive an inverted CRC16 of the data just read from this page. This CRC allows the master to prove data integrity even for byte applications by comparing the received CRC with its own calculations. If the master continues reading, the next bytes to be received are the redirection byte of the next page followed by an inverted CRC16 of this redirection byte. The next bits to be received are data from memory.

Unprogrammed redirection bytes read FF hexadecimal. If the redirection byte is different from FFH, this will signal the master that the data is not up–to–date. The one's complement of the redirection byte is the page address of the new data. Without significant loss of time, the master can start another read access, this time to the page that is indicated by the redirection byte. This process can be repeated until finally a page without redirection is accessed, containing the latest information. Reading redirected pages may also be of interest if intermediate versions of data are important.

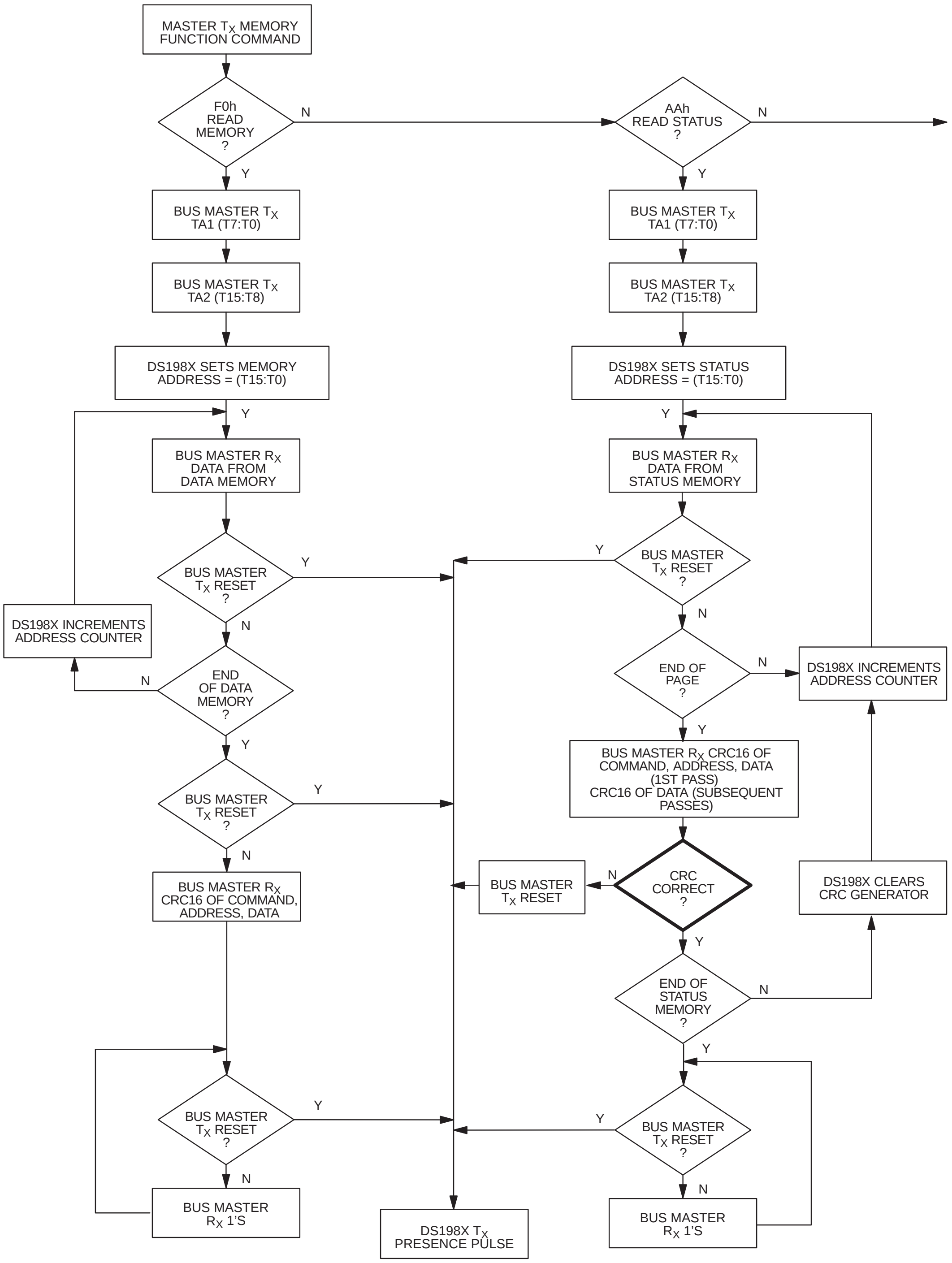
The Extended Read Memory command provides two CRC16s for every pass. The CRC16 at the end of a data page is generated by clearing the CRC accumulator first and then shifting in all data bytes starting at the first addressed location of the page. The calculation of the CRC after the redirection byte depends on whether this is the first pass or a subsequent pass through the command flow without a Reset in between. For the first pass the CRC is generated by clearing the CRC accumulator, shifting in the command, address bytes TA1 and TA2 and the redirection byte. For subsequent passes the CRC generator is also cleared and then only the redirection byte is shifted in.

A flowchart similar to the one for reading memory data applies to the READ STATUS command. The major difference is the command code AAH and the inverted CRC16 sent to the master at the end of each 8–byte status memory page. Since the information inside the status memory consists of independent bits and bytes that are likely to be updated later, it is not possible to store an inverted CRC16 value in the data. The only means to check for data integrity is the inverted CRC16 generated by the device itself.

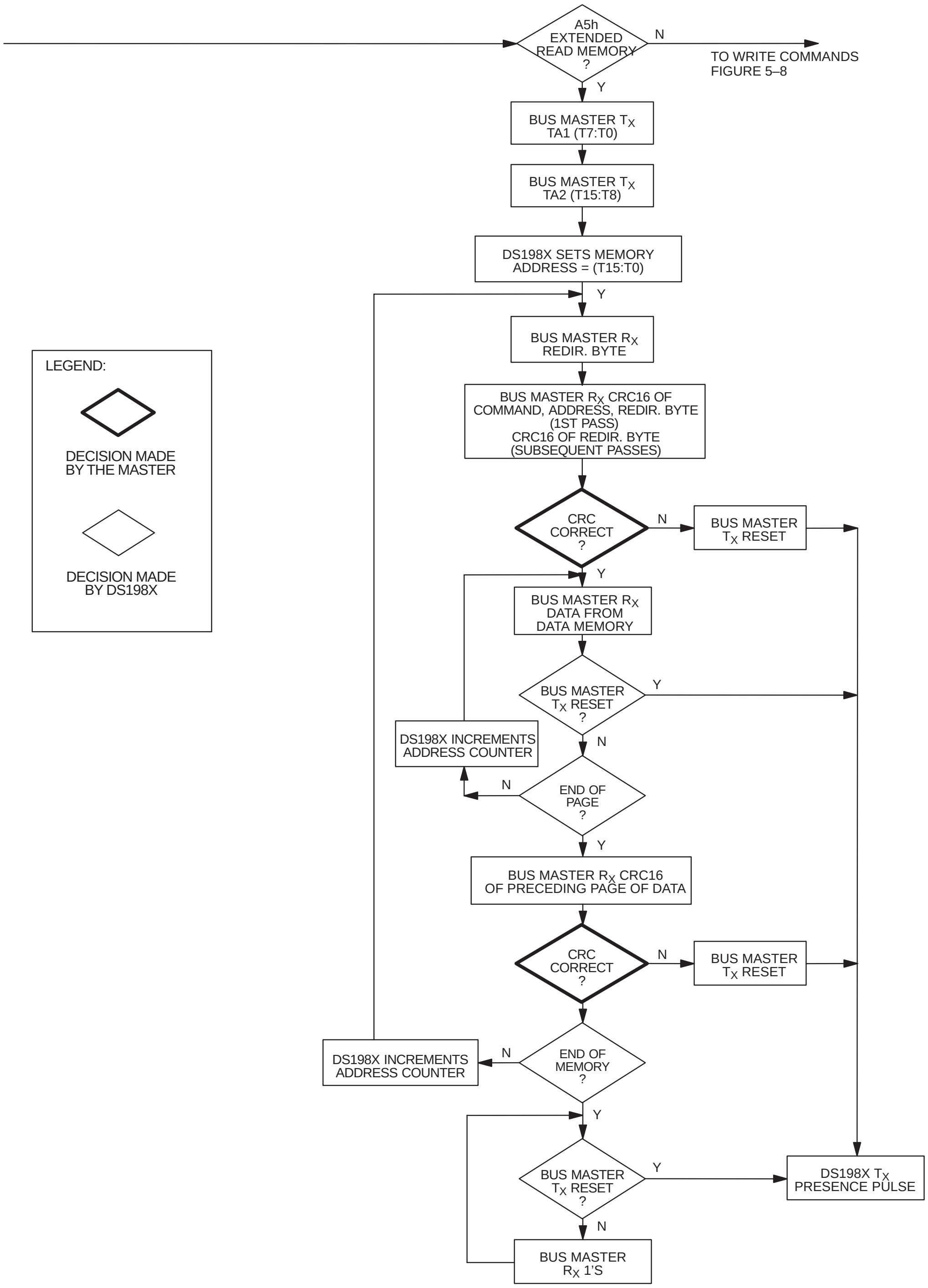
For all read commands the on–chip CRC generator is initially cleared to zero. With the Read Memory command, the CRC16 at the end of memory is generated by shifting in the command, address bytes TA1 and TA2 and all data bytes starting at the first addressed memory location.

For Read Status the calculation of the CRC at the end of a page depends on whether the end–of–page condition is encountered at the first pass or at a subsequent pass through the command flow without a Reset in between. For the first pass the CRC is generated by clearing the CRC accumulator, shifting in the command, address

MEMORY FUNCTION FLOW CHART EPROM READ COMMANDS Figure 5–7


MASTER T_X MEMORY FUNCTION COMMAND
F0h READ MEMORY ?
N
Y
AAh READ STATUS ?
N
Y
BUS MASTER T_X TA1 (T7:T0)
BUS MASTER T_X TA1 (T7:T0)
BUS MASTER T_X TA2 (T15:T8)
BUS MASTER T_X TA2 (T15:T8)
DS198X SETS MEMORY ADDRESS = (T15:T0)
DS198X SETS STATUS ADDRESS = (T15:T0)
Y
Y
BUS MASTER R_X DATA FROM DATA MEMORY
BUS MASTER R_X DATA FROM STATUS MEMORY
BUS MASTER T_X RESET ?
Y
Y
BUS MASTER T_X RESET ?
N
N
DS198X INCREMENTS ADDRESS COUNTER
END OF DATA MEMORY ?
N
END OF PAGE ?
N
DS198X INCREMENTS ADDRESS COUNTER
Y
Y
BUS MASTER R_X CRC16 OF COMMAND, ADDRESS, DATA (1ST PASS) CRC16 OF DATA (SUBSEQUENT PASSES)
BUS MASTER T_X RESET ?
Y
N
BUS MASTER R_X CRC16 OF COMMAND, ADDRESS, DATA
BUS MASTER T_X RESET
N
CRC CORRECT ?
DS198X CLEARS CRC GENERATOR
Y
END OF STATUS MEMORY ?
N
Y
BUS MASTER T_X RESET ?
Y
Y
BUS MASTER T_X RESET ?
N
N
BUS MASTER R_X 1'S
DS198X T_X PRESENCE PULSE
BUS MASTER R_X 1'S

MEMORY FUNCTION FLOW CHART EPROM READ COMMANDS Figure 5–7 (cont'd)


A5h EXTENDED READ MEMORY ?
N
TO WRITE COMMANDS FIGURE 5–8
Y
BUS MASTER T_X TA1 (T7:T0)
BUS MASTER T_X TA2 (T15:T8)
DS198X SETS MEMORY ADDRESS = (T15:T0)
Y
BUS MASTER R_X REDIR. BYTE
BUS MASTER R_X CRC16 OF COMMAND, ADDRESS, REDIR. BYTE (1ST PASS) CRC16 OF REDIR. BYTE (SUBSEQUENT PASSES)
CRC CORRECT ?
N
BUS MASTER T_X RESET
Y
BUS MASTER R_X DATA FROM DATA MEMORY
BUS MASTER T_X RESET ?
Y
N
DS198X INCREMENTS ADDRESS COUNTER
N
END OF PAGE ?
Y
BUS MASTER R_X CRC16 OF PRECEDING PAGE OF DATA
CRC CORRECT ?
N
BUS MASTER T_X RESET
DS198X INCREMENTS ADDRESS COUNTER
N
END OF MEMORY ?
Y
BUS MASTER T_X RESET ?
Y
DS198X T_X PRESENCE PULSE
N
BUS MASTER R_X 1'S
LEGEND:
DECISION MADE BY THE MASTER
DECISION MADE BY DS198X

bytes TA1 and TA2 and the data bytes starting at the first addressed location of the page. For subsequent passes the CRC generator is also cleared and then all data from the first to the last byte of the status memory page are shifted in.

D.2.d. Writing with Verification

Due to their different technology, writing or programming Add–Only iButtons differs significantly from writing NV RAM–based iButtons. Generally, the write procedure for Add–Only iButtons is much simpler, since it involves only one command. Figure 5–8 shows details.

After the master has sent the command code 0FH (WRITE MEMORY), the next data to follow is target address TA1, TA2 and one data byte. So far this is exactly the same as writing one byte to the scratchpad of an NV RAM iButton. With the next sixteen read data time slots, the master receives an inverted CRC16 of the command, target address and data byte. Then the master checks the CRC. If it is correct, the master sends a programming pulse to the bus. This pulse will program the addressed EPROM memory location using the data byte value loaded into the scratchpad. With the next eight read data time slots, the master will read back the byte that has just been programmed. By comparison of this byte with the data byte sent to the device, the master will decide if the programming was successful. If unsuccessful, the master can attempt to program this byte again by issuing a Reset Pulse, going through the ROM command level, and giving another WRITE MEMORY command along with TA1, TA2 and the desired data byte. If successful and the programmed byte was not the last byte of the data memory, the Add–Only iButton will advance its internal address counter, load the new address into its internal CRC generator, and wait for the next data byte. After the next data byte is received, the master will read back the CRC16 of the new address and data byte. Now another programming pulse may be applied, and so on.

The calculation of the CRC depends on whether the byte to be written is the first during the programming command flow or any subsequent byte without a Reset in between. For the first byte the CRC is generated by clearing the CRC accumulator, shifting in the command, address bytes TA1 and TA2 and the data byte itself. For subsequent bytes the CRC generator is loaded with the new (incremented) address and then the data byte is shifted in.

Programming status bytes is done in the same way as with data bytes. The only difference is the command code 55H. For calculation of the CRC the same algorithm as for programming data bytes applies.

Attempts to write to gaps within the status memory are ignored. If the uppermost memory location has just been programmed and the master continues communication with the device, the device will be in an idle state until it gets a reset pulse.

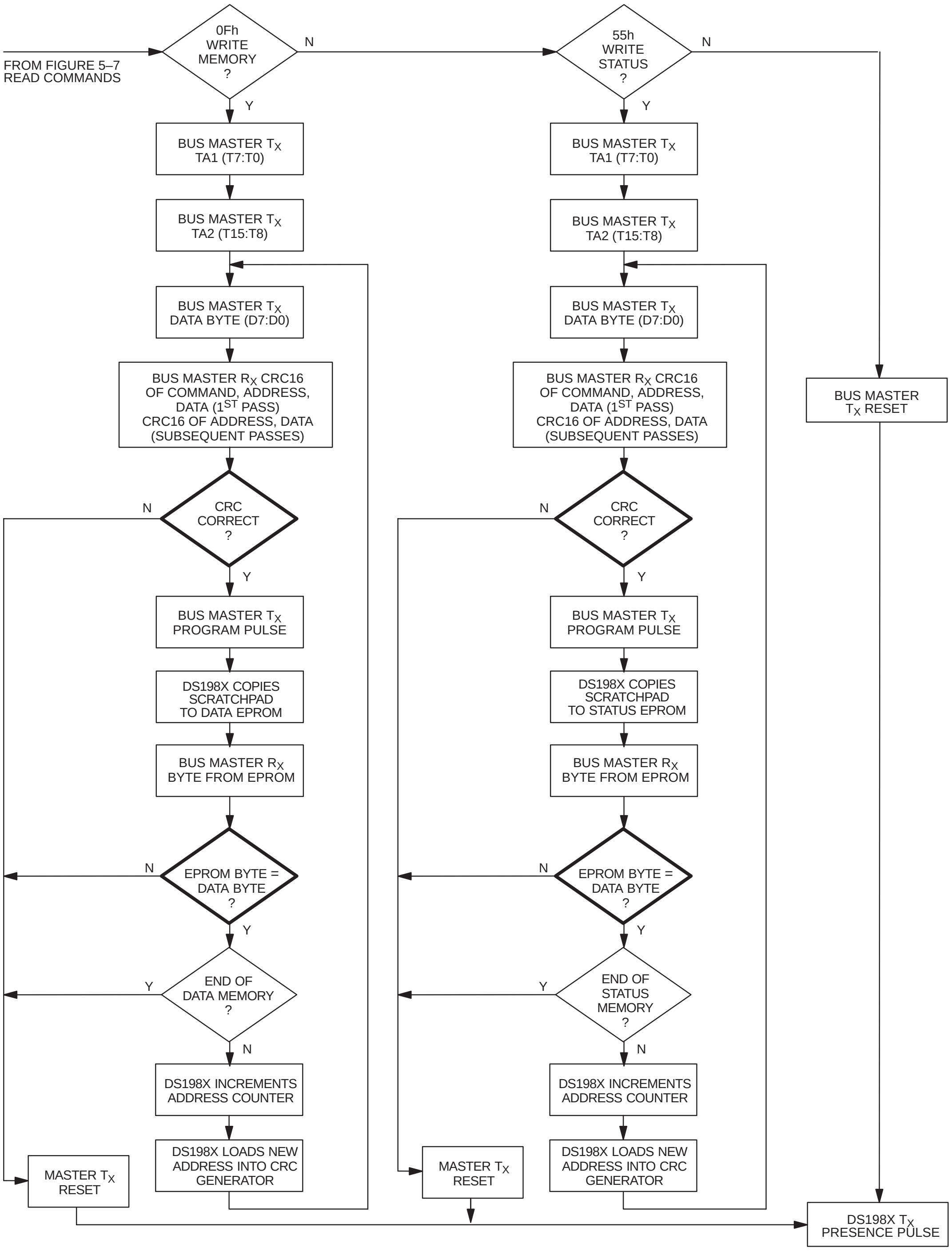
If the master sends a target address that is beyond the available address range, the device will set the uppermost address bits (i.e., those pointing out of the device) to zero to signal this condition by a non–matching CRC. The master must not send a programming pulse unless the CRC received from the device and the CRC calculated by the master match. Otherwise the address wraps around and an unwanted programming may occur.

If Add–Only iButtons are intended for use with the extended file structure, it is highly recommended not to format the device before writing application data. All data that the application requires to be stored in the device should be assembled in the master and then be written to the device as one process. This has the advantage that the directory pages need not be redirected and rewritten with every file that is initially copied to the device, saving memory pages for future use.

It is important to note that reading unprogrammed Add–Only Memories always produces FFH or 1's. Programming can only change the status of a bit cell from 1 to 0. Although the process of programming is byte–oriented, it is always possible to program individual bits from 1 to 0 (never from 0 to 1). If certain bits of a memory location need to be programmed to 0's, this can be done as follows: a) read the complete byte, b) set the desired bits to 0 with all other bits within the data byte set to 1, c) access the Add–Only iButton, d) send write command, target address, data byte, e) verify and send programming pulse. This algorithm is applicable for the status memory as well as for the data memory.

WARNING: The 12V programming pulse should not be applied if there are non–EPROM types of iButtons on the 1–Wire bus besides those specified by their family codes to tolerate 12 volts. Damage to the other devices may occur. The presence of other iButton types can be determined with the Search ROM command.

MEMORY FUNCTION FLOW CHART EPROM WRITE COMMANDS Figure 5–8


FROM FIGURE 5–7
READ COMMANDS
0Fh WRITE MEMORY ?
N
Y
55h WRITE STATUS ?
N
Y
BUS MASTER T_X TA1 (T7:T0)
BUS MASTER T_X TA2 (T15:T8)
BUS MASTER T_X DATA BYTE (D7:D0)
BUS MASTER R_X CRC16 OF COMMAND, ADDRESS, DATA (1ST PASS) CRC16 OF ADDRESS, DATA (SUBSEQUENT PASSES)
CRC CORRECT ?
N
Y
BUS MASTER T_X PROGRAM PULSE
DS198X COPIES SCRATCHPAD TO DATA EPROM
BUS MASTER R_X BYTE FROM EPROM
EPROM BYTE = DATA BYTE ?
N
Y
END OF DATA MEMORY ?
Y
N
DS198X INCREMENTS ADDRESS COUNTER
DS198X LOADS NEW ADDRESS INTO CRC GENERATOR
MASTER T_X RESET
BUS MASTER T_X TA1 (T7:T0)
BUS MASTER T_X TA2 (T15:T8)
BUS MASTER T_X DATA BYTE (D7:D0)
BUS MASTER R_X CRC16 OF COMMAND, ADDRESS, DATA (1ST PASS) CRC16 OF ADDRESS, DATA (SUBSEQUENT PASSES)
CRC CORRECT ?
N
Y
BUS MASTER T_X PROGRAM PULSE
DS198X COPIES SCRATCHPAD TO STATUS EPROM
BUS MASTER R_X BYTE FROM EPROM
EPROM BYTE = DATA BYTE ?
N
Y
END OF STATUS MEMORY ?
Y
N
DS198X INCREMENTS ADDRESS COUNTER
DS198X LOADS NEW ADDRESS INTO CRC GENERATOR
MASTER T_X RESET
BUS MASTER T_X RESET
DS198X T_X PRESENCE PULSE

III. Summary

The logical behavior of iButtons can be grouped into several communication levels or layers, such as Link, Network and Transport. Each level uses its own means to provide fault–tolerant data transfer. All iButtons are designed to operate as nodes in a 1–Wire, multi–drop network called MicroLAN. Special registers within NV RAM–based iButtons and a structured 3–step procedure using these registers guarantees data integrity when writing to iButtons. Data integrity during reads is checked by software means such as length byte indicating the number of valid bytes to follow and an inverted CRC16 double byte at the end of the data packet, which is usually the length of one page of 32 bytes or less. The CRC16 is initialized with the iButton page number to guarantee reading from the correct page. EPROM iButtons or Add–Only iButtons are read–compatible with NV RAM iButtons. The EPROM technology requires special programming that provides the same level of data integrity by means of an on–chip CRC generator. Add–Only iButtons also support the extended file structure. In addition, the on–chip CRC generator and a special read command make them well suited for byte applications, where the CRC information cannot be stored as part of the application data.

SPECIAL FUNCTIONS

CHAPTER 6: SPECIAL FUNCTIONS

I. Introduction

Chapter 5 discussed the logical behavior common to all iButton Memories. This chapter explains device–specific functions or extensions to the logical standards. Since the DS1991, as a password–secured memory, targets different applications than the other iButtons, its logical behavior is different. Therefore, it is detailed here. In addition to the features of other Memory iButtons, the DS1994 Memory Plus Time iButton has a real–time clock with alarm and interrupt facilities. Details about these extra features are also found in this chapter. The DS1982 1K bit Add–Only iButton also targets at different applications than the high capacity EPROM devices (byte storage rather than files). This requires special adaptations. Details of this and preliminary information on the Temperature iButton and the Addressable Switch are also given in this chapter.

II. ROM/NV RAM Devices

A. DS1990A Serial Number iButton

As explained in Chapter 2, the DS1990A includes only a 64–bit, laser–programmed ROM. It understands the commands Read ROM and Search ROM. Applying Match ROM or Skip ROM to the DS1990A will cause no further activity on the 1–Wire bus. This makes the DS1990A completely compatible with the Network layer.

To be used as drop–in replacement for the DS1990, the DS1990A also accepts the command code 0FH as a read command. The DS1990 used 0FH for the Read ROM command rather than the standard 33H and did not support the Search ROM command. For this reason only one DS1990 could be used on a 1–Wire bus along with other iButtons.

The DS1990A is intended as a simple device for applications where absolute identification is required. Since the DS1990A takes all of the required energy from the data line, it has a nearly unlimited lifetime. Further details about the logical behavior of the DS1990A are found in Chapter 5.

B. DS1991 MultiKey iButton

B.1. Overview

By design, the DS1991 does not supply data directly like other iButton Memories so that information can be kept secret. In addition to the ROM, the DS1991 incorporates a 64–byte NV RAM scratchpad and three password–protected memory pages, called Subkeys. This makes the DS1991 differ from other iButton Memories at the Transport layer and higher. Instead, the DS1991 has a hardware file system implemented in the device: each subkey can be used as an independent file, starting with a file name, followed by the password and 48 bytes of protected NV RAM. According to its intended application, the DS1991 supports a different command set at the Transport layer. Some of the commands are the same as with other Memory iButtons, but binary codes and addressing are different. The DS1991 supports the scratchpad commands Write Scratchpad, Read Scratchpad, Copy Scratchpad, the memory commands Read Subkey, Write Subkey and the command Write Password.

B.2. Scratchpad

The scratchpad can be used, as with other Memory iButtons, as intermediate storage before data is transferred to the final memory. Unlike other Memory iButtons, the DS1991 allows the user to select blocks of eight bytes or the complete scratchpad to be copied. Copying with the DS1991 means that the copied section of the scratchpad will be cleared to 0's after copying is finished.

B.3. Memory

Instead of memory pages, the DS1991 contains three subkeys. A subkey is very similar to a memory page. The first eight bytes of a subkey contain the identifier that can be used as the file name of the stored data. The identifier is available for public reading with the Read Subkey command. The next eight bytes store the password, which is write–only. The remaining 48 bytes of a subkey are the password–secured memory. These locations can only be accessed if the password is known. Unlike other Memory iButtons, the DS1991 allows direct writing to the subkey. The command Write Subkey has a relatively high potential of transferring data incorrectly if the electrical contact is poor. Therefore, it is recommended in touch applications to use the scratchpad as intermediate storage for verifying before the Copy Scratchpad command transfers the data.

Although the password and the identifier can be redefined using the scratchpad, there is a special command to write directly both password and identifier. This command, called Write Password, is essential to initialize a subkey, since the Copy Scratchpad command requires knowledge of the current password to write a new one. Directly writing the password of a subkey clears the identifier, the current password, and the contents of the protected memory to 0s. Copying the new password from the scratchpad does not clear the identifier or the protected memory.

B.4. Data Transfer

Flow charts of the commands implemented in the DS1991 at the Transport layer are shown in Figures 6–1 and 6–2. Before the DS1991 reaches the Transport layer, a reset and one ROM command have to be executed. Like the other iButtons, the DS1991 accepts only one data transfer command every time the Transport layer is reached.

The binary structure of DS1991 data transfer commands is different from other iButtons. Every DS1991 command consists of three bytes. The first of these bytes is the command itself, similar to other iButtons. The second byte provides addressing information. The two most significant bits contain the requested subkey number. Valid subkey numbers are 0, 1 and 2. Scratchpad read and write operations require the subkey number to be set to 3. The six lower bits of the address byte contain the start address where data has to be read from or written to. Valid addresses within subkeys are 10H to 3FH, since the first 16 bytes are reserved for the identifier and password. Within the scratchpad, all addresses between 00H and 3FH can be used. To write the password or copy the scratchpad, no byte address is required; the corresponding bits in the addressing byte must be 0. Copying data from the scratchpad to a subkey does not change the data addresses. Which of the eight data blocks of eight bytes each, or whether the entire scratchpad will be copied, is defined in the block selector code that must be provided together with the password of the desired subkey. The third byte of a DS1991 data transfer command is simply the 1's complement (bit inversion) of the address byte. Table 6–1 summarizes the command structure. Like other iButtons, the DS1991 requires commands and data to be sent starting at the least significant byte and the least significant bit.

The DS1991 has no user–readable registers for target address, byte offset and ending offset. Also, the Partial flag, Overflow flag and Authorization Accepted flag are missing.

Thus a different method is required to provide data integrity. If the scratchpad is used as intermediate storage for writing, reading it back will be sufficient to verify the integrity of the new data. After the data in the scratchpad is determined to be correct, a single command is issued to transfer the new data to the target subkey. In order to guarantee integrity of this transfer, one of nine different eight–byte codes is required, each one differing from the other by at least 32 bits. The subkey data format may include a preceding length byte and a following inverted CRC16 double byte to allow for easy checking of data integrity when reading.

Using Table 6–1, it is very easy to build up the complete binary code for each command. Figures 6–1 and 6–2 explain what has to be sent by the master and what the DS1991 will do. The commands Write Scratchpad and Read Scratchpad don't need further explanation. It is evident that data cannot be written or read outside of available address space. The Copy Scratchpad command, however, requires a Block Selector code to define which bytes must be copied to the selected subkey. Block Selector codes (Table 6–2) are 8–byte binary numbers. This highly redundant coding was chosen to avoid inadvertent modification of subkey data. The Block Selector code is sent starting at the least significant byte with the least significant bit. If several but not all blocks have to be copied, first the scratchpad can be filled with the new data and then the Copy Scratchpad command with the appropriate Block Selector code must be sent. Every copy cycle requires a Reset Pulse and a ROM command to be executed before the new Copy command can be issued by the master. Therefore, it can be more economical to load the complete scratchpad with data and to copy it in one stroke to the subkey, even if some of the blocks remain effectively unchanged.

Every access to the password–protected memory requires sending the 8–byte password. For the DS1991, the password is always eight bytes; every bit is significant.

All commands shown in Figure 6–2 include reading the subkey's identifier before the master has to send the password. As with the password, for the DS1991 the length of the identifier is always eight bytes. The DS1991 will count the time slots to decide if it has to listen or to answer. Therefore, it is important to always read all of the bits of the identifier.

