A Maxim Integrated Products Brand

71M653X
Energy Meter IC

APPLICATION NOTE

AN_653X_016 **December 2010**

Field Download of Code to the 71M653x ICs

This document describes three methods to reliably download code to a 71M6531, 71M6533 or 71M6534 ICs.

Introduction

Modern smart meters are sometimes required to update their code in the field via a network. The Teridian 71M653x family of energy metering ICs has been designed to support this requirement. Due to the wide variety of network systems in use, it is not practical to provide a complete demonstration solution. This Application Note describes the main parts of a solution, how they work with the hardware, and how they cope with failures and side effects.

Code samples for key processes of the flash download procedure are presented.

Theory of Operation

Network Interfaces

There are two UARTs on all ICs of the 71M653x family of ICs. The UARTs operate from 300 to 76,800 bd, including the common meter interface speeds 300 bd and 9600 bd. The 71M6534 has an additional switch (operated with the registers *UMUX_SEL*, *UMUX_E*, at 0x2007, bits 7,6) to switch UART 1, the optical UART between two alternate sets of pins. The CMOS outputs are fully compatible with both 5V current-loop and RS-485 interfaces. In most cases, the 71M6534 can effectively act as if it has three UARTs. All ICs have optical modulation available on UART1, and DIO pins to control standard RS-232 control lines or to interface directly to highly integrated PSTN modem ICs, such as the Teridian 73M2901CE.

Like all 8051 MPUs, with various programming, the 71M653x MPUs can operate at 7N2, (7 bits, no parity, 2 stop bits, also usable in systems needing 7N1, common ASCII teleprinters), 7E1 (7 bits even parity, 1 stop bit, used for the FLAG protocol), 7O1 (odd parity), 8N1 (used for DLMS/COSEM and ANSI), 8E1, 8O1, and 8N2 (offering higher reliability than 8N1 at high speeds).

Many modems depend on UARTs, but not all. To access other types of modems, the 3-wire EEPROM interface (see below) can be enhanced by adding a 1kΩ resistor between the input and outputs of the modem, and then be operated as a half-duplex master SPI. This gives access to a wide variety of single-chip industry-standard modems for Ethernet, PLC, T-1 and E-1.

When operated with current transformers and multiple-MELF-resistor voltage-dividers, the unit is well isolated from power line transients.

Flash Memory

Table 1 shows the flash memory available for the ICs of the 71M653x family.

Table 1: Flash memory allocation

FL_BANK [2:0]	Address Range for Lower Bank (Common) (0x000-0x7FFF)	Address Range for Upper Bank (0x8000-0xFFFF)	6531 128 KB	6533 128 KB	6534 256 KB
000	0x0000-0x7FFF	0x0000-0x7FFF	X	X	X
001	0x0000-0x7FFF	0x8000-0xFFFF	X	X	X
010	0x0000-0x7FFF	0x10000-0x17FFF	X	X	X
011	0x0000-0x7FFF	0x18000-0x1FFFF	X	X	X
100	0x0000-0x7FFF	0x20000-0x27000			X
101	0x0000-0x7FFF	0x28000-0x2FFFF			X
110	0x0000-0x7FFF	0x30000-0x37FFF			X
111	0x0000-0x7FFF	0x38000-0x3FFFF			X

The flash memory allocation of the 71M653x ICs is very similar to the ROM arrangement in Keil's example "Banking with Common Area" of chapter 9 (linker) of Keil's "Macro assembler and Utilities" manual.

In all 71M653x ICs, there is a 32-KB area from code address 0x0000 to 0x7FFF. The code in this area is always available to the 8051. This area is called "common" and it is the same memory area as "bank 0". Since it is always present, it never needs to be switched into the bank area.

The upper memory area with the address range 0x8000 to 0xFFFF can be allocated to any 32-KB memory bank. The 32-KB bank is selected by writing the bank's number in the register *FL_BANK* (SFR at 0xB6). After the bank is selected with *FL_BANK*, the bank's code is visible to the MPU in addresses 0x8000 to 0xFFFF.

The 71M6531 has four 32-KB banks (128 KB bytes total). Bank 0 is the common area. Banks 1, 2, and 3 are the banked code areas selected by *FL_BANK* (see Table 1).

The 71M6534 has eight 32-KB banks (256 KB total). Bank 0 is the common area. Banks 1 through 7 are the banked code areas selected by *FL_BANK* (see Table 1).

A reset sets *FL_BANK* to 1, so any 71M653x IC can run 64 KB of non-bank-switching code.

The 71M653x ICs have two write-protect registers to protect ranges at the beginning and end of flash, called *BOOT_SIZE* and *CE_LCTN*, activated by the enable bits *WRPROT_CE* and *WRPROT_BT*, in SFR 0xB2.

EEPROM

The 71M6531, 71M6533 and 71M6534 provide hardware support for either a two-pin or a three-wire (µ-wire) type of EEPROM interface. The interfaces use the *EECTRL* and *EEDATA* registers for communication.

The demo code provides a number of source code modules to drive different EEPROMs. EEPROMs up to 1 MBit (AT24C1024, 128 Kbytes) using an I^{2}C interface have been demonstrated to work with these ICs.

Two-pin EEPROM Interface

The dedicated 2-pin serial interface communicates with external EEPROM devices. The interface is multiplexed onto the DIO4 (SCK) and DIO5 (SDA) pins and is selected by setting *DIO_EEX[1:0]* = 01. The MPU communicates with the interface through the SFR registers *EEDATA* and *EECTRL*. If the MPU wishes to write a byte of data to the EEPROM, it places the data in *EEDATA* and then writes the Transmit code to *EECTRL*. This initiates the transmit operation which is finished when the *BUSY* bit falls. INT5 is also asserted when *BUSY* falls. The MPU can then check the *RX_ACK* bit to see if the EEPROM acknowledged the transmission.

A byte is read by writing the Receive command to *EECTRL* and waiting for the *BUSY* bit to fall. Upon completion, the received data is in *EEDATA*. The serial transmit and receive clock is 78 kHz during each transmission, and then holds in a high state until the next transmission. The *EECTRL* bits when the two-pin interface is selected are shown in Table 2.

Table 2: *EECTRL* Bits for 2-pin Interface

Status Bit	Name	Read/ Write	Reset State	Polarity	Description
7	*ERROR*	R	0	Positive	1 when an illegal command is received.
6	*BUSY*	R	0	Positive	1 when serial data bus is busy.
5	*RX_ACK*	R	1	Negative	0 indicates that the EEPROM sent an ACK bit.
4	*TX_ACK*	R	1	Negative	0 indicates when an ACK bit has been sent to the EEPROM.
3:0	*CMD[3:0]*	W	0000	Positive	

CMD[3:0]	Operation
0000	No-op command. Stops the I^{2}C clock (SCK, DIO4). If not issued, SCK keeps toggling.
0010	Receive a byte from the EEPROM and send ACK.
0011	Transmit a byte to the EEPROM.
0101	Issue a STOP sequence.
0110	Receive the last byte from the EEPROM and do not send ACK.
1001	Issue a START sequence.
Others	No operation, set the *ERROR* bit.

