Application Note 159
Ultra-Reliable 1-Wire Communications

INTRODUCTION

Dallas Semiconductor 1-Wire® devices communicate using a single data line and well-defined, time-tested protocols. These include special provisions to handle highly intermittent ("touch") connections that are common when using 1-Wire devices in the iButton® form factor. As 1-Wire devices expand into applications involving monetary value and high-security identification, the demand for highly reliable communications has grown. This Application Note will discuss techniques that can be used by programmers to make 1-Wire communications "ultra-reliable", and when these techniques should be considered.

This paper will focus on the iButton form of 1-Wire devices, as these devices are most often used in touch contact applications where communications can be most difficult. Despite the direct references to iButton devices, however, these guidelines apply generally to all 1-Wire devices in all packages. Embedded (microcontroller) applications are the focus of this discussion, although most of these techniques could apply in any 1-Wire bus master situation.

This paper assumes that the reader is already familiar with the Dallas 1-Wire bus protocol and basic 1-Wire device communications methods and algorithms. (See *AN153, Embedded 1-Wire Bus Master Programming Guide,* for more information.)

RELIABLE iButton COMMUNICATIONS

General 1-Wire device communication usually involves functions such as searching (to identify the devices present on the bus), reading device network addresses (serial numbers), reading device data or status, and writing memory or control data. Software performing these operations must be prepared to handle intermittent connections, user errors and bus short-circuits, external interference or loss-of-contact with the device at any point while an operation is being performed.

In some cases, communication failures are simple for the software to detect and rectify. When a failure occurs while reading an iButton, for example, the software may attempt to read the iButton again, or may rely on the user to remove and represent the iButton to instigate a new transaction. A failure reading data from an iButton often causes only a minor delay or annoyance to the user. A failure while writing to the device, however, can be much more serious. If an iButton contains monetary data, for example, debits (purchases) usually include writing revised monetary amounts to the iButton each time a purchase is made. Should one of these updates fail and leave the iButton data corrupted, the user will have lost their entire monetary balance—a generally unacceptable outcome, even when it happens very rarely.

A precise definition of "ultra-reliable" is difficult. As a rule, most normal 1-Wire bus data exchanges are reasonably reliable. Reading a device network address (serial number) using a READ ROM sequence, if followed by a check of the family code and CRC is reasonably reliable (see *FAMILY CODE AND CRC-8* later in this document). Even if the probe is bombarded by the worst-case kind of interference (random bits), the odds that a bogus iButton address would get through undetected are only about 1-in-256. Because the typical iButton-to-probe environment is considerably better than worst-case, the odds are

much lower than this. Nonetheless, some applications cannot tolerate even the occasional misread because the consequences would be too great.

For the sake of this document, we will consider ultra-reliable to be on the order of the same likelihood that a 64-bit iButton network address, read twice in a row and compared bit-for-bit, could be misread. In the presence of the worst possible type and amount of interference (continuous random bits), the odds of this are about 1-in-4,230,000,000 (about 1-in-SQRT(2^{64})). An operation will be considered ultra-reliable if, in the face of worst-case interference, it meets or exceeds this level of reliability.

WHEN TO GO ULTRA-RELIABLE

It is important to remember that ultra-reliable techniques are not always required. Normal software methods that have reasonably low odds of allowing an error to slip through are sufficient for most applications. However, when the consequences of a communication error are significant, or the psychological effect of errors would damage customer confidence, then ultra-reliable methods should be employed. Some examples are as follows:

In a monetary (eCash) scheme, a misread of the iButton address (serial number) would cause a complete failure of all the subsequent device accesses, as well as failure of the cryptographic checks that are (or should be) cryptographically bound to the device identity. The impact of a rare misread of the device identity is only a temporary denial of a product or service. When the user re-presents the iButton the transaction will complete normally, so the impact on the customer is minimal. In this case, the odds of a bogus transaction due to an iButton misread are astronomically small, so ultra-reliable methods to read the iButton address are probably not required.

Later in the transaction process, however, when the software is attempting to read the customer's account balance from the iButton, the tables are turned. At this point, an error could create or destroy monetary value, and creating or destroying money is entirely unacceptable, even if it occurs very rarely. Even worse, a failure while updating the balance in the iButton (during a write) could do permanent damage by leaving the customer iButton data corrupted and unusable in future transactions. Ultra-reliable methods are required where the consequences of an error are serious, and can be sacrificed where the consequences of an error are minor.

Another (often overlooked) reason for employing ultra-reliable techniques involves customer confidence. In an access control system based on iButton addresses (serial numbers), the odds that an invalid iButton key will be presented at a lock and be misread are small (less than 1-in-256 even in the very worst case situations). The odds that the misread iButton address will happen to match a valid key for that door are astronomically small (less than one-in-many-billions). Security is not an issue due to these immense odds, so one would assume that ultra-reliable methods are not required. However, consider the more frequent case where a valid iButton key is misread and the user is denied entry. The user will have only to try again, so this is only a minor annoyance and occurs very rarely. The security manager looking over the access logs, however, sees what appears to be an attempt by an invalid iButton key to gain entrance, which is cause for alarm. Sometimes ultra-reliable techniques are necessary for customer confidence even if not critical for secure system operation.

These examples demonstrate situations where ultra-reliable techniques should be considered. The following sections will discuss various iButton operations and methods to make them ultra-reliable. Many approaches might be employed to recover from communication errors. The methods described in the following sections are some of those that have proven successful in real-world applications. The programmer should use these ideas, extending them to other iButton operations that are not specifically addressed in this document.

THE NATURE OF THE 1-WIRE BUS

The 1-Wire bus is an inverted-logic wired-OR arrangement. This means that any of a number of things can take the bus to a low level. This includes electrical short-circuits caused by the steel iButton container improperly seated in the probe, reset pulses and time slots generated by the master, presence pulses generated by arriving iButton devices, and presence pulses or zero bits generated by other devices that might be present on the bus. (Not to mention rainwater, tampering attempts, etc.)

The 1-Wire bus is pulled down by low-impedance drivers in the master and slave devices, and returns to a high level due to weaker pull-up currents provided by the master. For an external source of interference to turn a 1 bit into a 0 bit, all that is required is a short circuit across the bus. For a 0 bit to be turned into a 1 bit, the interference would require sufficient energy to overcome the low-impedance driver that is generating the 0 bit. For this reason, errors more frequently involve *one* bits turning into 0 bits.

However, error modes are not always this simple. A noise pulse (of either polarity) that causes a slave device to be falsely clocked (i.e., that appears to the slave as an extra time slot) will cause all the subsequent bits to be returned out-of-step with what the master expects. Loss of contact when the slave is returning a 0 bit will also result in a 1 bit being read by the master. The programmer should expect that bit errors of all types (inverted bits, dropped bits, extra bits) are possible and must be dealt with. Good programming practices demand that the bus always be assumed unreliable, and that every precaution be taken to assure valid data communication.