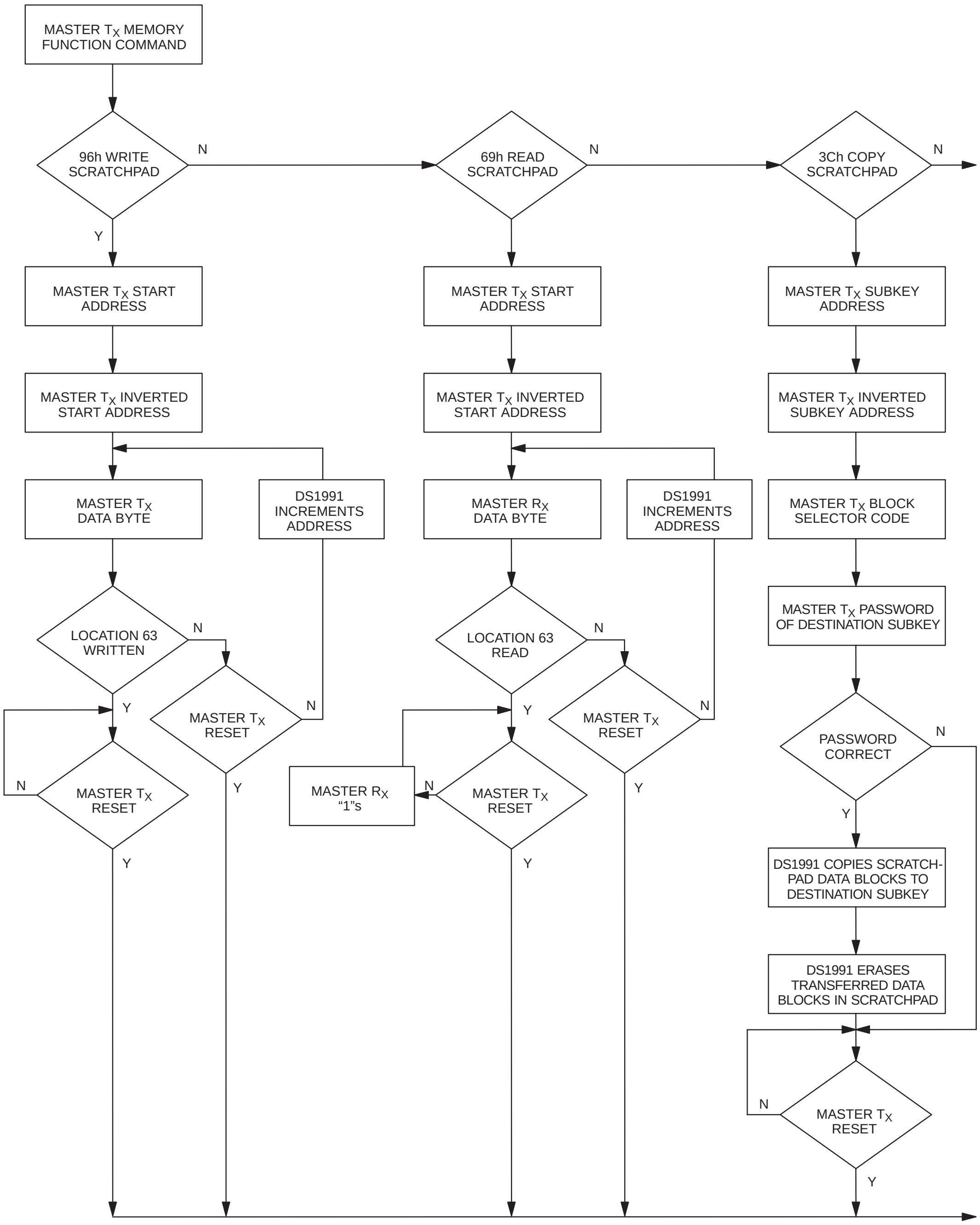
DS1991 COMMAND STRUCTURE Table 6–1

Command	1st byte	2nd byte								3rd byte
		B7	B6	B5	B4	B3	B2	B1	B0	
write scratchpad	96H	1	1	any value 00H to 3FH						ones complement of 2nd byte
read scratchpad	69H									
copy scratchpad	3CH	Sub–Key Nr.: 0 0 or 0 1 or 1 0		0	0	0	0	0	0	
read SubKey	66H			any value 10H to 3FH						
write SubKey	99H									
write password	5AH			0	0	0	0	0	0	

BLOCK SELECTOR CODES OF THE DS1991 Table 6–2

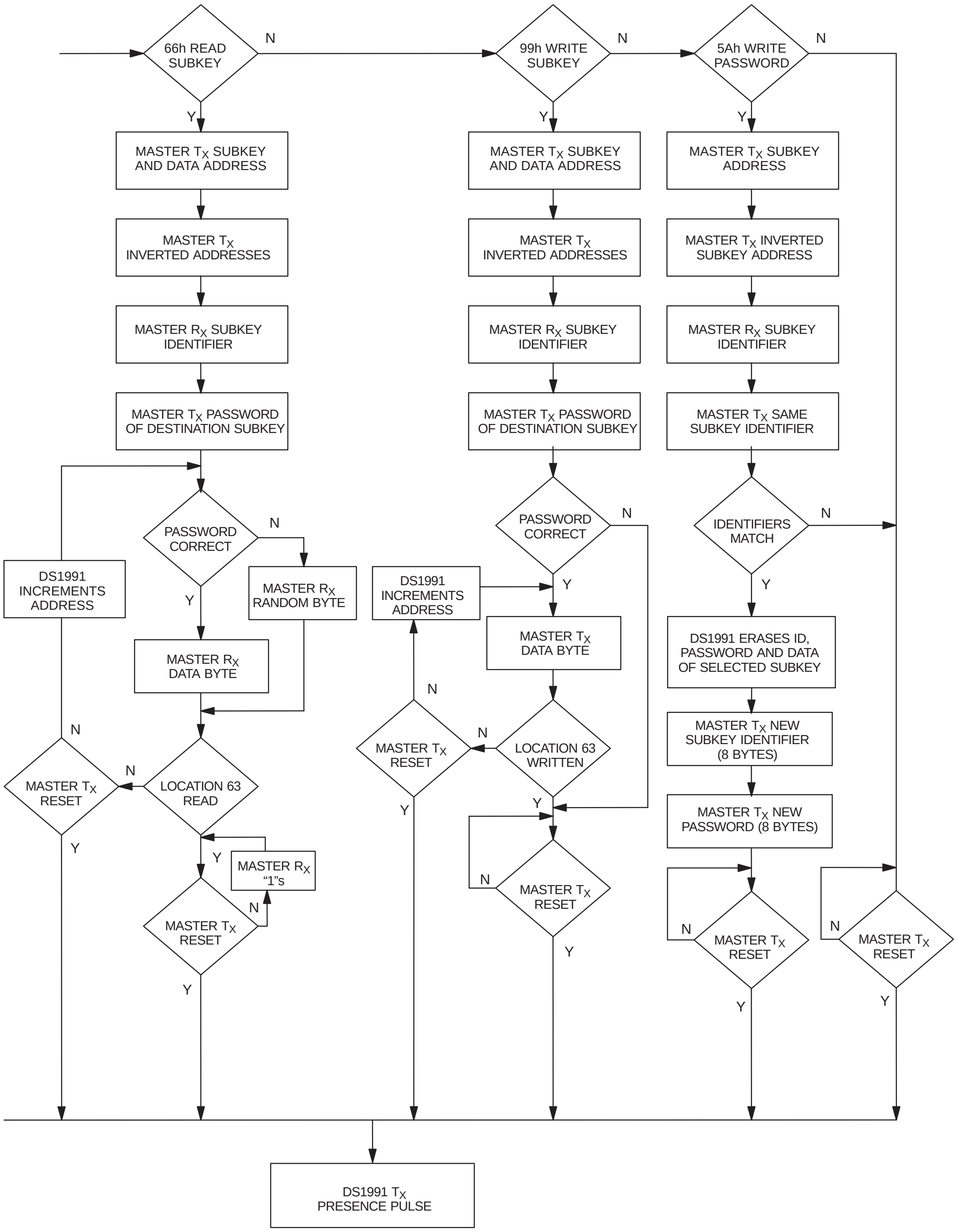
Block Nr.	Address Range	LS Byte			Codes				MS Byte
0 to 7	00 to 3FH	56	56	7F	51	57	5D	5A	7F
0	identifier	9A	9A	B3	9D	64	6E	69	4C
1	password	9A	9A	4C	62	9B	91	69	4C
2	10H to 17H	9A	65	B3	62	9B	6E	96	4C
3	18H to 1FH	6A	6A	43	6D	6B	61	66	43
4	20H to 27H	95	95	BC	92	94	9E	99	BC
5	28H to 2FH	65	9A	4C	9D	64	91	69	B3
6	30H to 37H	65	65	B3	9D	64	6E	96	B3
7	38H to 3FH	65	65	4C	62	9B	91	96	B3

MEMORY FUNCTIONS FLOW CHART Figure 6–1



MASTER T_X MEMORY FUNCTION COMMAND
96h WRITE SCRATCHPAD
N
69h READ SCRATCHPAD
N
3Ch COPY SCRATCHPAD
N
Y
MASTER T_X START ADDRESS
MASTER T_X START ADDRESS
MASTER T_X SUBKEY ADDRESS
MASTER T_X INVERTED START ADDRESS
MASTER T_X INVERTED START ADDRESS
MASTER T_X INVERTED SUBKEY ADDRESS
MASTER T_X DATA BYTE
DS1991 INCREMENTS ADDRESS
MASTER R_X DATA BYTE
DS1991 INCREMENTS ADDRESS
MASTER T_X BLOCK SELECTOR CODE
MASTER T_X PASSWORD OF DESTINATION SUBKEY
LOCATION 63 WRITTEN
N
Y
MASTER T_X RESET
N
Y
N
MASTER T_X RESET
Y
LOCATION 63 READ
N
Y
MASTER T_X RESET
N
Y
MASTER R_X "1"s
N
MASTER T_X RESET
Y
PASSWORD CORRECT
N
Y
DS1991 COPIES SCRATCH-PAD DATA BLOCKS TO DESTINATION SUBKEY
DS1991 ERASES TRANSFERRED DATA BLOCKS IN SCRATCHPAD
N
MASTER T_X RESET
Y

MEMORY FUNCTIONS FLOW CHART (Continued) Figure 6–2


66h READ SUBKEY
N
Y
99h WRITE SUBKEY
N
Y
5Ah WRITE PASSWORD
N
Y
MASTER T_X SUBKEY AND DATA ADDRESS
MASTER T_X SUBKEY AND DATA ADDRESS
MASTER T_X SUBKEY ADDRESS
MASTER T_X INVERTED ADDRESSES
MASTER T_X INVERTED ADDRESSES
MASTER T_X INVERTED SUBKEY ADDRESS
MASTER R_X SUBKEY IDENTIFIER
MASTER R_X SUBKEY IDENTIFIER
MASTER R_X SUBKEY IDENTIFIER
MASTER T_X PASSWORD OF DESTINATION SUBKEY
MASTER T_X PASSWORD OF DESTINATION SUBKEY
MASTER T_X SAME SUBKEY IDENTIFIER
PASSWORD CORRECT
N
Y
DS1991 INCREMENTS ADDRESS
MASTER R_X RANDOM BYTE
MASTER R_X DATA BYTE
LOCATION 63 READ
N
Y
MASTER T_X RESET
N
Y
MASTER R_X “1”s
MASTER T_X RESET
N
Y
PASSWORD CORRECT
N
Y
DS1991 INCREMENTS ADDRESS
MASTER T_X DATA BYTE
LOCATION 63 WRITTEN
N
Y
MASTER T_X RESET
N
Y
MASTER T_X RESET
N
Y
IDENTIFIERS MATCH
N
Y
DS1991 ERASES ID, PASSWORD AND DATA OF SELECTED SUBKEY
MASTER T_X NEW SUBKEY IDENTIFIER (8 BYTES)
MASTER T_X NEW PASSWORD (8 BYTES)
MASTER T_X RESET
N
Y
MASTER T_X RESET
N
Y
DS1991 T_X PRESENCE PULSE

B.5. Initialization

If new DS1991s need to be initialized, the password actually stored is always unknown. In this situation, only the command Write Password starts the devices' preparation for the application. Although the flow chart for this command looks very similar to Write Subkey, it differs in one very important point: the public readable identifier is compared to allow access, not the password. That is, instead of the unknown password, the master has to transmit exactly the subkey's identifier that it has just read from the device. Writing the password will automatically erase the data of the subkey. The command Write Password also writes a new identifier for the subkey. As soon as the new identifier (eight bytes) and the password (eight bytes) have safely arrived in the device, further changes of identifier and password should be copied from the scratchpad.

C. DS1992 Memory iButton: 1K Bit NV RAM

With a nonvolatile RAM capacity of four pages of 32 bytes each, the DS1992 is the smallest standard–feature Memory iButton. It reflects the complete standard implementation of the Network and the Transport layers, and therefore also supports the iButton file system. Refer to Chapter 5 for details about the logical behavior.

D. DS1993 Memory iButton: 4K Bit NV RAM

With a nonvolatile RAM capacity of 16 pages of 32 bytes each, the DS1993 is a typical standard–feature Memory iButton. Like the DS1992, it reflects the complete standard implementation of the Network layer and the Transport layer and supports the iButton file system.

E. DS1994 Memory iButton: 4K Bit NV RAM with Real Time Clock

E.1 Introduction

The DS1994 is a combination of all features of the DS1993 with the addition of a real–time clock, interval timer and cycle counter. Each of these counters can generate an alarm or interrupt. All additional registers are located in a separate memory page. Like the DS1992 and DS1993, the DS1994 reflects the complete standard implementation of the Network layer and the Transport layer. Excluding the memory page containing the timers and corresponding registers, the DS1994 also supports the iButton file system. In this section, only the extra features of the device will be discussed.

E.2. Register Map, Device Control, Device Status

The additional registers in the DS1994 that the DS1993 does not have are located in the highest page of the device, starting at address 200H. Figure 6–3 shows the address map of this page. The real–time clock occupies five bytes. Time representation is a completely binary number. Starting at a user–defined time point 0 (typically January 1st, 1970, 0:00:00 hours), the counter runs at a speed of one increment every 1/256 second. This time representation implies that the four most significant bytes of the Real–Time Clock register contain the number of seconds elapsed since the zero date. Minutes, hours, days, months, years and day of week are calculated via calendar software run on the master. This time representation, as it is also used with the UNIX operating system, allows country–specific daylight savings algorithms to be easily implemented by software. It also allows easy correction for a fast or slow clock and makes it easy to calculate differences between two dates/times. The five bytes used for the real–time clock are sufficient to code any time point within 136 years starting from the zero date.

ADDRESS MAP REGISTER PAGE OF DS1994, TIMEKEEPING AND SPECIAL FUNCTIONS

Figure 6–3

DEVICE STATUS REGISTER	200H		
DEVICE CONTROL REGISTER	201H		
REAL TIME COUNTER REGISTERS 5 BYTES	202H 206H	to	LSByte MSByte
INTERVAL TIME COUNTER REGISTERS 5 BYTES	207H 20BH	to	LSByte MSByte
CYCLE COUNTER REGISTERS 4 BYTES	20CH 20FH	to	LSByte MSByte
REAL TIME COUNTER ALARM REGISTERS 5 BYTES	210H 214H	to	LSByte MSByte
INTERVAL TIME ALARM REGISTERS 5 BYTES	215H 219H	to	LSByte MSByte
CYCLE COUNTER ALARM REGISTERS 4 BYTES	21AH 21DH	to	LSByte MSByte

Addresses 21EH and 21FH not available

The next counter is the interval timer. Like the real–time clock, it has a capacity of five bytes and a resolution of 1/256 second. Depending on device control configuration, this timer can automatically count the time that the DS1994 is connected to a pulled–up data bus; alternatively, it can be started and stopped under program control to measure time intervals.

The third counter is called cycle counter. It employs only four bytes. Every time the DS1994 is detached from the probe or the data line stays at a low level longer than required for communication and control, the cycle counter will increment. The interval time counter together with the cycle counter allow easy monitoring of active time and number of power on/off cycles if the DS1994 is built into a machine using, for example, a DS9098 MicroCan Retainer.

The memory locations following the counter registers act as alarm registers. They have the same lengths as the corresponding counters. If one of the counters matches the value of its alarm register, an alarm flag will be set . All three alarm flags are found in the Device Status register, address 200H (see Figure 6–4). The alarm flags for the Real Time Counter (RTF), Interval Timer (ITF) and Cycle Counter (CCF) are read–only. Reading the Device Status register will clear any of the alarm flags to 0.

The master has two ways to determine whether an alarm condition has occurred. Either it polls the Device Status register or it enables the timer/counter interrupts by setting the interrupt enable bits $\overline{\text{RTE}}$, $\overline{\text{ITE}}$ or $\overline{\text{CCE}}$ of the Device Status register to 0. How interrupts are signalled to the master and the consequences of using interrupts are explained later in this chapter.

The Device Control register, address 201H, defines in which mode the interval timer will operate and the minimum delay that the cycle counter and interval timer will need to recognize ”end of activity.” Figure 6–5 shows the bit assignments of all functions. The Delay SELect bit DSEL of the Device Control register allows the user to select between 3.5 ± 0.5 ms (bit cleared) and 123 ± 2 ms (bit set). If the DS1994 is firmly connected to the master and the interrupt facility is not used, the short time delay is sufficient. To avoid inadvertent recognition of end of activity in an application with bouncing probes or enabling DS1994 interrupts, the long value should be used. Otherwise, low times generated by the DS1994 itself, i.e., interrupts, could also be recognized as end of activity if the short delay is programmed.

DEVICE STATUS REGISTER OF DS1994 Figure 6–4

MSB							LSB
DON'T CARE	DON'T CARE	$\overline{\text{CCE}}$	$\overline{\text{ITE}}$	$\overline{\text{RTE}}$	CCF	ITF	RTF

Address = 200H; CCF, ITH, RTF read only

DEVICE CONTROL REGISTER OF DS1994 Figure 6–5

MSB							LSB
DSEL	STOP $\overline{\text{START}}$	AUTO $\overline{\text{MAN}}$	OSC	RO	WPC	WPI	WPR

Address = 201H

Bit 5 of the Device Control register, called AUTO/MAN, defines the mode of the interval timer. If this bit is set, the interval timer is in the automatic mode; the interval timer will be enabled by a high level on the data bus. If this bit is cleared to 0, the interval timer is in the manual mode. It will run only if the STOP/START bit, bit 6 of the Device Control register, is cleared to 0. The level of bit 6 is a don't care if bit 5 is set to 1 (automatic mode).

The purpose of bit 4 of the Device Control register is to control operation of the 32 KHz oscillator of the device. This bit must be set to 1 to enable time and interval counting. Only if the device is not in use should this bit be cleared to conserve the energy of the embedded lithium cell. When the DS1994 is shipped from the factory, the OSC bit is cleared.

The DS1994 is well suited to act as a programmable expiration controller. This function could be used to force a service call if a machine has been used for a predefined number of operating hours, a certain number of power on/off cycles, or if a certain date is reached. To avoid manipulation in such applications, all counters with alarm registers can be individually write protected after they have been loaded with the desired values. Each of the counters has its own write protect flag located in the least significant three bits of the Device Control register. The programmable expiration takes place when one or more write protect bits have been set and one corresponding alarm condition occurs. Bit 3 of the Device Control register, called RO or Read Only, defines the functionality of the DS1994 after the expiration has occurred. If this bit is set to 1, the device becomes read only. If it is cleared, only the ROM contents can be read after expiration. Before expiration the RO bit has no impact on operation. Setting one or more of the three write protect bits enables the device as expiration controller and prevents modification of the corresponding timer/counter and alarm registers, all other write protect bits and the RO bit. The OSC bit becomes write 1 only; i.e., the oscillator can still be started, but not stopped. If the cycle counter is write protected also, the Delay SELect bit DSEL cannot be changed anymore. Write protecting the interval timer also protects the AUTO/MAN bit and forces the STOP/START bit to be cleared to 0. In this case, the interval timer can either run permanently (manual mode) or it can run in the automatic mode, depending on the setting of the AUTO/MAN bit.

With exception of the write protect bits, all other bits of the Device Control register can be set like writing a byte to any other location using the scratchpad as intermediate storage. There is no danger of inadvertently setting a write protect bit. To set the write protect bits, first the scratchpad must be loaded with the desired new data of the device control register. Next, the scratchpad must be verified as usual. Then the Copy Scratchpad command must be performed three times. After the Copy Scratchpad command is performed the first time, the AA bit of the Transfer Status register E/S will be set. For the next two Copy Scratchpad commands, the AA bit must also be set; otherwise, the write protect bits still remain cleared. Each of the three Copy Scratchpad commands requires a full command cycle, including a Reset Pulse and ROM command. If it is desired to set more than one write protect bit, the bits must be set at the same time. Once a write protect bit is set, it cannot be undone, and the remaining write protect bits, if not set, cannot be set.

E.3. Interrupt Signalling and Processing

E.3.a. Alarm Versus Interrupt

If a DS1994 detects an alarm condition, it will automatically set the corresponding alarm flag in the Device Status register. The master does not get this information unless it reads the alarm flags. Doing this generates permanent communication on the 1–Wire line, puts a permanent load on the master, and consumes lithium energy from all addressed devices. If the electrical connection to the devices is poor, data from the Device Status Register may not be received correctly. The register page provides no CRC to detect transmission errors. Since reading the Device Status Register clears the alarm flags, a second reading will not give any help and information may be lost. Thus especially in a touch environment the alarm flags do not provide enough security in signalling alarm conditions.

The interrupt facility, however, makes sure that the master is always informed about alarms. Either the master is informed immediately or with the next Reset Pulse. Using the Conditional Search command (see section E.3.e of this chapter), it is possible to identify all interrupting devices without losing any status information. Bad contacts do not disturb the identification process, since the interrupt is signalled as long as the Device Status register is not read. Even if while reading the Device Status register the contact should be poor, the master still knows the device identity. Re–reading the Register Page allows reconstruction of the type of alarm by comparison of counter registers with alarm registers.

E.3.b. Interrupt Types

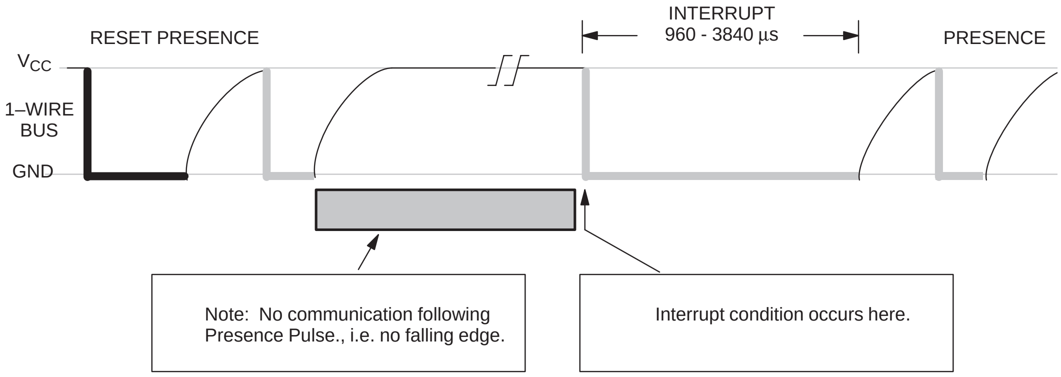
The DS1994 recognizes two types of interrupts: spontaneous interrupts, called type 1, and delayed interrupts, type 2. A spontaneous interrupt is generated as soon as the alarm condition occurs. This requires that the master allows interrupts to be generated from the 1–Wire bus. If the application does not allow this, delayed interrupts are still superior to alarms. No matter when the alarm condition occurs, the information about it will never inadvertently get lost. The master simply has to check the status of the data bus after it has released it from a Reset Pulse. This makes interrupts an important feature to secure alarm signalling in typical touch applications.

E.3.c. Interrupt Signaling

Spontaneous interrupts need to be armed by a Reset Pulse after the master has finished the actual communication on the 1–Wire line. They require that the data line is pulled high and that there is no other activity on the data line. A single falling slope or detaching the device from the 1–Wire line will disarm this type of interrupt. If an alarm condition occurs while the device is disarmed, the interrupt will be signaled at first as a type 2 (regular or special case) interrupt. Spontaneous interrupts are signaled by the DS1994 by pulling the data line low for 960 to 3840 µs. After this long low pulse, a Presence Pulse will follow as if the device had received an ordinary Reset Pulse from the master. This waveform (Figure 6–6) occurs only once with every spontaneous interrupt. If the alarm condition occurs just after the master has sent a Reset Pulse, i.e., during the high or low time of the presence pulse, the DS1994 will not assert its Interrupt Pulse until the Presence Pulse is finished (see Figure 6–7).

If the DS1994 has no chance to directly signal an interrupt, either because the data line was not pulled high, communication was in progress, or the interrupt was not armed, it will extend the next Reset Pulse sent by the master to a total length of 960 to 3840 µs. If the alarm condition occurs during the reset low time of the Reset Pulse, the DS1994 will immediately assert its interrupt pulse; thus the total low time of the pulse can be extended up to 4800 µs (see Figure 6–8). If a DS1994 with a not previously signaled alarm is attached to a probe, it will as usual send a Presence Pulse and wait for the Reset Pulse sent by the master to extend it and to subsequently issue a Presence Pulse (see Figure 6–9). Extended Reset Pulses do not confuse other iButtons since there is no true maximum time limit for the reset low time. The only limitation exists with the DS1994; to detect interrupts, the reset low time must be limited to a maximum of 960 µs; otherwise, it could mask or conceal Interrupt Pulses.

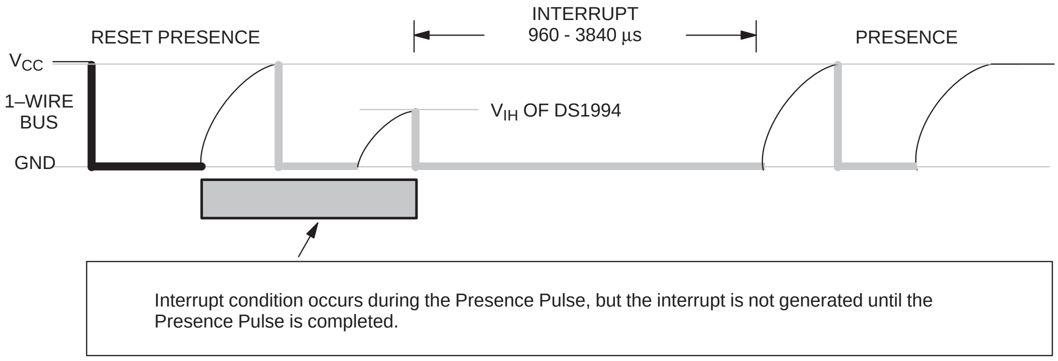
TYPE 1 INTERRUPT Figure 6–6



Note: No communication following Presence Pulse., i.e. no falling edge.

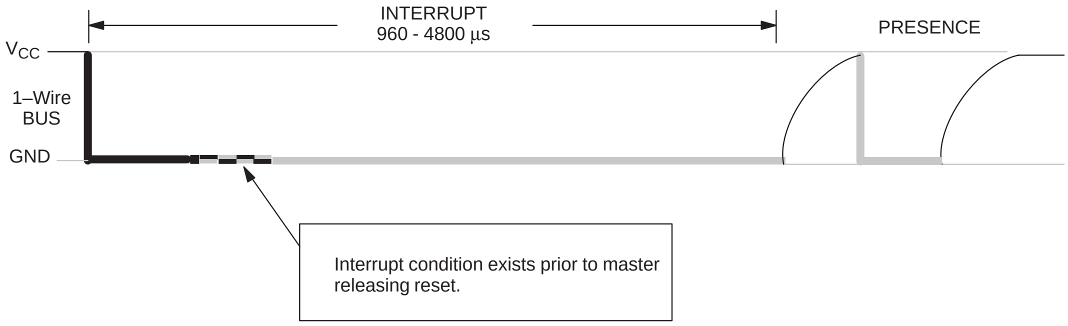
Interrupt condition occurs here.

TYPE 1A INTERRUPT (SPECIAL CASE) Figure 6–7



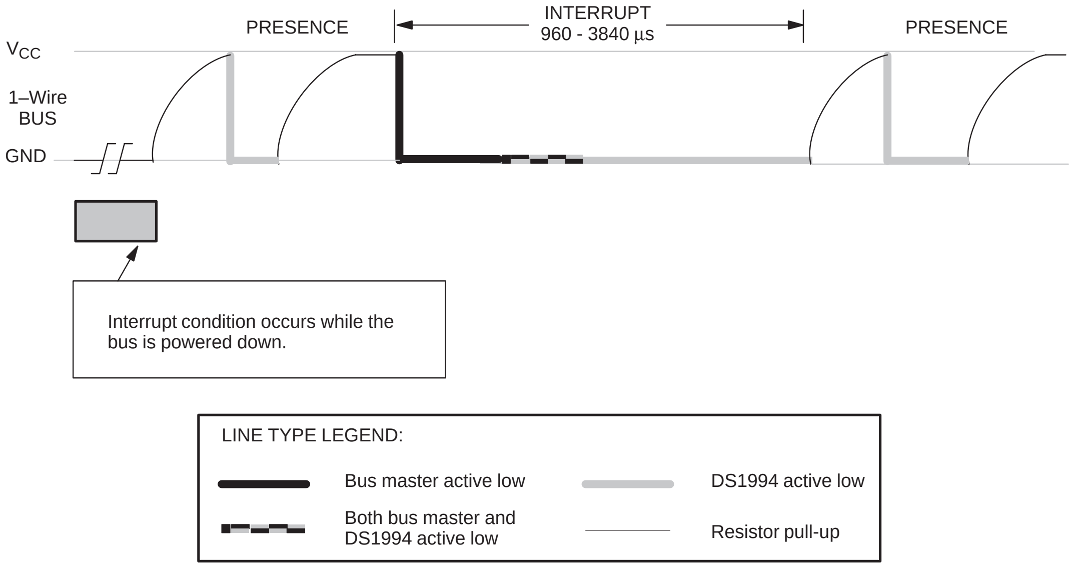
Interrupt condition occurs during the Presence Pulse, but the interrupt is not generated until the Presence Pulse is completed.

TYPE 2 INTERRUPT Figure 6–8



Interrupt condition exists prior to master releasing reset.

TYPE 2 INTERRUPT (SPECIAL CASE) Figure 6–9



Interrupt condition occurs while the bus is powered down.

The interrupt signaling discussed so far is valid for the first opportunity the device has to signal an interrupt. It is not required for the master to acknowledge an interrupt immediately. If an interrupt is not acknowledged the DS1994 will continue signaling the interrupt with every Reset Pulse. To do so, DS1994 devices of Revision B will always use the waveform of the Type 2 Interrupt (Figure 6–8). Devices of Revision C will either use the waveform of the Type 2 Interrupt (Figure 6–8) or the waveform of the Type 1A Interrupt (Figure 6–7). The waveform of the Type 2 Interrupt will be observed after a communication to a device other than the interrupting one; after successful communication to the interrupting device (without acknowledging the interrupt) the waveform of the Type 1A Interrupt will be found. If the revision of a particular DS1994 is not known, it can be determined by the waveform used for interrupt signaling or by the information branded on the lid of the MicroCan. The field RR of Figure 3–2, just above the family code, will read Bx for Revision B and Cx for Revision C. The character ”x” represents a 1–digit number that is not related to the chip inside.

E.3.d. Interrupt Acknowledge

As long as an interrupt has not been acknowledged by the master, the DS1994 will continue sending interrupt pulses. This has little impact on the communication in a network environment if the master waits for the data line to be high before a command is sent out, but it will slow down the communication. Therefore, the master should acknowledge interrupts as soon as possible. This is done by reading the Device Status Register, address 200H, which automatically clears all alarm flags. The three least significant bits tell the master which of the possible alarms has occurred. Comparing the contents of the corresponding alarm registers with the actual counter registers' contents gives information about when the alarm condition occurred.