✓ The EEPROM interface can also be operated by controlling the DIO4 and DIO5 pins directly. In this case, a resistor has to be used in series with SDA to avoid data collisions due to limits in the speed at which the SDA pin can be switched from output to input. However, controlling DIO4 and DIO5 directly is discouraged, because it may tie up the MPU to the point where it may become too busy to process interrupts.

Three-wire (µ-Wire) EEPROM Interface

A 500 kHz three-wire interface, using SDATA, SCK, and a DIO pin for CS (chip select) is available. The interface is selected by setting *DIO_EEX[1:0]* = 2. The *EECTRL* bits when the three-wire interface is selected are shown in Table 3. When *EECTRL* is written, up to 8 bits from *EEDATA* are either written to the EEPROM or read from the EEPROM, depending on the values of the *EECTRL* bits.

CAUTION The µ-Wire EEPROM interface is only functional when *MPU_DIV[2:0]* = 000.

Table 3: *EECTRL* Bits for the 3-wire Interface

Control Bit	Name	Read/ Write	Description
7	*WFR*	W	Wait for Ready. If this bit is set, the trailing edge of BUSY will be delayed until a rising edge is seen on the data line. This bit can be used during the last byte of a Write command to cause the INT5 interrupt to occur when the EEPROM has finished its internal write sequence. This bit is ignored if HiZ=0.
6	*BUSY*	R	Asserted while the serial data bus is busy. When the BUSY bit falls, an INT5 interrupt occurs.
5	*HiZ*	W	Indicates that the SD signal is to be floated to high impedance immediately after the last SCK rising edge.
4	*RD*	W	Indicates that *EEDATA* is to be filled with data from EEPROM.
3:0	*CNT[3:0]*	W	Specifies the number of clocks to be issued. Allowed values are 0 through 8. If *RD*=1, CNT bits of data will be read MSB first, and right justified into the low order bits of *EEDATA*. If *RD*=0, CNT bits will be sent MSB first to the EEPROM, shifted out of the MSB of *EEDATA*. If *CNT[3:0]* is zero, SDATA will simply obey the HiZ bit.

The timing diagrams in Figure 1 through Figure 5 describe the 3-wire EEPROM interface behavior. All commands begin when the *EECTRL* register is written. Transactions start by first raising the DIO pin that is connected to CS. Multiple 8-bit or less commands such as those shown in Figure 1 through Figure 5 are then sent via *EECTRL* and *EEDATA*.

When the transaction is finished, CS must be lowered. At the end of a Read transaction, the EEPROM will be driving SDATA, but will transition to HiZ (high impedance) when CS falls. The firmware should then immediately issue a write command with CNT=0 and HiZ=0 to take control of SDATA and force it to a low-Z state.

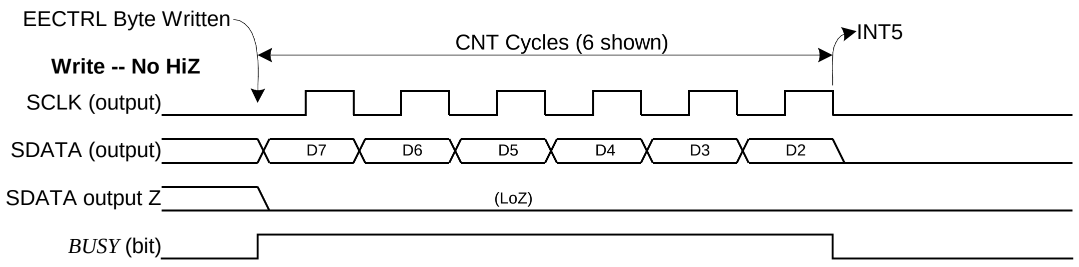

Figure 1: 3-wire Interface. Write Command, HiZ=0.

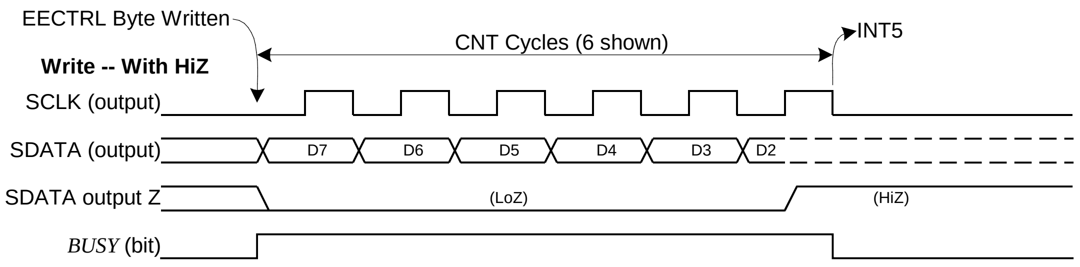

Figure 2: 3-wire Interface. Write Command, HiZ=1

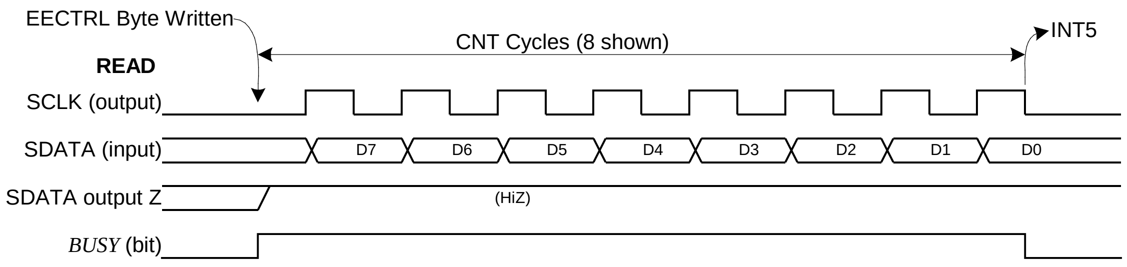

Figure 3: 3-wire Interface. Read Command.