The tools that the programmer may use to assure reliable data exchange despite the nature of the 1-Wire bus include:

- CRC-8 and CRC-16 checks generated by the hardware
- Embedded checksums or CRC checks included in the data
- Returned-bit checks on bus writes
- Multiple (redundant) reads and read-back of data that has been written
- Cryptographic validation of data and device ID

ERROR RECOVERY AND RETRIES

Programs that perform 1-Wire bus communications must have a well thought-out plan for handling retries when failures occur. Whenever a 1-Wire function fails, it can usually be attempted again, but the programmer must consider all of the things that could have occurred during the failed attempt. When contact is lost or the 1-Wire bus is shorted, the iButton device may no longer be addressed (selected). If the device had been placed into Overdrive (high speed) communication mode, it may have been reset to standard speed by the interruption. Subsequent attempts to retry an operation at Overdrive speed will fail because the device is in the incorrect speed mode.

In some situations, the number of retry attempts may be limited so that time will not be wasted in retires when what is needed is to start the transaction over from the beginning.

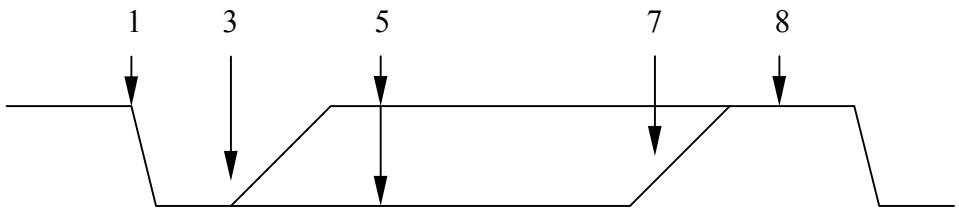
EARLY ERROR DETECTION

Because the iButton touch contact is intermittent and workable electrical contact may occur only for short periods, detecting errors as early as possible in the course of the transaction can make a substantial difference in performance. Detecting an error in the first few bytes of a communication sequence will allow the software to abort the exchange and start a retry attempt sooner. An error that is not detected until a long transfer of data is completed means that much precious time has been wasted in what has been essentially a doomed operation.

Most embedded 1-Wire subroutines that manually generate time slots (bits) on the bus perform the functions of reading and writing at the same time. For the most part, a read time slot is just a write-1 time slot where the slave device turns the 1 bit into a 0 bit as required to return the desired data. Typical bit-level (read and write) master subroutines are structured as follows (see reference waveform sketch below):

1) Take the bus LOW to start the time slot.
2) Wait 6µs.
3) Output the bit to be written to the bus. (If a 1, release the bus and allow it to return high. If a 0, continue to hold the bus low.)
4) Wait 9µs.
5) Sample the level on the bus (Low level = 0, high level = 1) @15µs after start of slot.
6) Wait 45µs.
7) Allow the bus to float high (ends any write-0 in progress) @60µs after start of slot.
8) Wait 10µs for bus recovery time.
9) Return the sampled bit.

The resulting waveform is similar to below (numbers refer to steps above):



This subroutine always performs the same basic 1 or 0 time slots. If the data provided to send out is a 0, it will generate a write-0 time slot where the bus remains low for 60µs. If the data to be written is a 1, then the time slot could be a write-1 or it could be a read, where the master takes the bus low for 6µs and then releases it. Each time a bit is written to the bus, the bus state is also read back and returned to the caller, so the same subroutine is used to perform bus writes as for bus reads. Protocol determines when a slave is being written-to or read-from. To read a data bit from an iButton slave, the subroutine is called upon to send out a 1 bit. If the bit comes back as a 0, then the slave returned a 0 bit by holding the bus low during the time slot sample time, turning a 1 time-slot into a zero time-slot. Reading from slaves is done by writing all 1s and allowing the slave to modify the 1 bits into 0 bits as necessary. Byte-level subroutines are simply loops that call the bit-level routines eight times and accumulate the bits into bytes.

It would seem that the returned value during a write time slot would be pointless—that it will always be the same as the value that was written. If all is well on the 1-Wire bus, then this is true, and the return value following writes can be (and often is) ignored. However, a short-circuit condition on the bus, an unexpected presence pulse from a new arrival, a noise pulse, or a 0 bit from an errant slave device can be detected by comparing each bit that is written with the bit that is returned. When the software is generating write time slots, and the data returned by the bit subroutine does not match the data that was sent, a bus error has occurred. To detect errors quickly and thereby not waste precious contact time completing a doomed transaction, ultra-reliable software should test the value of data written to the bus against the data returned when possible.

Applications running on PCs (non-embedded) or using serial-to-1-Wire interfaces (i.e., the DS2480) do not form 1-Wire time slots directly, but the result is the same. As each bit or byte is written to the bus, a

corresponding bit or byte is returned that, in the case of writes, can be tested for errors. (Applications on PCs may be at the mercy of serial port and port driver constraints and may be unable to respond to these errors until the entire command transaction is completed.)

SHORT CIRCUIT DETECTION

Shorted bus conditions are common as an iButton is moved into position on the probe and the steel can crosses the probe contacts. Short-circuit checks are often added to Bus Reset and Bit time-slot subroutines, but this must be done with care. If the bus is tested for a short circuit (low state) state too soon after it has been driven low and then released by the master or the iButton slave, the capacitance of the probe and iButton may make the bus appear shorted when it is not. It can take as long as 15µs for a bus to rise from a low level to a valid high level. Testing the bus for a shorted state should always be done just prior to driving the bus low at the start of a Bus Reset or Bit operation, when the bus has had ample time to recover to the high state from any previous low level pulse.

A shorted bus also appears to many Bus Reset subroutines as a valid presence pulse, so short circuit detect may be necessary to discriminate bus shorts from iButton arrivals.

CRC-8 AND CRC-16

iButtons use hardware generated Cyclic Redundancy Checks (CRCs) to validate data and identity information. These error-catching tools are powerful and reliable in most circumstances, but may be weak in the face of some types of interference. When a communication channel is good enough that the majority of bits arrive correctly, a CRC check can detect virtually any small error that may occur. When a channel encounters noise that is essentially random, however, these error control methods do a much poorer job than might be expected.

As an example, if 1000 bytes are sent with a CRC-8 check character at the end, the odds that a single missed or corrupted byte could get through are very small. If the same message contains 1000 randomly *generated* bytes, then the odds are 1-in-256 that one of these entirely erroneous messages would be accepted and deemed legitimate. In other words, the same error checking scheme that works very well in the presence of rare, small errors may perform very poorly in the presence of a flood of random errors.