E.3.e. Conditional Search

In a multi–drop environment with many and different types of iButtons on the same data bus, it is time–consuming to find the devices that signal the interrupt condition. The Search ROM command would reveal all device identifiers. Searching for a certain family code would reduce the number of search cycles. But in general, this method can be tedious. Therefore, the DS1994 supports a special command called Conditional Search, code ECH, with the non–acknowledged interrupt being the condition. This command works exactly as the normal Search ROM, but it will identify only devices with interrupts that have not yet been acknowledged. The conditional search algorithm may reveal parts other than DS1994 which respond to this command. They will be identified with different family codes and can be disregarded.

As soon as an interrupting device is identified, the master should read the Device Status register to acknowledge the interrupt. Depending on the interrupt source, it might be necessary to continue reading the other registers of the Register Page to get information required by the interrupt service routine. If several devices are signalling an interrupt, the master will issue further Conditional Search commands to find the other devices until all of them are serviced. In an environment where the DS1994 operates as an expiration controller (write protect bits set) and interrupts are also enabled, the read–only flag RO in the Device Control register should be set if it is necessary to clear the interrupt that occurs with the expiration.

F. DS1995 Memory iButton: 16K Bit NV RAM

The DS1995 reflects the complete standard implementation of the Network and Transport layers and supports the iButton file system. It can be regarded as a larger version of the DS1993. The memory is organized as 64 pages of 32 bytes. All software development for the DS1995 can be done using the DS1993 as a test vehicle. For more details on the DS1995, refer to the data sheet.

G. DS1996 Memory iButton: 64K Bit NV RAM

The memory of the DS1996 is organized as 256 pages of 32 bytes. The DS1996 incorporates all iButton standards. For more details on the DS1996, refer to the data sheet.

III. Add–Only iButtons

A. DS1982 Add–Only iButton: 1K–Bit OTP EPROM

On the Network Layer the DS1982 is completely in compliance with other iButtons. Due to its different application area (storage of bytes rather than files) some modifications are required. The main differences between DS1982 and DS1985/6 are the 8–bit CRC generator instead of a 16–bit version. This 8–bit CRC is calculated with the same algorithm as is used for the ROM–section of iButtons and is also output in the true form (not inverted). Instead of the Extended Read command the DS1982 supports a Read Data and Generate CRC command (code C3H) that omits the redirection byte. The flowcharts on Figures 6–10 and 6–11 show details.

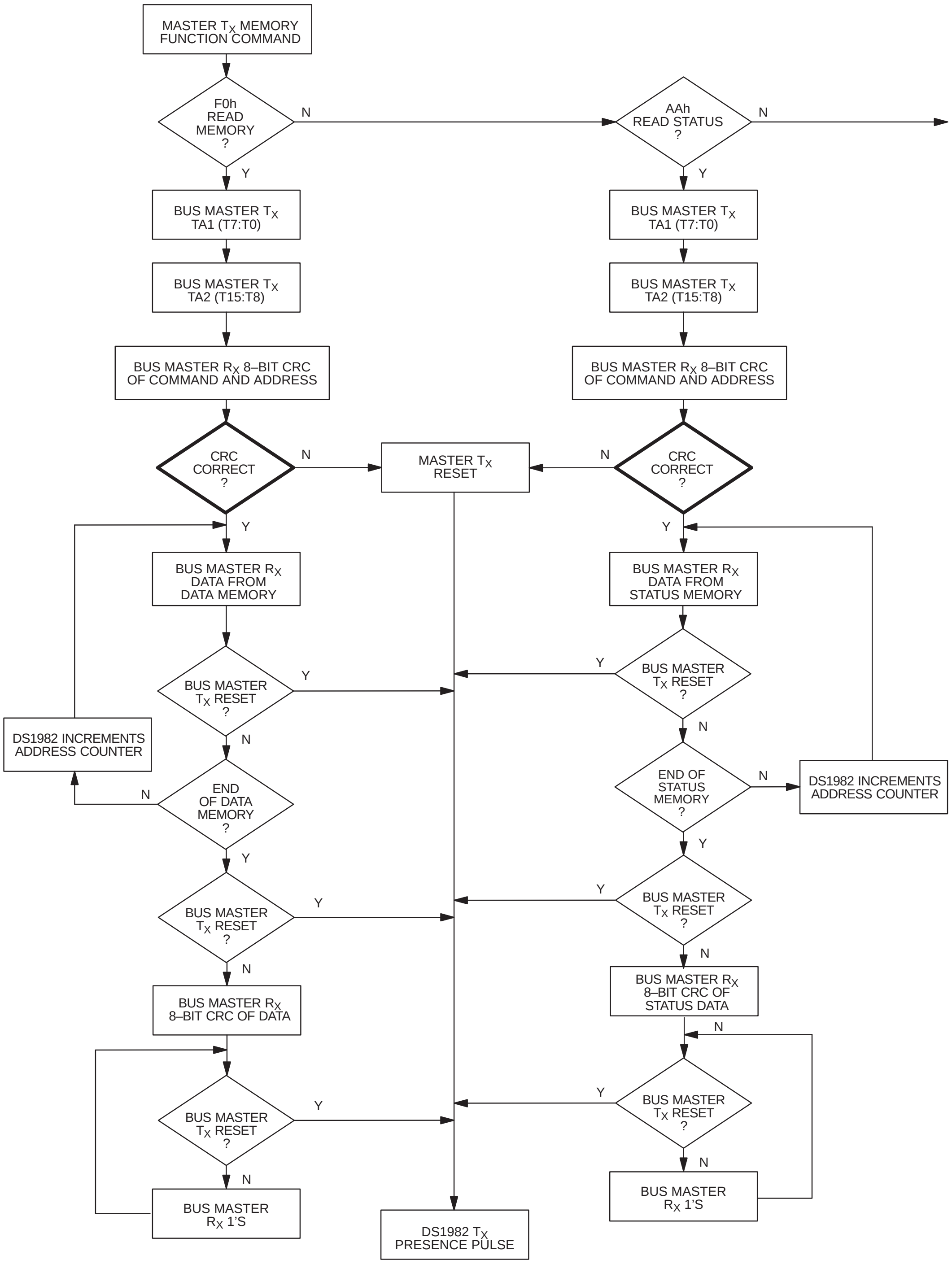
For all read commands, a CRC is generated and transmitted by the device before the first byte from either data or status memory is received by the master. This CRC is calculated by clearing the CRC–accumulator to 0 and then shifting in the command followed by the address information TA1 and TA2. This allows the master to check the correct transmission of the address before it continues reading data from memory. For the standard Read Memory command (code F0H) the device provides another CRC that is transmitted after the last byte of data has been read. This CRC is calculated starting with a cleared CRC accumulator and shifting in every byte beginning at the first addressed memory location. In the same way the CRC at the end of the status memory is calculated. With the Read Data & Generate CRC command the device also sends a CRC at the end of each data page. For this CRC the CRC accumulator is also cleared and then every byte beginning at the first addressed location of the page is shifted in.

For writing to either the data or status memory the CRC is calculated differently. For the first byte to be written the CRC calculation begins with a cleared CRC accumulator; then the command byte, address TA1 and TA2 and the data byte are shifted in. As long as the write command is not finished by a Reset Pulse, for every subsequent byte to be written the CRC accumulator is loaded with the least significant byte of the incremented address and then the data byte sent by the master is shifted in.

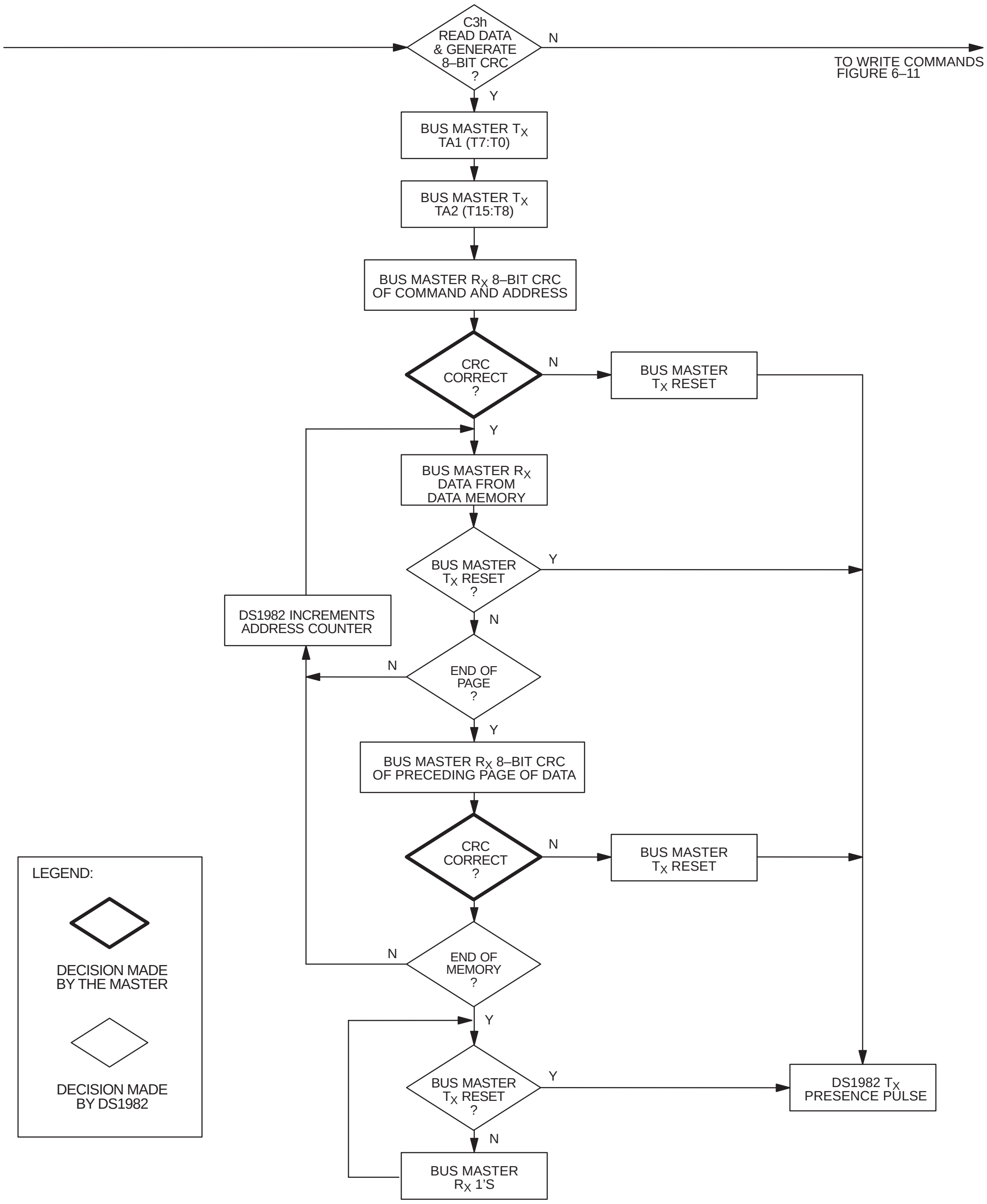
Since this device has a capacity of only 4 pages of 32 bytes it is not efficient to keep the standard memory map of status memory for this device. Instead, the status memory of the DS1982 is reduced to just one page of 8 bytes (Figure 6–12). In analogy to the standard, the lowest byte of the status memory contains the write protect bits for the memory. The lowest four bits are used to hardware–protect the memory pages. Since there are no write protect bits for the redirection bytes, the upper four bits of the write protect byte have no function. Depending on the application, the bitmap of used pages may be stored in these bits. Byte addresses 1 to 4 contain the redirection bytes, one for each page of memory. Of the remaining 3 bytes within the status memory page, the highest one is factory–programmed to 00H. The other two bytes are not dedicated to a special purpose.

The DS1982 with its capacity of 1K bit targets byte–oriented applications where the serial number is required for identification and up to 128 memory bytes are useful to store additional information. With its ability to store data according to the extended file structure, the DS1982 is a one–time–programmable counterpart of the DS1992. Page redirection is handled differently, however, and it deviates from the iButton Standards on the Transport layer and in the status memory map.

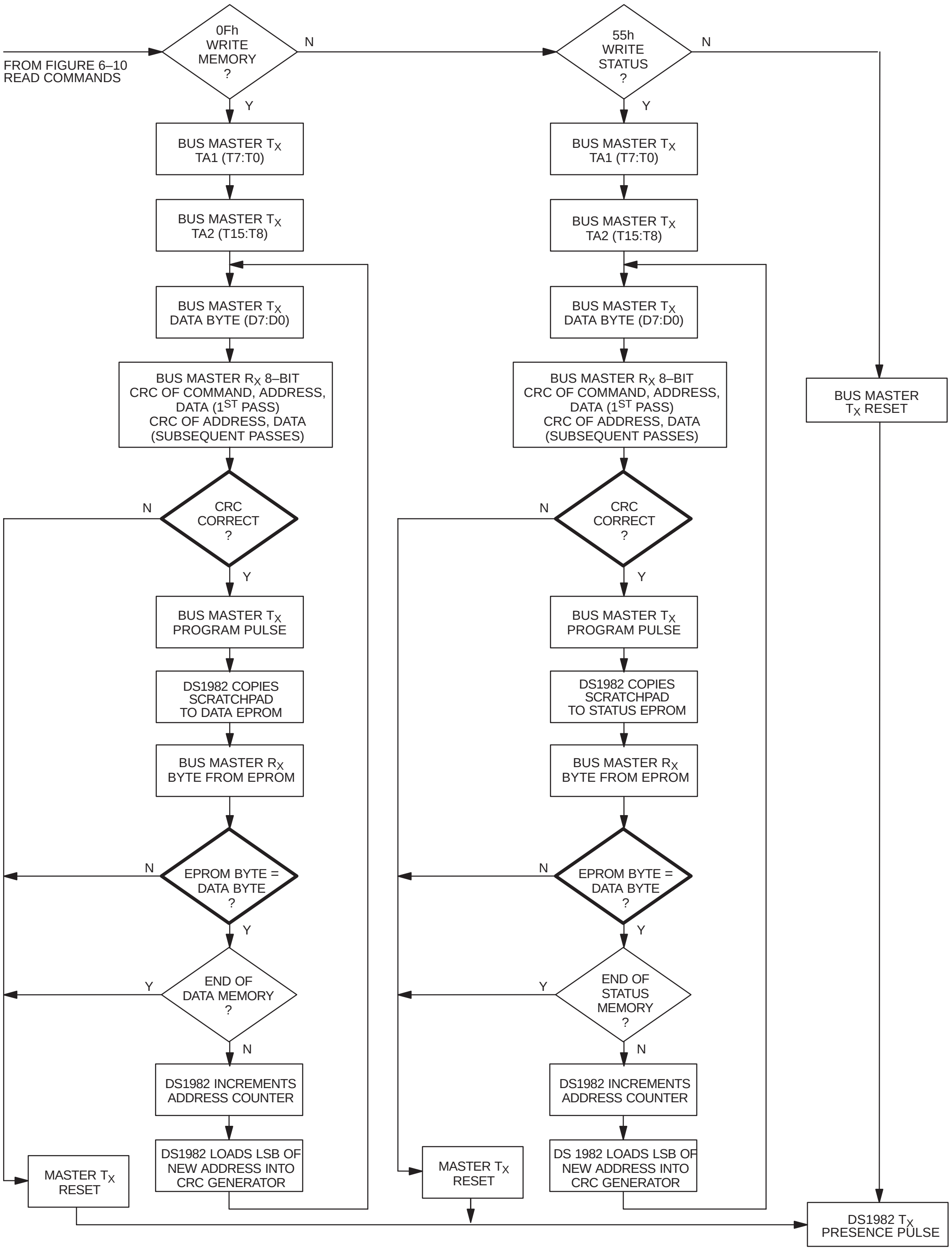
MEMORY FUNCTION FLOW CHART DS1982 READ COMMANDS Figure 6–10


MASTER TX MEMORY FUNCTION COMMAND
F0h READ MEMORY ?
N
Y
AAh READ STATUS ?
N
Y
BUS MASTER TX TA1 (T7:T0)
BUS MASTER TX TA2 (T15:T8)
BUS MASTER RX 8–BIT CRC OF COMMAND AND ADDRESS
BUS MASTER TX TA1 (T7:T0)
BUS MASTER TX TA2 (T15:T8)
BUS MASTER RX 8–BIT CRC OF COMMAND AND ADDRESS
CRC CORRECT ?
N
MASTER TX RESET
N
CRC CORRECT ?
Y
Y
BUS MASTER RX DATA FROM DATA MEMORY
BUS MASTER RX DATA FROM STATUS MEMORY
BUS MASTER TX RESET ?
Y
Y
BUS MASTER TX RESET ?
N
DS1982 INCREMENTS ADDRESS COUNTER
N
END OF DATA MEMORY ?
N
END OF STATUS MEMORY ?
N
DS1982 INCREMENTS ADDRESS COUNTER
Y
Y
BUS MASTER TX RESET ?
Y
Y
BUS MASTER TX RESET ?
N
N
BUS MASTER RX 8–BIT CRC OF DATA
BUS MASTER RX 8–BIT CRC OF STATUS DATA
N
BUS MASTER TX RESET ?
Y
Y
BUS MASTER TX RESET ?
N
N
BUS MASTER RX 1'S
BUS MASTER RX 1'S
DS1982 TX PRESENCE PULSE

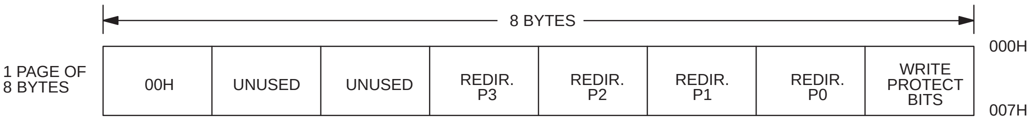
MEMORY FUNCTION FLOW CHART DS1982 READ COMMANDS Figure 6–10 (cont'd)


C3h READ DATA & GENERATE 8–BIT CRC ?
N
TO WRITE COMMANDS FIGURE 6–11
Y
BUS MASTER T_X TA1 (T7:T0)
BUS MASTER T_X TA2 (T15:T8)
BUS MASTER R_X 8–BIT CRC OF COMMAND AND ADDRESS
CRC CORRECT ?
N
BUS MASTER T_X RESET
Y
BUS MASTER R_X DATA FROM DATA MEMORY
BUS MASTER T_X RESET ?
Y
N
DS1982 INCREMENTS ADDRESS COUNTER
END OF PAGE ?
N
Y
BUS MASTER R_X 8–BIT CRC OF PRECEDING PAGE OF DATA
CRC CORRECT ?
N
BUS MASTER T_X RESET
LEGEND:
DECISION MADE BY THE MASTER
DECISION MADE BY DS1982
END OF MEMORY ?
N
Y
BUS MASTER T_X RESET ?
Y
DS1982 T_X PRESENCE PULSE
N
BUS MASTER R_X 1'S

MEMORY FUNCTION FLOW CHART DS1982 WRITE COMMANDS Figure 6–11


FROM FIGURE 6–10
READ COMMANDS
0Fh WRITE MEMORY ?
N
Y
55h WRITE STATUS ?
N
Y
BUS MASTER T_X TA1 (T7:T0)
BUS MASTER T_X TA2 (T15:T8)
BUS MASTER T_X DATA BYTE (D7:D0)
BUS MASTER R_X 8–BIT CRC OF COMMAND, ADDRESS, DATA (1ST PASS) CRC OF ADDRESS, DATA (SUBSEQUENT PASSES)
BUS MASTER T_X RESET
CRC CORRECT ?
BUS MASTER T_X PROGRAM PULSE
DS1982 COPIES SCRATCHPAD TO DATA EPROM
DS1982 COPIES SCRATCHPAD TO STATUS EPROM
BUS MASTER R_X BYTE FROM EPROM
EPROM BYTE = DATA BYTE ?
END OF DATA MEMORY ?
END OF STATUS MEMORY ?
DS1982 INCREMENTS ADDRESS COUNTER
DS1982 LOADS LSB OF NEW ADDRESS INTO CRC GENERATOR
DS 1982 LOADS LSB OF NEW ADDRESS INTO CRC GENERATOR
MASTER T_X RESET
DS1982 T_X PRESENCE PULSE

STATUS MEMORY MAP OF DS1982 1K BIT ADD–ONLY iButton Figure 6–12



B. DS1985 Add–Only iButton: 16K–Bit OTP EPROM

The DS1985 is the first Add–Only iButton with the full implementation of the Network and Transport layers that supports the EPROM–adapted iButton file system. The memory of the DS1985 is organized as 64 pages of 32 bytes. For details on the logical behavior, refer to Chapter 5. For further information on the DS1985, refer to the data sheet.

C. DS1986 Add–Only iButton: 64K–Bit OTP EPROM

The memory of the DS1986 is organized as 256 pages of 32 bytes. The DS1986 shows the same logical behavior as the DS1985 except for the memory capacity, which is four times greater. For more details on the DS1986, refer to the data sheet.

IV. Other MicroCan Products

This section is dedicated to non–memory products packaged in MicroCans. These products, like Memory iButtons, communicate in time slots and use the standard ROM commands of the Network layer for addressing. Everything else is device–specific. An adapted command set on the Transport layer supports the special features.

DS1920 Temperature iButton

The Temperature iButton is a very accurate and easy–to–operate temperature sensor. It can be read like a Memory iButton or wired as a MicroLAN to read the temperature of many locations remotely. As with the DS1990A and all Add–Only iButtons, the Temperature iButton does not contain an internal energy source. Energy is needed only to do temperature conversion or to write to its nonvolatile EEPROM memory locations. For this reason, a strong pull–up is required just after the command for temperature conversion has been sent by the master or when data is being written to the EEPROM cells.

The memory map of the DS1920 is comprised of an 8–byte scratchpad with embedded registers for temperature reading in the first two bytes, EEPROM backed–up temperature alarm registers (TH, TL or user bytes) in the next two bytes, two dummy bytes and two registers providing data for temperature interpolation.

All temperature reading and setting of the alarm thresholds is done through the scratchpad. With a memory map of 8 bytes, a random access addressing mode is not implemented. Reading the scratchpad always starts with the temperature register and ends with the interpolation register. For data integrity check the DS1920 transmits an 8–bit CRC (”ROM–type”) over the entire scratchpad after the last byte of the scratchpad has been read.

The data representation within the DS1920 is very straightforward. Temperature reading is the two’s complement of the temperature in degrees Celsius. The temperature increments are 0.5°C.

In addition to the four standard ROM commands, the DS1920 supports the Conditional Search ROM, command code ECH. A DS1920 will respond to the conditional search if the temperature alarm registers TH and TL are loaded with valid temperature values (upper and lower thresholds) and the latest temperature conversion has revealed a value outside the interval TH to TL. Since the temperature alarm registers are one byte each and the most significant bit represents the sign, the resolution of the temperature alarm is reduced to ±1°C.

The DS1920 is specified for operation over the temperature range –55°C to +100°C. Within the range of 0°C to +70°C, the accuracy of temperature reading is ±1 LSB or ±0.5°C. In the range of +70°C to +85°C, the accuracy decreases to ±1°C. In the remaining ranges the accuracy is typically better than ±2°C.

V. Solder–Mount Products

In this section devices are discussed that require more than one connection and therefore do not fit into a MicroCan. These devices support the communication standards and networking commands of iButtons. On the Transport layer, the devices may deviate from iButtons.

A. DS2407 Addressable Switch

The DS2407 (formerly referenced as DS2405A) is an enhanced version of the DS2405. It works in all applications of the DS2405 but is easier to use, more flexible and much faster in achieving specific operations. Furthermore, it is completely in compliance with iButton standards up to the Network layer and partly in compliance on the Transport Layer.

The DS2407 comprises two PIO–channels (”switches”) A and B that can be controlled or sensed (either directly or by use of Conditional Search ROM) through the 1–Wire bus. In addition, the DS2407 provides 1K–bit of EPROM data memory and 8 bytes of Status Memory (7 bytes EPROM, 1 byte SRAM) similar to the DS1982. Embedded in the Status Memory are user–programmable power–on conditions for the PIO–channels, Conditional Search settings (address 6), and SRAM cells to override the power–on default values (address 7). The memory function command set of the DS2407 is identical to the DS1985 plus a new command for channel access.

The channel access command selects one or both channels, reads from or writes to the selected channel(s), or toggles between read and write without having to reset the device. When reading, a 16–bit CRC generator can add CRC values to safeguard the data stream, if desired. Before data is written to or read from a PIO–channel, the DS2407 transmits a channel info byte indicating the status of each channel (i. e., if the pull down transistor is conducting or not), the logical level sensed at each channel and the status of each channel's activity latch, a flip flop that is set whenever a change of the logical level at a PIO occurs. Channel A is designed for high sink currents (minimum 40 mA) and voltages of 12V; channel B can handle 6V and sink at least 6 mA. There are several more unique features of the DS2407 that, due to space limitations, cannot be discussed here. Please refer to the DS2407 data sheet.

The DS2407 will be available in a 6–pin C–lead surface mount package as well as in a TO–92 package for through mount applications. With the 3–lead TO–92 package, however, channel B will not be accessible.

B. DS2404S–C01 Dual Port Memory Plus Time

This product is a special version of the chip that is inside the DS1994 Memory Plus Time iButton. Initially, this device was developed as a custom part for the iButton Recorder, but has been authorized for public availability. To be distinguished from the DS1994, the DS2404S–C01 has the family code 84H. In addition to this, the 12 most significant bit of the serialization field are coded 001H, leaving 28 bits for serialization. The communication with the DS2404S–C01 through the 1–Wire port is identical to the DS1994; all functions of the DS1994 are available.

The second port of the DS2404S–C01 is a 3–Wire serial interface providing the signals Data, Clock and $\overline{\text{Reset}}$ for communication speeds up to 2 Mbits/s. The 3–Wire interface directly accesses the scratchpad, memory locations and special registers without requiring a ROM command to address the device. For the 3–Wire interface the same command codes and transaction flowcharts apply as for the 1–Wire interface; the lasered ROM itself and the commands of the network layer are not accessible. The arbitration between ports is done according to the method first come, first serve. Housed in a 16–pin SOIC package, the DS2404S–C01 provides a separate open drain $\overline{\text{IRQ}}$ pin for interrupt signalling and a 1 Hz clock output. Depending on the application the device can either operate on V_{CC} from 2.8V to 5.5V with battery backup or on battery only. The two interfaces of the DS2404S–C01 bridge other electronic equipment to the MicroLAN.

VI. Chapter Summary

Several iButton types have special features that are designed for a special application (DS1991 MultiKey, DS1982 1K bit Add–Only iButton) or add timekeeping functions (DS1994 with its real–time clock, interval counter, cycle counter, expiration controller, and alarm generator). Special features require additional commands and thus may reduce the software compatibility (DS1991, DS1982). The Temperature iButton is another MicroCan device that can operate in a touch environment as well as on a hard–wired 1–Wire bus. To sense and to control the state of nodes using the digital selection capability of the 1–Wire bus, the Addressable Switch has been introduced. All of these devices are electrically compatible in the 1–Wire environment; they don't disturb each other.

iButton FILE STRUCTURE

CHAPTER 7: iButton FILE STRUCTURE

I. Introduction

Most flexibility is obtained from a travelling data storage medium if data of different independent applications or for different purposes can be stored, retrieved and updated independently. These requirements describe well known features of common operating systems and file structures. For use with iButton Memories and their special operating environment, a fully featured file structure, called “extended file structure” has been developed. Together with the extended file structure the so–called “default data structure” exists. The default data structure, as a subset of the extended file structure, does not allow one to store multiple independent files. However, it may be sufficient for very simple applications, or for devices such as the DS1991, which provide a hardware file system. For details on the default data structure please refer to Chapter 10, Validation of iButton Standards.

This chapter describes the fundamentals of the extended file structure and gives sufficient information to generate file structures for Memory iButtons that are fully compatible with iButton TMEX. The DS0621 TMEX Professional Developer’s Kit contains the complete definition of the extended file structure which includes provisions for subdirectories, file attributes, passwords, date stamps, owner identification, etc.

In order to keep compatibility with DOS files on a PC and to save time writing application–specific software, TMEX is available from Dallas Semiconductor for IBM–compatible PCs as the DS0621 TMEX Professional Developer’s Kit. The DS0621 is a software package and requires hardware components of the DS9092K iButton Starter Kit for operation. The DS0621 also includes all details of iButton Memory data structure, nested subdirectories, attributes and guidelines for backward–compatibility, as well as utilities that implement file–oriented iButton Memory functions, including device formatting.

II. Data Organization

As indicated in Chapter 5, ”Logical Standards,” the data organization of iButtons is very similar to floppy disks. A sector of a floppy roughly corresponds to a page of an iButton. The directory tells which files are stored, where the data is found in the device, and how many pages it occupies. In this way information can be randomly accessed for quick response.

The basic structure of iButton data files is shown in Figure 7–1. Each page of the file begins with a length byte, contains a continuation pointer, and ends with an inverted CRC16 check. The continuation pointer is the page address where the file is continued. A continuation pointer 0 marks the last page of a file. The length byte indicates how many valid bytes a page contains, not counting the length byte and the CRC. The CRC calculation, however, also includes the length byte. The CRC accumulator is initialized by setting it equal to the iButton page number. Every byte of a page is transmitted least significant bit first. The length byte is the first to be transmitted. Of the two CRC bytes, the least significant will be sent first.

The basic rules of the data file also apply to the directory file. Figure 7–2 shows details. Instead of data, the directory contains management information and file entries. The control field of seven bytes has the same length as a file entry. The bitmap supports the operating system in allocation of memory space for writing files. In the bitmap of NV RAM–type iButtons, used pages are marked with a 1, empty pages with a 0. The least significant bit corresponds to page 0. The most significant bit of the device flags must be set to 1. All other device flags must be 0. The device directory including control field is created during the process of formatting the device.

File entries consist of the 4–byte file name, one–byte file extension, the start page address where the file begins, and the number of pages the file occupies (Figure 7–3). File names must consist of ASCII characters only, as with DOS. The most significant bit of the file extension is set if the file is read–only. The extension 255 (all bits set) is reserved for the operating system. Continuation pages of the directory don’t need a control field; this space is available to store another file entry.

The extended file structure is also valid for Add–Only iButtons. Since updating a bitmap inside the directory page would result in a non–matching CRC, the bitmap becomes part of the status memory. Since it is only possible to alter an EPROM–bit from 1 to 0, a used page of an Add–Only iButton is marked with a 0 (instead of a 1 as it is with NV RAM iButtons).

Add–Only iButtons that are programmed once are read–compatible to NV RAM type iButtons, as long as no further data is added. By design, Add–Only iButtons provide read compatibility to NV RAM iButtons. Reading the DS1982, however, requires skipping or discarding the first eight data bits received. These bits represent an 8–bit CRC over the read command byte and the specified target address. For full documentation of the file structure adapted to Add–Only iButtons, please refer to the DS0621 TMEX Kit.