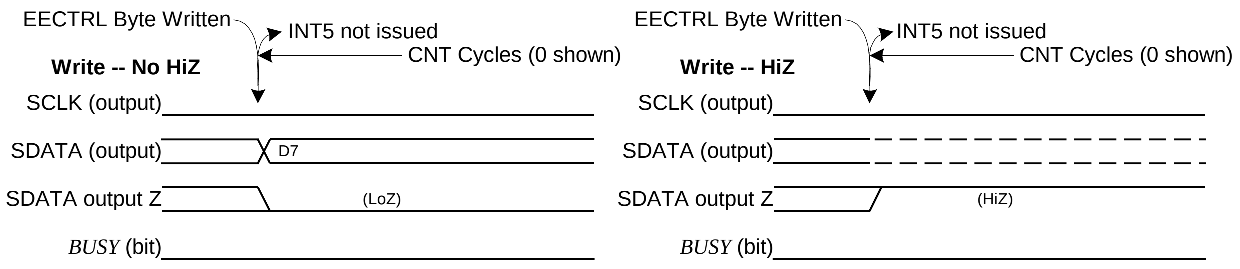

Figure 4: 3-Wire Interface. Write Command when CNT=0

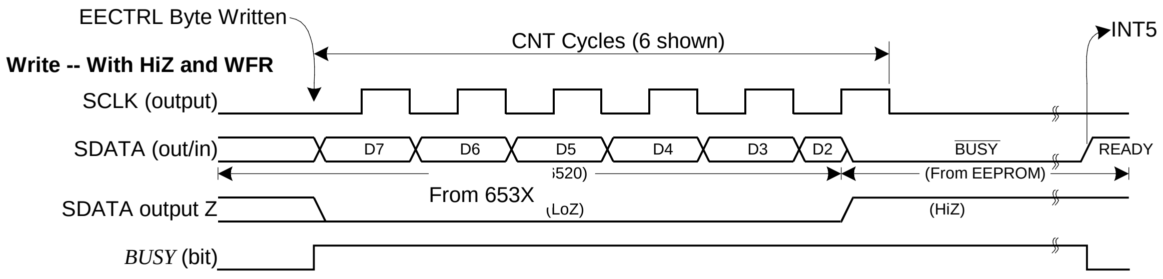

Figure 5: 3-wire Interface. Write Command when HiZ=1 and WFR=1

Software Designs for Flash Update

Three approaches for field update of the flash code will be presented below. First, the general requirements will be examined.

Requirements

The meter has to remain in service if the download of new code fails due to a power failure or an EMI event. This means that some subset of working firmware must always be present in the meter.

A download must have a CRC, longitudinal parity or checksum at a known location, usually at the end of the downloaded data. This feature serves to determine if the downloaded code is free of faults.

A download must have a date or revision in a standard place, usually close to the CRC. A standard, convenient way to store a date and time is as a count of seconds from Jan 1, 1970; this is the standard 32-bit date code available from the standard C I/O function `time_t time(time_t *)`. Dates are usually better than revisions, because dates can be generated automatically, have meaning for both computers and people, and always increment.

A download should have a destination (target) part number in itself at a standard place, usually close to the CRC. That is, it should have the part number for the meter on which it will run.

The CRC or checksum must cover all of the download data, including the code, date and destination part number.

Code generation should be automated as much as possible, in order to reduce human error. Keil C's output or user options tab can run a user program to do this.

A meter in the field is not responsible for validating downloaded code beyond checking its CRC and possibly its date.

The download features must be tested before the new code is released for use in the field.

A meter's network should have some method of reporting configuration data and errors. The minimum for download is to report the meter's part number, whether the last download failed, and the code dates for the boot code and main code.

Approach 1: Save a Temporary Copy in EEPROM

In this approach, the meter has three pieces of software:

- Boot code is in the common bank.
- The main meter software is in common and the upper flash banks
- The alternate copy of meter software ("new code") is in the EEPROM.

The main advantage of this approach is that the meter continues to measure and communicate during most of the download. Also, unlike approach 3, the downloads are interchangeable. Also, an EEPROM can be used instead of a larger, more expensive flash option in the 71M653x.

A disadvantage is slower access to the copy of the software in EEPROM, and a possible extra expense from the larger EEPROM. However, unlike approach 2 and 3, slow access to the EEPROM does not disable the MPU. The MPU can continue to respond to interrupts and other events while it is waiting to read from or write to the EEPROM.

Boot code: This includes the reset and other vectors. The reset vector has some special code to manage the downloads. The other vectors jump to a vector table provided by the main code. The boot code also has a CRC, destination part number and date, in a known place. After reset, the boot code has control. It runs the latest code with a valid CRC and destination part number. If the latest valid code is in the EEPROM, the boot code erases flash page by page, and copies the EEPROM code into the main flash array. After that, it jumps to the reset vector in the main code.

Main code: This contains the network logic. The network logic performs all security, and loads new code to EEPROM. After reset and a load, the main code checks the EEPROM. If it contains invalid code, the main code copies itself to the EEPROM, so that the meter always has two copies of valid code. If a boot code defect is ever discovered, on command from the network operator the main code may check the date, destination part number and CRC of the boot code, and replace old, bad boot code.

Download sequence:

1. The meter operates normally using main code.
2. Download is securely authorized by the network operator, communicating to the main code.
3. The network operator transmits or broadcasts the code using some method that has error detection and correction.
4. The main code copies the new code from the network to the EEPROM.
5. When the copy in EEPROM is complete, the main code tests the EEPROM code's CRC, date and destination part number.
6. If the CRC and destination (target) part number of the EEPROM code is correct, and the EEPROM code date is later than that of the code in flash, the main code in flash saves its revenue registers, and resets the meter.
7. If the new code has a bad CRC or destination part number, the main code notes a download error, copies itself to the EEPROM array, and resumes normal operation.
8. The boot code starts up after a reset. It should turn on a status light.
9. The boot code compares the date of the main code and the EEPROM code. If the EEPROM code is later, and the destination part number matches, it checks the EEPROM code's CRC. If the CRC is valid, the flash is erased, which takes several seconds, (up to 124 pages x40ms = 5s for a 6533, up to 252 pages x 40ms = 10s for a 6534) and the EEPROM code is copied to the main flash. **Metering is stopped during this brief period.**
10. The boot code tests the CRC of the code in flash. If it is not valid, the boot code goes to step 7.
11. The boot code jumps to the main code's reset vector.
12. The main code restores the revenue registers, adding an estimate of the power used during the time it was out of service.
13. On command, the main code may check and replace the boot code. It should replace the boot code only if the boot code has an old version or bad CRC, because there is no recovery if power fails or there is an EMI event while the boot code is being loaded. It should replace boot code only on command so that the network operator can minimize the chance of a power failure. It should write a temporary vector table (routing the vectors to the main code) at the start of replacement, so that vectors are in place during most of the boot code replacement. The flash page with the boot code's final vector table can be written last.
14. The main code resumes normal operation. It should blink the status light (different from the boot code).

Table 4 summarizes the failure modes and effects for approach 1.

Table 4: Failure modes and effects analysis of Approach 1

Failure	Effect	How to Fix
Revenue lost due to download.	Revenue lost to utility while meter is out of service, multiplied by the number of meters serviced.	Most of the download occurs while the meter is on line. Revenue registers are explicitly saved before ending operations. Download process assures re-dundant paths to return to service. The meter is off line for a brief time, about three seconds, and can re-liably estimate the power usage for this interval.