CRC-type check codes also have some other potential weaknesses. One weakness is that the CRC of all zeros is zero. This means that a shorted bus (which returns all zero bits) will show valid CRC despite the fact that the data is entirely in error. Another problem is that an iButton device may contain many data pages, each with valid CRC on the contents. Should an error occur in the transfer of the page selection bits, the data that is returned will appear valid when it is actually the data from the wrong page. Steps can be (and are) taken to overcome these weaknesses, but it is nonetheless important that they be understood.

CRCs and checksums provide a good working error control mechanism, but may alone be insufficient for ultra-reliable 1-Wire bus programming. When a data error would have very serious consequences, additional checks or redundancy must be used to back-up these error-checking methods.

DETECTING iButton PRESENCE

Most systems that use iButtons presented at probes (readers) by humans are first faced with determining the best way to detect the initial arrival of the device. The method selected depends on the nature of the system. All 1-Wire devices generate a presence pulse when power is applied to them, and following a Bus Reset. This presence pulse can be used to signal the arrival of an iButton at a reader probe. (In battery-powered applications, this pulse is often used to wake-up a ‘sleeping’ microcontroller when an iButton arrives.) An iButton reader may also allow the presence pulse to cause an interrupt and thereby invoke

program code that will process the arrival of the iButton. Lastly, a reader may simply poll, or issue repeated bus reset pulses, and watch for the appearance of a presence pulse response to indicate when a device is present.

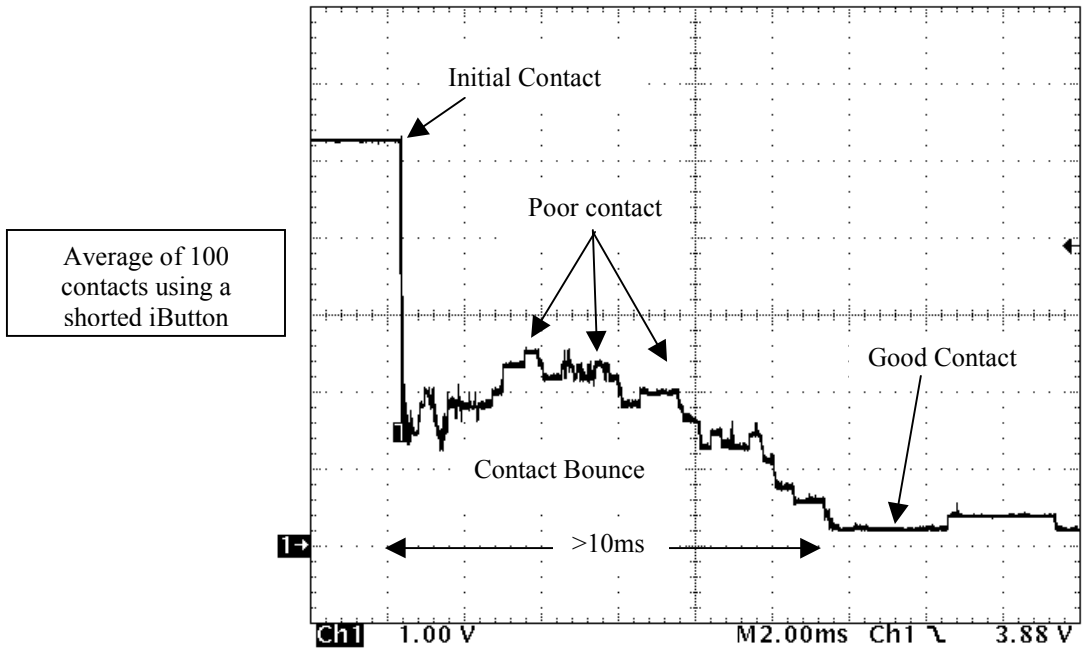
If there are other 1-Wire devices present on the bus, then the polling option will not work because the other devices will always issue presence pulses in response to each Bus Reset pulse. The reader will not be able to determine when a new device has arrived. Detecting the arrival of a new iButton on a bus that has existing 1-Wire devices on it requires more complex programming. (See *ARRIVAL AND DEPARTURE DETECTION* below.)

A system using repeated Bus Reset sequences to poll for the arrival of a single iButton device may find that it occupies the CPU to the detriment of other activities. For this reason, the polling rate may be slowed. A 1ms Bus Reset sequence issued once every 50ms is certainly sufficient for detecting the arrival of an iButton, and consumes only 2% of the CPU time.

READING iButton IDENTITY

Once the reader becomes aware that a new iButton has made contact at the probe, it will usually attempt to read the network address (serial number) of the device. However, the iButton may not be readable until electrical connection becomes stable. A program that observes the arrival of an iButton and then immediately attempts to read the iButton identity will most likely fail. Repeated attempts must be made to read an arriving iButton.

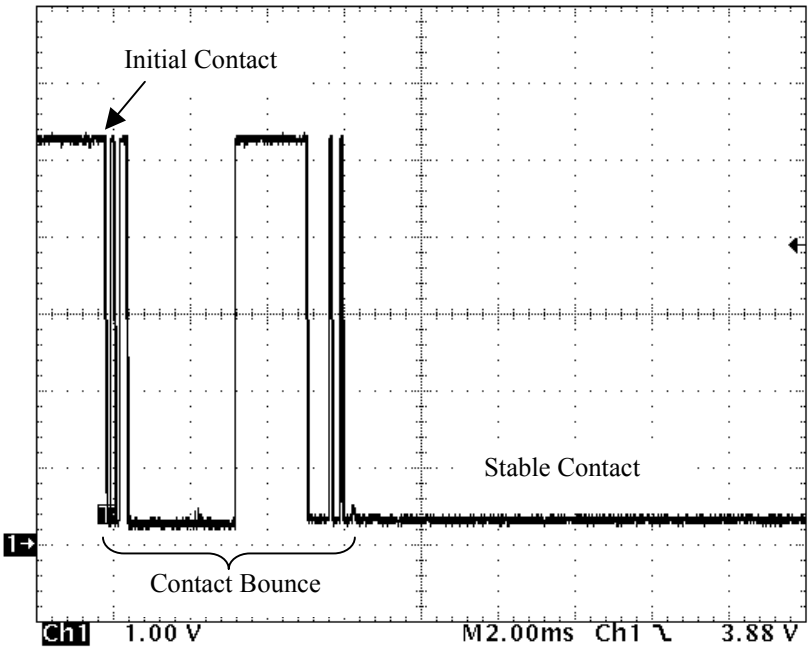
Below is an oscilloscope trace showing a 100-sample average of repeated contacts, by hand, of a short-circuited iButton on a typical 1-Wire reader probe (1kΩ pull-up):



The waveform above demonstrates that the electrical contact between the reader probe and the iButton does not stabilize until as long as 10ms or more after the initial contact is made. More than 40% of the iButton arrivals in this test had no stable connection even 6ms after the initial contact.