STRUCTURE OF A PAGE OF A DATA FILE Figure 7–1

Length **Binary 1 . . . 29**	**Data** **ASCII or Binary**	**Cont.– Pointer** **Binary**	**$\overline{\text{CRC16}}$** **Binary**	**(Unused)**
1 byte	0 to 28 bytes	1 byte	2 bytes	28 to 0 bytes

STRUCTURE OF THE FIRST PAGE OF THE DEVICE DIRECTORY Figure 7–2

Length **Binary 8 . . . 29**	**Control** **Field**	**File Entries** **ASCII & Binary**	**Cont.– Pointer** **Binary**	**$\overline{\text{CRC16}}$** **Binary**	**(Unused)**
1 byte	7 bytes	0 to 21 bytes	1 byte	2 bytes	21 to 0 bytes

Directory Mark **“AA”**	**Attributes** **Binary**	**Device Flags** **10000000**	**Bitmap of** **Used Pages** **2 bytes binary**	**Do Not Change!** **2 bytes binary**
1 byte	1 byte	1 byte	LS/MS–byte	“00” “00”

STRUCTURE OF A FILE ENTRY Figure 7–3

File Name **ASCII, Blank Filled,** **Left Justified**	**File Extension** **Binary**	**Start Page** **Binary**	**Number of Pages** **Binary**
4 bytes	1 byte	1 byte	1 byte

III. Features

The extended file structure is carefully designed to provide high speed and best performance in a touch environment. Every memory page of an NV RAM type iButton can be read, CRC–checked or written without the need to access other pages. If a file is modified, only the affected pages need to be rewritten. Pages of a file need not be contiguous; files can be extended by redefining continuation pointers. Files can be grouped into nested subdirectories. Attributes defined for a directory apply for all files within it. The extended file structure comprehends future types of iButton Memories with up to 256 pages with a maximum of 256 bytes per page.

IV. Utilities

The TMEX functions referring to the Extended File Structure of iButton Memories are implemented for the MS–DOS environment as interrupt calls in the same way as the Interrupt–Level 1–Wire I/O drivers. For MS–Windows, the TMEX functions are provided in a Dynamic Link Library (DLL). (Details are in Chapter 9, ”Systems Integration Software.”) In addition to the interrupts and DLLs, a set of utility programs has been developed that can be executed to perform standard file–oriented data transfers to and from iButton Memories. The utility programs TFormat, TType, TCopy, TDir, TAttrib, and TDel perform for iButton Memories the same operations as the corresponding DOS utilities for disk files. TView is a special utility program that allows easy inspection of the contents of all the various data files on an iButton Memory, and TOpt is a utility program that will perform diagnostics on the file structure and defragment the data on a Memory iButton in the same manner as a disk optimizer program for hard disks. Defragmentation reorganizes the pages of each file to make them contiguous. This results in maximum speed for reading data and directories. The TChk utility performs diagnostic tests on the file structure and displays a report on the integrity of the data, along with any abnormal conditions encountered. All these utilities including documentation are shipped together with TMEX as part of the DS0621 Professional Developer’s Kit.

V. Chapter Summary

The extended file structure is very similar to that of floppy disks. It is optimized for data integrity and speed in a touch environment. This chapter provides basic information about the file structure. Complete documentation is available in the DS0621 Professional Developer’s Kit, for use with the DS9092K iButton Starter Kit.

SYSTEMS INTEGRATION HARDWARE

CHAPTER 8: SYSTEMS INTEGRATION HARDWARE

I. Introduction

This chapter describes a range of hardware interfaces for communicating with iButtons from a variety of different platforms, ranging from microcontrollers and microprocessors to PCs and workstations. Each hardware interface is controlled by a specific set of software or firmware drivers to provide the basic I/O functions called TouchReset, TouchByte, and TouchBit.

II. ESD Protection

Since the 1–Wire bus directly connects peripheral devices to components within a computer, it is necessary to provide circuits that protect against damage by electrostatic discharge (ESD). Figure 8–1 shows an example of ESD protection circuitry applicable for bi–directional ports and for ports with programmable data direction. A protection circuit for fixed–direction ports is shown in Figure 8–3.

Figure 8–1 shows the minimum protection circuitry. The Zener diode CR1 limits positive spikes to a harmless level and clamps negative spikes one diode drop below the ground level. R1 is the pull–up resistor required for 1–Wire operation; it is usually located close to a port pin of a microcontroller. CR1 is located as close to the 1–Wire probe as possible.

ESD is extensively covered in technical literature. More elaborate protection circuits can provide even greater tolerance to electrostatic discharge.

III. IBM–Compatible PCs

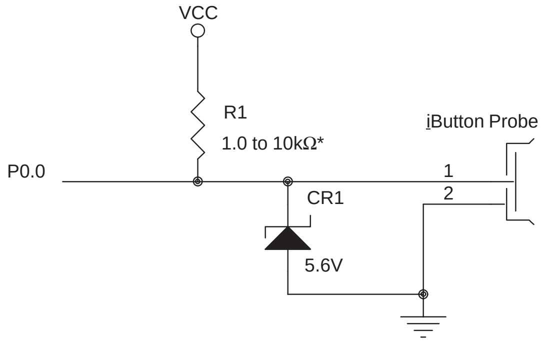
A. DS9097 COM Port Adaptor

The DS9097 COM Port Adapter provides a simple iButton reader for PCs and other computers having an RS232C serial port capable of transmitting and receiving at 115,200 bits per second. The DS9097 is a simple, low–cost passive adapter circuit that performs RS232C level conversion, allowing an iButton Probe to be connected to a PC so that an iButton can be read and written. Due to its passive operation, this adapter is not completely in compliance with the RS232C standard, but generally it works well with IBM–compatible PCs.

The DS9097 is well suited to read all iButton products, to write SRAM devices and to support temperature measurements with one Temperature iButton at a time. For writing Add–Only iButtons the DS9097E, an upgraded version of the DS9097 is available. Due to the limited available energy at the COM port, either a special programming algorithm (a bit at a time) or an external DC supply may be required.

To operate with the DS9097, the serial port must support a data transmission rate of 115,200 bits per second in order to form the 1–Wire time slots correctly, even though the actual rate of data transmission to the iButton is 14,400 bits/second (somewhat below its full data rate of 16,400 bits/second). Nearly all PCs support the required bit rate and are fully compatible with the DS9097. For details, please refer to the brochure ”50 Ways to Touch Memory,” which is included in the DS9092K iButton Starter Kit.

BIDIRECTIONAL PORT WITH ESD–PROTECTION Figure 8–1



* Value selected on the basis of wire capacitance, contact resistance and leakage currents.

B. Add–In Cards

Add–In cards for PCs interfacing up to eight 1–Wire lines are commercially available. The upgrade for Add–Only iButtons and the Temperature iButton is in preparation. For addresses of manufacturers, please contact Dallas Semiconductor.

IV. Interfaces To Other Computers And Operating Systems

For computers other than IBM PCs, a variety of possible interfaces may exist to allow iButton communication. Some of these hardware interfaces are discussed below. A method of forming a 1–Wire bus from two fixed direction ports is explained in section V, later in this chapter.

A. 8250 UART For Serial Communication

If a computer utilizes an 8250 UART or equivalent to provide RS232C serial communication, allows a bit rate of 115,200 bps, and allows user code or installed device drivers to communicate directly with the registers of the 8250, then it will provide iButton communication in the same manner as an IBM PC with a DS9097 COM Port Adaptor. The software techniques required to transmit and receive iButton data on the 1–Wire bus are identical to those used on an IBM PC, and can be coded entirely in a high–level language since the 8250 UART provides the critical timing for the 1–Wire bus. For programming Add–Only iButtons and for operation of the Temperature iButton, additional hardware and software is required to provide the 12V programming pulse and the strong 5V pull–up, respectively. An application note will be available from Dallas Semiconductor.

B. iButton Peripheral Control Card

An add–in card can be designed for any given computer to provide the necessary I/O capabilities to operate the 1–Wire bus. This add–in peripheral might be as simple as a general–purpose parallel input/output (PIO) card, where the 1–Wire timing is provided in software. Alternatively, a controller card can be designed specifically to provide hardware driving and timing for the 1–Wire bus including the requirements for Add–Only iButtons and the Temperature iButton.

C. Phantom Bus Interface

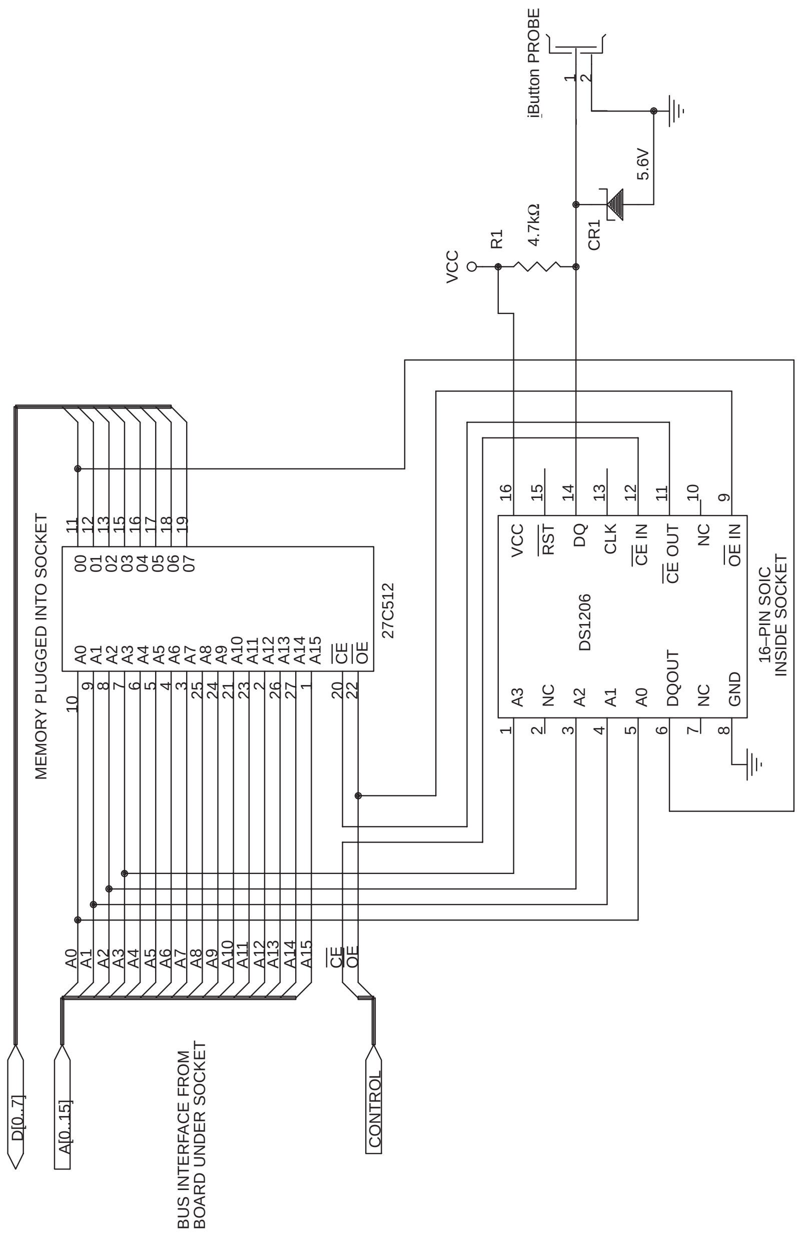
The DS1206 Phantom Serial Interface (Figure 8–2) allows a low–cost connection to a parallel data bus such as that used in IBM PCs and other computers. The Phantom is activated by a coded sequence of address accesses. It then operates as a data latch which allows the software to read or write the 1–Wire bus with the correct timing to send and receive data. After the communication session is completed, the Phantom can be deactivated by reference to a specific address, causing it to disappear from the parallel data bus until it is activated again.

This simple interface can be used to share memory address space with another memory–mapped device. Because of its principle of operation, the Phantom Bus Interface is not capable of supporting all types of interrupts generated by iButtons. Alarms or interrupts can be recognized only when the computer polls the 1–Wire bus via the Phantom Bus Interface. The phantom interface is not capable of supporting the requirements of Add–Only iButtons and the Temperature iButton without additional hardware.

D. RS232C iButton Terminal Interface

For interfacing to larger computers such as workstations and central data processing systems, a standard RS232 communication channel supported by the operating system can be employed. Protocol and level conversion is provided by a simple add–on device, called iButton Terminal Interface. The iButton Terminal Interface is a microprocessor–controlled circuit that allows the data read from an iButton to be injected into an RS232C data link between a central computer and a remotely located ”dumb terminal.” Conversely, data can be transmitted from the remote computer, captured by iButton Terminal Interface, and written into an iButton Memory. Besides its normal function as a bi–directional buffer between computer and terminal, the iButton Terminal Interface also continuously polls for and reads CRC16 validated data from an iButton Memory. If the data is valid, it is injected into the RS232C serial data stream to the computer as if it had been typed on the remote terminal. In the reverse direction, specially coded data received from the central computer is stored in the iButton Terminal Interface to be written into the next Memory iButton that comes into contact with the probe. More details about this can be found in ”50 Ways to Touch Memory,” available from Dallas Semiconductor. The iButton Terminal Interface can be upgraded to comply with the requirements of Add–Only iButtons and the Temperature iButton.

PHANTOM 1–WIRE INTERFACE Figure 8–2



V. Microcontroller Interfaces

The standard hardware interface between the iButton Probe and a microcontroller is a direct connection between the data wire of the probe and an I/O port pin of the microcontroller. In the following sections, the general port pin hardware interfacing issues are discussed. For programming Add–Only iButtons and operation of the Temperature iButton, two more port pins are required to control the 12V supply and the strong 5V pull–up, respectively. Special care must be taken to keep the high programming voltage isolated from the I/O pins of the microcontroller and other 5V devices.

A. General Interfacing Considerations

There are three common implementations of I/O on port pins of a microcontroller: bi–directional, programmable data direction, and fixed direction. Some microcontrollers use one type of I/O exclusively, and others mix two or more of the above types in the same device. The interfacing techniques for communicating on the 1–Wire bus differ slightly for the three different types, as described further below.

B. Bi–Directional Port

A bi–directional port pin is pulled strongly to ground when a zero is written to it, and is either floated (high impedance) or weakly pulled up when written with a one. When the port pin is read, it returns a 1 if the pin is at high voltage (V_{CC}) and a 0 if the port pin is low (ground). To input on such a port pin, the pin must have previously been written to a 1 so that the external circuit drives a high impedance input. Bi–directional port pins are ideally suited for communicating with iButtons on a 1–Wire bus. The port pin is written with a 0 to drive the 1–Wire bus low, and is written with a 1 to allow the 1–Wire bus to be pulled high by the internal or external pull–up resistor. The data on the 1–Wire bus is sampled simply by reading the port pin. All data transfer to and from an iButton is therefore accomplished by writing to or reading from a single pin.

C. Programmable Data Direction

A port pin with programmable data direction is either a low impedance output or a high impedance input, depending on the state of the corresponding bit in the data direction register for that port. This type of port pin can be configured to drive a 1–Wire bus as follows:

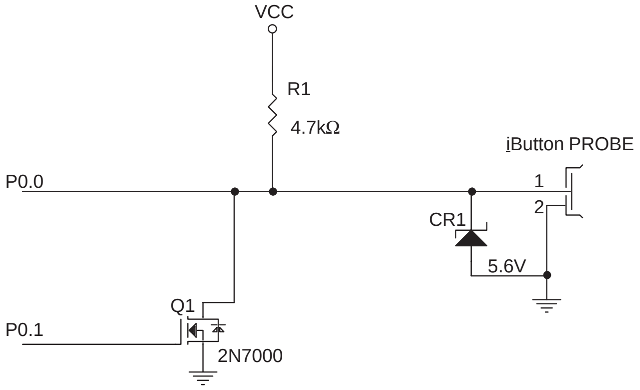
- Write a 0 to the port pin initially, so that it will be driven low when it is in the low impedance output state. (This step need be done only once, when the micro is first powered up.)
- Output to the 1–Wire bus by writing data to the data direction bit corresponding to the chosen port pin. (This data must be complemented prior to output if a 1 on the data direction bit signifies output.)
- Input from the 1–Wire bus by reading data from the chosen port pin.

Note that the method described above has the effect of converting a programmable data direction port pin into a bi–directional port pin (with a low impedance, low voltage state and a high impedance, high voltage state). The only difference in the firmware is that the instructions to input from the 1–Wire bus reference the desired port bit itself, whereas the instructions to output to the 1–Wire bus reference the corresponding data direction bit.

D. Fixed Direction

A fixed direction port pin is one which cannot be converted between output and input states under program control. (Included in this category is a port pin which operates as an input until it is used for the first time as an output. It then becomes an output and remains an output until the micro receives a hardware reset.) In order to construct a 1–Wire bus driver from fixed direction port pins, it is necessary to combine an output port pin with an input port pin as shown in Figure 8–3. Q1 inverts the logical level of the output port and converts it into an open drain output. Data is written to the 1–Wire bus by writing complement data to the output port pin. Reading the 1–Wire bus is accomplished by an input command referring to the input port pin.

FIXED DIRECTION PORT Figure 8–3



VI. Microprocessor Interfaces

Microprocessor interfaces take a variety of forms. iButtons can interface to a microprocessor by way of a standard peripheral adaptor chip. Any system that has a peripheral I/O (PIO) or serial I/O (SIO) can accommodate the 1–Wire protocol. Popular PIO's such as the 8255 or 8256 have software–programmable I/O pins (ports). The 1–Wire interface requires only a single bidirectional port pin with a 5 kΩ pull–up resistor tied from Data to V_{CC}. For programming Add–Only iButtons and operation of the Temperature iButton, two more logic signals are required to control the 12V supply and the strong 5V pull–up, respectively. Special care must be taken to keep the high programming voltage isolated from the microprocessor and other logic circuits.

Two configurations for the 1–Wire bus are possible: either one port pin is dedicated to iButtons or iButtons share a port pin with another device, e.g., a keypad.

VII. Chapter Summary

This chapter presents hardware considerations for communication with iButtons from a variety of host processors, including IBM PCs and compatibles, workstations, microcontrollers and microprocessors. The simplicity of the 1–Wire interface allows a variety of effective communication interfaces to be developed easily for any type of host computer. For further details on hardware and software for programming Add–Only iButtons and operation of the Temperature iButton, please contact Dallas Semiconductor.

SYSTEMS INTEGRATION SOFTWARE

CHAPTER 9: SYSTEMS INTEGRATION SOFTWARE

I. Introduction

This chapter describes a range of software standards and implementations for communicating with iButtons from a variety of different platforms, ranging from microcontrollers to PCs to mainframes. In developing these modules, attention has been given to achieving a hierarchical structure that promotes maximum flexibility and compatibility with existing and future iButton products. The structures and contents of these modules are described in detail in the sections that follow.

II. IBM–Compatible PC Implementations

This section describes integration software designed for the IBM–compatible PC. This category includes not only desktop and laptop PCs but also hand–held MS–DOS data collectors and embedded DOS controllers. A variety of different software platforms for iButton I/O have been developed to satisfy the diverse needs of software developers and system integrators working in the PC and MS–DOS environments. iButtonTMEX is available from Dallas Semiconductor as part number DS0621. Subsection II of this chapter gives a detailed description of TMEX and other PC–compatible platforms.

The basic design of iButton TMEX is patterned after the International Standards Organization (ISO) reference model of Open System Interconnection (OSI), which specifies a layered protocol having up to seven layers, denoted as Physical, Link, Network, Transport, Session, Presentation, and Application. According to this model, the electrical and timing requirements of iButtons and the characteristics of the 1–Wire bus comprise the Physical layer. The software functions TouchReset, TouchByte, and TouchBit correspond in this model with the Link layer, which also includes hardware initialization and fault detection facilities. The multidrop access system functions First, Next, Access, etc., that support selection of individual network nodes correspond to the Network layer. The software that transfers NV RAM data to and from individual network nodes corresponds to the Transport layer. In multi–user or multi–tasking environments such as Microsoft Windows, a Session layer effectively manages sharing of the 1–Wire bus among multiple instances (simultaneous invocations) of iButton application programs. The Presentation layer provides a file structure that allows iButton data to be organized into independent files and randomly accessed (as with a diskette). The Application layer represents the final application program designed by the software developer.

The Physical layer supported explicitly by this software is the DS9097 COM Port Adapter, which connects to the RS232C serial port of the PC and adapts the RS232C signal levels for communication on the 1–Wire bus. The DS9097 works correctly with nearly all PCs, and the RS232C serial port of the PC provides the critical timing required by the 1–Wire bus, so that programming with the DS9097 adapter is time–independent. The higher layers of the protocol (above the link layer) support any possible implementation of the Physical layer.

The description given in the subsequent sections II. A, B, C applies to TMEX Revision 1.10, which requires the DS9097 COM–port adapter and is limited to reading and writing SRAM based iButtons with data densities of up to 4K–bits and reading the DS1982. TMEX 2.00, a newer version of TMEX, also supports reading and writing high–density SRAM and EPROM based devices. The DS9097 COM–Port adapter will be sufficient for reading and writing SRAM–based devices and reading EPROM iButtons. For writing EPROM–based iButtons, the DS9097E enhanced COM–port adapter is required. TMEX 2.00 also supports 1–Wire communication through the parallel port using the DS1410D adapter. Since the parallel port operates on 5–Volt logic levels, it is not suited for writing EPROM–based devices. Due to space limitations a full description of TMEX 2.00 cannot be given in the Book of DS19xx iButton Standards. A short summary of the new functions, however, is found at the end of this chapter.

A. Device drivers for MS–DOS

For the DOS programming environment, the software layers (Link, Network, etc.) are provided as interrupt–level I/O drivers. This is accomplished by use of Terminate and Stay Resident (TSR) programs to install the interrupt handlers in system RAM. (Any or all of these drivers could also be installed by encoding them in the BIOS ROM or the control ROM of an add–in interface card designed to support iButton I/O.)

A.1. Advantages of Interrupt–Level Device Drivers

For DOS programmers, the use of interrupt–level device drivers to support iButton communication offers several advantages over alternative modes of support. Some of the advantages of interrupt–level I/O calling in the DOS environment are listed below:

A.1.a. Hardware Independence

When the iButton I/O functions are called as system interrupts, the hardware specific interface to the 1–Wire data bus is separated from the software that uses it. This gives the hardware designer the maximum latitude in selecting the most suitable hardware interface to iButtons for a particular PC or embedded MS–DOS controller. (In some instances, a ”spare” I/O pin may be available on a PIO which can be adapted easily to serve as the 1–Wire bus driver. In this case the hardware design task is minimal, and the firmware to control the 1–Wire bus can be incorporated into the BIOS ROM.) The software that invokes the system interrupts operates completely independently of the particular hardware implementation.

A.1.b. Interchangeability

An additional advantage is that the user of the software can substitute a different hardware interface simply by overriding the existing iButton I/O interrupt service of the Link layer with a TSR containing a different set of hardware drivers. This could be used to give the software user his choice of iButton I/O technologies and to allow him to switch iButton readers simply by installing different interrupt control procedures. Examples of possible alternatives to the DS9097 COM port adapter include interfaces through a game control port, an LPT port, or a direct interface to the PC bus with the DS1206 Phantom Bus Interface.

A.1.c. Add–In Card Compatibility

The interrupt installation code and interrupt support firmware can be supplied in the ROM of an add–in card that provides iButton I/O capabilities. In this way, both the hardware and firmware support for iButton I/O can be added simply by inserting the card in an expansion slot of the PC. This support for iButton interface cards allows an easy upgrade path as better driver circuits are devised to support long transmission paths, localized signaling (LED or beeper in iButton) and other features.

A.1.d. Convenient Linkage to High Level Languages

Most high–level languages support the means to make system interrupt calls. These interrupt calls are independent of procedure calling conventions, memory model size, segment naming rules, and other issues that complicate mixed–language programming. (In the few instances of high–level languages that support assembly language linkage but not direct interrupt calling, it is generally possible to construct short assembly language procedures which make the required interrupt calls and pass the results through to the calling program.)

A.2. Specification of the Interrupt Level Interface

The text below provides the specification of the interrupt–level hardware–independent interface to the 1–Wire bus used with Dallas Semiconductor iButton products. According to this specification, all iButton device drivers, procedure libraries, and application programs can communicate on the 1–Wire bus by invoking a software interrupt, in the same manner as other I/O devices supported by BIOS. The interface consists of an interrupt vector, designated below as the Dallas One–Wire (DOW) Interrupt. (The default interrupt–type for this vector is 61 Hex.) The DOW Interrupt is a software interrupt which can be called on to perform polled data I/O with any Dallas Semiconductor 1–Wire product.

The file management functions for iButton data files are provided by a second interrupt vector, designated below as the TMEX Interrupt. (The default interrupt–type for this vector is 63H.) The TMEX Interrupt is a software interrupt which provides the file control and I/O functions for reading and writing iButton data files.

A.2.a. The DOW Interrupt (Link, Network and Transport Layers)

The DOW Interrupt is installed by means of two separate TSRs, named BASFUN and EXTFUN. BASFUN installs the Basic Functions associated with the Link Layer, and EXTFUN installs the Extended functions associated with the Network and Transport Layers.

BASFUN provides the basic functions of reset, bit and byte I/O, and hardware fault detection for the DS9097 COM Port Adapter. It must always be installed whenever iButton TMEX is to be used. (To use a different hardware interface to the 1–Wire bus, a new TSR must be written to provide the same basic functions for the new interface.) BASFUN will try to install itself on interrupt 61H. If that interrupt is already used, then another interrupt 60H through 66H must be provided on the command line. For example, installing BASFUN on interrupt 65H could be done by the following: BASFUN 65

EXTFUN provides the network control functions that allow a program to identify and communicate separately with each iButton on the 1–Wire bus. It also provides Transport functions that read and write low–level packets to iButtons. EXTFUN is hardware independent, since it uses the functions provided by BASFUN to communicate with the 1–Wire bus. With no command line parameter, EXTFUN will attempt to search through interrupts 60H to 66H and layer itself on the first BASFUN type interrupt it finds. EXTFUN can be forced to use a particular BASFUN type interrupt by providing it on the command line. For example, if there are BASFUN interrupts on 60H and 64H, and the 64H interrupt is desired, then EXTFUN must be invoked as follows: EXTFUN 64

(1) Structure of the DOW Interrupt

The DOW Interrupt contains a vector to an interrupt service routine having the following structure:

(a) Field #1
This field contains a five byte far jump to the main entry point of the service routine (field #4). The contents of the field are:

i. EA Hex (The one byte far jump instruction.)

ii. Two byte offset address of field #4.

iii. Two byte segment address of field #4.

(b) Field #2
This field consists of one byte specifying the length of the ID string which follows in field #3.

(c) Field #3
This field contains the body of the ID string. The string must begin with the ASCII characters ”DOW”, followed immediately by an ASCII character in the range ”0”–”9” or ”A”–”E” which specifies in hexadecimal the number of the highest numbered function code supported by the interface excluding the CLOSE function. (Function codes are described in section (2) below.) Additional text may optionally be included in the ID string to specify version number, type of 1–Wire interface, etc. However, only the first four characters are required to be present.

(d) Field #4
Actual interrupt service executable code. (The placement of the code at this offset from the base of the structure is optional, since the far jump of field #1 allows placement of this field number anywhere within the range of the far jump.)

The purpose of the above structure is to make it possible for the calling program to determine whether the interrupt is supported before invoking the interrupt. (If the interrupt were not supported, invoking it could cause an error that would require the computer to be rebooted.) To determine that the interrupt is supported, the calling program reads the interrupt vector and uses it as a pointer to address the structure described above and find the string beginning with the characters ”DOW”. The calling program can also read the following character to determine the highest level of support offered by the interrupt service. If the characters ”DOW” are successfully found, then it is safe to invoke the interrupt.

An additional purpose of this structure is to make possible a search of all interrupts to determine which interrupt has been selected for the iButton I/O drivers. This search will be needed when the default interrupt–type 61 hex cannot be used because of conflict with an existing interrupt. (In this case, other interrupt–types in the range 60 – 66 hex may be used instead.)

(2) Functions provided by the DOW Interrupt

The DOW Interrupt is invoked with a function code in the AH register to specify the operation to be performed, and any required output parameter in the AL register. The interrupt returns with the carry bit cleared if the operation was allowed and set otherwise. If the carry bit is set, then an error code is returned in the AL register to indicate the reason that the function was not performed. The error code is as follows:

Physical, Link, and Network Layer Errors
1 => Specified 1–Wire port has not been initialized with a call to the Setup function.
2 => Specified 1–Wire port nonexistent.
3 => Function not supported.

Transport Layer Errors
4 => Error reading or writing a packet.
5 => Packet larger than provided buffer.
6 => Not enough room for packet on device.
7 => Device not found.
8 => Block transfer too long.
9 => Wrong type of device for this function.

If the carry bit is cleared, then the input parameter (result of the function call) is returned in the AL register. Additional information may be returned in other registers, as described in further detail below.

The function code is constructed by placing the number of the desired 1–Wire port (n) in the most significant nibble of AH, and the number of the desired function in the least significant nibble. This allows any of 16 possible functions to be performed on any of 16 separate 1–Wire ports. The functions provided may be classified as basic functions, extended functions, or interrupt control functions as described below:

(a) Basic Functions Provided by BASFUN (Link Layer)
The basic functions described in this section are essential functions which must always be provided in any implementation of the interrupt service. These functions are complete in that there are no other essential functions, and they are universal in the sense that they can communicate with any present or future iButton device that follows the Dallas Semiconductor 1–Wire communication protocol.

i. SETUP: The SETUP function must be called before any other functions will work. It is intended to be called once at the beginning of a communication session to initialize the 1–Wire port and verify the physical integrity of the 1–Wire bus. Note that execution of this function will reset all the parts on the specified 1–Wire bus.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 00 hex.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the error code. If carry is not set then AL has 00 hex if the 1–Wire is shorted and 01 otherwise.
- There may be diagnostic results in DX (see text following this function list).

ii. TOUCHRESET: The TOUCHRESET function transmits the 1–Wire Reset signal to the specified 1–Wire bus and listens for the Presence signal returned by one or more 1–Wire parts present on the bus.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 01 hex.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the error code. If carry is not set then AL has 01 hex if a Presence indicator was detected and 00 hex otherwise.
- There may be diagnostic results in DX (see text following this function list).

iii. TOUCHBYTE: The TOUCHBYTE function transmits a byte least significant bit first to the 1–Wire bus and returns the byte received concurrently from the bus.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 02 hex.
- Byte to transmit in AL.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the error code. If carry is not set then AL contains the byte received from the 1–Wire bus.
- There may be diagnostic results in DX (see text following this function list).

iv. TOUCHBIT: The TOUCHBIT function transmits a bit to the 1–Wire bus and returns the bit received concurrently from the bus.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 03 hex.
- LSB of AL has the bit to transmit.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the error code. If carry is not set then the LSB of AL contains the bit received from the 1–Wire bus.
- There may be diagnostic results in DX (see text following this function list).

v. CLOSE: The CLOSE is the inverse of the SETUP function. When the 1–Wire port is no longer needed for I/O operations, this function can be called to power–down or otherwise release resources that are used by the 1–Wire port. Battery conservation in hand–held and laptop units is thus facilitated. It is intended that this function will be called only once at the end of a 1–Wire communication session. Note that SETUP and CLOSE are potentially time–consuming operations since power cycling of circuitry might be involved.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 0F hex.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the error code.
- There may be diagnostic results in DX (see text following this function list).