Failure	Effect	How to Fix
Boot code is corrupted.	The meter's reset vector is uninitialized. The download process is impossible, and most safety features fail. The meter becomes unusable without depot service.	The 71M653x meter ICs have several interlocks to prevent accidental flash erases and writes. (See section below on writing to flash.) In addition the 71M653x meter ICs have a special hardware interlock to prevent writes and erases of boot code. The last page number of the boot code should be placed in *BOOT_SIZE* (0x20A7), and then set the bit *WRPROT_BT* (SFR 0xB2, bit 5).
Main code is corrupted.	Unlocalized corruption of the download in flash.	The 71M653x meter ICs have several interlocks to prevent accidental flash erases and writes. (See section below on writing to flash.) The CRC test of the code in flash causes the boot code to refuse to run the corrupted code in flash, and reload the reliable copy still in EEPROM.
EEPROM code is corrupted.	Unlocalized corruption of the download in eeprom.	Most EEPROMs contain write locks, as well as voltage level locks to prevent corruption. These should be used. The CRC test of the EEPROM causes the boot code to refuse to run the bad EEPROM code, and run reliable main code. Main code restores a viable copy of code (itself) to the EEPROM.
Detected communication bit error	Corruption of a section of download data. Code will fail to communicate.	Network code retries the transmission, correcting the defective data.
Undetected bit errors, occurring in storage, or transmission of the download.	Unlocalized corruption of the download to EEPROM. Code will fail to communicate.	The main code's CRC test of the EEPROM fails. Consequently, the main code rejects the download and replaces it with a copy of the working main code in flash.
Network operator error, data entry error, or misconfiguration causes the wrong code to be loaded to meters.	Incompatible code. Code will fail to communicate.	The boot code and main code both test the destination part number. The main code rejects the incom-patible code immediately after down-load, and replaces it with itself. If not, the boot code tests the de-stination part number and rejects a download with the wrong part number. In both cases, the working main code continues to operate.

Failure	**Effect**	**How to Fix**
EMI event or power failure during download from the network to the EEPROM.	Unlocalized corruption of the download in EEPROM. Code will fail to communicate.	The CRC test of the EEPROM causes the boot code to refuse to run the bad EEPROM code, and run reliable main code. The main code restores a viable copy (of itself) to the EEPROM.
EMI event or power failure during the code download from EEPROM to flash.	Unlocalized corruption of the download in flash.	The CRC test of the code in flash causes the boot code to refuse to run the bad code in flash, and reload the reliable copy still in EEPROM.
Boot code defect is discovered	Unknown effects, but it will at least start the main code, because it was tested to do so.	The boot code can be reloaded by the main code. The boot code includes a CRC, date, and destination part number so that it can be downloaded.
EMI event or power failure during load of boot code from main code.	The meter's reset vector is uninitialized. The meter becomes unusable without depot service.	No perfect fix is possible, because the 8051 MPU has only one reset vector. Planned power failures can be avoided by performing boot code replacement only on command of the network operator. EMI effects can be reduced by performing the boot code re-placement with all interrupts dis-abled. EMI effects can be further reduced by first writing a temporary vector table to pass all vectors to the main code. After that, the rest of the boot code is replaced. The last flash page written should install the boot code's vector table.

Approach 2: Save the Temporary Copy in Upper flash.

This approach is very similar to approach 1, except that flash memory in the meter IC is used to store the downloaded code, instead of an external EEPROM.

In this approach, the meter has three pieces of software:

- Boot code is in common (the bottom of bank 0).
- The main meter software is in common in the bottom half of the flash banks.
- An alternate copy of meter software is in the upper flash banks.

In a 71M6533, the boot code is in a few pages, at the bottom of bank 0. The main code is in banks 0 and 1, and the alternate copy is in banks 2 and 3. In a 71M6534, the boot code is also in a few pages at the beginning of bank 0, main code is in banks 0 to 3, and the alternate copy is in banks 4 to 7.

The main advantages are that the download process is more reliable and less subject to tampering because no external devices are needed to do it. Also, unlike approach 3, the downloads are interchangeable. The alternate copy in flash is also very fast to access. Assembly costs are unchanged. A smaller, inexpensive EEPROM is still needed to store the revenue registers. The flash memory on the 71M653x series can only perform 20,000 write operations, and more operations are usually needed to preserve revenue registers.

The main disadvantage of approach 2 is that the meter is disabled while the flash is erased during the download. During this period, the CE must be disabled completely. The MPU is disabled for periods of 40 milliseconds while each flash page is being erased. Erasing two banks, (e.g. banks 2 and 3) on a 71M6533 takes about 2.6 seconds (64 pages x 40ms/page). Erasing four banks (e.g. banks 4 to 7) on a 71M6534 takes about 5.2 seconds (128 pages x 40ms/page). A larger, more expensive flash option in the 71M653x is also needed.

Boot code: This includes the reset and other vectors. The reset vector has some special code. The other vectors jump to a vector table provided by the main code. The boot code has a CRC and date, in a known place. After reset, the boot code has control. It runs the latest code with a valid CRC and destination part number. If the latest valid code is in the alternate flash array, the boot code copies the alternate code into the main flash array. After that, it jumps to the reset vector in the main code.

Main code: This contains the network logic. The network logic performs all security, and loads new code to alternate flash array. After reset, the main code checks the alternate flash array. If it has invalid code, the main code copies itself to the alternate flash array, so that the meter always has two copies of valid code. If a boot code defect is ever discovered, the main code may check the date and CRC of the boot code on command from the network operator, and replace old, bad boot code.

Download sequence:

1. The meter operates normally using main code.
2. Download is securely authorized by the network operator, communicating to the main code.
3. The main code disables the CE and erases the alternate code. **The meter stops communicating and measuring for several seconds.**
4. The lost billing data is estimated from the last accumulation interval, and added to the billing registers.
5. The network operator transmits or broadcasts the code, using some method that has error detection and correction. The main code copies the new code from the network to the alternate flash array. Write operations to a byte in the flash array take about 20 µs/byte, and a special hardware protocol (faster than most EEPROMs).
6. When the copy in the alternate flash is complete, the main code tests the alternate flash code's CRC, date and destination part number.
7. If the CRC and destination part number of the alternate code is correct, and the date is later than the code in flash, the main code in flash saves its revenue registers, and resets the meter.
8. If not, the main code notes a download error, erases the alternate array, copies itself to the alternate array, and resumes normal operation. (This failure ends the download sequence.)
9. The boot code starts up after a reset. It should turn on a status light.
10. The boot code compares the date of the main code and the alternate code. If the alternate code is later, and the destination part number matches, the boot code checks the alternate code's CRC. If the CRC is valid, the main flash is erased (2.6s or 5.2s), and the alternate code is copied to the main flash.
11. The boot code tests the CRC of the code in flash. If it is not valid, the boot code goes to 7.
12. The boot code jumps to the main code's reset vector.
13. The main code restores the revenue registers, adding an estimate of the power used during the time it was out of service.
14. On command, the main code may check and replace the boot code. It should replace the boot code only if the boot code has an old version or bad CRC, because there is no recovery if power fails or there is an EMI event while the boot code is being loaded. It should replace it only on command so that the network operator can minimize the chance of a power failure. It should write a temporary vector table (routing the vectors to the main code) at the start of replacement, so that vectors are in place during most of the boot code replacement. The flash page with the boot code's final vector table can be written last.
15. The main code resumes normal operation. It should blink the status light.

Table 5 summarizes the failure modes and effects analysis of Approach 2.