The waveform below is a single (typical) iButton contact on the probe. Note that contact bounce is prevalent (and severe) in the first 6ms to 8ms of after contact. A Read-ROM command sequence (at

standard speed) requires about 6ms to 8ms to complete, so these contact failures would prove catastrophic if the read was attempted following initial contact and no retries were performed.



For example, it might be argued that a delay of 20ms following the iButton arrival would solve the problem by allowing the connection time to stabilize. However, for those arrivals that stabilize sooner, transaction time would be wasted, and there is no guarantee that the contact will stabilize in 20ms or even 200ms as the iButton is moved about on the probe. The penalty suffered for each failed read attempt is only 6ms, so the better practice is to perform reads quickly and frequently, with error checking (and perhaps redundant reads as well for ultra-reliable device identification).

READ ROM Method: When the reader probe may expect only a single iButton device to be present at any one time, the direct read method using a READ ROM command may be used. The READ ROM function obtains the 64-bit address of any single iButton device. (If more than one device is present, they will all return their addresses at the same time and the resulting collisions will make the data invalid.)

Using the Read ROM method, there is no guarantee that the data obtained from the iButton is correct. Any loss of contact or corruption of the data would result in an erroneous address from the device. The CRC-8 byte that is part of the address must be used to further validate the address that was read. The READ ROM command performs a non-interactive communication with the iButton. If the iButton was not present, the READ ROM command would not fail, although it would return an all-ones device address. Without short detection, a shorted bus would return an all-zero device address. If the contact was lost during the read, some bits might be valid and others invalid.

Ultra-reliable technique demands a more stringent validation of the device address. In a system where a misread of the device address would cause serious problems (i.e., loss of monetary value), it will be necessary to read the device address twice and compare the reads to make sure that the address has been read correctly.

SEARCH ROM Method: Another way to obtain the address of an arriving device is to use the SEARCH ROM function and seek-out the device using a more interactive method. The search function is used to single-out one device among many on a 1-Wire bus. It does this using a binary search technique in which the iButton participates in a two-way communication over 192 time slots. Instead of just asking an iButton to return its address, the search function interactively determines the device address, so errors in the communication path are more easily detected. Successful completion of a SEARCH ROM operation is a strong indication that a device is present and ready for subsequent communications.

FAMILY CODE AND CRC-8

The leading byte of the eight-byte address defines the type, or family, of the device. This byte is called the Family Code. The last byte in the address is a CRC-8 check byte. It is important that the program test the value of the family code as well as the CRC of the device address. If the family code is zero (which by definition is never a valid family code) then there has been a misread (most likely a short circuit at the probe). If the software does not verify that the family code is non-zero, the entire address may be all zero bits due to a shorted bus, and the resulting CRC check will not detect the error (because the CRC-8 result of all zeroes is zero).

ARRIVAL AND DEPARTURE DETECTION

In many iButton systems, the software must reliably detect the arrival of new iButtons on the 1-Wire bus and the departure of iButtons from the bus. Because the ROM SEARCH function provides a roll call of devices, and because errors on such a bus are common, special steps must be used to make an ultra-reliable scheme for detecting iButton arrivals and departures.

In order to detect arrivals and departures, the software must maintain a table of the devices that are known to be present on the bus and their status. Because errors may cause devices to be missed in some scans, it must also manage an "aging" scheme so that it can de-bounce device departures. A premature decision that a device has departed the bus will produce a subsequent erroneous arrival event. A false arrival event may cause an unwanted action (like a monetary debit) to be performed. There are many ways that arrivals and departures can be reliably detected. The typical scheme involves an algorithm such as this:

1) The bus is continually scanned using SEARCH ROM to discover each device address in turn.
2) As each device is discovered, a table is referenced to determine if that device has been seen before. If the device just discovered is not already in the table, then the new device is added to the table, and an arrival event is reported for that device.
3) When a new device is added to the table, or when a device just discovered is found to already be in the table, an "age" byte for that device entry is set to a constant value (n).
4) Each time a scan has been made of all the devices present on the bus and a new scan begins, all the age bytes in the table are decremented. If any age byte reaches zero, then the device has not been observed in (n) scan passes, so that entry is removed from the table and a departure event is reported for that device.

When each new device is observed, its age is set to (n). Whenever it is missed, its age is decremented. If the age reaches zero, then it is assumed that the device has actually departed the bus. This scheme requires a table of nine-byte entries that is large enough to contain as many records as there could be devices on the bus. Problems with the scheme above include the following:

1) When more devices are present on the bus, the time to scan them all is greater and so the debounce time is greater. An age byte reload value (n) of 5 might debounce a single device departure by only

5ms, but with 100 devices present on the bus, it would take over three seconds to report the departure of any one device.
2) A shorted bus, if the short circuit lasts long enough, causes the departure of all the devices, although they are still present. When the short is cleared, all the same devices will arrive again. (Some algorithms stop the aging process when the 1-Wire bus is shorted to prevent this.)
3) There is no debounce for arrivals. This means that a misread iButton address will be reported as an arrival, and then a subsequent departure, when no such device actually was ever present at all. Despite CRC checks and the added reliability of the SEARCH ROM algorithm, erroneous device addresses still do get through, although only rarely.

Ultra-reliable methodology requires that the algorithm be expanded. Debounce of the arrival involves a scheme to make new arrivals tentative until they are observed twice and so their arrival can be safely announced.

1) The bus is constantly scanned using SEARCH ROM to find each device address in turn.
2) Each device address is checked for non-zero family code and then for the correct CRC-8.
3) As each address is found, a table is referenced to determine if that device has been seen before.
4) If the device is not already in the table, then the new device is added to the table. When a device is added to the table, the age byte is set to (3) and a flag is set that marks the device as a tentative arrival.
5) When a device is found to already be in the table and flagged as tentative, the age byte is incremented by two. If the age byte exceeds (3), then the age byte is reset to (n), the tentative flag is cleared, and an arrival event is reported for the new device.
6) When a device that was just discovered is found to already be in the table and not flagged as tentative, the age byte is incremented up to, but not past, (n).
7) Each time a scan has been completed of all the devices present on the bus, and the bus is found to not be shorted, all the age bytes in the table are decremented by one. If any age byte reaches zero, that entry is removed from the table. If the removed entry was not marked tentative, then a departure event is generated for that device.

The table will have age bytes that reflect the general health and population of the 1-Wire bus. If the bus is performing well, then none of the active entries will be flagged as tentative, and the age bytes will all be equal to (n). If devices are occasionally missed in the scanning, those devices will have age bytes less than (n).

An erroneous device address will be added to the table and marked as tentative, and then removed as it ages and no subsequent matching device is discovered to validate it. The erroneous device address will therefore not be reported as a false arrival event.