Any (or all) of the basic functions can perform optional diagnostics on the 1–Wire port. Upon return from the interrupt service, register DH can be examined to determine which tests (if any) were made. If DH contains a non–zero value then DL will contain test results. A value of zero in DH indicates that no tests were performed. If these optional diagnostic tests are implemented then the format of the results is as follows:

Register DH bits indicate which test(s) were performed. A bit is set for each test performed.
Bit 0: Test for short to ground (1–Wire stuck low).
Bit 1: Test for short to V_{CC} (1–Wire stuck high).
Bit 2: Test for Alarm Interrupt.
Bits 3–7: Reserved for future expansion.

Register DL bits contain the results of tests indicated by DH. A bit is set for each tested condition that was detected.
Bit 0: Fail ground short test (1–Wire is shorted).
Bit 1: Fail V_{CC} short test (1–Wire is shorted).
Bit 2: Alarm Interrupt condition was detected.
Bits 3–7: Reserved for future expansion.

A special case exists where the 1–Wire interface hardware is not able to explicitly detect the polarity of a short (all that is known is that the interface is somehow non– compliant). In this case the function performing the test will set both bits 0 and 1 of register DH to indicate that a test for shorts was performed and if an error is detected both bits 0 and 1 of register DL will be set. The nature of the test should then be clear to the calling routine since normally only one of those bits in DL would be set after any given call to the function.

Please note that the dynamic nature of iButton contact with the 1–Wire bus will occasionally cause some of the above conditions to be detected. The conditions actually occur for brief periods in normal operation. Only a persistent condition should be interpreted as indicating a hardware fault.

(b) Extended Functions Provided by EXTFUN (Network and Transport Layer)
The extended functions described in this section are provided to support the multidrop capability of the 1–Wire protocol, allowing the calling program to locate and individually communicate with one of many iButton devices connected in parallel on the 1–Wire bus. The availability of extended functions may be determined either by checking the highest implemented function number that follows the characters ”DOW” or by calling the desired extended function and checking the carry flag on return.

i. FIRST: The FIRST function transmits a TouchReset signal and returns false if no Presence signal is detected. If a Presence signal is detected, it executes the ROM search algorithm to find the first ROM data pattern on the 1–Wire bus. The ROM data pattern that was found is stored in an internal 8–byte buffer.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 04 hex.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set, then AL contains the error code. If carry is not set, then AL contains a 00 hex if no Presence signal was detected and 01 if a valid ROM was read and placed in the internal 8–byte buffer.
- There may be diagnostic results in DX (see text prior to this function list).

ii. NEXT: The NEXT function transmits a TouchReset signal and returns false if no Presence signal is detected. If a Presence signal is detected, it executes the ROM search algorithm to find the next ROM data pattern on the 1–Wire bus. The ROM data pattern that was found is stored in an internal 8–byte buffer.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 05 hex.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the error code. If carry is not set then AL contains a 00 hex if no Presence signal was detected and 01 if a valid ROM was read and placed in the internal 8–byte buffer.
- There may be diagnostic results in DX (see text prior to this function list).

iii. ACCESS: The ACCESS function transmits a TouchReset signal and returns false if no Presence signal is detected. If a Presence signal is detected, it accesses the device whose ROM code is in the internal 8–byte buffer. The access readies the iButton to accept memory function commands such as read scratchpad. There are three types of access. The first and default type uses the Match ROM command to select the device. This method has the advantage of being fast but the disadvantage of not knowing if the device was on the 1–Wire or not. The second and third type use the Search ROM command to access and verify that the device is on the 1–Wire. The disadvantage of these two 'strong' access commands is that they are slower than the first type. The third type differs from the second in that the device is only accessed if it is alarming.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 06 hex.
- AL contains 00 hex for the default access, F0 hex for a StrongAccess and EC hex for a StrongAlarmAccess.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the error code. If carry is not set then AL contains a 00 hex if no Presence signal was detected and 01 if the access was successful.
- There may be diagnostic results in DX (see text prior to this function list).

iv. ROMDATA: The ROMDATA function provides the means for the calling program to read or write data in the internal 8–byte buffer by returning a far pointer to the buffer. The buffer is arranged least significant byte first.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 07 hex.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the error code.
- ES:BX is a far pointer (Segment:Offset) to the 8–byte internal ROM buffer.
- There may be diagnostic results in DX (see text prior to this function list).

v. FIRSTALARM: The FIRSTALARM function transmits a TouchReset signal and returns false if no alarming Presence signal is detected. If an alarming Presence signal is detected, it executes the ROM search algorithm to find the first alarming ROM data pattern on the 1–Wire bus. The ROM data pattern that was found is stored in an internal 8–byte buffer. This function allows a program to limit the scope of the search algorithm to only those parts that may require attention because an alarm condition has been set.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 08 hex.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the error code. If carry is not set then AL contains a 00 hex if no alarming Presence signal was detected and 01 if a valid ROM was read and placed in the internal 8–byte buffer.
- There may be diagnostic results in DX (see text prior to this function list).

vi. NEXTALARM: The NEXTALARM function transmits a TouchReset signal and returns false if no alarming Presence signal is detected. If an alarming Presence signal is detected, it executes the ROM search algorithm to find the next alarming ROM data pattern on the 1–Wire bus. The ROM data pattern that was found is stored in an internal 8–byte buffer. This function allows a program to limit the scope of the search algorithm to only those parts that may require attention because an alarm condition has been set.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 09 hex.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the error code. If carry is not set then AL contains a 00 hex if no alarming Presence signal was detected and 01 if a valid ROM was read and placed in the internal 8–byte buffer.
- There may be diagnostic results in DX (see text prior to this function list).

vii. READPACKET: The READPACKET function reads an iButton Default Data Structure from the NV RAM of a DS1992, 1993 or 1994 and DS1982 and stores it in a provided buffer. The iButton Default Data Structure is discussed in Chapter 10. Note that the ROM pattern for the desired iButton must already be in the internal 8–byte buffer before this function is called. This constraint enables this function to be multi–drop compatible.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 0A hex.
- AL is the page number that the packet begins on (current devices range from 0 to 15).
- CX is the maximum number of bytes that can be read into the buffer.
- ES:BX is a far pointer (Segment:Offset) to an empty buffer for the data to be read.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the error code.
- CX is the number of bytes read into the buffer.
- There may be diagnostic results in DX (see text prior to this function list).

viii. WRITEPACKET: The WRITEPACKET function writes an iButton Default Data Structure to the NV RAM of a DS1992, 1993 or 1994 starting at a specified page. The iButton Default Data Structure is discussed in Chapter 10. Note that the ROM pattern for the desired iButton must already be in the internal 8–byte buffer before this function is called. This constraint enables this function to be multi–drop compatible.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 0B hex.
- AL is the page number that the packet begins on (current devices range from 0 to 15).
- CX is the number of bytes to write.
- ES:BX is a far pointer (Segment:Offset) to the buffer of data to write.

Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the error code.
- CX is the number of bytes written.
- There may be diagnostic results in DX (see text prior to this function list).

ix. BLOCKIO: The BLOCKIO function is a general–purpose block transfer function. A TouchReset signal is done on the 1–Wire bus and returns false if no Presence signal was found. If a device is present on the 1–Wire bus then the data is transferred one byte at a time using TouchByte. The value that is returned is then placed back into the same place in the data buffer. This function is employed to communicate with devices such as the DS1991 and the clock and register page of the DS1994 (see Chapter 6)

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 0C hex.
- CX is the number of bytes to transfer (1024 max).
- ES:BX is a far pointer (Segment:Offset) to the data buffer to read and write.

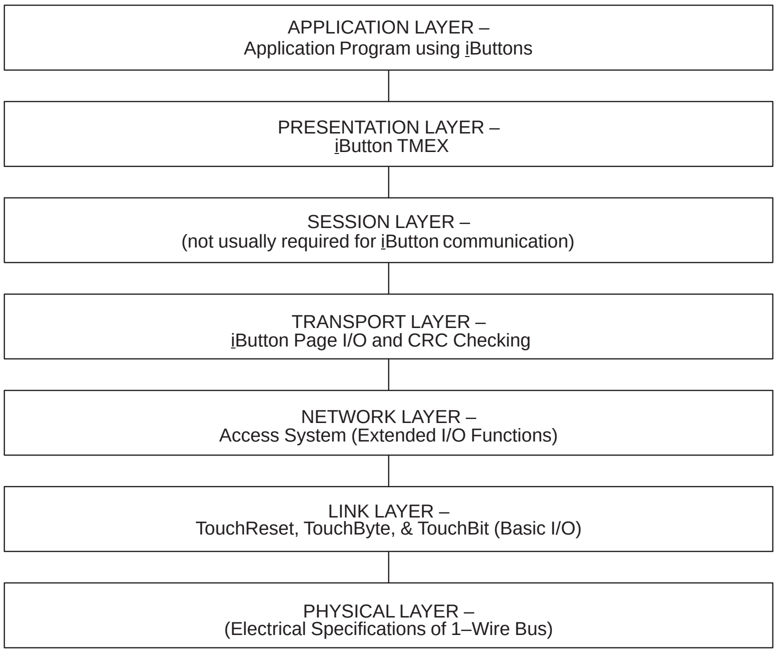
Upon return from function:
- Carry is set if there is an error in execution.
- If carry is set then AL contains the error code.
- CX is the number of bytes transferred (0 if no device found).
- There may be diagnostic results in DX (see text prior to this function list).

(c) Hardware Interrupt Control Functions
Function codes beginning with AH = nD are reserved for the interrupt control functions that will be defined in a future revision of this software. These functions will be used to support hardware interrupt capability for detecting the arrival of iButtons, addressing the problems of enabling and disabling hardware interrupts, handling near–simultaneous interrupts on different 1–Wire lines, and determining which line or lines require interrupt service.

A.2.b. The TMEX Interrupt (Presentation Layer)

In order to insulate the programmer from the details of reading and writing into the Extended File Structure (see Chapter 7), iButton TMEX was developed. This system provides the presentation layer of the layered architecture described above, allowing data to be transferred to and from individual files in an iButton. Figure 9–1 illustrates the organization of systems integration software for iButton communication into a layered structure. The TMEX Interrupt is provided by a TSR named TMEXFUN. TMEX functions are implemented as interrupt calls in the same manner as the DOW Interrupts (see Section A.2.a of this chapter). The default interrupt–type for this interrupt is 63 hex, although any other available interrupt–type in the range 60–66 hex may be specified on the command line. (Note that one of the features provided by the TMEX Interrupt is the ability to locate the DOW Interrupt automatically, even if it is not installed at 61 hex.) A specific DOW Interrupt can be specified on the command line but only if the TMEX interrupt is also specified. For example, if the DOW interrupt is on 65 hex and you want the TMEX interrupt to be on 62 hex, then use the command line: TMEXFUN 62 65

iButton COMMUNICATION LAYERED SOFTWARE STRUCTURE Figure 9–1


APPLICATION LAYER –
Application Program using iButtons
PRESENTATION LAYER –
iButton TMEX
SESSION LAYER –
(not usually required for iButton communication)
TRANSPORT LAYER –
iButton Page I/O and CRC Checking
NETWORK LAYER –
Access System (Extended I/O Functions)
LINK LAYER –
TouchReset, TouchByte, & TouchBit (Basic I/O)
PHYSICAL LAYER –
(Electrical Specifications of 1–Wire Bus)

The TMEX interrupt has the same structure as the DOW Interrupt defined in section II.A.2.a.(1), except that Field #3 contains an ID string that begins with the ASCII characters "TMEX" followed immediately by an ASCII character in the range "0"–"9" or "A"–"E" which identifies the level of support provided by the interrupt.

In much the same way as interrupt 21H (the "DOS" interrupt) provides the facility to read, write, and otherwise manipulate disk files, the TMEX interrupt implements a subset of the standard disk interrupt functions to read and write into the Extended File Structure of an iButton.

In all of the TMEX Interrupt calls, the most significant nibble of AH contains the number of the desired 1–Wire port. The least significant nibble of AH contains the TMEX function number. AL contains the result of the function call or error code if the carry bit is set. Additional information may be returned in other registers. The error codes and their meanings are given below:

1 => NO_DEVICE – no device found on 1–Wire
2 => WRONG_TYPE – wrong type of device, must be DS1992, 1993, 1994 or DS1982 in case of a read
3 => FILE_READ_ERR – file read error, including directory
4 => BUFFER_TOO_SMALL – buffer is smaller than read file
5 => HANDLE_NOT_AVAIL – no more file handles are left
6 => FILE_NOT_FOUND – file specified is not on this device
7 => REPEAT_FILE – file already exists with name provided
8 => HANDLE_NOT_USED – given handle is not assigned a file
9 => FILE_WRITE_ONLY – trying to read a write file handle
10 => OUT_OF_SPACE – not enough room on device to write
11 => FILE_WRITE_ERR – write error, part may have expired

12 => FILE_READ_ONLY – trying to write a read file handle
13 => FUNC_NOT_SUP – function not supported
14 => BAD_FILENAME – illegal filename
15 => CANT_DEL_READ_ONLY – trying to delete read only file
16 => HANDLE_NOT_EXIST – handle does not exist
17 => ONE_WIRE_PORT_ERROR – error in the 1–Wire port, 1–Wire port may not be set up.

Some of the TMEX functions return a pointer to a file entry structure. This structure is given below as a 'C' example. Note that 'uchar' is an 'unsigned char' and would be the same as a 'BYTE' in pascal.

```
typedef struct {
   uchar name[4];                            /* four–character file name */
   uchar extension;                  /* extension number, range 0 – 127 */
   uchar startpage;                      /* page number where file starts */
   uchar numpages;                  /* number of pages occupied by file */
   uchar readonly;                            /* 1 if read only, 0 otherwise */
   uchar bitmap[4];                       /* current bitmap of the device */
} FileEntry;
```

The ROM pattern for the desired iButton must already be in the internal eight–byte buffer before any TMEX function is called. This can be accomplished by direct writing to the internal buffer or by use of the extended functions First or Next. This constraint enables TMEX to be multi–drop compatible. The functions provided by the TMEX Interrupt are specified below. Note that TMEX Version 1.10 does not support bitmap files or sub–directories. Bitmap files are required for devices providing more than 32 pages of memory, e.g., DS1995 or DS1996.

(1) FirstFile: The FirstFile function reads the directory of a specified iButton and returns a pointer to the file information of the first file.

Prior to function call:
– Upper nibble of AH is 1–Wire port number.
– Lower nibble of AH is 01 hex.

Upon return from function:
– Carry is set if there is any error in execution.
– AL is the number of directory entries found or an error code if the carry is set.
– ES:BX is a far pointer (Segment:Offset) to the directory entry of the first file. The directory entry is a structure containing the file name, extension, location, size, attributes, etc.

(2) NextFile: The NextFile function reads the directory of a specified iButton if it has not already been done and returns a pointer to the file information of the next file.

Prior to function call:
– Upper nibble of AH is 1–Wire port number.
– Lower nibble of AH is 02 hex.

Upon return from function:
– Carry is set if there is any error in execution.
– AL is 01 hex if a next file was found or 00 hex if one was not. AL is an error code if the carry is set.
– ES:BX is a far pointer (Segment:Offset) to the directory entry of the next file. The directory entry is a structure containing the file name, extension, location, size, attributes, etc.

(3) OpenFile (for reading): The OpenFile function reads the directory of a specified iButton if it has not already been done and searches for a file name. If it finds it then it assigns a file handle to it. The handle can then be used to read the file with the ReadFile function.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 03 hex.
- ES:BX is a far pointer (Segment:Offset) to the name of the file to be opened. See the FileEntry structure above.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is the file handle number or an error code if the carry is set.

(4) CreateFile (for writing): The CreateFile function reads the directory of a specified iButton if it has not already been done and searchs for a file name. If it does not find it then it assigns a file handle to it. The handle can then be used to write the file with the WriteFile function.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 04 hex.
- ES:BX is a far pointer (Segment:Offset) to the name of the file to be opened. See the FileEntry structure above.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is the file handle number or an error code if the carry is set.
- CX is the maximum number of bytes available for a file.

(5) CloseFile: The CloseFile function closes the specified file handle. This function should be called after a file handle is no longer needed, such as after it has been used in a ReadFile function.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 05 hex.
- AL (lower nibble) is the file handle to be closed.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is an error code if the carry is set.

(6) ReadFile: The ReadFile function reads and stores the file specified by the file handle number. The data is stored in a buffer provided by the caller.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 06 hex.
- AL (lower nibble) is the file handle to be read.
- CX is the maximum number of bytes to read.
- ES:BX is a far pointer (Segment:Offset) to an empty buffer for the data to be read.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is an error code if the carry is set.
- CX is the number of bytes read.

(7) WriteFile: The WriteFile function writes the file specified by the file handle number. The length of the data and the data location are provided by the caller.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 07 hex.
- AL (lower nibble) is the file handle to be read.
- CX is the number of bytes to write.
- ES:BX is a far pointer (Segment:Offset) to the buffer containing the data to be written.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is an error code if the carry is set.
- CX is the number of bytes written.

(8) DeleteFile: The DeleteFile function reads the directory of a specified iButton if it has not already been done and searches for a file name. If it finds it then it deletes the file from the directory and updates the bitmap of used pages.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 08 hex.
- ES:BX is a far pointer (Segment:Offset) to the name of the file to be Deleted.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is the error code if the carry is set.

(9) Format: The Format function writes an empty directory into the specified iButton.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 09 hex.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is the error code if the carry is set.

(10) Attribute: The Attribute function changes the write protect attribute of a specified file.

Prior to function call:
- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 0A hex.
- AL is the file attribute. 01 hex for write protect and 00 hex for no write protect.
- ES:BX is a far pointer (Segment:Offset) to the name of the file.

Upon return from function:
- Carry is set if there is any error in execution.
- AL is the error code if the carry is set.

(11) ReNameFile: The ReNameFile function changes the name of a file that was previously opened using the OpenFile function. The file handle of the opened file and a new name are provided by caller.

Prior to function call:

- Upper nibble of AH is 1–Wire port number.
- Lower nibble of AH is 0B hex.
- AL (lower nibble) is the handle of the file to be re–named.
- ES:BX is a far pointer (Segment:Offset) to the new name of the file.

Upon return from function:

- Carry is set if there is any error in execution.
- AL is an error code if the carry is set.

B. Dynamic Link Libraries for Microsoft Windows (TMEXGEN and TMEXCOM)

In the Microsoft Windows operating environment, the functions of iButton TMEX are provided in the Dynamic Link Libraries (DLLs) named TMEXGEN and TMEXCOM. Use of DLLs to provide common utility functions is a Windows standard, and software development environments for Windows programs include the means to call on the functions provided by DLLs. One of the main advantages of DLLs is that many different iButton programs (or many different instances of the same program) can share the same DLL while executing simultaneously in the multitasking environment of Windows.

When using TMEXGEN.DLL for communication on the 1–wire bus, the TSR named BASFUN must be executed prior to entering the Windows environment, in order to install the link layer for the DS9097 COM Port Adapter. The link layer is kept separate from the DLL because the link layer is hardware dependent. This separation allows a different hardware interface to the 1–Wire bus to be used in place of the serial port adapter without affecting the DLL. The functions provided by TMEXGEN.DLL make interrupt calls to perform basic I/O functions on the 1–Wire bus. If a different link layer for a different hardware interface is installed instead, then the functions provided by TMEXGEN will automatically operate with the new hardware interface.

TMEXCOM.DLL provides exactly the same I/O functions for 1–wire communication as TMEXGEN.DLL, but TMEXCOM is specialized to use the DS9097 COM port adapter exclusively. Use of TMEXCOM instead of TMEXGEN has two advantages:

* The 1–wire communication functions execute faster because the overhead required to call an interrupt is eliminated.

* It is not necessary to execute BASFUN or any other TSR to provide 1–wire interrupt service before entering the Windows environment.

The primary disadvantage of TMEXCOM is that it is not hardware independent. Programs written to use TMEXCOM will therefore have to be revised to take advantage of future advances in 1–wire communication hardware technology.

Because Windows is a multitasking environment, the DLL functions must have as one of their parameters a pointer to a packet of local state variables. This allows each simultaneous instance of an iButton program to maintain its own state specification in order to operate independently of all the others. This is not required in the DOS environment because only one program can be executed at a time. In addition, the multitasking environment requires support for a Session Layer, allowing each instance of an iButton program to request exclusive control of the 1–Wire bus while executing a critical sequence of iButton I/O functions. This is accomplished by means of a semaphore in the data segment of the DLL, which is common to all simultaneous instances. Session functions are provided in the DLL to request and release exclusive control of the 1–Wire bus by means of this semaphore. (Note that the non–preemptive nature of Windows 3.1 multitasking often makes these session functions unnecessary. The session functions will assume greater importance with future Windows environments that support preemptive multitasking.)

The C function prototypes of the functions provided by TMEXGEN and TMEXCOM are specified below. All functions have the FAR PASCAL qualifiers required of DLL functions and return a result of type integer. In all of the functions below in which the parameter ”Handle” appears, Handle is the session handle returned by the function TMStartSession. (A valid session must have been started before any of the functions requiring a session handle can be executed.) All of the DLL functions will return –200 if the session handle ’Handle’ is not valid and –201 if the BASFUN interrupt is not found if it is required by the DLL. A TMEX session in Windows has a time limit so that one iButton program can not monopolize a 1–Wire line. The time limit is initially 1 second. If there is 1–Wire activity in that 1 second interval then the session is extended another second. These extensions are limited to 10 seconds total. A session that has timed–out will invalidate the ”Handle” and all functions will return –200.

The data types used in the DLL prototypes have the following definitions:

FAR – type attribute to make a pointer long
PASCAL – use the pascal calling convention for a function
int – signed 16–bit integer
BYTE – unsigned character byte
LPINT – long pointer to a signed 16–bit integer
BYTE FAR * – long pointer to an unsigned byte

B.1. Version Function

int FAR PASCAL Get_Version(BYTE FAR *idbuf);

This special function is used to copy the identification string from the DLL being called into the buffer ’idbuf’. The format for the identification string is ”xx_DLLNAME_Vz.zz_month/day/year” where:

_	represent spaces
xx	hardware type code 00 to 99 that the DLL uses.
	00 general DLL requiring a BASFUN type interrupt
	01 com port DLL
DLLNAME	the name of the DLL that ranges in length from 1 to 8 characters.
Vz.zz	version number. (i.e. V1.00)
month/day/year	date DLL released

For example the identification string for this version TMEXCOM is

”01 TMEXCOM V1.10 1/1/94”.

If the DLL requires the use of an interrupt, such as TMEXGEN’s use of BASFUN, then the interrupt’s identification string will be appended to ’idbuf’ with a ’&’. For example if TMEXGEN can find BASFUN then the identification string may look like this:

”00 TMEXGEN V1.10 1/1/94 & DOW3 V1.10 1/1/94”

This functions also returns:

1 => identification string has be copied to ’idbuf’
0 => identification string was not copied possibly due to an error in finding required interrupts.

B.2. Basic Functions

These functions provide the same services as the Basic iButton Interrupts installed by the TSR named BASFUN.

B.2.a. int FAR PASCAL TMSetup(int Handle);

This function verifies the existence of the 1–Wire port and returns its condition as follows:

0 => Setup failed.
1 => Setup ok.
2 => Setup ok but 1–Wire bus shorted.
3 => 1–Wire bus does not exist.
4 => Setup not supported.

B.2.b. int FAR PASCAL TMTouchReset(int Handle);

This performs the Reset function on the 1–Wire bus, resetting all of the devices on the 1–Wire port. The function returns the result of the operation as follows:

0 => No presence pulse detected.
1 => Non–alarming presence pulse detected.
2 => Alarming presence pulse detected.
3 => 1–Wire bus is shorted.
4 => Setup has not been run on 1–Wire port.
5 => TouchReset not supported.

B.2.c. int FAR PASCAL TMTouchByte(int Handle, int outch);

This function transmits the least significant byte of the variable outch on the 1–wire bus and concurrently receives a byte value from the 1–wire bus. The received byte is returned as the value of the function.

B.2.d. int FAR PASCAL TMTouchBit(int Handle, int outbit);

This function transmits the least significant bit of the variable outbit on the 1–wire bus and concurrently receives a bit from the 1–wire bus. The received bit is returned as the value of the function.

B.2.e. int FAR PASCAL TMClose(int Handle);

This function closes a particular 1–Wire port. After this function is called on a 1–Wire port, the only way to use it again is to call TMSetup.

B.3. Extended Functions

These functions provide the same services as the Extended iButton Interrupts installed by the TSR named EXTFUN. These functions all require a pointer to the local state variables which must be declared as an array "GB" in the application program. The length of the array in bytes is 208 times the number of the highest numbered port used, plus 208.

B.3.a. int FAR PASCAL TMFirst(int Handle, BYTE FAR *GB);

Call the extended function "First" to look for the first multi–drop device on the 1–Wire bus. Returns a value of 1 if a part is found and 0 otherwise.

B.3.b. int FAR PASCAL TMNext(int Handle, BYTE FAR *GB);

Call the extended function "Next" to look for the next multi–drop device on the 1–Wire bus. Returns a value of 1 if the next part is found and 0 otherwise.

B.3.c. int FAR PASCAL TMAccess(int Handle, BYTE FAR *GB);

Call the extended function "Access" to reset and start a new communication session with a particular device on the 1–Wire bus. The selected device is the one whose ROM contents are located in the ROM data buffer. The ROM data buffer can be filled with the ROM contents of a device by calling First or Next, or by placing a specific ROM code in the buffer using the function TMRom described below. Returns a value of 1 if a part is on the 1–Wire and 0 otherwise.

B.3.d. int FAR PASCAL TMStrongAccess(int Handle, BYTE FAR *GB);

Call the extended function "StrongAccess" to reset and start a new communication session with a particular device on the 1–Wire bus. This is the same as TMAccess except that instead of using a match ROM command sequence the search ROM command sequence is used. This accesses the iButton and also verifies that it is on the 1–Wire. Returns a value of 1 if the selected part is on the 1–Wire and 0 otherwise.

B.3.e. int FAR PASCAL TMStrongAlarmAccess(int Handle, BYTE FAR *GB);

Call the extended function "StrongAlarmAccess" to reset and start a new communication session with a particular device on the 1–Wire bus. This is the same as TMStrongAccess except this function requires that the iButton must have an alarm interrupt condition to be accessed. Returns a value of 1 if the selected part is on the 1–Wire and alarming, and 0 otherwise.

B.3.f. int FAR PASCAL TMRom(int Handle, BYTE FAR *GB, LPINT ROM);

This function transfers a ROM data pattern between the internal eight–byte buffer maintained by the DLL and an array ”ROM” of eight integers declared in the application program. The direction of data transfer is specified by the value of the first integer in the array. If the first integer is zero, then the 8–bytes from the internal buffer are transferred into the eight integer variables of the integer array. If the first integer is non–zero, then the least significant bytes of the eight integers are transferred into the internal eight–byte buffer. (This function allows an application program to obtain the ROM data of a device that has been found with the TMFirst or TMNext functions. It also allows the application program to specify the ROM data of a specific device to be addressed with the TMAccess function.)

B.3.g. int FAR PASCAL TMFirstAlarm(int Handle, BYTE FAR *GB);

The function TMFirstAlarm operates exactly like the function TMFirst described above, except that the search is limited to those parts with an active alarm interrupt pending. This allows a program to limit the scope of the search algorithm to only those parts that may require attention because an alarm condition has been set.

B.3.h. int FAR PASCAL TMNextAlarm(int Handle, BYTE FAR *GB);

The function TMNextAlarm operates exactly like the function TMNext described above, except that the search is limited to those parts with an active alarm interrupt pending.

B.3.i. int FAR PASCAL TMReadPacket(int Handle, BYTE FAR *GB, int StartPage, BYTE FAR *ReadBuf, int MaxRead);

The function TMReadPacket reads a Default Data Structure packet starting on ”StartPage” in a Memory iButton. The data is placed into ”ReadBuf” up to a maximum of ”MaxRead” bytes. Returns a read count length greater then or equal to 0 for success or one of the following negative values for a failure:

–1 / 1–Wire not initialized with SETUP /
–2 / specified 1–Wire port nonexistent /
–3 / function not supported /
–4 / error reading or writing a packet /
–5 / packet larger than provided buffer /
–6 / not enough room for packet on device /
–7 / device not found
–8 / block transfer too long /
–9 / wrong type of device for this function /

B.3.j. int FAR PASCAL TMWritePacket(int Handle, BYTE FAR *GB, int StartPage, BYTE FAR *WriteBuf, int NumWrite);

The function TMWritePacket writes a Default Data Structure packet starting on ”StartPage” in a Memory iButton. The ”NumWrite” bytes of data in ”WriteBuf”is written. Returns a byte length greater then or equal to 0 for success or one of the negative values described in ”TMReadPacket” for a failure.

B.3.k. int FAR PASCAL TMBlockIO(int Handle, BYTE FAR *GB, BYTE FAR *TranBuf, int NumTran);

The function TMBlockIO is a general purpose block transfer function. A TouchReset is done followed by TouchBytes of all of the ”NumTran” bytes in the ”TranBuf” data buffer. The values returned from the TouchBytes are placed back into the ”TranBuf” data buffer. Returns a byte length greater then or equal to 0 for success or one of the negative values described in ”TMReadPacket” for a failure.

Note that this is a ”raw” I/O function. To address a particular part on a multidrop 1–Wire bus, it is necessary to supply the match ROM command byte (55H) and the eight ROM bytes from the internal 8–byte buffer before giving any commands to read or write the NVRAM memory.

B.4. TMEX Functions

The following functions return an integer value representing the result of the operation. Negative results indicate that a TMEX error has occurred. The TMEX error codes and their meanings are given below:

–1 => NO_DEVICE – no device found on 1–Wire
–2 => WRONG_TYPE – wrong type of device, must be DS1992,3,4
–3 => FILE_READ_ERR – file read error, including directory
–4 => BUFFER_TOO_SMALL – buffer is smaller than read file
–5 => HANDLE_NOT_AVAIL – no more file handles are left
–6 => FILE_NOT_FOUND – file specified is not on this device
–7 => REPEAT_FILE – file already exists with name provided
–8 => HANDLE_NOT_USED – given handle is not assigned a file
–9 => FILE_WRITE_ONLY – trying to read a write file handle
–10 => OUT_OF_SPACE – not enough room on device to write
–11 => FILE_WRITE_ERR – write error, part may have expired
–12 => FILE_READ_ONLY – trying to write a read file handle
–13 => FUNC_NOT_SUP – function not supported
–14 => BAD_FILENAME – illegal filename
–15 => CANT_DEL_READ_ONLY – trying to delete read only file
–16 => HANDLE_NOT_EXIST – handle does not exist
–17 => ONE_WIRE_PORT_ERROR – error in the 1–Wire port, 1–Wire port may not be setup.
–200 => INVALID_SESSION – the session handle provided is invalid

The TMEX functions and the operations performed by them are specified as follows:

B.4.a. int FAR PASCAL TMFirstFile(int Handle, BYTE FAR *GB, BYTE FAR *FileName);

This function finds the first file in the device on the 1–Wire port and copies information about the file into the buffer ”FileName”. The information is in the form of a structure. The organization of the structure is as follows:

```
typedef struct {
   uchar name[4];                                /* four–character file name */
   uchar extension;                      /* extension number, range 0 – 127 */
   uchar startpage;                           /* page number where file starts */
   uchar numpages;                      /* number of pages occupied by file */
   uchar readonly;                                  /* 1 if read only, 0 otherwise */
   uchar bitmap[4];                              /* current bitmap of the device */
} FileEntry;
```

This function returns:
>0 => number of file entries in directory, first file entry is in buffer ”FileName”
0 => if the device has an empty directory
<0 => a TMEX error has occurred

B.4.b. int FAR PASCAL TMNextFile(int Handle, BYTE FAR *GB, BYTE FAR *FileName);

This function finds the next file in the device on the 1–Wire port and copies information about the file into the buffer ”FileName”. The information is in the form of a structure. The organization of the structure is described in ”TMFirstFile” above.