Table 5: Failure modes and effects analysis of Approach 2

Failure	Effect	How to Fix
Revenue lost due to download.	Revenue lost to utility while meter is out of service, multiplied by the number of meters serviced.	Most of the download occurs while the meter is on line. Revenue registers are explicitly saved before ending operations. Download process assures redundant paths to return to service. The meter is off line for a brief time, about three seconds, and can reliably estimate the power usage for this interval.
Boot code is corrupted.	The meter's reset vector is uninitialized. The download process is impossible, and most safety features fail. The meter becomes unusable without depot service.	The 71M653x meter ICs have several interlocks to prevent accidental flash erases and writes. (See section below on writing to flash.) In addition the 71M653x meter ICs have a special hardware interlock to prevent writes and erases of boot code. The last page number of the boot code should be placed in *BOOT_SIZE* (0x20A7), and then set the bit *WRPROT_BT* (SFR 0xB2, bit 5).
Main code is corrupted.	Unlocalized corruption of the download in flash.	The 71M653x meter ICs have several interlocks to prevent accidental flash erases and writes. (See section below on writing to flash.) The boot code's CRC test of the code in flash, causes boot code to refuse to run the bad code in flash, and reload the reliable copy still in alternate flash.
Alternate flash code is corrupted.	Unlocalized corruption of the download in alternate flash.	The 71M653x meter ICs have several interlocks to prevent accidental flash erases and writes. (See section below on writing to flash.) In addition the 71M653x meter ICs have a special hardware interlock that can be utilized to prevent writes and erases of the alternate flash area. The CE's location should be placed at the end of the main flash area. The CE's location has to be set in *CE_LCTN* for the meter to run. The alternate flash will be after this. Then set the bit *WRPROT_CE* (SFR 0xB2, bit 4). This protects all flash after the start of the CE code. When

Failure	Effect	How to Fix
		the boot code is writing flash, it should use this same mechanism to protect the alternate flash. The boot code's CRC test of the alternate flash causes boot code to refuse to run the bad alternate flash code, and run reliable main code. Main code restores a viable copy of code (itself) to the alternate flash.
Detected communication bit error	Corruption of a section of download data. Code will fail to communicate.	Network code retries the transmission, correcting the defective data.
Undetected bit errors, occurring in storage, or transmission of the download.	Unlocalized corruption of the download to alternate flash. Code will fail to communicate.	The main code's CRC test of the alternate flash fails. So, the main code rejects the download, replaces it with a copy of the working main code in flash.
Network operator error, data entry error, or misconfiguration causes the wrong code to be loaded to meters.	Incompatible code. Code will fail to communicate.	The boot code and main code both test the destination part number. The main code rejects the incompatible code immediately after download, and replaces it with itself. If not, the boot code tests the destination part number and rejects a download with the wrong part number. In both cases, the working main code continues to operate.
EMI event or power failure during download from the network to the alternate flash.	Unlocalized corruption of the download in alternate flash. Code will fail to communicate.	The boot code's CRC test of the alternate flash causes boot code to refuse to run the bad alternate flash code, and run reliable main code. Main code restores a viable copy (of itself) to the alternate flash.

Approach 3: Alternation of Address Ranges

This approach is very similar to approach 2, except that both programs remain executable at the same time, so that the meter always has at least one hot spare of working firmware that is instantly accessible. It resembles some systems used in military avionics.

In this approach, the meter has three pieces of software:

- Boot code is in common (the bottom of bank 0).
- Two working copies of software are in the top and bottom halves of flash. That is, in a 71M6533, the "low code" is in banks 0, 1, and the "high code" is in banks 2, 3. In a 71M6534, the "low code" is in banks 0 to 3, and the "high code" is in banks 4 to 7.

The main advantages are that the download process is more reliable and less subject to tampering because no external devices are needed to do it. Also, the meter always has a hot spare of the firmware, so switching in the spare has no delay to erase and copy a flash area. Assembly costs are unchanged. A smaller, inexpensive EEPROM is still needed to store the revenue registers. The flash memory in the 71M653x series can only perform 20,000 write operations, and more write operations are needed to preserve revenue registers.

The main disadvantage is that the meter is disabled while the flash is erased during the download. Another disadvantage is that there are two incompatible pieces of code, with two software builds, that execute in two regions, a “high download” and a “low download.” A larger, more expensive flash option in the 71M653x is also needed.

During the erase, the CE must be disabled completely. The MPU is disabled for periods of 40 milliseconds while each flash page is being erased. Erasing two banks, (e.g. banks 2 and 3) on a 71M6533 takes about 2.6 seconds (64 pages x 40ms/page). Erasing four banks (e.g. banks 4 to 7) on a 71M6534 takes about 5.2 seconds (128 pages x 40ms/page).

The “high” and “low” downloads require the download process to define a starting address. Ideally, successive downloads should alternate between high and low areas on odd and even software revisions.

Boot code: This includes the reset and other vectors. The reset vector has some special code to qualify downloads. The other vectors in the boot code have short routines that test a semaphore bit, and then jump to either the high download’s vector table or the low download’s vector table.

The boot code has a download address, CRC and date, in a known place. After reset, the boot code has control. It finds the latest code with a valid CRC and destination part number. After that, it sets a bit to indicate which download is current, then jumps to the reset vector in the current code.

Main code: This contains the network logic. The network logic performs all security, and loads new code to the other flash array. It should ignore code that tries to overwrite the current main code. After reset, the main code checks the other flash array. If it has invalid code, the main code erases the date, destination part number and CRC. If a boot code defect is ever discovered, on command from the network operator, the main code may check the date and CRC of the boot code, and replace old, bad boot code.

Download sequence:

1. The meter operates normally using main code.
2. Download is securely authorized by the network operator, communicating to the main code.
3. The main code disables the CE and erases the alternate code. The meter stops communicating and measuring for several seconds.
4. The lost billing data is estimated from the last accumulation interval, and added to the billing registers.
5. The network operator transmits or broadcasts the code using a method that has error detection and correction. The main code examines the destination address. If it does not overlap the current code, it copies the new code from the network to flash. Write operations to a byte in the flash array take about 20 µs/byte, and a special hardware protocol (faster than most EEPROMs).
6. When the copy in flash is complete, the main code tests the CRC, date and destination part number of the alternate flash code.
7. If the CRC and destination part number of the alternate code are correct, and the date is later than the code in flash, the main code in flash saves its revenue registers, and resets the meter.
8. If not, the main code notes a download error, erases the alternate code’s date, destination aprt number and CRC, then resumes normal operation. (This failure ends the download sequence.)
9. The boot code starts up after a reset. It should turn on a status light.
10. The boot code compares the date of the high code and the low code. If the high code is later, and the destination part number matches, the boot code checks the high code’s CRC. If the CRC is valid, the boot code sets the selection bit to “high” and executes the reset vector of the high code.
11. The boot code tests the CRC of the “low” code in flash. If it is not valid, the boot code goes to step 7.
12. The boot code sets the selection bit to “low” and then jumps to the “low” code’s reset vector.
13. The selected main code restores the revenue registers, adding an estimate of the power used during the time it was out of service.
14. On command, the main code may check and replace the boot code. It should replace the boot code only if the boot code has an old version or bad CRC, because there is no recovery if power fails or if there is an EMI event while the boot code is being loaded. Boot code should be replaced only on command so that the network operator can minimize the chance of a power failure. It should write a temporary vector table (routing the vectors to the main code) at the start of replacement, so that vectors are in place during most of the boot code replacement procedure. The flash page with the boot code’s final vector table can be written last.
15. The main code resumes normal operation. It should blink the status light (different from the boot code).