This algorithm will require that a device be discovered on the bus in two subsequent scans before it is reported as an arrival, and that it must not be found to be on the bus in (n) consecutive scans before it will be reported as a departure. Given the astronomical odds that an error in the Rom Search will net the same incorrect address value twice, satisfy the SEARCH ROM requirements, and pass the CRC-8 check, this algorithm will provide ultra-reliable reporting of arrivals and departures.

The problem of debounce times changing with varying numbers of devices can be handled by aging the devices in the table on a real-time basis instead of on the completion of each discovery pass. This keeps the aging and debounce time consistent, but limits the maximum number of devices that may exist on the bus. If it takes longer to discover all the devices than it takes to age them, this scheme breaks down.

Of course, other algorithms could be devised for arrival and departure validation and debounce, but the same general redundancy concept should be used to make these events ultra-reliable.

DEVICE ADDRESSING

Once the arriving device address is known reliably, this address will be used to select the device for subsequent communications. In systems where only one device may exist on the bus, there is a temptation to simply use the SKIP ROM command and forego sending the device selection each time a new 1-Wire operation is to be performed. However, secure systems and ultra-reliable techniques demand rigorous controls. Selecting the device using its unique address for each operation guarantees that a substitution of iButtons by the user cannot accidentally corrupt the new device. Ultra-reliable techniques demand direct addressing of devices at the start of each transaction using a MATCH ROM sequence.

Some ultra-reliable systems may find it useful to use a method once called Strong Access to select a device for communication. This method performs a normal SEARCH ROM operation except that it follows the binary search path already known to be correct for the specific device to be addressed. In other words, the device is discovered in a search that already knows the device it expects to find. The end result is the same as for a MATCH ROM operation (i.e., the device is selected and ready for use) but the SEARCH ROM process will have required 128 correct bits of feedback from the device, and so the likelihood that the device is actually present and selected is much more positively assured. Remember that a MATCH ROM function, a memory function command, two bytes of memory address and any number of read time slots could be performed to a device that does not exist, and the error would not be detected until the examination of the Family Code and CRC check bytes, and of the bogus data that was read. The SEARCH ROM method will detect a missing or mis-addressed device within a few bit times as it fails to return the proper responses.

The decision to use Strong Access methods may be affected by the timing required. A MATCH ROM sequence at standard speed requires a little more than 6ms to complete (0.6ms at Overdrive speed). A SEARCH ROM sequence requires almost 17ms to complete (1.6ms at Overdrive speed). A typical fully-secured SHA iButton eCash debit requires about 14 iButton address sequences. This means that the longer access method will add 154ms to the transaction time at standard speed (14ms at Overdrive speed), which may be prohibitive.

READING iButton DATA

Reading data from iButtons is an easy programming task. Once the device is selected (using MATCH ROM or a SEARCH ROM), a Read Memory operation is performed, an address is provided, and then read-time slots are generated to read the device data. Errors while reading from iButtons are common due to the nature of the iButton connection. Even if we assume that the device has been properly selected, and the Read Memory command properly transferred, errors can cause the memory address to be corrupted as well as the data from the memory itself.

To combat these problems, CRC-16 checks are usually embedded in the iButton data records to verify their integrity. Because writing to iButton devices is usually based on 32-byte page sizes, file structure schemes have typically divided iButton memory areas into 32 byte pages. In each page, these rules require that data length and CRC-16 information will be included. These checks provide a reasonable level of error control when reading iButton memory data. To prevent page-selection (address byte) errors, the CRC-16 is typically seeded with the page number for each page.

Once again, however, ultra-reliable techniques may demand more stringent methods. When misread data from an iButton device would have serious consequences, the programmer should consider requiring multiple, redundant reads of the data before concluding that it is correct.

WRITING iButton DATA

While reading device identity and data may have serious consequences in some systems, most systems will reject a misread device and the user will experience a slight delay or will have to re-present the iButton for another try at completing the transaction. Writing to an iButton, however, can be a much more critical task. Once the data is written, an iButton may leave the probe and not return. If the data was written in error, there is no opportunity to re-write it, and no chance to fix the problem. In fact, there may be no way to even know that the data was mis-written at all, because the device may have departed before the write could be read back and verified. This makes the technique of writing to iButtons the most critical of all.

Coming to the rescue are special mechanisms inside all iButtons that are intended for this purpose. In general, there are only two acceptable outcomes in an attempt to write new data to an iButton. The first outcome is failure, in which the old data remains unchanged and the new data is simply lost. This is usually an acceptable outcome because the original value in the iButton has not been lost and it can simply be tried again. The other outcome is, of course, complete success, in which case the iButton departs with the new data intended for it.

The unacceptable outcome, where the iButton has been left corrupted or partially written and partially unwritten, must be prevented at all costs. To do this, writes to iButton pages must be atomic. That is to say, an entire page must be written in a single, uninterruptible operation that is not affected by external sources of error. Because of the serial nature of iButton communication, this cannot be done without the help of some special hardware in the iButton device.

The iButton contains a special data page, aside from the normal memory, called the scratchpad. This (usually but not always one-page-sized) memory area is a temporary workspace where new data is assembled and validated before being copied, in a single atomic operation, to the real device memory. To write to an iButton device, the program must first write the proposed data to the scratchpad memory. Some iButton devices return a CRC-16 of the data written as a check that the data was written correctly. Ultra-reliable techniques, however, demand that the program read the scratchpad back and verify it, byte for byte. When the scratchpad has been written, checked for correct CRC-16, and then read back and checked byte-for-byte with the original data, it is ready to copy to the real memory. The Copy Scratchpad command is issued and the iButton performs an internal, atomic copy of the new data over the old data. The iButton electronics guarantee by design that loss of contact, short-circuit of the device, or interference on the 1-Wire bus will not corrupt the copy of the scratchpad to memory.

After new data has been written to a device, the next step is to make sure that the Copy Scratchpad actually occurred. To do this, the device memory page itself must be read back and checked to make sure it matches the new data that was intended for it. Remember that the scratchpad mechanism guarantees that the contents will either be the original, unchanged data (if the copy failed), or it will be the new, correct data (if the copy succeeded). There can be no other outcome. It is up to the programmer to handle either outcome properly.

