AN1796
APPLICATION NOTE

FIELD UPDATES FOR FLASH BASED ST7 APPLICATIONS USING A PC COMM PORT

by Dennis Nolan

INTRODUCTION

Having the capability to update software in a microprocessor while the part remains in-circuit has proven to be a very valuable feature of the new flash based micro controllers. Generally, this has been accomplished via a special dedicated connector (something like a 10 pin header) on the board which talks to a specialized programmer which, in turn, is controlled by a PC. While the cost of these programmers has come down steadily, mostly due to increased capabilities of the MCUs to virtually program themselves, they are still not very available to the end user. The purpose of this note is to present practical software techniques which can be built into virtually any application which includes an RS232 serial port to allow the flash based ST7 MCU to be updated by any end user in the field.

The software presented here is a small (less than 500 bytes) boot loader which will allow for the operating software in the ST7 to be replaced safely and reliable by use of a standard S-Records format file which can either be emailed to an end user or made available for download on a Web site. The PC side of the software update operation can easily be accomplished using any of a number of terminal emulation programs. In this note the operation will be described using the HyperTerminal program which is available to all Windows based PC users since it comes bundled with the operating system.

Rev. 1.0

1 FLASH MEMORY SECTORS

We will divide flash memory within the ST7 into two sectors by selection within the option byte when the chip is programmed using a standard programmer. These sectors are referred to as sector0 and sector1. The selection at program time is to decide what will be the size of sector0, with the balance of the available memory going to sector1. The boot loader program is less than 500 bytes in size so when programming the part select the smallest size for sector0, which is 0.5K or 512 bytes. The only real distinction between sector0 and sector1 is that sector0 is always write protected as it pertains to in application programming (under direct ST7 program control). Sector0 can always be programmed by a dedicated programmer but it cannot be written to by the MCU itself. Sector1 can be programmed either by a dedicated programmer or by software executing on the MCU itself. Sector0 is always the uppermost part of memory so that it includes the reset and interrupt vectors (data that one always wants to be write protected in order to make a self updateable system “bulletproof” in the face of catastrophes such as power failures in the middle of an update). Since the entire boot loader code and the vectors reside in the write protected sector0 it is always possible to recover from a problem during an update transfer by simply resetting the ST7, which will get us back to the boot loader. The example program was written for the ST72264 which has 8K of flash so sector1 occupies from $E000 to $FDFF and sector0 from $FE00 to $FFFF.

2 ABSOLUTE ADRESSES AND VECTORS

The example program was written for the ST72264 but, so long as the reset and interrupt vector locations are the same, it can be executed without change on any ST7 regardless of the size of its flash memory. This is because whatever the size of the flash, it will occupy the top of the 0000 to $FFFF 64k possible addressing space of an ST7. We would thus always want sector0 to occupy the top 0.5k, the space from $FE00 to $FFFF. We would also want the skip-jump table (which is technically part of sector1) to always be located in the area just below sector0 (also the top of sector1). In the example program this table occupies from $FDD0 to $FDFF. Looking at the vector section at the end the program, the segment directive is used to place the program pointer at each of the absolute memory locations for the various interrupt vectors (and the reset vector) as described in the data sheet for the MCU. At each of these locations, a dc.w directive is used to place a 16 bit absolute address vector. In most micro controller software we would be more accustomed to seeing a symbolic (and relocatable) address reference used here and this points out an important concept that must be understood. After we make our first version of the software application and send it “out into the world” by programming it into the MCU with a standard programmer (sector0 and sector1), sector0 will probably (if all goes according to plan) never be changed again, although sector1 can have many revisions. When sector1 code is revised, the addresses of any or all of the interrupt service routines associated with the interrupt vectors will move. Since sector0 and the vector table will not change we need the address references in the vector table to be fixed. This limitation would cause a severe loss of flexibility in the program if it were not for the use of the skip-jump table located at $FDD0. This table has one entry for each interrupt vector. The table consists of long jump (to cover the entire 64k space) instructions spaced every four bytes. Actually a long jump only takes up three bytes (one for the jp op code and two for the address) but it’s easier to count by four.

Please note that, in the example program, all of the jumps are to the symbolic (and re-locatable) address “dummyisr” except for the one associated with the reset vector, which jumps to “start”. This is because the example program does not make use of any interrupts. The program must provide a mechanism to use any of the interrupts since we cannot “get at” the vector table once we send our program out into the world. If we send out a version 1.0 of our application which perhaps uses only the SCI interrupt and later decide to revise the program and make use of the TIMERA interrupt we can do that by simply replacing the appropriate “jp dummyisr” instruction in the skip-jump table with a jump to our TIMERA interrupt service routine. Since the skip-jump table resides in sector1 we are free to change the destination of the jumps whenever we please although the absolute location of the jump instruction itself is fixed so that it always agrees with sector0.

A careful examination of the reset vector reveals that it is treated specially. This vector does use a symbolic (and re-locatable) address reference, “BANK0”. This is permissible because

the destination is within sector0 and so the two will always stay “in sync”. At power-up reset we always want to execute code in sector0 which is write protected and gives the decision point to either read in an S-Records file to update sector1 or exit to execute normal code in sector1. The exit to sector1 is done via a jump to absolute address $FDFC which is an entry in the skip-jump table containing a re-directing jump to “start”.

Note: Time delays within this program are implemented as simple countings loops. If the main crystal clock is other than 8 Mhz then times will change accordingly.