Table 6 summarizes the failure modes and effects analysis of Approach 3.

Table 6: Failure modes and effects analysis of Approach 2

Failure	Effect	How to Fix
Revenue lost due to download.	Revenue lost to utility while meter is out of service, multiplied by the number of meters serviced.	Most of the download occurs while the meter is on line. Revenue registers are explicitly saved before ending operations. The download process assures re-dundant paths to return to service. The meter is off line for a brief time, about equal to approach 1, and half of approach 2, and can reliably estimate the power usage for this interval.
Boot code is corrupted.	The meter's reset vector is un-initialized. The download process is impossible, and most safety features fail. The meter becomes unusable without depot service.	The 71M653x meter ICs have several interlocks to prevent accidental flash erases and writes. (See section below on writing to flash.) In addition, the 71M653x meter ICs have a special hardware interlock that prevents overwriting and erasure of boot code. The last page number of the boot code should be placed in *BOOT_SIZE* (0x20A7), and then set the bit *WRPROT_BT* (SFR 0xB2, bit 5).
Main code is corrupted.	Unlocalized corruption of the download in flash.	The 71M653x meter ICs have several interlocks to prevent accidental flash erases and writes. (See section below on writing to flash.) The boot code's CRC test of the code in flash, causes boot code to refuse to run the bad code in flash, and run the reliable alternate copy.
Alternate flash code is corrupted.	Unlocalized corruption of the download in alternate flash.	The 71M653x meter ICs have several interlocks to prevent accidental flash erases and writes. (See section below on writing to flash.) In addition the 71M653x meter ICs have special hardware interlocks that can be utilized to prevent writes and erases of the high and low flash areas. The low flash area can be protected by using the boot block protection, *BOOT_SIZE* (0x20A7), and then set the bit *WRPROT_BT* (SFR 0xB2, bit 5). To protect the high memory area, the CE's location should be placed

Failure	Effect	How to Fix
		at the start of the main flash area. The CE's location has to be set in *CE_LCTN* for the meter to run. The rest of the high flash will be after this. Then set the bit *WRPROT_CE* (SFR 0xB2, bit 4). This protects all flash after the start of the CE code. When the boot code is writing flash, it should use this same mechanism to protect the alternate flash. The boot code's CRC test of the alternate flash causes boot code to refuse to run the bad alternate flash code, and run reliable main code. Main code restores a viable copy of code (itself) to the alternate flash.
Detected communication bit error	Corruption of a section of download data. Code will fail to communicate.	Network code retries the trans-mission, correcting the defective data.
Undetected bit errors, occurring in storage, or transmission of the download.	Unlocalized corruption of the download to alternate flash. Code will fail to communicate.	The main code's CRC test of the alternate flash fails. So, the main code rejects the download, and invalidates it.
Network operator error, data entry error, or misconfiguration causes the wrong code to be loaded to meters. Either the code is for the wrong meter, or it's to the wrong address.	Incompatible code. Code will fail to communicate.	The main code tests the destination address to assure that the main code does not overwrite itself. It refuses to load code that would overwrite itself. The boot code and main code both test the destination part number. The main code rejects the incompatible code immediately after download, and invalidates it. If not, the boot code tests the destination part number and rejects a download with the wrong part number. In both cases, the working main code continues to operate.
EMI event or power failure during download from the network to the alternate flash.	Unlocalized corruption of the download in alternate flash. Code will fail to communicate.	The CRC test of the alternate flash causes the boot code to refuse to run the bad alternate flash code, and run reliable main code. Main code invalidates the alternate flash area.

CRC Code

This code is a CRC calculation that can be used to validate code. It has been fully tested. It's believed to be compatible with Teridian's checksum utility (delivered with the demo code, executable on PCs), but the compatibility has not been tested.

```
//  The standard 16-bit CRC polynomial specified in ISO/IEC 3309 is used.
//             16   12   5
//  Which is: x  + x  + x + 1
//  This is the most secure error detection in common use.  It detects all
//  1 and 2 bit errors, and most multiple bit errors.  It detects misordered
//  bytes.  It fails to detect leading zeros.  It is slower than a
//  longitudinal parity (also provided), and the practical difference
//  is hard to measure. It is much more secure than a checksum because
//  all bits are of equal significance and misordering is detected.
// This is small reentrant in case it is called as part of the error
// recording in a sag.
//
#pragma save
#pragma NOAREGS
uint8_t data_ok (uint8x_t *ptr, uint16_t len, uint8_t set) small reentrant
{
    uint8_t d;
    uint16_t CRC;

    if (set)
       len -= 2;                        // Do not include CRC in calculation.

    CRC = 0xFFFF;
    do
    {
       d = *ptr++ ^ (CRC & 0xFF);       // Compute combined value.
       d ^= d << 4;
       CRC  = (d << 3) ^ (d << 8) ^ (CRC >> 8) ^ (d >> 4);
    } while (--len);

    if (set)
    {                                   // Store complement of CRC.
       *ptr++ = (CRC & 0xFF) ^ 0xFF;
       *ptr   = (CRC >> 8)   ^ 0xFF;    // Verify.

       return ((*(ptr    ) == ((CRC >> 8)   ^ 0xFF))
            && (*(ptr - 1) == ((CRC & 0xFF) ^ 0xFF)));
    }
    else
       return (CRC == 0xF0B8);
}
#pragma restore
```

LRC Code

This code is an example of a longitudinal parity calculation that can be used to validate code. It has been fully tested. It is much smaller and faster to execute than a CRC, and has better security than a simple checksum.

```
// Performs longitudinal parity.
// It is less secure than a CRC, (also provided) but much faster.
// It is about as fast as a checksum, but more secure because all bits
// have equal significance.
// It detects single-bit errors, many multiple bit errors and
// grossly invalid data.
// It can fail to detect canceling errors, or errors that
// interchange data bytes. Like a CRC and checksum it fails to detect
// extra leading zeros.
// Unlike a CRC, it can have a check digit of one byte, although this
// routine pads the check digit to two bytes.
// This is small reentrant in case it is called as part of the error
// recording in a sag.
#pragma save
#pragma NOAREGS
uint8_t data_ok (uint8x_t *ptr, uint16_t len, uint8_t set) small reentrant
{
    uint8_t LRC;

    if (set)
       len -= 2;                // Do not include LRC in calculation.

    LRC = 0x55;                 // Detect leading zeros.
    for(; len != 0; --len)
    {
        LRC ^= *ptr++;
    }

    if (set)
    {
        *ptr++ = 0;             // Pad to use same space as CRC.
        *ptr = LRC;             // Store complement of LRC.
        LRC ^= *ptr;
    }

    return (LRC == 0);
}
#pragma restore
```

EEPROM Code

Many EEEPROM drivers are available in the demo code. There are too many drivers to fit in this document.