For example: a vending machine system uses encrypted eCash in iButtons to pay for food items. When a customer presents an iButton, the balance is read from the iButton and decrypted, the price of the item deducted, the new balance is encrypted and written back into the customer iButton, and the product is delivered to the customer. The basic flow is this:

1) The customer iButton arrives.
2) The vending machine reads the iButton address (serial number), checks for a non-zero family code, and checks the CRC-8 result.
3) The vending machine reads the eCash balance from the iButton. The record structure and CRC-16 are examined to be sure that the data from the iButton is correct. Because a misread of the customer balance could result in the creation of value, which is a serious failure in an eCash system, the customer balance is read again to make the read ultra-reliable. The balance is decrypted and examined to be sure it is sufficient for the purchase.
4) The vending machine reduces the balance by the purchase amount, encrypts the new balance, generates the new CRC-16 values for the new record, and writes the new record to the iButton scratchpad.
5) The CRC-16 generated by the iButton hardware is tested, and then the scratchpad is read back and checked to make sure that it is a match to the new data being written. If all is well, the Copy Scratchpad command is issued and the iButton is now officially debited.
6) Because the Copy Scratchpad command could have failed (or have even been defrauded), the vending machine reads the iButton again and verifies that the copy occurred and that the new data is correct.
7) When the iButton has been read back and the debit is assured the vending machine deliver the product to the customer.

As can be seen above, ultra-reliable techniques require a strongly pessimistic view of all iButton 1-Wire bus operations—an assumption that failures can and will occur at any step in the communication process. This is even more important in light of the fact that an attacker might manipulate the data or hardware in an attempt to defraud the system. The errors might indeed be generated intentionally.

Note:

The example above represents a simplified version of an iButton eCash application, and would be vulnerable to a variety of attacks if implemented only as described. Several iButton devices are available that have strong security mechanisms that thwart these attacks and should be used for any real eCash application such as this. See *AN149, SHA 1-Wire Embedded Application Guide,* for more information about these secure monetary devices.

SPECIAL CASES

Some iButton devices have special security features that complicate error handling during writes. Some devices intended for eCash use contain write cycle counters that increment each time a write is performed to the device memory. These counter values are included in the encryption of the monetary data to create unique instances of the monetary value, and thereby prevent replay attacks. These counters make error recovery more complicated because a failed write may change the counter value and render the encrypted monetary balance data invalid. Attempts to recover from a failed write will have to go back several steps so that the new counter value can be properly included in the encryption of the new data.

CRYPTOGRAPHIC SIGNATURES AND ERROR DETECTION

When the data stored in an iButton memory page is encrypted, or is accompanied by a signature or MAC (Message Authentication Code), the very fact that the decryption or validation has failed will serve as an indication of a data communication error. However, the response to an error such as this may depend upon the nature of the error. A communication error may cause software to re-read the device, but a cryptographic failure may be handled as an attack or fraud attempt. For this reason, it is often undesirable to rely on a failure of cryptographic validation as the sole means to handle what may be a common, transient data communications error. Ultra-reliable techniques mandate that cryptographic validation be performed only when the data is known to be free of communication errors. Only then can the proper action be taken, and the proper statistics collected.

OVERDRIVE SPEED CONSIDERATIONS

In many iButton applications, cable length and loading preclude the use of the Overdrive (high-speed) mode. In those applications where the embedded electronics are in close proximity to the iButton probe, however, this high-speed communication mode can greatly reduce transactions times.

iButton devices that are able to communicate in Overdrive mode must first be addressed at standard speed and then switched to Overdrive speed. The ROM function commands that place a device in Overdrive speed mode are OD MATCH ROM and OD SKIP ROM. A typical procedure to place an iButton into Overdrive mode is as follows:

1) Issue a RESET at standard speed.
2) Send an OD MATCH ROM command at standard speed.
3) Send the device address (8 bytes) at Overdrive speed.
4) Send memory function commands at Overdrive speed.
5) Send or receive data at Overdrive speed.

The device will remain in Overdrive speed mode until a standard speed reset is issued.

When the reader may only have one iButton device on the bus at any one time, it may use the READ ROM command to obtain the device network address (serial number), as follows:

1) Issue a RESET at standard speed.	(1024µs)
2) Send a READ ROM command at standard speed.	(560µs)
3) Read the 8-byte network address from the device at standard speed.	(4,480µs)

However, this can be done much faster if Overdrive mode is used:

1) Issue a RESET at standard speed.	(1024µs)
2) Send an OD SKIP ROM command at standard speed.	(560µs)
3) Issue a RESET at Overdrive speed.	(96µs)
4) Send a READ ROM command at Overdrive speed.	(56µs)
5) Read the 8-byte network address from the device at Overdrive speed.	(448µs)

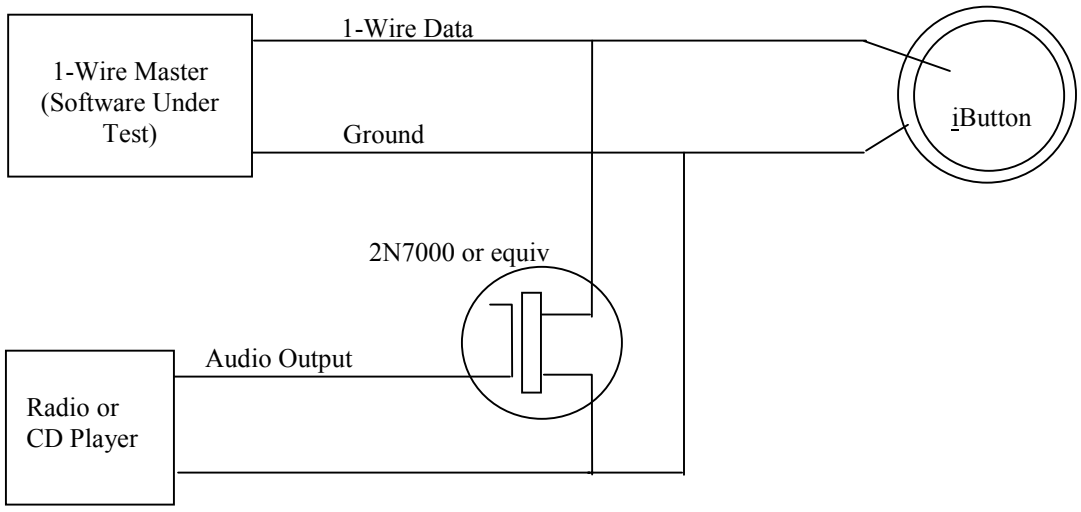
Despite the fact that two bus resets are issued in the process of getting to Overdrive speed mode, the overdrive obtains the device address in 2,184µs. The standard speed method is simpler, but takes 6,064µs to complete.

VERIFICATION OF iButton BUS SOFTWARE

Embedded system software for iButton communication should be first closely examined, and then tested, to prove that it is effective. A close examination will consider each 1-Wire communication sequence and evaluate each of the possible outcomes, no matter how rarely they may occur.

Some of the best proof-of-design and statistical information about iButton communications reliability and error-control can be had by testing real systems with exaggerated noise and interference. Random noise or pulse generators can be used to induce bus errors, and then the software performance and statistics can be examined as transactions are performed.