This function returns:

1	=>	if the next file was found, data is in buffer ”FileName”
0	=>	if there are no more files in the directory
<0	=>	a TMEX error has occurred

B.4.c. int FAR PASCAL TMOpenFile(int Handle, BYTE FAR *GB, BYTE FAR *FileName);

This function finds the filename specified in the buffer ”FileName” and returns a handle for that file. The filename must be in the format specified in ”TMFirstFile.” Only the ”name[4]” and ”extension” are used, however.

This function returns:

>= 0	=>	file found, and this is the file handle
< 0	=>	a TMEX error has occurred

B.4.d. int FAR PASCAL TMCreateFile(int Handle, BYTE FAR *GB, LPINT maxwrite, BYTE FAR *FileName);

This function searches for the file specified in the buffer ”FileName” to verify that it does not already exist, and returns the file handle.

This function returns:

>= 0	=>	file created, and this is the file handle
< 0	=>	a TMEX error has occurred

B.4.e. int FAR PASCAL TMCloseFile(int Handle, BYTE FAR *GB, int FileHandle);

This function closes the file specified by ”FileHandle.”

The function returns:

1	=>	file closed
< 0	=>	a TMEX error has occurred

B.4.f. int FAR PASCAL TMReadFile(int Handle, BYTE FAR *GB, int FileHandle, BYTE FAR *ReadBuf, int maxread);

This function reads from the file specified by the file handle number ”FileHandle” and copies it into the buffer ”ReadBuf,” not to exceed ”maxread” characters.

The function returns:

>= 0	=>	file read, and this is the number of bytes
< 0	=>	a TMEX error has occurred

B.4.g. int FAR PASCAL TMWriteFile(int Handle, BYTE FAR *GB, int FileHandle, BYTE FAR *WriteBuf, int numwrite);

This function writes the file specified by the file handle ”FileHandle” with the data from the buffer ”WriteBuf,” containing ”numwrite” characters.

The function returns:

>= 0	=>	file written, and this is the number of bytes
< 0	=>	a TMEX error has occurred

B.4.h. int FAR PASCAL TMDeleteFile(int Handle, BYTE FAR *GB, BYTE FAR *FileName);

This function deletes the file named ”FileName.”

The function returns:

1	=>	file successfully deleted
< 0	=>	a TMEX error has occurred

B.4.i. int FAR PASCAL TMFormat(int Handle, BYTE FAR *GB);

This function writes an empty directory into the Memory iButton.

The function returns:

1	=>	Memory iButton successfully formatted
< 0	=>	a TMEX error has occurred

B.4.j. int FAR PASCAL TMAttribute(int Handle, BYTE FAR *GB, int attr, BYTE FAR *FileName);

This function changes the attributes of the file ”FileName” to the attributes ”attr.”

The function returns:

1	=>	file attribute changed to ”attr”
< 0	=>	a TMEX error has occurred

B.4.k. int FAR PASCAL TMReNameFile(int Handle, BYTE FAR *GB, int FileHandle, BYTE FAR *FileName);

This function changes the name of a previously opened file specified by the ”FileHandle”. The new name is given in ”FileName”.

The function returns:

1	=>	file name changed to ”FileName”
< 0	=>	a TMEX error has occurred

B.5. Session Functions

The functions in this section are used to establish a session on the 1–Wire bus, return a session handle to use in subsequent communication, and to establish whether a valid session is in progress. (Because Windows is a multi–tasking environment, a session is needed to prevent interference between simultaneous instances of programs that communicate on the 1–Wire bus.)

B.5.a. int FAR PASCAL TMStartSession(int Prt);

This function is called with the port number ”Prt” of the 1–Wire bus to be used. (For example, using the DS9097 COM port adapter, this would be the COM port number.) The function returns the session handle number if the session has been established, and 0 to indicate that the 1–Wire bus is busy while another session is in progress. The session handle is good for at least 1 second and up to 10 seconds with continuous use.

B.5.b. int FAR PASCAL TMValidSession(int Handle);

This function is called to determine whether a session is still valid with the specified session handle. It returns a 1 if the session is established, and 0 otherwise.

B.5.c. int FAR PASCAL TMEndSession(int Handle);

This function is called to close a session of iButton communication on the 1–Wire bus. It returns a 1 if the session specified by ”Handle” was successfully closed, and 0 if there was no valid session established with the specified session handle.

It is the responsibility of the programmer to open and close sessions frequently, in order to allow other instances of iButton programs to continue executing in the multi–tasking environment. In general, a session should be opened, a specific indivisible task should be performed, and then the session should be closed to allow other programs to access the 1–Wire bus.

C. EXAMPLE PROGRAMS UTILIZING TMEX

This section describes the sample programs which have been provided to demonstrate the use of TMEX. The sample programs for the MS DOS environment are designed to perform familiar utility operations on Touch Memory data files, similar to those provided by standard DOS commands. These programs are written in Borland C. The C program examples call on a library of routines that are very similar to the DLL routines described above. The library in turn calls on the TMEX interrupt functions. The library source is provided with the DS0621 TMEX Professional Developer's Upgrade. All of these programs assume DS9097 COM–port adaptor and only one iButton of type DS1992/3/4 on the 1–Wire bus.

C.1. MS DOS Programming Examples in C

The following utility programs are provided as C source code and also as executables. These programs demonstrate functions provided by the interrupt calls of TMEX. For these programs to function, the interrupt service routines 'BASFUN', 'EXTFUN', and 'TMEXFUN' must be installed first (for installation see sections A.2.a and A.2.b).

C.1.a. TTYPE

The TMEX utility 'TTYPE' is similar to the DOS command 'TYPE' in that it prints the file specified to standard output. The output can be redirected into another file. Providing 'TTYPE' with a single argument of '?' will give the following usage message:

usage: ttype filename <OneWirePort>

– displays the iButton data from the file 'filename' through OneWirePort
– argument 1 specifies filename
– 'filename' must have the format NAME.XXX, where 'NAME' is up to a 4 digit alphanumeric name and XXX is a number extension in the range 0–126.
– <optional> argument 2 specifies OneWirePort
– default OneWirePort may be specified in file default.prt
– output may be redirected. Example: ttype NAME.124 > NAME.124
– version 1.10

'TTYPE' has the following source files:

TTYPE.C	–	the main ttype program.
TUTIL.C	–	varies utility routines common to all of the TMEX programs.
TMEXLIB.C	–	library of routines that use the interrupt calls to implement basic, extended and TMEX functions. The function prototypes and return values are very similar to the TMEX DLLs.
TMEXUTIL.H	–	include file containing includes and global variables common to all of the TMEX programs.

C.1.b. TDIR

The TMEX utility 'TDIR' is similar to the DOS command 'DIR' in that it prints a list of the files on a device. It also provides the length, starting location, and attributes of each file. The output can be redirected into another file. Providing 'TDIR' with a single argument of '?' will give the following usage message:

usage: tdir <OneWirePort>

– displays the root directory of an iButton on OneWirePort
– <optional> argument 1 specifies OneWirePort
– default OneWirePort may be specified in file default.prt
– version 1.10

'TDIR' has the following source files:

TDIR.C – the main tdir program.
TUTIL.C, TMEXLIB.C, and TMEXUTIL.H – as described in 'TTYPE'.

C.1.c. TDEL

The TMEX utility 'TDEL' is similar to the DOS command 'DEL' in that it deletes a file from a device. It can only delete a single file at a time. If all files must be deleted, use 'TFORMAT'. Providing 'TDEL' with a single argument of '?' will give the following usage message:

usage: tdel filename <OneWirePort>
- delete the file 'filename' on OneWirePort
- argument 1 specifies filename
- 'filename' must have the format NAME.XXX, where 'NAME' is up to a 4 digit alphanumeric name and XXX is a number extension in the range 0–126.
- <optional> argument 2 specifies OneWirePort
- default OneWirePort may be specified in file default.prt
- version 1.10

'TDEL' has the following source files:
TDEL.C – the main tdel program.
TUTIL.C, TMEXLIB.C, and TMEXUTIL.H – as described in 'TTYPE'.

C.1.d. TATTRIB

The TMEX utility 'TATTRIB' is similar to the DOS command 'ATTRIB' in that it changes the attribute of a file on a device. For TMEX Version 1.10, the only attribute is read–only. Providing 'TATTRIB' with a single argument of '?' will give the following usage message:

usage: tattrib (+R | –R) filename <OneWirePort>
- changes the read only attribute of the file 'filename' through OneWirePort
- argument 1 specifies if the attribute is to be added '+R' or taken away '–R'
- argument 2 specifies filename
- 'filename' must have the format NAME.XXX, where 'NAME' is up to a 4 digit alphanumeric name and XXX is a number extension in the range 0–126.
- <optional> argument 3 specifies OneWirePort
- default OneWirePort may be specified in file default.prt
- version 1.10

'TATTRIB' has the following source files:
TATTRIB.C – the main tattrib program.
TUTIL.C, TMEXLIB.C, and TMEXUTIL.H – as described in 'TTYPE'.

C.1.e. TCOPY

The TMEX utility 'TCOPY' is similar to the DOS command 'COPY' in that it can copy a DOS file to an iButton Memory device. It can not copy a file from an iButton to a DOS file however. To copy a file from an iButton, use 'TTYPE' and redirection. Providing 'TCOPY' with a single argument of '?' will give the following usage message:

usage: tcopy source destination <OneWirePort>
- copies the DOS file 'source' to the iButton on OneWirePort
- argument 1 specifies the DOS source filename
- argument 2 specifies the iButton destination filename. NAME.XXX where 'NAME' is up to a 4 digit alphanumeric name and XXX is a number extension in the range 0–126.
- <optional> argument 3 specifies OneWirePort
- default OneWirePort may be specified in file default.prt
- version 1.10

'TCOPY' has the following source files:
TCOPY.C – the main tcopy program.
TUTIL.C, TMEXLIB.C, and TMEXUTIL.H – as described in 'TTYPE'.

C.1.f. TFORMAT

The TMEX utility 'TFORMAT' is similar to the DOS command 'FORMAT' in that it re–formats an iButton device. Unlike DOS however, this is a quick operation. 'TFORMAT' will delete all files on an iButton device regardless of the attributes. Providing 'TFORMAT' with a single argument of '?' will give the following usage message:

```
usage: tformat <OneWirePort>
      –  formats an iButton on OneWirePort
      –  <optional> argument 1 specifies OneWirePort
      –  default OneWirePort may be specified in file default.prt
      –  version 1.10
```

'TFORMAT' has the following source files:
TFORMAT.C – the main tformat program.
TUTIL.C, TMEXLIB.C, and TMEXUTIL.H – as described in 'TTYPE'.

C.1.g. TREN

The TMEX utility 'TREN' is similar to the DOS command 'REN' in that it renames a file. It can only rename a single file at a time. Providing 'TREN' with a single argument of '?' will give the following usage message:

```
usage: tren oldfilename newfilename <OneWirePort>
      –  rename the file 'oldfilename' to 'newfilename' on OneWirePort
      –  argument 1 specifies the old filename
      –  'oldfilename' must have the format NAME.XXX, where 'NAME' is up to a 4 digit alphanumeric
         name and XXX is a number extension in the range 0–126.
      –   'newfilename' must be in the same format as 'oldfilname'.
      –  <optional> argument 3 specifies OneWirePort
      –  default OneWirePort may be specified in file default.prt
      –  version 1.10
```

'TREN' has the following source files:
TREN.C – the main tren program.
TUTIL.C, TMEXLIB.C, and TMEXUTIL.H – as described in 'TTYPE'.

C.1.h. TPEEK

The TMEX utility 'TPEEK' examines the interrupts 60H through 66H and displays the TSR ID strings found there. This utility can be used to see if the TMEX TSR's have already been installed. Providing 'TPEEK' with a single argument of '?' will give the following message:

```
usage: tpeek
      –  displays the ID strings from any TMEX driver found on interrupts 60H to 66H.
      –  version 1.10
```

'TPEEK' has the following source files:
TPEEK.C – main tpeek program.
TUTIL.C, TMEXLIB.C, and TMEXUTIL.H – as described in 'TTYPE'.

C.2. Other iButton Utilities for MS DOS

The following utility programs provide services similar to those in section D.1. above. Since these programs are rather long and do not demonstrate any new techniques, they are provided in executable form only.

C.2.a. TVIEW

The TMEX utility ’TVIEW’ displays a list of the files on an iButton device and also some device information. Using the arrow keys, a file can be selected and its contents displayed. Providing ’TVIEW’ with a single argument of ’?’ will give the following usage message:

usage: tview <OneWirePort> <bw>
- displays the root directory of an iButton on OneWirePort
- allows the individual files to be examined
- <optional> argument 1 specifies OneWirePort
- default OneWirePort may be specified in file default.prt
- <optional> argument 2 selects black and white mode.
- version 1.10

C.2.b. TCHK

The TMEX utility ’TCHK’ checks the extended file structure of an iButton device for errors or fragmentation. Each file is checked and its status given. Providing ’TCHK’ with a single argument of ’?’ will give the following usage message:

usage: tchk <OneWirePort>
- checks the directory and files of an iButton on OneWirePort
- <optional> argument 1 specifies OneWirePort
- default OneWirePort may be specified in file default.prt
- version 1.10

C.2.c. TOPT

The TMEX utility ’TOPT’ checks the extended file structure as in ’TCHK’ and then corrects any problems found. Providing ’TOPT’ with a single argument of ’?’ will give the following usage message:

usage: tchk <OneWirePort>
- checks the directory and files of a Memory iButton on OneWirePort
- <optional> argument 1 specifies OneWirePort
- default OneWirePort may be specified in file default.prt
- version 1.10

C.2.d. TMEMCOPY

The TMEX utility ’TMEMCOPY’ copies the contents of one iButton Memory to another. This utility searches the entire iButton for valid data structures and copies them to any iButton that has corresponding pages. A data structure is considered to be valid if it is according to the description given in Chapter 7, “iButton File Structure.” Providing ’TMEMCOPY’ with a single argument of ’?’ will give the following messages:

usage: tmemcopy <OneWirePort>
- copy all valid data packets from one iButton to another
- <optional> argument 1 specifies OneWirePort
- default OneWirePort may be specified in file default.prt
- version 1.10

D. Programming Considerations

D.1. General Considerations

D.1.a. BIOS Support

It is envisioned that in a typical implementation, code to support both the basic and the extended 1–Wire interrupt functions would be incorporated into the BIOS ROM. In the event that BIOS ROM space is limited, the basic interrupt functions could be incorporated in ROM and the extended functions, being hardware–independent, could be loaded into RAM, either with a device driver or TSR program. (A TSR or device driver could also be used at any time to substitute a new set of interrupt support routines for use with a different hardware interface, such as the RS232C serial port adaptor.) The primary advantage of the BIOS implementation of the basic functions is hardware independence, i.e., the functions operate the same way regardless of the hardware implementation. This permits a wide latitude in designing the 1–Wire support hardware. (In many cases the required I/O pins are already available for use and need only to be suitably connected on the PC motherboard.)

D.1.b. Calling from Application Programs

An MS–DOS application program can perform any desired iButton I/O operation by making direct calls to the interrupts defined in this document. An MS–Windows program can achieve the same result by calling the functions defined in the TMEX DLL.

D.2. Linkable Device Drivers

While the interrupt–level device drivers described in section A above provide comprehensive modular support for iButton I/O, there are some cases in which it would be preferable to include the iButton I/O procedures as part of the main application program. This is accomplished by linking the desired procedures from an iButton I/O library designed to be linked to a variety of different high–level languages. These linkable I/O procedures provide essentially the same basic and extended functions as the interrupt–level device drivers described in section II.A.2.a.(2) above. Some of the advantages of this approach are listed below.

D.2.a Independence

With a built–in set of iButton I/O procedures, the program does not require that the I/O firmware be installed (e.g., by executing a TSR) before the program is run. Also, once execution of the program is completed, the code for iButton I/O is automatically removed from memory, thereby releasing that memory for other uses. Also, the built–in procedures do not interfere with existing interrupts in the PC that might otherwise be in conflict with the interrupts designated for the iButton I/O drivers.

D.2.b. Security

In applications such as software access control and usage auditing for which the iButton provides a security function, the linked iButton I/O procedures provide a greater degree of security because the information flow is more difficult to trace. (The interrupt–level device drivers can easily be traced with a filter TSR that monitors activity on the iButton I/O interrupts. While this would normally be considered an advantage for diagnostic purposes, it is disadvantageous for security purposes because the data flow to and from the iButton is no longer concealed.)

The main difficulty in linking iButton I/O procedures to high–level languages is that each language has different calling conventions, parameter passing requirements, and segment naming rules. Even within a single language, the choice of memory model affects calling and parameter passing requirements. This problem can be dealt with by providing the basic and extended iButton I/O procedures as linkable .OBJ modules designed specifically for use with various common language compilers. The .OBJ modules can be made more generally linkable by putting them in the large memory model with the Pascal calling convention. They can then be called from Pascal directly, and from C by declaring function prototypes with the ”far” and ”pascal” qualifiers to override the default attributes. To call them from other languages (Basic, Fortran, etc.), it may be necessary to use a short patch code which converts between the different calling conventions. (Alternatively, the iButton I/O procedures can be coded directly in the appropriate high–level language and recompiled to insure compatibility. While this approach guarantees correct linkage, it sacrifices the code size and speed advantage derived from hand coding in assembly language.)

When security is not an issue, the greatest flexibility can be achieved by writing the application code so that it first checks whether the interrupt–level device drivers are installed. If they are installed, it uses them for communication with iButtons. If they are not installed, it uses instead the linkable device drivers for communication with a particular hardware interface (such as the DS9097 COM Port Adapter) as a default. This provides the end user with a program which can operate independently, but which can also automatically take advantage of specialized iButton reading hardware and interrupt–level device drivers if they are installed in his computer system.

III.iButton Usage With Other Computers And Operating Systems

A variety of different options are available from Dallas Semiconductor for communication with iButtons from computers other than IBM–compatible PCs. For computers other than IBM PCs and compatibles, several possible interfaces can be used for iButton communication. These include the 8250 UART, the iButton Peripheral Control Card, and the Phantom Bus Interface previously described in Chapter 8. With each of these interfaces, the key to communication lies in the implementation of the basic 1–Wire functions of the link layer (TouchReset, TouchByte, and TouchBit) to operate with the desired interface. The network layer, transport layer, session layer (if needed) and presentation layer can be coded in a portable, high–level language.

IV. Microcontroller Programming Support For iButtons

Microcontrollers offer the simplest and most flexible interface to iButton products because the electrical connection is extremely simple and because the event timing needed to communicate with iButtons in microcontroller applications can be performed with high precision. On the other hand, it is impractical to provide a single set of software which is suitable for all microcontrollers because the instruction sets of various microcontrollers are different and because, even with a single microcontroller, the coding of iButton I/O procedures depends on clock crystal frequency, limitations on available microprocessor resources, choice of firmware development tools, and other factors. This section presents design factors that should be considered in constructing microcontroller firmware for iButton I/O. It should be noted that many of the design factors presented here apply equally well to any implementation of iButton I/O software.

In microcontrollers, it is generally necessary to write the code to generate the reset/presence signals and the time slot I/O signals in assembly language to achieve the correct timing. All other code can be written in non–time–critical assembly language or in a high–level language designed for microcontrollers. This section describes the recommended structure of microcontroller firmware for communicating with iButtons.

A. TouchReset

The procedure TouchReset delivers the Reset signal to the 1–Wire bus and returns a Boolean result indicating whether a Presence signal was detected. (The Presence signal is indicated by a low pulse occurring within 60 µs after the rising edge of the Reset signal and lasting between 60 and 240 µs.) In addition, it is possible for the TouchReset function to detect and report several other conditions which may or may not be needed in any particular application. These other conditions are described below.

A.1. Alarm Pending

It is possible to detect a pending alarm condition during the presence detect phase, and this condition may also be reported by the TouchReset function. (The pending alarm condition is indicated when the 1–Wire bus remains low continuously for at least 960 µs after the falling edge of the Reset signal, but no longer than 3840 µs.)

A.2. Short Circuit

The TouchReset function may detect that the 1–Wire bus remains low continuously for at least 3840 µs after the falling edge of the Reset signal, indicating a short circuit on the 1–Wire bus. Depending on the application, the TouchReset function may return only a Presence result, or it may also discriminate the alarm pending and short circuit conditions. (When only the Presence result is returned, it should be returned True if an alarm condition is present but False if a short circuit is detected.)

A.3. Adaptive Timing

During the Reset sequence, the Presence signal received from the 1–Wire bus can be used to optimize the timing for communication with the particular devices connected on the bus when the TouchReset procedure is executed. This information is derived from measurements of the length of time between the rising edge of the Reset signal and the beginning of the Presence pulse and the length of the Presence pulse itself. The potential advantage of adaptive timing is that with a typical iButton with an internal time base of 30 µs, adaptive timing allows communication at approximately twice the non–adaptive rate, and also allows approximately twice the recovery time to overcome the effect of long–line capacitance, thereby increasing the reliability of long–line communication. When multiple iButtons are on the 1–Wire bus, the time base of the slowest part and the time base of the fastest part can be calculated from the measured times. The time base of the slowest part limits the data transmission rate, whereas the time base of the fastest part limits the signal recovery time. (Non–adaptive procedures assume the worst–case range of 15 to 60 µs for the internal time base. Adaptive procedures become identical to non–adaptive procedures when connected to a 1–Wire bus having both a 15 µs and a 60 µs iButton.) Note that when adaptive timing is used, the timing should be recalibrated after every reset pulse to insure that the communication to follow is correctly timed.

B. Touchbit

This procedure transfers a single bit of information on the 1–Wire bus, and returns a single bit which indicates the information returned by one or more of the iButton devices on the bus. (The use of a single procedure for both output and input parallels the bi–directional port pin hardware philosophy. The advantage of using a procedure in which the direction of data flow is unspecified is that it is easier to relay iButton data from one processor to another because the intermediate processors do not have to know which direction information is flowing.)

C. Touchbyte

This is a derived function which transmits and receives all eight bits of a byte. The bits of the byte are transmitted and received least significant bit first. (The function is easily implemented by calling the TouchBit procedure eight times.)

D. Access System

The Access System provides the networking support functions which are required to handle multiple iButtons on a common 1–Wire bus. These functions are described in section II.A.2.a.(2)(b). Since these functions communicate with the 1–Wire bus through the procedures TouchReset, TouchByte, and TouchBit, they are themselves hardware– and timing– independent. They may therefore be coded in a high–level language.

E. Hardware Interrupt Handling

One of the advantages of an iButton is that it is able to generate an interrupt request on the 1–Wire bus when it requires attention. In order to be able to respond to these interrupts, the port pin to which the 1–Wire bus is attached must support a hardware interrupt triggered by the falling edge of the voltage signal on the bus. Interrupts may result from one of the following circumstances:

1. When an iButton first makes contact with the 1–Wire bus, it generates a Presence signal which may be used to trigger a hardware interrupt, provided that hardware interrupts are enabled when the contact is made. This interrupt signals that a new part has arrived on the bus, and that a search of the bus should be performed to identify and communicate with the new part.
2. When an iButton with alarm capability (e.g., DS1994) detects an alarm condition and the last signal on the 1–Wire bus was the Reset/Presence signal, the part will apply the alarm signal to the 1–Wire bus, again triggering a hardware interrupt. (To insure that this interrupt source can be detected, the processor interrupt capability must be enabled by setting the interrupt enable bit for the desired port pin, and the iButton interrupt capability must be enabled by transmitting the Reset signal on the 1–Wire bus.) This indicates that a predefined alarm condition in the iButton has been satisfied, and that the special function registers of the iButton should be read to identify the condition that caused the alarm. (Note that when using alarm interrupts in a multidrop environment, the alarm search command EC hex is very desirable as a means to limit the search to those parts which may have caused the interrupt.)

V. Usage Of iButtons With Centralized Computers

There are several possible interfaces to centralized computers. All of the following methods remove the critical 1–Wire timing from the computer so that the software resident in the computer is language– and operating system–independent. Common to all larger computers is the RS232C interface.

The simplest interface for this is the iButton Terminal Interface. This device is inserted into the serial link between the centralized computer and a dumb terminal. Data from iButtons is injected into the serial data stream and sent to the centralized device as if it had been entered from the keyboard of the terminal. Specially coded data from the central computer can be buffered and written into the next iButton that is placed into contact with the probe. With the iButton Terminal Interface, a centralized computer need only be concerned with a high level representation of the data contained in the iButtons. Hardware information on centralized computer interfacing can be found in ”50 Ways to Touch Memory.”

VI. TMEX2.00 Enhancements

As indicated earlier in this chapter, TMEX 2.00 provides functions to support high–density SRAM and EPROM iButtons. The enhancements apply equally to the DOS and WINDOWS implementation. These new functions are:

- TMEpromPulse, to generate a program pulse,
- TMFamilySearch, to perform a ROM search for a particular family code only,
- TMSkipFamily, to perform a ROM search disregarding a particular family code,
- TMReadEpromPage, to read a particular page of EPROM status or data memory
- TMWriteEpromByte, to write a byte of EPROM data or status memory,
- TMDirectoryMR, to make or remove a (sub–) directory,
- TMChangeDirectory, to change from the current directory to another one,
- TMCreateEpromJob, to prepare a data structure to be written to an EPROM device and
- TMDoEpromJob to write the data generated with TMCreateEpromJob to an EPROM–based iButton.

Due to the implementation of (sub–) directories with TMEX2.00, the group of DOS utilities (section C of this chapter) was enhanced by:

- TMD, to make a (sub–) directory
- TRD, to remove a (sub–)directory,
- TCD, to change the directory and
- TTREE, to list the directory structure of a device.

These and the other DOS–utilities (exception: TOPT) now apply to all iButtons, SRAM and EPROM, all densities. A full description of all new functions will be found on the TMEX disk.

VII. Chapter Summary

This chapter presents the various implementations of systems integration software, including IBM PCs and compatibles, other computers and operating systems, microcontrollers, and centralized computers. The system integration software is organized in a layered structure, with the lowest software layer (link layer) providing the basic 1–Wire I/O procedures, the intermediate layer (network layer) handling multi–drop communication, and the highest layer (presentation layer) providing file structure for iButtons similar to that of a diskette.

NOTE:
At the time this document was converted from “Touch Memory” to “iButton” the current TMEX was Revision 3.00. For owners of the DS9092k all TMEX DOS utilities mentioned in this chapter as well as a TMEX–based Windows Application (3.1 and 95/NT) called “iButton Viewer” including documentation are available for free as executable programs if downloaded from the Dallas Semiconductor web site.

VALIDATION OF iButton STANDARDS

CHAPTER 10: VALIDATION OF iButton STANDARDS

I. Introduction

This book defines mechanical, electrical, and logical iButton standards. As with any standard, there are ways to ensure that an application meets the standards. For applications in full compliance with the standard, Dallas Semiconductor will authorize the use of Certified Dallas Touch^{TM} labels, which can be placed on a company's products.

II. Touch Validator

The Touch Validator is a Touch Pen with the addition of validation firmware executed by the internal processor. The Touch Validator checks timing characteristics, ROM contents, and the structure of data stored in the nonvolatile RAM of a Memory iButton. Beyond that, it runs a diagnostic on the integrity of the file structure (not applicable for the DS1990A and DS1991). If any deviation from the standards is detected, the complete RAM contents are copied for additional analysis by a PC. This mode of operation allows fast "go/no go" checking and validation directly at the location of the iButton. For information on Validator firmware, please contact Dallas Semiconductor.

III. Default Data Structure

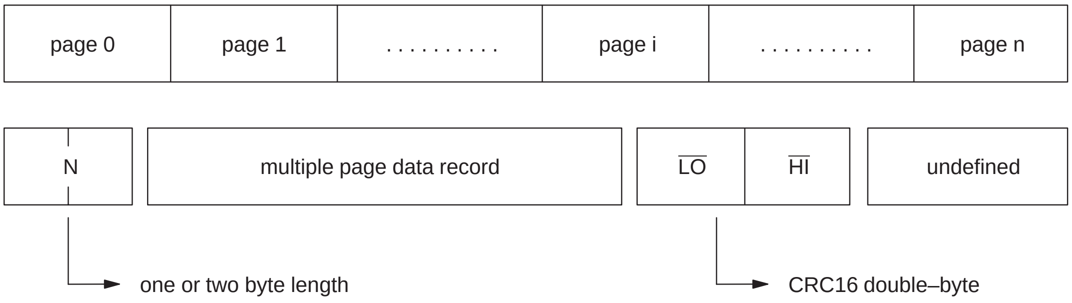
Since it is an official standard, the validation firmware will also accept the Default Data Structure, which is actually the predecessor of the file structure described in Chapter 7. The Default Data Structure has advantages in applications where a sophisticated directory structure is not required. It saves some space since one page can contain up to 32 bytes of application data instead of 28; it also simplifies the software, which will save ROM code of single–chip microcontrollers. In contrast to the Extended File Structure, however, the Default Data Structure does not support multiple files in the same device, CRC–checked reading of single memory pages, scattered continuation pages, and data records of more than 508 bytes.

The default data structure assumes that only one unnamed file will be stored in an iButton. It defines that the first one or two bytes on page 0 contain the length of the data record. If the data record is longer than 254 bytes, then the first byte of page 0 is 255, and the second byte is deployed to store the number of bytes exceeding 255. The first byte following the length byte(s) must be an ASCII code less than 128 (80H). After the application data, there must follow an inverted CRC16 double–byte (Figure 10–1). The length byte(s) itself and the CRC16 are not counted to determine the length. As with the extended file structure, the length byte(s) is included in the CRC16 calculation. Before the CRC16 calculation, the CRC16 accumulator is initialized to zero.