Proven Flash Access Code

This code is the closest thing to download code that is presently available. It reads data at one speed, and writes code to flash at another, faster speed. It loads Intel extended hex files via RS-232 at 38.4KBaud to a 71M6533. It has been fully tested.

```
#include <ctype.h>
#define SDCC 0
#include "reg80515.h"    // Common 80515 structures.

/* Compile switches */
// #define UART1 1        // 1= use uart1, 0 = uart 0; set by C51 options
#define BAUD_RATE_38K 1   // 1=38,400 BAUD, 0 = 9600 baud
#define UART_CHK_FREQ 0   // 1= DIO toggles on uart check, 0 = for data record

/* constants */
#define ERASED 0xff             // Value in flash after it is erased.
#define XON 0x11                // Value of XON
#define PWE 0x01
#define PAGE_ERASE 0x55  // Initiate Flash Page Erase cycle.. ..must be proceeded by a write to PGADR.
sfr FL_BANK = 0xB6;      // Flash Bank select.
sfr INTBITS  = 0xF8;      // multiplexed interrupts & watchdog
#define WD_RST  0xFF     // WatchDog bit.
//sfr P0  = 0x80;        // Port 0, GPIO 00-07.
sbit DIO_7 = P0^7;       // VARh pulse's DIO
sfr DIR0  = 0xA2;        // 1 => output, 0 => input pin.
xdata unsigned char CONFIG0 _at_ 0x2004;
xdata unsigned char CONFIG2 _at_ 0x2007;
xdata unsigned char DIO0 _at_ 0x2008;
xdata unsigned char WAKE _at_ 0x20A9;
#define SLEEP 0x40

/* select port */
#if UART1
#define SxCON  S1CON
#define SxBUF  S1BUF
#define SxRELL S1RELL
#define SxRELH S1RELH
#else
#define SxCON  S0CON
#define SxBUF  S0BUF
#define SxRELL S0RELL
#define SxRELH S0RELH
#endif


/* local functions */
void flash_erase(void);             // Erase all of flash, except first 2K
void flash_write (                  // Write data to flash
        unsigned char code *dest_ptr,   // The destination in flash
        unsigned char pdata  *src_ptr,  // The source, in xdata RAM
        unsigned char length);          // The number of bytes to write
void putc(unsigned char  Data);         // Put a character out
unsigned char getc(void);               // Get a character (from the queue)
void uart_check (void);                 // Put characters into the queue

/* local data */
#define MAX_BYTE 128
#define SIZE (5 + MAX_BYTE)
unsigned char pdata byte_ary[SIZE]; // Intel byte array
unsigned char byte_idx;             // index to insert to byte_ary
unsigned char checksum;             // Intel hex checksum of data in byte_ary
unsigned char lsb;                  // 1 = next character is lower hex
unsigned char ok;                   // 1 = no errors ever detected

unsigned char pdata char_ary[16];   // serial input array
unsigned char char_insert_idx;      // index to put byte into the array
unsigned char char_extract_idx;     // index to get byte out of array
unsigned char data page_erase;      // Interlock for page erase
unsigned char data pwe;             // Interlock for flash write

unsigned char delay_cnt;            // counts delay loops
unsigned int xon_timer;             // counts time before XON sent

//============================================
void feed_wd (void)    /* push off (feed) the watchdog */
{
    if (0 == (--delay_cnt)) /* don't feed it too often */
    {
        delay_cnt = 255;
        INTBITS = WD_RST;
        ++xon_timer;
    }
}

void flash_erase(void)
{
    unsigned char bank;
    unsigned int byte_idx;
    FL_BANK = 1;
    for(byte_idx=8;byte_idx<128;byte_idx+=4) // from 0x800 byte to 0x7FFF, in 1K pages
    {
        FPAGE = byte_idx;               /*Set PageNo*/
        /* only erase and write on command;
         * An EMI event can corrupt the PC, executing erase code */
        if (PAGE_ERASE != page_erase)   /* Interlock valid?*/
        {
            ERASE = 0;
            return;
        }
        ERASE = page_erase;             /* Initiate page erase, MPU halted.*/
        ERASE = 0;                      /* Finish page erase. */
        INTBITS = WD_RST;
    }
    for(bank=1;bank<4;bank++)         // For 6531,6532,6533
        // for(bank=1;bank<8;bank++)         // For 6534
    {
        FL_BANK = bank;
        // from 0x8000 byte to 0xFFFF, 1K pages
        for(byte_idx=128;byte_idx<256;byte_idx+=4)
        {
            FPAGE = byte_idx;               /*Set PageNo*/
            /* only erase and write on command;
             * An EMI event can corrupt the PC, executing erase code */
            if (PAGE_ERASE != page_erase)   /* Interlock valid?*/
            {
                ERASE = 0;
                return;
            }
            ERASE = page_erase;            /* Initiate page erase, MPU halted.*/
            ERASE = 0;                      /* Finish page erase. */
            INTBITS = WD_RST;
        }
    }
    FL_BANK = 1;
}

/* length is intentionally limited to 256 */
void flash_write (
        unsigned char code *dest_ptr,
        unsigned char pdata  *src_ptr,
        unsigned char length)
{
    unsigned char byte;

    uart_check();                   /* try to get a character */
    /* only erase and write on command;
     * An EMI EVENT can corrupt the PC, executing erase code */
    while (length--)
    {
        byte = *src_ptr++;
        if (ERASED != byte)         /* the fastest write is the one not done */
        {
            FCTRL = pwe;            /* Enable Flash Write */
            if (PWE != pwe)         /* Interlock valid? */
            {  // not valid. Abort the write loop.
                FCTRL = 0;
                return;
            }
            // MOVX @DPTR,A writes single byte to FLASH.
            *(unsigned char xdata *)dest_ptr++ = byte;
            FCTRL =  0;              // Clear Flash Write
        } else {
            ++dest_ptr;
        }
        uart_check();                  /* try to get a character */
    }
}

void uart_check(void)    /* try to get a character */
{
    #if UART_CHK_FREQ
    P0 ^= 0x40;                 /* flash the Wh and alternate pulse LED */
    #endif
    if(SxCON & 0x01 ) /*  if RI FLAG SET  */
    {   /* queue the character */
      char_insert_idx &= 0x0F;
      char_ary[char_insert_idx++] = SxBUF;
      SxCON &= ~0x01;
    }
    feed_wd ();
}

/* get a character out of the character queue */
unsigned char getc(void)
{
    char_extract_idx &= 0x0f;
    return char_ary[char_extract_idx++];
}

/* put a character out */
void putc(unsigned char  ch)
{
    while(0 == (SxCON & 0x02 ));
    SxBUF = ch;
}

void record_write(void)
{
    unsigned int address;

    if(0 != checksum)   /* all the checksummed bytes sum to zero */
        ok = 0;
    /*
     * byte_ary[0] = record length (first byte of intel hex record)
     * byte_ary[1],[2] = 16-bit address (record type 0), unused (record type 4)
     * byte_ary[3]=intel hex record type (0=data, 4=extended address)
     * byte_ary[4]=first code byte (record type 0) or addr31..24 (record type 4)
     * byte_ary[5]=2nd code byte (record type 0) or addr23..16 (record type 4)
     */
    address = byte_ary[1] << 8;
    address |= byte_ary[2];
    if (0 == byte_ary[3])           /* code record */
    {
        if (0 == (byte_ary[1] & 0x80))      /* put adr15 into FL_BANK */
            FL_BANK &= 0xFE;
        else
            FL_BANK |= 1;
        if (0 != FL_BANK)           /* bank 0? */