As an example, eCash debiting schemes have been tested by having the software perform repeated debits of a fixed iButton while injecting extreme interference on the 1-Wire bus. This was done using a music source (radio or CD player) and an N-Channel FET (see diagram below). The transistor converts the audio waveform into intermittent (effectively random) short-circuits on the 1-Wire bus. The method creates widely varied forms of interference and does not drive the bus to illegal conditions that would not occur in real situations, or that might damage components. The simple test circuit is shown below:



(Various improvements could be made to this test circuit, but that is beyond the scope of this document. The bus should be intermittently shorted but not driven to levels that violate the component maximum ratings.)

The software should be configured so that the 1-Wire master performs repeated transactions with the iButton. Playing music or inter-channel noise, raise the audio signal level (output volume) until the 1-Wire bus begins to experience errors. Continue to raise the level until large numbers of the transactions fail due to bus errors—bombard the bus with interference—and watch what happens. Numerous non-fatal errors will occur and many debits will fail to complete, and every recovery mechanism is put to the test. The methods for detecting errors and performing retries can be adjusted for the best performance under these test conditions. If fatal errors occur, the programmer can now examine the system and determine what went wrong so that the flaw can be found and fixed. Reduced music or noise levels may be used to create rare, single-bit errors on the bus. High levels of music or noise can produce extended periods of disruption, which may bring different error recovery mechanisms into play. Percussive musical passages produce widely spaced, momentary data disruptions, while sustained notes produce large periods of disruption spanning many retries. Testing should include a wide variety of signal sources and levels so that many types of errors are experienced.

Note:
There is no reason why the software should not be able to reach a level of reliability where no fatal errors occur under any bus-interference circumstances, even if the interference test is run for millions of transactions (days, weeks, or months).

ERROR TRACKING AND STATISTICS

Any iButton-based system should be able to collect and report statistics regarding performance. In order to program for ultra-reliable operation the programmer should build-in statistics gathering algorithms and software 'hooks' to monitor the way that the software detects and responds to bus errors. The statistics that are often of use in monitoring software performance are as follows:

1) Number of arrivals of iButtons
2) Number of successful transactions with iButtons
3) Number of 1-Wire bus communication errors of various types
4) Average time to complete a transaction (from arrival of iButton to delivery of product)
5) Number of iButtons rejected due to security or cryptographic failures
6) Number of times an iButton was written and then read back and found to have failed to take on the new data
7) Number of times iButton data was read back and found corrupted

These statistics can give a good indication of system performance and reliability, and the effect of program changes can be directly observed.

STRUCTURING CODE FOR SPEED AND ERROR RECOVERY

Many programs working interactively with iButtons use Overdrive (high) speed to minimize transaction times and increase the odds of a completing a transaction in an intermittent contact environment. There can be problems, however, if algorithms are not able to cope with mode changes brought on by contact problems. To demonstrate how these problems can be handled, we will consider an example application that allows a user to pass through a locked gate 10 times, opening the gate in response the presentation of an iButton. The system keeps a count in the user's iButton of the number of times the door has been opened for that user. When the count reaches zero, the door is no longer opened. (We will assume that a single byte in the iButton has been programmed with the value "10" when the iButtons are deployed.)

The software detects the arrival of an iButton device by polling for presence, reads a single-byte counter from the device, decrements the counter and writes the decremented counter back to the iButton. It then opens the gate. To keep these debits fast, the system will use Overdrive-speed communications.

We will start with a simple (and very unreliable) version of this program in unstructured pseudo-code and then discuss improvements to make it more robust. (This assumes that the customer's iButton is the only device on the 1-Wire bus.)

Note:

(SS) refers to commands issued at standard speed, and (OD) refers to commands issued at Overdrive speed. The iButton device is switched to Overdrive speed from standard speed by issuing an OD MATCH ROM (in this example). The command itself is issued at standard speed, and then the device changes modes and subsequent communications are conducted at Overdrive speed. See lines 10, 11, and 12 below for an example of the switch to Overdrive speed.

1. Start:

2. ; Loop until an iButton arrives -

3. Do:
4. Issue a RESET pulse (SS)
5. Loop until Presence is detected

6. ; We have detected an iButton device arrival. Read its device address -

7. Send a READ ROM command (SS)
8. Read 8 bytes of the Device Address (SS)

9. ; We have read a valid iButton device address, now try to read the count from it –

10. Send a RESET pulse (SS)
11. Send an OD SKIP ROM command (SS)
12. Send a READ MEMORY command (OD) *; Switched to overdrive speed here*
13. Send address of desired page (OD)
14. Read address byte from device -> Balance (OD)

15. ; We have the balance, now decrement it and write it back to the iButton –

16. If Balance = 0, then “Insufficient Funds”, go to Start
17. Balance = Balance – 1

18. Send a OD RESET pulse (OD)
19. Send an OD SKIP ROM command (OD)
20. Send a WRITE SCRATCHPAD command (OD)
21. Send address of desired page (OD)
22. Send Balance byte (OD)

23. Send a RESET pulse (OD)
24. Send an OD SKIP ROM command (OD)
25. Send a COPY SCRATCHPAD command (OD)
26. Send address and E/S byte (OD)
27. Wait for copy to be completed

28. ; Open the gate –

29. Open the gate
30. Go to Start

Although the preceding program appears to perform all the necessary steps to detect and debit an iButton, it has many serious reliability flaws:

1) Lines 7–8 provide no protection against misread device addresses, which will occur often in an intermittent environment, and no mechanism for retry.
2) Lines 10–14 will read the count byte from the iButton, but there is no error control and so bogus counts might be read due to bus errors.
3) Lines 18–27 have no protection against errors and no mechanism for retry.
4) Line 11 switches the iButton to Overdrive speed (OD) and then Overdrive speed is assumed to be in effect for the remainder of the transaction. A bus problem could cause the iButton to revert to standard speed, and then the rest of the operations would fail.
5) The gate is opened without checking to make sure that the device was actually debited. Departure of the iButton or a bus error as the COPY SCRATCHPAD command is issued (either by accident or as an intentional attack) would open the door without debiting the iButton.
6) There is no mechanism to make sure that the iButton leaves the probe when the transaction is completed. If the user holds the iButton on the probe after the transaction is completed (which is likely, because the transaction occurs in milliseconds) then the transaction will be performed again and user will lose value.