IV. Chapter Summary

The Touch Validator is a device that checks whether an iButton Application is in compliance with the iButton standards as described in this book. In addition to the iButton File Structure described in Chapter 7, the Default Data Structure is accepted as standard for simple applications.

iButton DEFAULT DATA STRUCTURE Figure 10–1



APPENDICES

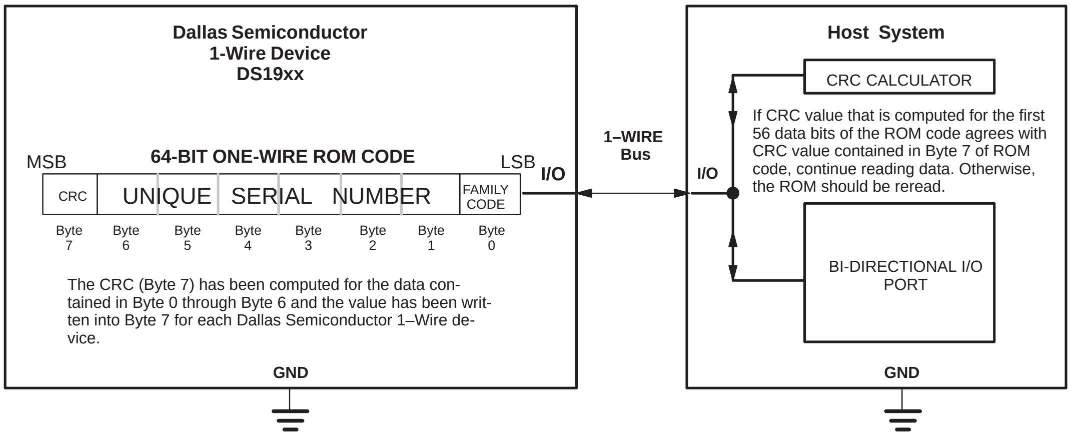
APPENDIX 1
UNDERSTANDING AND USING CYCLIC REDUNDANCY CHECKS WITH iButon PRODUCTS

INTRODUCTION

The Dallas Semiconductor iButton products are a family of devices that all communicate over a single wire following a specific command sequence referred to as the 1–Wire^{TM} Protocol. A key feature of each device is a unique 8–byte ROM code written into each part at the time of manufacture. The components of this 8–byte code can be seen in Figure 1. The least significant byte contains a family code that identifies the type of iButton product. For example, the DS1990A has a family code of 01 Hex and the DS1991 has a family code of 02 Hex. Since multiple devices of the same or different family types can reside on the same 1–Wire bus simultaneously, it is important for the host to be able to determine how to properly access each of the devices that it locates on the 1–Wire bus. The family code provides this information. The next six bytes contain a unique serial number that allows multiple devices within the same family code to be distinguished from each other. This unique serial number can be thought of as an "address" for each device on the 1–Wire bus. The entire collection of devices plus the host form a type of miniature local area network, or Micro-LAN; they all communicate over the single common wire. The most significant byte in the ROM code of each device contains a Cyclic Redundancy Check (CRC) value based on the previous seven bytes of data for that part. When the host system begins communication with a device, the 8–byte ROM is read, LSB first. If the CRC that is calculated by the host agrees with the CRC contained in byte 7 of ROM data, the communication can be considered valid. If this is not the case, an error has occurred and the ROM code should be read again.

Some of the iButton products have up to 8K bytes of RAM in addition to the eight bytes of ROM that can be accessed by the host system with appropriate commands. Even if iButtons do not have CRC hardware onboard, if the host has the capability to calculate a CRC value for the ROM codes, then a procedure to store and retrieve data in the RAM portion of the devices using CRCs can also be developed. Data can be written to the device in the normal manner; then a CRC value that has been calculated by the host is appended and stored with the data. When this data is retrieved from the iButton, the process is reversed. The host compares the CRC value that was computed for the data bytes to the value stored in memory as the CRC for that data. If the values are equal, the data read from the iButton can be considered valid. In order to take advantage of the power of CRCs to validate the serial communication on the 1–Wire bus, an understanding of what a CRC is and how they work is necessary. In addition, a practical method for calculation of the CRC values by the host will be required for either a hardware or software implementation.

iButton SYSTEM CONFIGURATION USING DOW CRC Figure 1



The CRC (Byte 7) has been computed for the data contained in Byte 0 through Byte 6 and the value has been written into Byte 7 for each Dallas Semiconductor 1–Wire device.

If CRC value that is computed for the first 56 data bits of the ROM code agrees with CRC value contained in Byte 7 of ROM code, continue reading data. Otherwise, the ROM should be reread.

BACKGROUND

Serial data can be checked for errors in a variety of ways. One common way is to include an additional bit in each packet being checked that will indicate if an error has occurred. For packets of 8–bit ASCII characters, for example, an extra bit is appended to each ASCII character that indicates if the character contains errors. Suppose the data consisted of a bit string of 11010001. A ninth bit would be appended so that the total number of bits that are 1's is always an odd number. Thus, a 1 would be appended and the data packet would become 111010001. The underlined character indicates the parity bit value required to make the complete 9–bit packet have an odd number of bits. If the received data was 11101000 1, then it would be assumed that the information was correct. If, however, the data received was 111010101, where the 7th bit from the left has been incorrectly received, the total number of 1's is no longer odd and an error condition has been detected and appropriate action would be taken. This type of scheme is called odd parity. Similarly, the total number of 1's could also be chosen to always be equal to an even number, thus the term even parity. This scheme is limited to detecting an odd number of bit errors, however. In the example above, if the data was corrupted and became 111011101 where both the 6th and 7th bits from the left were wrong, the parity check appears correct; yet the error would go undetected whether even or odd parity was used.

DESCRIPTION

Dallas Semiconductor 1–Wire CRC

The error detection scheme most effective at locating errors in a serial data stream with a minimal amount of hardware is the Cyclic Redundancy Check (CRC). The operation and properties of the CRC function used in Dallas Semiconductor products will be presented without going into the mathematical details of proving the statements and descriptions. The mathematical concepts behind the properties of the CRC are described in detail in the references. The CRC can be most easily understood by considering the function as it would actually be built in hardware, usually represented as a shift register arrangement with feedback as shown in Figure 2. Alternatively, the CRC is sometimes referred to as a polynomial expression in a dummy variable X, with binary coefficients for each of the terms. The coefficients correspond directly to the feedback paths shown in the shift register implementation. The number of stages in the shift register for the hardware description, or the highest order coefficient present in the polynomial expression, indicate the magnitude of the CRC value that will be computed. CRC codes that are commonly used in digital data communications include the CRC–16 and the CRC–CCITT, each of which computes a 16–bit CRC value. The Dallas Semiconductor 1–Wire CRC (DOW CRC) magnitude is eight bits, which is used for checking the 64–bit ROM code written into each 1–Wire product. This ROM code consists of an 8–bit family code written into the least significant byte, a unique 48–bit serial number written into the next six bytes, and a CRC value that is computed based on the preceding 56 bits of ROM and then written into the most significant byte. The location of the feedback paths represented by the exclusive–or gates in Figure 2, or the presence of coefficients in the polynomial expression, determine the properties of the CRC and the ability of the algorithm to locate certain types of errors in the data. For the DOW CRC, the types of errors that are detectable are:

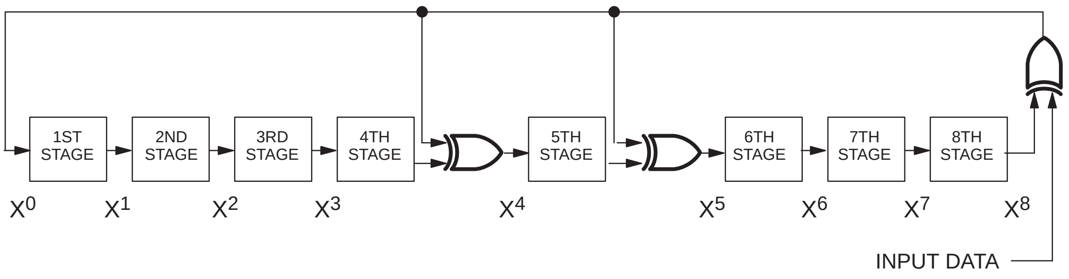
1. Any odd number of errors anywhere within the 64–bit number.
2. All double-bit errors anywhere within the 64–bit number.
3. Any cluster of errors that can be contained within an 8–bit "window" (1–8 bits incorrect).
4. Most larger clusters of errors.

The input data is Exclusive–Or'd with the output of the eighth stage of the shift register in Figure 2. The shift register may be considered mathematically as a dividing circuit. The input data is the dividend, and the shift register with feedback acts as a divisor. The resulting quotient is discarded, and the remainder is the CRC value for that particular stream of input data, which resides in the shift register after the last data bit has been shifted in. From the shift register implementation it is obvious that the final result (CRC value) is dependent, in a very complex way, on the past history of the bits presented. Therefore, it would take an extremely rare combination of errors to escape detection by this method.

The example in Figure 3 calculates the CRC value after each data bit is presented. The shift register circuit is always reset to 0's at the start of the calculation. The computation begins with the LSB of the 64–bit ROM, which is the 02 Hex family code in this example. After all 56 data bits (serial number + family code) are input, the value that is contained in the shift register is A2 Hex, which is the DOW CRC value for that input stream. If the CRC value which has been calculated (A2 Hex in this example), is now used as input to the shift register for the next eight bits of data, the final result in the shift register after the entire 64 bits of data have been entered should be all 0's. This property is always true for the DOW CRC algorithm. If any 8–bit value that appears in the shift register is also used as the next eight bits in the input stream, then the result that appears in the shift register after the 8th data bit has been shifted in is always 00 Hex. This can be explained by observing that the contents of the 8th stage of the shift register is always equal to the incoming data bit, making the output of the EXOR gate controlling the feedback and the next state value of the first stage of the shift register always equal to a logic 0. This causes the shift register to simply shift in 0's from left to right as each data bit is presented, until the entire register is filled with 0's after the 8th bit. The structure of the Dallas Semiconductor 1–Wire 64–bit ROM uses this property to simplify the hardware design of a device used to read the ROM. The shift register in the host is cleared and then the 64 ROM bits are read, including the CRC value. If a correct read has occurred, the shift register is again all 0's which is an easy condition to detect. If a non-zero value remains in the shift register, the read operation must be repeated.

DALLAS 1–WIRE 8–BIT CRC Figure 2

Polynomial = $X^8 + X^5 + X^4 + 1$



Until now, the discussion has centered around a hardware representation of the CRC process, but clearly a software solution that parallels the hardware methodology is another means of computing the DOW CRC values. An example of how to code the procedure is given in Table 1. Notice that the XRL (exclusive or) of the A register with the constant 18 Hex is due to the presence of the EXOR feedback gates in the DOW CRC after the fourth and fifth stages as shown in Figure 2. An alternative software solution is to simply build a lookup table that is accessed directly for any 8–bit value currently stored in the CRC register and any 8–bit pattern of new data. For the simple case where the current value of the CRC register is 00 Hex, the 256 different bit combinations for the input byte can be evaluated and stored in a matrix, where the index to the matrix is equal to the value of the input byte (i.e., the index will be I = 0–255). It can be shown that if the current value of the CRC register is not 00 Hex, then for any current CRC value and any input byte, the lookup table values would be the same as for the simplified case, but the computation of the index into the table would take the form of:

New CRC = Table [I] for I=0 to 255 ;
where I = (Current CRC) EXOR (Input byte)

For the case where the current CRC register value is 00 Hex, the equation reduces to the simple case. This second approach can reduce computation time since the operation can be done on a byte basis, rather than the bit-oriented commands of the previous example. There is a memory capacity tradeoff, however, since the lookup table must be stored and will consume 256 bytes compared to virtually no storage for the first example except for the program code. An example of this type of code is shown in Table 2. Figure 4 shows the previous example repeated using the lookup table approach. Two properties of the DOW CRC can be helpful in debugging code used to calculate the CRC values. The first property has already been mentioned for the hardware implementation. If the current value of the CRC register is used as the next byte of data, the resulting CRC value will always be 00 Hex (see explanation above). A second property that can be used to confirm proper operation of the code is to enter the 1's complement of the current value of the CRC register. For the DOW CRC algorithm, the resulting CRC value will always be 35 Hex, or 53 Decimal. The reason for this can be explained by observing the operation of the CRC register as the 1’s complement data is entered, as shown in Figure 5.

ASSEMBLY LANGUAGE PROCEDURE Table 1

```
DO_CRC:       PUSH   ACC                ;save accumulator
              PUSH   B                  ;save the B register
              PUSH   ACC                ;save bits to be shifted
              MOV    B,#8               ;set shift = 8 bits ;

CRC_LOOP:     XRL    A,CRC              ;calculate CRC
              RRC    A                  ;move it to the carry
              MOV    A,CRC              ;get the last CRC value
              JNC    ZERO               ;skip if data = 0
              XRL    A,#18H             ;update the CRC value
                                        ;

ZERO:         RRC    A                  ;position the new CRC
              MOV    CRC,A              ;store the new CRC
              POP    ACC                ;get the remaining bits
              RR     A                  ;position the next bit
              PUSH   ACC                ;save the remaining bits
              DJNZ   B,CRC_LOOP         ;repeat for eight bits
              POP    ACC                ;clean up the stack
              POP    B                  ;restore the B register
              POP    ACC                ;restore the accumulator
              RET
```

EXAMPLE CALCULATION FOR DOW CRC Figure 3

Complete 64–Bit 1–Wire ROM Code: A2 00 00 00 01 B8 1C 02

Family Code:	0	2	Hex										
	0000	0010	Binary										
Serial Number:	0	0	0	0	0	0	0	1	B	8	1	C	Hex
	0000	0000	0000	0000	0000	0000	0000	0001	1011	1000	0001	1100	Binary

CRC VALUE	INPUT VALUE	
00000000	0	
00000000	1	
10001100	0	2
01000110	0	
00100011	0	
10011101	0	
11000010	0	0
01100001	0	
10111100	0	
01011110	0	
00101111	1	C
00010111	1	
00001011	1	
00000101	0	
10001110	0	1
01000111	0	
10101111	0	
11011011	0	
11100001	0	8
11111100	1	
11110010	1	
11110101	1	
01111010	0	B
00111101	1	
00011110	1	
10000011	0	
11001101	0	1
11101010	0	
01110101	0	
10110110	0	
01011011	0	0
10100001	0	
11011100	0	
01101110	0	
00110111	0	0
10010111	0	
11000111	0	
11101111	0	
11111011	0	0
11110001	0	
11110100	0	
01111010	0	
00111101	0	0
10010010	0	
01001001	0	
10101000	0	
01010100	0	0
00101010	0	
00010101	0	
10000110	0	
01000111	0	0
10101101	0	
11011010	0	
01101101	0	
10111010	0	0
01011101	0	

10100010 = A2 Hex = CRC Value for [00000001B81C (Serial Number) + 02 (Family Code)]

CRC VALUE	INPUT VALUE	
10100010	0	
01010001	1	
00101000	0	2
00010100	0	
00001010	0	
00000101	1	
00000010	0	A
00000001	1	

00000000 = 00 Hex = CRC Value for A2 [(CRC) + 00000001B81C (Serial Number) + 02 (Family Code)]

DOW CRC LOOKUP FUNCTION Table 2

```
Var
   CRC : Byte;
Procedure Do_CRC(X: Byte);
{
   This procedure calculates the cumulative Dallas Semiconductor 1–Wire CRC of all bytes passed to it.  The result
accumulates in the global variable CRC.
}
Const
   Table : Array[0..255] of Byte = (
          0,    94,   188,   226,    97,    63,   221,   131,   194,   156,   126,    32,   163,   253,    31,    65,
        157,   195,    33,   127,   252,   162,    64,    30,    95,     1,   227,   189,    62,    96,   130,   220,
         35,   125,   159,   193,    66,    28,   254,   160,   225,   191,    93,     3,   128,   222,    60,    98,
        190,   224,     2,    92,   223,   129,    99,    61,   124,    34,   192,   158,    29,    67,   161,   255,
         70,    24,   250,   164,    39,   121,   155,   197,   132,   218,    56,   102,   229,   187,    89,     7,
        219,   133,   103,    57,   186,   228,     6,    88,    25,    71,   165,   251,   120,    38,   196,   154,
        101,    59,   217,   135,     4,    90,   184,   230,   167,   249,    27,    69,   198,   152,   122,    36,
        248,   166,    68,    26,   153,   199,    37,   123,    58,   100,   134,   216,    91,     5,   231,   185,
        140,   210,    48,   110,   237,   179,    81,    15,    78,    16,   242,   172,    47,   113,   147,   205,
         17,    79,   173,   243,   112,    46,   204,   146,   211,   141,   111,    49,   178,   236,    14,    80,
        175,   241,    19,    77,   206,   144,   114,    44,   109,    51,   209,   143,    12,    82,   176,   238,
         50,   108,   142,   208,    83,    13,   239,   177,   240,   174,    76,    18,   145,   207,    45,   115,
        202,   148,   118,    40,   171,   245,    23,    73,     8,    86,   180,   234,   105,    55,   213,   139,
         87,     9,   235,   181,    54,   104,   138,   212,   149,   203,    41,   119,   244,   170,    72,    22,
        233,   183,    85,    11,   136,   214,    52,   106,    43,   117,   151,   201,    74,    20,   246,   168,
        116,    42,   200,   150,    21,    75,   169,   247,   182,   232,    10,    84,   215,   137,   107,    53);
Begin
   CRC := Table[CRC xor X];
End;
```

TABLE LOOKUP METHOD FOR COMPUTING DOW CRC Figure 4

Current CRC Value (= Current Table Index)	Input Data	New Index (= Current CRC xor Input Data)	Table (New Index) (= New CRC Value)
0000 0000 = 00 Hex	0000 0010 = 02 Hex	(00 H xor 02 H) = 02 Hex = 2 Dec	Table[2]= 1011 1100 = BC Hex = 188 Dec
1011 1100 = BC Hex	0001 1100 = 1C Hex	(BC H xor 1C H) = A0 Hex = 160 Dec	Table[160]= 1010 1111 = AF Hex = 175 Dec
1010 1111 = AF Hex	1011 1000 = B8 Hex	(AF H xor B8 H) = 17 Hex = 23 Dec	Table[23]= 0001 1110 = 1E Hex = 30 Dec
0001 1110 = 1E Hex	0000 0001 = 01 Hex	(1E H xor 01 H) = 1 F Hex = 31 Dec	Table[31]= 1101 110 = DC Hex = 220 Dec
1101 1100 = DC Hex	0000 0000 = 00 Hex	(DC H xor 00 H) = DC Hex = 220 Dec	Table[220]= 1111 0100 = F4 Hex = 244 Dec
11110100 = F4 Hex	0000 0000 = 00 Hex	(F4 H xor 00 H) = F4 Hex = 244 Dec	Table [244]= 0001 0101 = 15 Hex = 21 Dec
0001 0101 = 15 Hex	0000 0000 = 00 Hex	(15 H xor 00 H) = 15 Hex = 21 Dec	Table[21]= 1010 0010 = A2 Hex = 162 Dec
1010 0010 = A2 Hex	10100010 = A2 Hex	(A2 H xor A2 H) = Hex = 0 Dec	Table[0]=0000 0000 = 00 Hex = 0 Dec

CRC REGISTER COMBINED WITH 1'S COMPLEMENT OF CRC REGISTER Figure 5

CRC Register Value								Input
X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_7*
1	X_0	X_1	X_2	X_3*	X_4*	X_5	X_6	X_6*
1	1	X_0	X_1	X_2*	X_3	X_4*	X_5	X_5*
1	1	1	X_0	X_1*	X_2*	X_3	X_4*	X_4*
0	1	1	1	X_0	X_1*	X_2	X_3	X_3*
1	0	1	1	0	X0*	X1*	X2	X2*
1	1	0	1	0	1	X0*	X1*	X1*
0	1	1	0	1	0	1	X0*	X0*
0	0	1	1	0	1	0	1	Final CRC Value = 35 Hex, 53 Decimal

Note: X_i* = Complement of Xi

CRC–16 COMPUTATION FOR RAM RECORDS IN iButtons

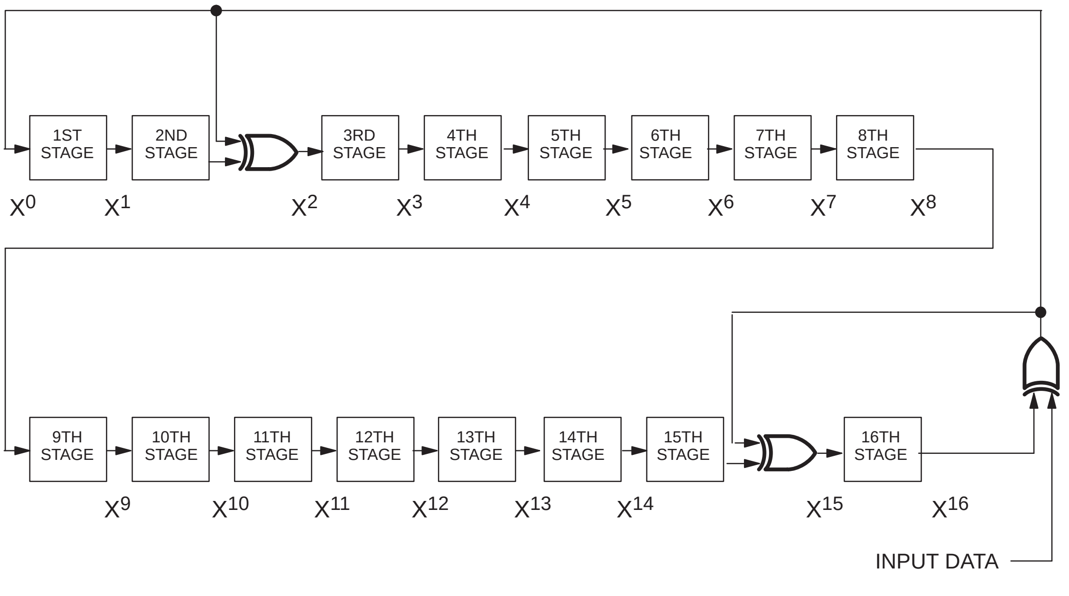
As mentioned in the introduction, some iButton devices have RAM in addition to the unique 8–byte ROM code found in all iButtons. Because the amount of data stored in RAM can be large compared to the 8–byte ROM code, Dallas Semiconductor recommends using a 16–bit CRC value to ensure the integrity of the data, rather than the 8–bit DOW CRC used for the ROM. The particular CRC suggested is commonly referred to as CRC–16. The shift register and polynomial representations are given in Figure 6. The figure shows that for a 16–bit CRC, the shift register will contain 16 stages and the polynomial expression will have a term of the sixteenth order. As stated previously, the iButton devices do not calculate the CRC values. The host must generate the value and then append the 16-bit CRC value to the end of the actual data. Due to the uncertainty of the iButton's "communication channel," i.e., the two metal contact surfaces, data transfers can experience errors that generally fall into three categories. First, brief intermittent connections can cause small numbers of bit errors to occur in the data, which the normal CRC–16 function is designed to detect. The second type of error occurs when contact is lost altogether, for example when the iButton is removed from the reader too quickly.

This causes the last portion of the data to be read as logic 1's, since no connection to an iButton will be interpreted as all 1's by the host. The normal CRC–16 function can also detect this condition under most circumstances. The third type of error is generated by a short circuit across the reader, which can be caused by an iButton that is not inserted correctly, or tilted significantly once in the reader. A short at the reader causes the data to be read as all 0's by the host. When using CRCs, this can cause problems, since the method to determine the validity of the data is to read the data plus the stored CRC value, and see if the resulting CRC computed at the host is 0000 Hex (for a 16–bit CRC.) If the reader was shorted, the data plus the CRC value stored with the data will be read as all 0's, and a false read will have occurred, but the CRC computed by the host will incorrectly indicate a valid read. In order to avoid this situation, Dallas Semiconductor recommends storing the complement of the computed CRC–16 value (CRC–16*) with the data that is written into the RAM. Using an uncomplemented CRC–16 value, the retrieval of data from the iButton is similar to the DOW CRC case. That is, if the CRC register in the host is initialized to 0000 Hex and then all of the data plus the CRC–16 value stored with the data is read from the iButton, the resulting calculation by the host should have a 0000 Hex, as a final result. If instead, the complement of the CRC–16 value is stored with the data in the iButton, then the CRC register at the host is initialized to 0000 Hex and the actual data plus the stored CRC–16* value is read. The resultant CRC value should be B001 Hex for a valid read. This greatly improves the operation of the system, since it can no longer be fooled by a short at the reader. The reason that the CRC–16 function has these properties can be shown in an analogous manner to the DOW CRC case (see Figures 3 and 5). The operation of the 16–bit CRC is identical in theory to the 8 bit version described earlier, but the properties of the CRC change since a 16–bit value is now available for error detection. For the CRC–16 function, the types of errors that are detectable are:

1. Any odd number of errors anywhere within the data record.
2. All double–bit errors anywhere within the data record.
3. Any cluster of errors that can be contained within a 16–bit "window" (1–16–bits incorrect).
4. Most larger clusters of errors.

CRC–16 HARDWARE DESCRIPTION AND POLYNOMIAL Figure 6

Polynomial = $X^{16} + X^{15} + X^2 + 1$



The hardware implementation of the CRC–16 function is straightforward from the description given in Figure 6. Table 3 shows a software solution that is analogous to the hardware operations which compute the CRC–16 values using single–bit operations. As before, a less computation–intensive software solution can be developed through the use of a lookup table. The basic concepts presented for the 8 bit DOW CRC lookup table also work for the CRC–16 case. A slight modification in procedure from the 8–bit case is required, however, because if the entire 16–bit result for the CRC–16 function were mapped into one table as before, the table would have 2^{16} or 65536 entries. A different approach is shown in Table 4, where the 16–bit CRC values are computed and stored in two 256–entry tables, one containing the high order byte and the other the low order byte of the resultant CRC. For any current 16–bit CRC value, expressed as Current_CRC16_Hi for the current high order byte and Current _CRC16_Lo for the current low order byte, and any new input byte, the equation to determine the index into the high order byte table for locating the new high order byte CRC value (New_CRC16_Hi) is given as:

New_CRC16_Hi = CRC16_Tabhi[I] for I=0 to 255;
where I = (Current_CRC16_Lo) EXOR (Input byte)

The equation to determine the index into the low order byte table for locating the new low order byte CRC value (New_CRC16_Lo) is given as:

New_CRC16_Lo = (CRC16_Tablo[I]) EXOR (Current_CRC16_Hi) for I=0 to 255;
where I = (Current_CRC16_Lo) EXOR (Input byte)

An example of how this method works is shown in Figure 7.

ASSEMBLY LANGUAGE FOR CRC–16 COMPUTATION Table 3

```
crc_lo          data    20h                         ; lo byte of crc calculation (bit addressable)
crc_hi          data    21h                         ; hi part of crc calculation

                                                 •

                                                 •

                                                 •

;--------------------------------------------------------
;               CRC16 subroutine.
;               - accumulator is assumed to have byte to be crc'ed
;               - two direct variables are used crc_hi and crc_lo
;               - crc_hi and crc_lo contain the CRC16 result
;--------------------------------------------------------
crc16:                                                          ; calculate crc with accumulator
                push    b                                       ; save value of b
                mov     b,                  #08h                ; number of bits to crc.
crc_get_bit:
                rrc     a                                       ; get low order bit into carry
                push    acc                                     ; save a for later use

                jc      crc_in_1                                ;got a 1 input to crc
                mov     c,                  crc_lo.0            ;xor with a 0 input bit is bit
                sjmp    crc_cont                                ;continue
crc_in_1:
                mov     c,                  crc_lo.0            ;xor with a 1 input bit
                cpl     c                                       ;is not bit.
crc_cont:
                jnc     crc_shift                               ; if carry set, just shift
                cpl     crc_hi.6                                ;complement bit 15 of crc
                cpl     crc_lo.1                                ;complement bit 2 of crc
crc_shift
```

```
mov     a,          crc_hi       ; carry is in appropriate setting
rrc     a                        ; rotate it
mov     crc_hi,     a            ; and save it
mov     a,          crc_lo       ; again, carry is okay
rrc     a                        ; rotate it
mov     crc_lo,     a            ; and save it

pop     acc                      ; get acc back
djnz    b,          crc_get_bit  ; go get the next bit

pop     b                        ; restore b
ret

end
```

ASSEMBLY LANGUAGE FOR CRC–16 USING A LOOKUP TABLE Table 4

```
crc_lo              data          40h       ; any direct address is okay
crc_hi              data          41h
tmp                 data          42h

                                       •

                                       •

                                       •

;-------------------------------------------------------
;             CRC16 subroutine.
;             - accumulator is assumed to have byte to be crc'ed
;             - three direct variables are used, tmp, crc_hi and crc_lo
;             - crc_hi and crc_lo contain the CRC16 result
;             - this CRC16 algorithm uses a table lookup
;-------------------------------------------------------
crc16:
              xrl     a,            crc_lo          ; create index into tables
              mov     tmp,          a               ; save index
              push    dph                           ; save dptr
              push    dpl                           ;
              mov     dptr,         #crc16_tablo    ; low part of table address
              movc    a,            @a+dptr         ; get low byte
              xrl     a,            crc_hi          ;
              mov     crc_lo,       a               ; save of low result

              mov     dptr,         #crc16_tabhi    ; high part of table address
              mov     a,            tmp             ; index
              movc    a,            @a+dptr         ;
              mov     crc_hi,       a               ; save high result

              pop     dpl                           ; restore pointer
              pop     dph                           ;
              ret                                   ; all done with calculation

crc16_tablo:
              db      000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
              db      001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
              db      001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
              db      000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
              db      001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
              db      000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
              db      000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
              db      001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
              db      001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
              db      000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
              db      000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
              db      001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
              db      000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
              db      001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
              db      001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
              db      000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
              db      001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
              db      000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
              db      000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
              db      001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
              db      000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
```

```
                db      001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
                db      001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
                db      000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
                db      000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
                db      001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
                db      001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
                db      000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
                db      001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
                db      000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
                db      000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
                db      001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
crc16_tabhi:
                db      000h, 0c0h, 0c1h, 001h, 0c3h, 003h, 002h, 0c2h
                db      0c6h, 006h, 007h, 0c7h, 005h, 0c5h, 0c4h, 004h
                db      0cch, 00ch, 00dh, 0cdh, 00fh, 0cfh, 0ceh, 00eh
                db      00ah, 0cah, 0cbh, 00bh, 0c9h, 009h, 008h, 0c8h
                db      0d8h, 018h, 019h, 0d9h, 01bh, 0dbh, 0dah, 01ah
                db      01eh, 0deh, 0dfh, 01fh, 0ddh, 01dh, 01ch, 0dch
                db      014h, 0d4h, 0d5h, 015h, 0d7h, 017h, 016h, 0d6h
                db      0d2h, 012h, 013h, 0d3h, 011h, 0d1h, 0d0h, 010h
                db      0f0h, 030h, 031h, 0f1h, 033h, 0f3h, 0f2h, 032h
                db      036h, 0f6h, 0f7h, 037h, 0f5h, 035h, 034h, 0f4h
                db      03ch, 0fch, 0fdh, 03dh, 0ffh, 03fh, 03eh, 0feh
                db      0fah, 03ah, 03bh, 0fbh, 039h, 0f9h, 0f8h, 038h
                db      028h, 0e8h, 0e9h, 029h, 0ebh, 02bh, 02ah, 0eah
                db      0eeh, 02eh, 02fh, 0efh, 02dh, 0edh, 0ech, 02ch
                db      0e4h, 024h, 025h, 0e5h, 027h, 0e7h, 0e6h, 026h
                db      022h, 0e2h, 0e3h, 023h, 0e1h, 021h, 020h, 0e0h
                db      0a0h, 060h, 061h, 0a1h, 063h, 0a3h, 0a2h, 062h
                db      066h, 0a6h, 0a7h, 067h, 0a5h, 065h, 064h, 0a4h
                db      06ch, 0ach, 0adh, 06dh, 0afh, 06fh, 06eh, 0aeh
                db      0aah, 06ah, 06bh, 0abh, 069h, 0a9h, 0a8h, 068h
                db      078h, 0b8h, 0b9h, 079h, 0bbh, 07bh, 07ah, 0bah
                db      0beh, 07eh, 07fh, 0bfh, 07dh, 0bdh, 0bch, 07ch
                db      0b4h, 074h, 075h, 0b5h, 077h, 0b7h, 0b6h, 076h
                db      072h, 0b2h, 0b3h, 073h, 0b1h, 071h, 070h, 0b0h
                db      050h, 090h, 091h, 051h, 093h, 053h, 052h, 092h
                db      096h, 056h, 057h, 097h, 055h, 095h, 094h, 054h
                db      09ch, 05ch, 05dh, 09dh, 05fh, 09fh, 09eh, 05eh
                db      05ah, 09ah, 09bh, 05bh, 099h, 059h, 058h, 098h
                db      088h, 048h, 049h, 089h, 04bh, 08bh, 08ah, 04ah
                db      04eh, 08eh, 08fh, 04fh, 08dh, 04dh, 04ch, 08ch
                db      044h, 084h, 085h, 045h, 087h, 047h, 046h, 086h
                db      082h, 042h, 043h, 083h, 041h, 081h, 080h, 040h
```

COMPARISON OF CALCULATION AND TABLE LOOKUP METHOD FOR CRC–16 Figure 7

Example:
CRC register starting value: 90 F1 Hex
Input Byte: 75 Hex

Calculation Method

Current CRC Value	Input
1001 0000 1111 0001	
0100 1000 0111 1000	1
0010 0100 0011 1100	0
1011 0010 0001 1111	1
1111 1001 0000 1110	0
1101 1100 1000 0110	1
1100 1110 0100 0010	1
1100 0111 0010 0000	1
0110 0011 1001 0000	0

New CRC Value = 63 90 Hex

Table Lookup Method

Current_CRC16_Lo = F1 Hex
Current_CRC16_Hi = 90 Hex
Input byte = 75 Hex

Tabhi Index= (Current_CRC16_Lo) EXOR (Input byte)
= F1 EXOR 75= 84 Hex = 132 Dec
New_CRC16_Hi = Tabhi[132] = 63 Hex (from Table 4.)