            address |= 0x8000;      /* not bank0=address the bank */
        if (0x07FF < address)       /* address not in boot code */
        {
            flash_write (
                    (unsigned char code *)address,
                    &byte_ary[4],
                    byte_ary[0]);
        }
        #if !UART_CHK_FREQ
        P0 ^= 0x44;                 /* flash the Wh and alternate pulse LED */
        #endif
    }
    else if (4 == byte_ary[3])      /* EA record type, Set upper 16 bits */
    {
        FL_BANK = (byte_ary [5] << 1) & 0x02;   /* adr16 for 6531,2,3 */
        // FL_BANK = (byte_ary [5] << 1) & 0x06;     /* adr17,16 for 6534 */
        #if !UART_CHK_FREQ
        P0 ^= 0x44;                 /* flash the Wh and alternate pulse LED */
        #endif
    }
    else if (1 == byte_ary[3])      /* end record */
    {
        if(ok) {
            putc('1');
        } else {
            putc('0');
        }
    }
    else
    {
        ok = 0;
    }
    xon_timer = 0;
}

void record_start (void)
{
    byte_idx = 0;         /* ready for the next record */
    lsb = 0;
    checksum = 0;
    byte_ary[0]=0;          /* clear the count */
    byte_ary[1]=0xff;       /* invalid address */
    byte_ary[2]=0xff;
    byte_ary[3]=0xff;       /* invalid record type */
}

void main(void)
{
    unsigned char data ch;

    /* only erase and write on command;
     * An EMI EVENT can corrupt the PC, executing erase code */
    page_erase = PAGE_ERASE;    /* enable flash erase */
    pwe = PWE;                  /* enable flash write */
    DIR0 = 0;                   /* make the DIOs readable */
    DIO0 = 0;       /* enable DIO7 and DIO8, the VARh and Wh DIOs */

    /* commanded to change flash? */
    /* Why poll a DIO?  If an EMI event corrupts the PC,
     * running this code, the DIO will eventually return
     * to a non-bootcode state, and the meter will start */
    if (!DIO_7)        /*if Var pulse output shorted low at reset*/
    {
        /* so, to erase flash, execution has to reset,
         * and also DIO_7 has to be true */
        INTBITS = WD_RST;   /* reset watchdog */
        /* And it should not be in brownout mode */
        WAKE = SLEEP;       /* if in battery mode, force sleep */

        ADRMSB = 0x04;      /* pdata at 0x400 in RAM */
        CONFIG0 &= ~0x07;   /* standard, fast clock rate */

        delay_cnt = 0;
        flash_erase();      /* erase flash */
        ok = 1;

        #if UART1
        SxCON = 0x9A;
        #if BAUD_RATE_38K
        SxRELH= 0xFF;       /*baud rate is set @38,400*/
        SxRELL= 0xFC;
        #else // 9600
        SxRELH= 0xFF;       /*baud rate is set @9600*/
        SxRELL= 0xF0;
        #endif // end baud rate
        CONFIG2 = 0x00;     /* enable OPT_TX, UART1 */
        #else
        ES0_SEL = 1;        /* bit 7 of SFR 0xD8, UART0 baud from MPU clock */
        SxCON = 0x5A;
        #if BAUD_RATE_38K
        SxRELH= 0xFF;       /*baud rate is set @38,400*/
        SxRELL= 0xFE;
        #else // 9600
        SxRELH= 0xFF;       /*baud rate is set @9600*/
        SxRELL= 0xF8;
        #endif // end baud rate
        CONFIG2 = 0x40;     /* enable DIO2, the alternate pulse output */
        #endif
        SxCON &= ~0x01;
        char_extract_idx = char_insert_idx = 0;
        xon_timer = 0;

        /* unit is ready for data */
        putc(':');
        #if !UART_CHK_FREQ
        P0 &= ~0x44;    /* turn on the Wh and alternate pulse LED */
        #endif
        DIR0 |= 0x44;

        record_start();         /* ready the next record */

        /* Why poll a DIO?  If an EMI event corrupts the PC,
         * running this code, the DIO will eventually return
         * to a non-bootcode state, and the meter will start */
        while (!DIO_7)          /* jumper is on */
        {
            uart_check();       /* try to find another character */

            /* while the character queue is not empty */
            if (char_extract_idx == char_insert_idx)
            {
                /* no characters for ~3 seconds; maybe XON is needed? */
                /* xon timer is incremented in watchdog & uart check fn */
                if (xon_timer > 0x300)
                {
                    #if !UART_CHK_FREQ
                    /* turn on the Wh and alternate pulse LED */
                    P0 &= ~0x44;
                    #endif
                    putc(XON);
                    xon_timer = 0;  /* also cleared when record found */
                }
            } else {
                /* queued characters */
                ch = getc();
                if (':' == ch || '\r' == ch || '\n' == ch) /* record ended? */
                {
                    if (5 < byte_idx)       /* record stored up? */
                    {   /* Set the actual data length. */
                        checksum -= byte_ary[0];    /* adjust the checksum */
                        byte_ary[0] = byte_idx - 5; /* set the byte length */
                        checksum += byte_ary[0];    /* adjust the checksum */
                        record_write();     /* write the stored record */
                    }
                    record_start();         /* ready the next record */

                } else if (isxdigit(ch)) {  /* get next hex character */

                    ch -= '0';
                    if (ch >= 10)
                    {
                        ch &= ~(0x20);                      /* map 'a' to 'A' */
                        ch -= 'A' - '0' - 10;   /* 'A' mapped to 10. */
                    }
                    if (lsb)
                    {
                        lsb = 0;
                        ch |= byte_ary[byte_idx];
                        checksum += ch;         /* add the checksum */
                        byte_ary[byte_idx++] = ch;
                        if (SIZE <= byte_idx)
                        {
                            byte_idx = SIZE - 1;
                            ok = 0;
                        }
                        /* past the record length? */
                        if (byte_idx >= (byte_ary[0]+5))
                        {
                            record_write();   /* write the stored record */
                            record_start();   /* ready the next record */
                        }
                    }
                    else
                    {
                        lsb = 1;
                        byte_ary[byte_idx] = ch << 4;
                    }
                }
                /* else, throw away other characters */
            } /* end if character available */
        } /* end while jumper on */

        /* wait until last character is sent */
        while(0 == (SxCON & 0x02 ));

    } /* end if jumper on */

    /* exit */
    page_erase = 0;     /* disable flash erase */
    pwe = 0;            /* disable flash write */
    ADRMSB = 0x00;      /* pdata at 0 in RAM */
    P0 =0xFF;           /* DIOs to input, off */
    DIR0 = 0;
    SxCON = 0;                /* disable UART */
    FL_BANK = 1;        /* bank to default */
    /* returns to start-up code, which jumps to application entry point */
}

/* History:
 * $Log: 6533_small_boot.c,v $
 * Revision 1.3  2009/12/18 18:31:50  tvander
*/
```