The following is an example of a re-written program that resolves these problems:

Start:

```
; Be sure that we start with no iButton present. This prevents us from reading the same iButton
; again if it lingers on the probe beyond the term of transaction -

        Retry = 100
        Do:
                Issue a RESET pulse (SS)
                If presence detected then
                        Retry = 100
                Else
                        Decrement Retry
                End If
        Loop Until Retry = 0

; Now wait to detect the arrival of an iButton -

        Do:
                Issue a RESET pulse (SS)
        Loop until Presence detected

; We have detected an iButton device arrival. Attempt to read its address. We will try up to 32 times to
; read the device address without error before giving up.

        Err = 1
        For Retry = 1 to 32
                Send an OD SKIP ROM command (SS)
                Issue an RESET pulse (OD)
```

```
            Send a READ ROM command (OD)
            Read 8 bytes → Device Address (OD)
            If Family Code >0 then
                  Check CRC8 of all eight byte
                  If CRC = OK Err = 0, Break
            End If
            Issue a RESET pulse (SS)
      Next Retry
      If Err>0 then go to Start

; We have read a valid iButton device address, now try to read the balance byte –

      Err = 1
      Speed = 1
      For Retry = 1 to 32
            If Speed = 1 then
                  Send a OD RESET pulse (OD)
                  Send an OD MATCH ROM command (OD)
            Else
                  Send RESET pulse (SS)
                  Send an OD MATCH ROM command (SS)
                  Set Speed = 1
            End If
            Send the 8-byte Device Address (OD)
            Send a READ MEMORY command (OD)
            Send address of desired page (OD)
            Read address byte from device → Balance (OD)
            Read two CRC-16 check bytes (OD)
            If CRC = OK then Err = 0, Break
            Speed = 0
      Next Retry
      If Err>0 then go to Start

; We have the balance, now check it, decrement it and write it back –

      If Balance = 0, then "Insufficient Funds", go to Start
      Balance = Balance – 1
      Compute new CRC16 bytes for the new balance byte

; We will try at Overdrive speed first. If that fails, we will take steps to get the device back
; to Overdrive speed before we try again -

WriteAgain:

      Err = 1
      Speed = 1
      For Retry = 1 to 32
            If Speed = 1 then
                  Send a OD RESET pulse (OD)
                  Send an OD MATCH ROM command (OD)
```

```
                Else
                        Send RESET pulse (SS)
                        Send an OD MATCH ROM command (SS)
                        Set Speed = 1
                End If
                Send the 8-byte Device Address (OD)
                Send a WRITE SCRATCHPAD command (OD)
                Send address of desired page (OD)
                Send Balance byte (OD)
                Send the CRC16 bytes (OD)
                Receive CRC bytes from iButton (OD)
                If CRC = OK then Err = 0, Break
                Speed = 0
        Next Retry
        If Err > 0 then go to Start

; Now perform the Copy Scratchpad -

        Err = 1
        Speed = 1
        For Retry = 1 to 32
                If Speed = 1 then
                        Send a OD RESET pulse (OD)
                        Send an OD MATCH ROM command (OD)
                Else
                        Send RESET pulse (SS)
                        Send an OD MATCH ROM command (SS)
                        Set Speed = 1
                End If
                Send the 8-byte Device Address (OD)
                Send a COPY SCRATCHPAD command (OD)
                Send address and E/S byte (OD)
                Wait for copy to be completed
                If copy completed OK then Err = 0, Break
                Speed = 0
        Next Retry
        If Err > 0 then go to Start


; Read the balance back and check to make sure that the iButton was actually debited. If the iButton
; still has the old balance, try to write it again -

        Send a RESET pulse (OD)
        Send an OD MATCH ROM command (OD)
        Err = 1
        For Retry = 1  to 32
                Send the 8-byte Device Address (OD)
                Send a READ MEMORY command (OD)
                Send address of desired memory byte (OD)
                Read balance byte from device (OD)
```

```
            Read CRC16 bytes from device (OD)
            If CRC16 = OK then:
                  Compare balance byte read to actual Balance
                  If OK then Err = 0, Break
                  Compare balance from device = previous (old) balance
                  If Match, then go to WriteAgain       ; Try the write again if nothing happened
            End If
            Send RESET pulse (SS)
            Send an OD MATCH ROM command (SS)
      Next Retry
      If Err>0 then go to Start

; Open the gate –

      Open the gate
      Go to Start
```

There are several important points to be made about this code. If there are no failures, this code performs the majority of the iButton communication at Overdrive speed. This keeps transactions fast and improves the odds of completing a transaction in a poor contact situation. However, if a communications error occurs, the code performs a standard speed RESET, returning the iButton to a known state. It then issues an OD MATCH ROM, putting the iButton back into Overdrive mode before attempting a retry.

This code also addresses the iButton device specifically each time it accesses it. This prevents an erroneous address from causing the device data to be altered. If the device address is misread, then the device will not be selected when the erroneous address is used, and the subsequent read and write attempts will fail. The OD SKIP ROM function would select any device present on the bus, and would allow its contents to be modified.

Also note that we have added a two-byte CRC-16 check on the balance byte in the iButton. This provides a reasonable level of protection against errors in reading the counter. This CRC-16 should also be seeded with the page number in which the balance resides to prevent page selection errors. If controlling the number of gate accesses was a very serious matter, then perhaps the count byte and CRC bytes should be read again to make sure, beyond any doubt, that they have been read correctly. (For this example, we have deemed the CRC-16 check to be sufficient.)

The added code at the beginning makes sure that there is no iButton present for at least 100 read attempts (about 100ms) before it starts looking for the arrival of an iButton. If this is not done, an iButton in intermittent contact with the probe could be debited multiple times. It is very important that the departure of the previous iButton be positively verified before attempting to perform another debit. If a simple presence check is used, and the arrival is not de-bounced in this manner, then the intermittent iButton contact could (and would) easily fool the system and cause multiple debits to be performed at one touch of the iButton.

Note:

The example above represents a simplified version of an iButton eCash application, and would be vulnerable to a variety of attacks if implemented only as described. Several iButton devices are available that have strong security mechanisms that thwart these attacks and should be used for any real eCash application such as this.

SOME FINAL PRECAUTIONS

Programmers should take great care to handle system interrupts properly. The low-level Bus Reset and Bit subroutines have time-critical periods that will be adversely affected if interrupts are not properly prevented. While the details of these low-level algorithms are outside of the scope of this document, improper interruptions during 1-Wire communications will cause errors that appear to be external and are often difficult to isolate. A careful review should be made of low-level 1-Wire routines to make sure that interrupts are prevented during time-critical operations, and testing with an iButton firmly affixed at the probe (i.e., all external sources of error removed) should yield 100% correct functionality without retries. Only after this is proven should the programmers attention turn to error-handling methods.

CONCLUSIONS

Systems that require ultra-reliable communication with iButtons through intermittent (touch) connections require careful consideration, special algorithms, and thorough testing. Program flow must be designed to provide fast transactions, early detection of errors, and effective retry methods. The quality of the iButton connection must always be considered untrustworthy and error-prone. If the proper precautions are taken, however, ultra-reliable 1-Wire iButton communication can be had despite the intermittent contact environment. The programmer must consider errors and error-recovery in the software design stages as well as in the design-proof-testing stage to guarantee an ultra-reliable result.