Tablo Index = (Current_CRC16_Lo) EXOR (Input byte) = 132 Dec
Tablo[132] = 00 Hex (from Table 4.)
New_CRC16_Lo = Tablo[132] EXOR (Current_CRC16_Hi)
= 00 EXOR 90 = 90 Hex

New CRC Value = 63 90 Hex

An interesting intermediate method is presented in Table 5. The code will generate a CRC–16 value for each byte input to it by operating on the entire current CRC value and the incoming byte using the equations shown in Figure 8. The derivations for the equations are also shown, using alpha characters to represent the current 16–bit CRC value and numeric characters to represent the bits of the incoming byte. The result after eight shifts yields the equations shown. These equations can then be used to precompute large portions of the new CRC value. Notice, for example, that the quantity ABCDEFGH01234567 (defined as the EXOR of all of those bits) is the parity of the incoming data byte and the low order byte of the current CRC. This method reduces computation time and memory space as compared to both the bit–by–bit and lookup table methods described above. Finally, two properties of the CRC–16 function that can be used as test cases are mentioned as an aid to debugging the code for any of the previous methods.

The first property is identical to the DOW CRC case. If the current 16–bit contents of the CRC register are also used as the next 16–bits of input, the resulting CRC value is always 0000 Hex. A second property of the CRC–16 function is also similar to the DOW CRC case, if the 1's complement of the current 16–bit contents of the CRC register are also used as the next 16–bits of input, the resulting CRC value is always B0 01 Hex. The proof for these two CRC–16 properties follows in an analogous way to the proof for the DOW CRC case.

REFERENCES:

Stallings, William, Ph.D., Data and Computer Communications. 2nd ed., New York: Macmillan Publishing. 107-112.

Buller, Jon, "High Speed Software CRC Generation", EDN, Volume 36, #25, pg. 210.

ASSEMBLY LANGUAGE PROCEDURE FOR HIGH–SPEED CRC–16 COMPUTATION Table 5

```
lo     equ        40h                                  ; low byte of CRC
hi     equ        41h                                  ; high byte of CRC
                                                              •

                                                              •

                                                              •

crc16:
       push       acc                                  ; save the accumulator.

       xrl        a,        lo
       mov        lo,        hi                        ; move the high byte of the CRC.
       mov        hi,       a                          ; save data xor low(crc) for later
       mov        c,         p
       jnc        crc0
       xrl        lo,        #01h                      ; add the parity to CRC bit 0
crc0:
       rrc        a                                     ; get the low bit in c
       jnc        crc1
       xrl        lo,        #40h                      ; need to fix bit 6 of the result
crc1:
       mov        c,        acc.7
       xrl        a,        hi                         ; compute the results for other bits.
       rrc        a                                    ; shift them into place
       mov        hi,       a                          ; and save them
       jnc        crc2
       xrl        lo,       #80h                       ; now clean up bit 7
crc2:
       pop         acc                                 ; restore everything and return
       ret
```

HIGH–SPEED CRC–16 COMPUTATION METHOD Figure 8

CURRENT CRC VALUE = XWUT SRQP HGFE DCBA
INPUT BYTE = 7654 3210

NOTATION: ABC = A EXOR B EXOR C

DEFINITION: DEF EXOR D = (D EXOR D) EXOR EF = 0 EXOR EF = EF

REGISTER STAGE (SEE FIGURE 6 FOR OPERATION)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	INPUT
X	W	U	T	S	R	Q	P	H	G	F	E	D	C	B	A	
A0	X	WA0	U	T	S	R	Q	P	H	G	F	E	D	C	AB0	0
AB01	A0	AB01X	WA0	U	T	S	R	Q	P	H	G	F	E	D	ABC01	1
ABC012	AB01	BC12	AB01X	WA0	U	T	S	R	Q	P	H	G	F	E	ABCD012	2
ABCD0123	ABC012	CD23	BC12	AB01X	WA0	U	T	S	R	Q	P	H	G	F	ABCDE0123	3
ABCDE01234	ABCD0123	DE34	CD23	BC12	AB01X	WA0	U	T	S	R	Q	P	H	G	ABCDEF01234	4
ABCDEF012345	ABCDE01234	EF45	DE34	CD23	BC12	AB01X	WA0	U	T	S	R	Q	P	H	ABCDEFG012345	5
ABCDEFG0123456	ABCDEF012345	FG56	EF45	DE34	CD23	BC12	AB01X	WA0	U	T	S	R	Q	P	ABCDEFGH0123456	6
ABCDEFGH01234567	ABCDEFG0123456	GH67	FG56	EF45	DE34	CD23	BC12	AB01X	WA0	U	T	S	R	Q	ABCDEFGHP01234567	7

THIS YIELDS THE FOLLOWING DEFINITIONS:

NEW CRC REGISTER VALUES AFTER EIGHT SHIFTS

Xnew = ABCDEFGH01234567
Wnew = ABCDEFG0123456 = ABCDEFGH01234567 H7
Unew = G6H7
Tnew = F5G6
Snew = E4F5
Rnew = D3E4
Qnew = C2D3
Pnew = B1C2
Hnew = A0B1X
Gnew = A0W
Fnew = U
Enew = T
Dnew = S
Cnew = R
Bnew = Q
Anew = P ABCDEFGH01234567

APPENDIX 2
USE OF ADD–ONLY iButton FOR SECURE STORAGE OF MONETARY EQUIVALENT DATA

I. INTRODUCTION

The fare payment system used by the Bay Area Rapid Transit (BART) system in San Francisco is an example of an application in which monetary equivalent data is read and written electronically. In this system, the user can obtain a transit ticket and deposit any desired amount of money into it from an automatic vending machine. The information is stored in the ticket magnetically in the form of encoded data written on a magnetic stripe. Each time the user travels from one place to another, the system deducts the fare from the amount represented by the magnetically encoded data, thus reducing the value of the ticket. When the value of the ticket is nearly exhausted, it can be restored to a high value by inserting it again into an automatic vending machine and depositing additional funds.

The BART system eliminates the need for handling money and making change at the point of entry to the transit system, thereby reducing labor costs and increasing efficiency. A similar advantage can be realized in many other circumstances where an electronically readable and alterable "token" can eliminate the costs and delays associated with money handling at the point of use. Such a token might therefore be used as a meal ticket on a college campus, as a ride ticket at an amusement park, or wherever tickets or tokens are now used to speed monetary payments and/or eliminate unnecessary labor.

The system described above suffers from three significant disadvantages:

A. Paper tickets with magnetic stripes deposited on them are subject to wrinkling or tearing which can cause loss of the monetary equivalent data. Also, the magnetic stripes are subject to erasure by environmental magnetic fields, even if the paper carrier and magnetic material are physically intact.

B. Since magnetic recording is a read/write technology, it is possible for a technologically sophisticated person to read the contents of the magnetic stripe when the ticket has a large monetary value, use the ticket until the value is nearly gone, then rewrite the original data into the ticket to restore its original value. It is not necessary for the person to understand the encoding of the monetary data in order to do this. Therefore, the use of a read/write technology makes the tickets vulnerable to counterfeiting.

C. The magnetic recording technology requires uniform motion of the magnetic material across the read/write heads in order to read and write data reliably. This makes it necessary to use a relatively complex mechanical ticket–handling mechanism to read, debit, and rewrite the monetary equivalent data.

II. ADD–ONLY iButton AS AN ALTERNATIVE TECHNOLOGY FOR MONETARY TOKENS

Add–Only iButton provides a viable alternative technology for storage of monetary equivalent data which delivers the advantages described above but does not suffer from the disadvantages. iButton is sealed in a durable stainless steel Microcan which protects it against environmental damage. Reading and writing data is accomplished with a momentary contact to a simple electrical probe, requiring no sophisticated mechanical handling mechanisms. The add–only attribute of Add–Only iButton provides protection against counterfeiting, since the data in these memories can never be restored to its original value once it has been modified.

Add–Only iButton contains many bits of information, with each bit having either a one or a zero value. Initially, all the bits in the memory are ones. The read/write probe can read these bits, and it can also selectively change one or more of the bits to zero. Once a bit has been changed to a zero, it cannot be changed back to a one. Writing a bit is therefore much like punching a hole in a meal ticket card. The electrically alterable bits are organized into memory pages having 256 bits each. In addition to these electrically alterable bits, each iButton also contains a unique 64 bit registration number which cannot be altered. No two iButtons ever have the same registration number. Finally, each page has a status register that can be read to determine which pages have been used up, and error detection circuitry (CRC) which allows the reader to determine if it has read the data correctly.

With this feature set, it is possible to design a system in which monetary equivalents can be added to or removed from the part many times before it must be replaced, and which is highly resistant to counterfeiting. The basic principle of this system is described below.

III. ELECTRONIC CREDITING AND DEBITING OF ADD–ONLY iButton

One possible technique which allows storage of credits and debits in Add–Only iButton is as follows. Monetary units are added by changing one bits to zero bits starting from the least significant bit of each page and progressing toward the most significant bit. Monetary units are debited by changing one bits to zero bits starting from the most significant bit of each page and progressing toward the least significant bit. As the memory is repeatedly debited and credited, the rows of zero bits grow toward the middle of the page. When they meet, the page is marked as exhausted with the status byte and the process continues on the next page. (It is possible to ignore pages and treat the entire memory as a single page, but that would require the reading of the entire memory, increasing the time needed to complete a transaction. The electronic read/write process is more efficient when only a portion of the stored data needs to be read). With this technique, assuming the credit units and debit units have equal value, a 1024 bit memory could credit and debit 512 monetary units before it was used up. If credit units are taken to represent some multiple of the debit unit, then more debits are allowed. (For example, if each credit unit is the equivalent of three debit units, then a 1024 bit memory would allow 768 debits).

The problem with the simple system described above is that anyone with the necessary knowledge and equipment to read and write data in Add–Only iButton can easily increase the value by adding additional credit units. This is possible because there is a direct, straightforward correspondence between a bit location and its value. If the bits were scrambled (permuted) in an apparently random manner, it would no longer be possible to determine how to add credit units to the memory. For example, if 15 bits on a page are still set to one, only one of these bits will add a credit unit to the memory. Similarly, only one of the bits will add a debit unit to the memory. If any one of the other 13 bits were written to zero, it would appear out of sequence and would signify that the memory had been tampered with, thereby invalidating it. Therefore, if a person guesses which bit to write next, he has one chance in 15 of adding a credit unit, one chance in 15 of adding a debit unit (decreasing the value), and 13 chances in 15 of invalidating the memory and flagging it as having been subject to tampering. Although there is a chance of guessing correctly which bit to change, the laws of probability are stacked against this event. (This is the kind of statistical analysis that makes lotteries predictable).

The unique registration number in each iButton can be used to permute the bits in each part differently, so that one cannot determine by studying the data in one part how to add credit units to a different part. Many different techniques are possible to determine a unique bit permutation from the unique registration number supplied with each part. A few of these techniques are described below.

IV. CALCULATING BIT PERMUTATIONS FROM UNIQUE REGISTRATION NUMBERS

The number of different permutations of the 256 bits in each page is very large, approximately ten to the power of 507. Only a minute fraction of these permutations can be enumerated with the unique registration number, since the registration number represents a range of 281 trillion unique numerical values, or about ten to the power of 14. The permutations that can be derived from the unique registration number are thus buried in the much larger population of possible permutations. 281 trillion is in fact an extremely large set of unique registration numbers that is sufficient for all practical purposes. The enormously larger number of different permutations greatly multiplies the task of deducing the permutation from the registration number. To select a permutation based on the registration number from this enormous population, the following method could be used.

A. Replace the CRC in the publicly readable registration number with the page number of the page to be scrambled and then encrypt it with a standard block cypher encryption algorithm (such as DES), using a secret encryption key. This produces a 64 bit encrypted number which is unique to each page of each part and is known only to the reader.

B. Divide the 64 bit encrypted number by 256 to obtain a quotient and a remainder that lies in the range 0–255. The value of the remainder gives the position of bit 1 in the scrambled data, and leaves 255 other bit positions unfilled.

C. Divide the quotient from step B by 255 to obtain another quotient and a remainder that lies in the range 0–254. The value of this remainder gives the position of bit 2 in the remaining 255 bits that were unfilled after step B.

D. Repeat step C for each successive bit, decreasing the divisor by 1 each time, until all 64 bits have been placed in their scrambled positions. Each time the quotient reaches zero during this process, replace it with the original encrypted number from step A.

The steps B – D above are numerically intensive and may not be suitable for microcontroller–based equipment. A simpler but less secure technique is to start with an initial, secret, randomly chosen permutation and then further permute it based on the 64 bit encrypted number by interchanging certain bits or not depending on whether a bit in the encrypted number is a one or a zero. This method provides a simpler set of permutations but may still provide adequate security in many applications. The complexity of the technique used to derive permutations from the unique registration number can be selected based on the degree of security needed in the application and the amount of computing power available in the equipment.

V. DESCRIPTION OF OPERATION

Using the methods described above, the automatic debiting equipment operates as follows to decrease the value of the memory by one monetary unit:

A. The equipment detects an Add–Only iButton by means of the presence pulse that it generates, reads the unique registration number, and checks its validity with the CRC.

B. The equipment reads the status registers to find the first page that has not been used up. It then reads that page, making use of the built–in CRC calculation circuitry to confirm the validity of the read.

C. Using a secret encryption key that can be changed periodically, the equipment applies a standard encryption algorithm (such as DES) to the unique registration number (with the CRC replaced with the active page number) to generate a unique secret number, and then uses this number to reorder the bits read from the active page using any of the techniques described in section IV above.

D. After the above reordering, the zero bits starting from the least significant bit represent credits, and the zero bits starting from the most significant bit represent debits. The data, beginning with the least significant bit, should therefore appear as an unbroken sequence of zero bits (credits), followed by an unbroken sequence of one bits (not yet used), followed by an unbroken sequence of zero or more zero bits (debits). The equipment checks the integrity of these three sequences. If there is a break in any of these sequences or if the number of debits exceeds the number of credits, then there is evidence of tampering and the equipment may take appropriate action (such as recording the registration number, or even sounding an alarm or summoning an official).

E. If the number of credits is greater than the number of debits, the equipment adds one more zero bit to the unscrambled sequence, checks to make sure that the page has not been used up, and then uses the bit permutation in reverse to determine where the debit bit falls in the original scrambled bit sequence. Any time a page is filled, the equipment writes the status bytes to mark the page as used up and proceeds to the next page.

F. The equipment performs a write operation to write the bit identified in step E above from a one to a zero, then reads back the page to make sure that the write operation was completed correctly. When a successful write of the debit bit is detected, the equipment activates a peripheral device (passenger gate, etc.) to signal a completed, successful operation.

The operation of the crediting equipment is similar to that described above. The crediting equipment receives cash from the user and sets one or more credit bits to zero to indicate the amount of added value. When a page is half full of credit bits, the equipment proceeds to the next page to add additional credits. The bits are written in the scrambled order so that it is impossible to distinguish the credit bits from the debit bits and the bits that have not yet been used.

Both the debiting and crediting equipment can make use of a secure microprocessor (such as the DS5002 secure micro) so that even if the equipment is stolen, it cannot be made to reveal the secret encryption key which is used in step C above. This makes it possible to limit the knowledge of this information to a very small number of individuals. It is important to note that a blank Add–Only iButton has no monetary value until it his been credited with monetary equivalents using its unique bit scrambling algorithm. Therefore, there is no advantage to a counterfeiter to obtain a supply of blank iButtons, and it is unnecessary to take special precautions to safeguard these supplies.

Assuming that a high–performance processor is used so that the time required to perform the calculations described above can be neglected, the minimum time required for a debiting transaction is the time required to read the unique registration number, read the status bytes, read the appropriate page, and write out the bit that represents the debit. This time, equal to 31.7 milliseconds, is scarcely perceptible and would be regarded as essentially instantaneous by the user.

VI. SUMMARY

Add–Only iButton has the following special characteristics which make it uniquely suitable for applications requiring secure crediting, debiting, and portable storage of monetary equivalent data:

A. A unique, unalterable registration number which allows the data on each different part to be encrypted differently. This makes it impossible to determine how to counterfeit a part by studying how data is written into a different part.

B. Random–access memory which is one–way alterable, that is, having bits that can be changed from a one to a zero but not from a zero back to a one. This makes it impossible to write into a part the data pattern it held earlier when it was more valuable. (This type of memory is commonly referred to as one–time–programmable EPROM, but this terminology is misleading in the current application because it suggests that the part can be written only once).

C. A small, durable Microcan package with a simple electrical connection, allowing data to be read or written with a momentary contact.

APPENDIX 3
EXAMPLE OF iButton USAGE IN A BANKING APPLICATION

In the example described below, a Memory iButton is used to emulate an American Banking Association (A.B.A.) credit card. Credit cards have a three–track magnetic stripe that can hold up to 1288 bits of information. The data from the three tracks can be stored in separate files of the Extended File Structure, designated IATA.0, ABA.0, and ATM.0. The information is encoded in these files in exactly the same format as the corresponding magnetic stripes. This facilitates the direct substitution of an iButton reader/writer in place of a magnetic stripe reader/writer. The following specifications describe the structure of the data in the three files representing the three magnetic stripes. The LRC character has even parity tracks 1 through 3 conform to the data layout and encoding described by the ISO 3554–1976 (E) Standard with the following two exceptions: a) leading zeros to the Start Sentinel character and zeros following the Longitudinal Redundancy Check (LRC) character are not included in the data; b) clocking bits are not included in the data.

IATA.0 FILE FORMAT AND DESCRIPTION (TRACK 1)

This track contains up to 79 characters of data encoded in a six–bit character set (Table 1) with an odd parity bit. This implies that a maximum of 553 bits of information are contained on this track in 70 bytes. Note that the least significant bit (LSB) is loaded first, followed by the other data bits and its parity bit. The unused bits in the last data byte are zero-filled. In a track that has all 79 characters in it, the seven most significant bits of byte 70, the last byte, are zero–filled and not used.

IATA FORMAT A

Field	**Length**
Start Sentinel	1
Format Code = ‘A’	1
Record:	
Surname	
”/”	
First Name	
” ”	
Title	
” ”	2 to 26 characters
Separator	1
Discretionary Data	balance up to maximum record length
End Sentinel	1
LRC	1
	Maximum of 79 characters

IATA FORMAT B

Field	**Length**
Start Sentinel	1
Format Code = ‘B’	1
Account Number	up to 19 characters
Record:	
Surname	
”/”	
First Name	
” ”	
Title	
” ”	2 to 26 characters
Separator	1
Discretionary Data	balance up to maximum record length
End Sentinel	1
LRC	1
	Maximum of 79 characters

ABA.0 FILE FORMAT AND DESCRIPTION (TRACK 2)

This track contains up to 40 characters of data encoded in a four–bit character set (Table 2) with an odd parity bit. This implies that a maximum of 200 bits of information are contained on this track in 25 bytes. Note that the least significant bit (LSB) is loaded first followed by the other data bits and its parity bit. The unused bits in the last data byte are zero-filled. In a track that has 39 characters in it, the five most significant bits of byte 25, the last byte, are zero–filled and not used.

ABA FORMAT DESCRIPTION

Field	Length
Start Sentinel	1
Account Number	up to 19 characters
Separator	1
Discretionary Data	balance up to maximum record length
End Sentinel	1
LRC	1
	Maximum of 40 characters

ATM.0 FILE FORMAT AND DESCRIPTION (TRACK 3)

This track contains up to 107 characters of data encoded in a four–bit character set (Table 2) with an odd parity bit. This implies that a maximum of 535 bits of information are contained on this track in 67 bytes. Note that the least significant bit (LSB) is loaded first followed by the other data bits and its parity bit. The unused bits in the last data byte are zero–filled. In a track that has 107 characters in it, the most significant bit of byte 67, the last byte, is zero–filled and not used.

AUTOMATIC TELLER FORMAT DESCRIPTION

Field	Length
Start Sentinel	1
Primary Account Number	up to 19 characters
Field Separator	1
End of Cycle Date	4
Amount Remaining	3
Authorized Amount/Cycle	3
Number Days/Cycle	2
Validity Date	4
Expiration Date	4
Service Restrictions	4
Card Member	1
Type Algorithm	2
Account Number ID	3
Offset to Secret Code	4
Retry Count	1
Security Method and Code	9
Discretionary Data	Primary Account Number + Discretionary Data < 60
End Sentinel	1
LRC	1
	Maximum of 107 characters

TABLE 1

Decimal Value	Graphic
0	’ ’
1	①
2	①
3	②
4	’ $ ’
5	③
6	①
7	①
8	’ (’
9	’) ’
10	①
11	①
12	①
13	’ – ’
14	’ . ’
15	’ / ’

Decimal Value	Graphic
16	’ 0 ’
17	’ 1 ’
18	’ 2 ’
19	’ 3 ’
20	’ 4 ’
21	’ 5 ’
22	’ 6 ’
23	’ 7 ’
24	’ 8 ’
25	’ 9 ’
26	①
27	①
28	①
29	①
30	①
31	’ ? ’③

Decimal Value	Graphic
32	’ @ ’
33	’ A ’
34	’ B ’
35	’ C ’
36	’ D ’
37	’ E ’
38	’ F ’
39	’ G ’
40	’ H ’
41	’ I ’
42	’ J ’
43	’ K ’
44	’ L ’
45	’ M ’
46	’ N ’
47	’ O ’

Decimal Value	Graphic
48	’ P ’
49	’ Q ’
50	’ R ’
51	’ S ’
52	’ T ’
53	’ U ’
54	’ V ’
55	’ W ’
56	’ X ’
57	’ Y ’
58	’ Z ’
59	④
60	④
61	④
62	’^’
63	①

① This character position is used for hardware control only.

② This character position is reserved for an additional graphic.

③ These characters have the following meanings:
character 5, ’ % ’ : represents the “Start Sentinel”,
character 31, ’ ? ’ : represents the “End Sentinel”,
character 62, ’ ^ ’ : represents the “Separator”.

④ This character position is reserved for additional national characters when required. Not to be used internationally.

TABLE 2

Decimal Value	Graphic
0	’ 0 ’
1	’ 1 ’
2	’ 2 ’
3	’ 3 ’
4	’ 4 ’
5	’ 5 ’
6	’ 6 ’
7	’ 7 ’
8	’ 8 ’
9	’ 9 ’
10	④
11	①
12	④
13	②
14	④
15	③

① “Start Sentinel”

② “Separator

③ “End Sentinel”

④ This character position is used for hardware control only.

APPENDIX 4
MANAGING CONCURRENT INTERRUPTS
IN iButton I/O SOFTWARE

All communication with iButtons are originated by a master using a bit–synchronous, half–duplex, 1–Wire serial link. Each iButton contains a self–timed serial communication controller which transmits or receives each bit within a specified period of time after the low–going edge originated by the master. However, the time interval between bits is determined by the master and may be arbitrarily long, allowing competing interrupts from other devices to be serviced between any two bits. Communication with iButtons is accomplished with falling edge activated time slots lasting 60 µs and with a reset/presence signal having a low and a high period each lasting 480 µs. These time–dependent signals are generated and detected by the software procedures TouchReset and TouchBit, which are usually coded in assembly language because they must produce short, accurate time intervals. Copies of these routines for a variety of microprocessors are available from Dallas Semiconductor.

It might appear from the timing requirements of iButton communication that iButton I/O would monopolize the time of the microprocessor, leaving no time to service competing interrupts from timers or other devices. In fact, this is not the case. When properly written, the TouchReset and TouchBit procedures inhibit interrupts for only a few microseconds at a time, allowing ample opportunity to service other interrupting devices. The following text describes typical timing constraints imposed by the requirements of interrupt service and the proposed method of dealing with them while performing iButton I/O operations.

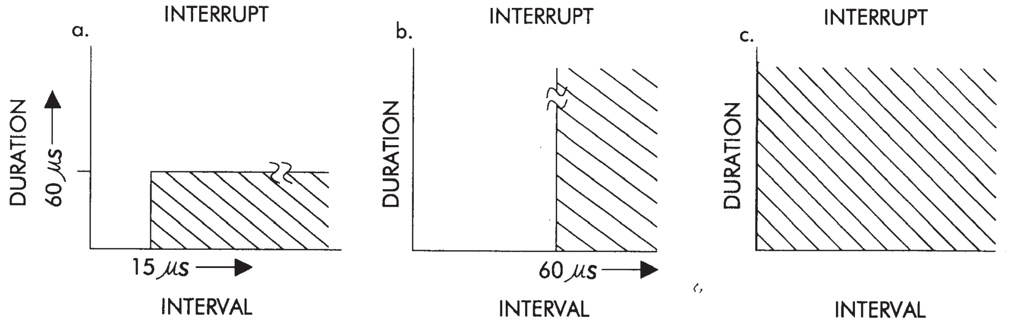
The first case to be considered is one in which interrupts generated by competing events may occur at intervals as short as 15 µs, but the interrupt service procedure requires less than 60 µs to execute. In this case, as depicted in Figure 1a, the interrupt system is left enabled at all times except during the first 15 microseconds of a write–one or read time slot. The interrupt system must be disabled while the 1–Wire Bus is driven low, released, and sampled, in order to insure that these operations all take place in a time of 15 µs or less. (The minimum allowable time between interrupts can be reduced by using less than 15 µs for these operations, but this fails to take advantage of the available recovery time and may affect performance of 1–Wire communication with long wire lengths or large numbers of iButtons on the bus.) Since interrupt services require less than 60 µs and every logic state in the response period lasts at least 60 µs, the TouchReset procedure can sample the 1–Wire Bus at a sufficiently high rate that no important characteristics of the response signal are missed. (Note that if more than half the time is spent servicing interrupts, the TouchBit procedure may need to reference an independent timer to produce a correctly timed write–zero signal, in order to insure that the write–zero signal is not extended to 120 µs and misinterpreted as a reset signal.) Figure 1a depicts the range of interrupt intervals and durations over which this software solution may be applied.

In the second case, as depicted in Figure 1b, the interrupts occur at intervals as short as 60 µs and the interrupt service procedures may require more or less than 60 µs to execute. This case is handled by disabling interrupts during the entire I/O time slot and also during the critical period of the TouchReset signal. The critical period is the 60 µs period immediately following the release of the 1–Wire Bus, after it has been held low for at least 480 µs. During the critical period, the procedure watches for the 1–Wire bus to go high and low again (presence pulse), to go high and remain high (no presence pulse), or to remain low (short circuit). The remainder of the TouchReset procedure is executed with interrupts enabled. Figure 1b depicts the range of interrupt intervals and durations over which this software solution may be applied.

In the event that the time interval between interrupts from the same competing source is so unpredictable that the 15 µs limit cannot be guaranteed, it is still possible to communicate successfully with an iButton if the typical interval between interrupts is long enough to allow some communication packets to be sent or received successfully. To use this method, the interrupt system is enabled at all times, allowing competing interrupts to operate normally. Assuming a competing interrupt is randomly distributed in time, it has approximately one chance in ten of causing a bit error in either a read or a write operation. It is therefore still possible, if the average competing interrupt rate is not excessive, to get a CRC–checked packet transmitted and confirmed without errors. iButton communication protocols are inherently error–tolerant to provide reliable communication with uncertain electrical contact, and this mechanism can also be used to compensate for bit errors introduced by competing interrupts. The efficiency of this technique depends on both the average interrupt rate and the packet size. That is, as the interrupt environment becomes more contentious, the effective data transfer rate goes down.

If interrupt timing is so unfavorable that none of the conditions described above are satisfied, the timing problems can still be eliminated completely by use of a hardware interface which is capable of performing the critical timing and buffering independently of the software. This is the case for communication using the PC serial port, where an 8250 or 16C450 UART is used to generate the critical timing and detect the response. In this case, as depicted in Figure 1c, TouchReset and TouchBit can be written in a high–level language and interrupts can be left enabled at all times. This is the simplest system to design since there is no possibility of interfering with other interrupt–driven processes. The 8250 and 16C450 UARTs have more capability than is needed for this task, and a much simpler bi–directional single–bit buffer with hardware–generated timing could suffice. Most UARTs capable of operation at 115200 bps or above can be used to handle the timing when performing iButton I/O. Recently introduced palmtop computers and PDAs like the Apple Newton, the HP100, and the AST/Tandy/Casio Zoomer all support the 115200 bps rate. Figure 1c depicts the range of interrupt intervals and durations over which this software solution may be applied.

1–WIRE COMMUNICATION IN A SHARED PROCESSOR ENVIRONMENT Figure 1
(SHADED AREA REPRESENTS ASSURED COMMUNICATION)


a.
INTERRUPT
DURATION
60 μs
15 μs
INTERVAL
b.
INTERRUPT
DURATION
60 μs
INTERVAL
c.
INTERRUPT
DURATION
INTERVAL