3 USING THE PROGRAM WITH HYPERTERMINAL

Configure HyperTerminal as follows:

1. Select FILES->PROPERTIES
2. Click the “CONNECT TO” tab
3. At “CONNECT USING” select DIRECT TO COM1, 2,3, or 4 as appropriate.
4. Click CONFIGURE and select

```
BITS PER SECOND: 9600
DATA BITS: 8
PARITY: none
STOP BITS: 1
FLOW CONTROL: none
```

5. Click the “SETTINGS” tab
6. Select EMULATION: ansi
7. Click ASCII SETUP and
   – Uncheck all boxes except “Append Line Feeds to Incoming Line Ends”
   – Set CHARACTER DELAY to 0 milliseconds
   – Set LINE DELAY to 20 milliseconds

NOTE: A proper line delay setting is crucial to proper operation of the system. The ST7 needs this time to store the data in an S-Records line before it is ready to receive the data from the next line.

After HyperTerminal has been configured select FILE-> “SAVE AS” to save the program configuration as a .HT file. Afterwards, just clicking the .HT file will open HyperTerminal with the required settings.

Load the example program into the MCU, connect the target to the PC via RS232, and start up HyperTerminal. After HyperTerminal has started, reset the MCU. After approximately five seconds the MCU will transmit a single dollar sign ($) character which will appear on the PC monitor. Be sure not to press any keys on the keyboard during the five second interval. If the $ is not observed, double check the program configuration and the COMM port selection.

Once the $ character has been observed, reset the MCU and, within five seconds, press any printable key on the keyboard. This Key will be echoed back to the monitor and will indicate that the ST7 is now in a mode where it will wait indefinitely to receive an S-Records file update for sector1. At this point, do the following with HyperTerminal:

1. Select TRANSFER->SEND TEXT FILE
2. Select FILES OF TYPE: all files (*.*)

3. Navigate to the location of your .S19 S-Records file and double click it.

At this point you should see the S-Records data scroll past on the screen since the ST7 echoes the data back to the terminal as it is received and stored. Once the entire file has been received the boot loader will force a cold reset on the processor in order to safely lock-up the flash against any further writes. After the five second wait interval another $ character will be transmitted. The program section which transmits the $ is at the label “start” and is actually the entire sector1 program in this example. After sending the $, execution falls into an endless loop at label “loop1”. As a suggested exercise, go to the example program source file and change the character that is transmitted from $ to some other printable character and then re-build the application. Now start up HyperTerminal and reset the ST7. After five seconds the $ character will appear (remember, we haven’t transmitted the new file yet). Now reset the MCU and, within five seconds, press any printable character on the keyboard. The character will be echoed back to the screen indicating that the ST7 is waiting for a file. Do the TRANSFER->SEND TEXT FILE procedure and send the .s19 file for the example (now slightly modified). The file will scroll by on the screen and, after five seconds, whatever character you have replaced the $ with will appear on the screen. CONGRADULATIONS: You’ve used HyperTerminal to do a field update of your application program via RS232.

4 DECISION POINT

At the label “bootupwait” the program enters a five second loop where it continually polls to see if a UART character has been received. If a character is detected then execution jumps to the label “doupdate” where the flash memory is unlocked and the program waits patiently to receive the S-Records update file. If the five second period expires without detection of a UART character then execution escapes to the entry point for sector1 code (the “jp $FDFC” instruction). While this is a good method to use for an example program because it will run properly “out of the box” on any ST7 based system that has a RS232 port it may not be the best choice in all situations. It may be undesirable, for example, that the system will always delay for five seconds after power up before starting to execute the normal program or it may be difficult to arrange for the triggering character to be send to the board within five seconds after power up reset.

One alternative method would be to dedicate a general purpose port pin to control the function. In the hardware, we could include a pull-up resistor to Vcc at the pin and provide a jumper that would ground the pin if installed. The jumper would normally be installed and we would only remove it if an update was desired. The code at “bootupwait” can be changed so that the port pin is examined. If the input is logic low, execute the jump to $FDFC without delay to start normal operation. If the pin is high then the program can wait until the jumper is re-installed and then jump to “doupdate” to receive a new program. Normally, the jumper would always be installed. To do an update we would remove the jumper, power up (reset) the ST7, replace the jumper and then transmit the file. After the file has been received a cold reset will follow and, because the jumper is installed, normal execution will commence.

If an application is tight on I/O and cannot afford to dedicate a pin to this function another approach is to insert a jumper between the output of the RS232 line receiver and the Rx input of the ST7 and then provide a 10k pull down resistor to ground at the pin of the ST7. Normally, the jumper would be installed. Since the normal marking state of the Rx signal is logic high, the code at the decision point can examine the pin (as just a general I/O pin) and, if it is high, exit normally to sector1. If the pin is low (jumper has been removed) then the program would wait until the pin goes high, then jump to doupdate and wait to receive a new file. To perform an update one would then remove the jumper and power up (reset) the ST7, install the jumper, and then transmit the file. After the file has been received a cold reset will follow and, because the jumper is installed, normal execution will commence.

5 APPENDEX1: SOFTWARE SOURCE

st7/

```
    #include "264regs.inc"

    segment byte at: E000 'program'

;******** Entry point for Sector1 Code **********
start:
       ld a,#'$'
       ld SCIDR,a                          ; transmit character
loop1:
       jrt loop1

;**** Default interrupt service routine ****
dummyisr:
       iret


;**************** Skip-Jump Table ******************

    segment byte at: FDD0 'program'                ; I^2C
                     jp    dummyisr
    segment byte at: FDD4 'program'                ; SCI
                     jp    dummyisr

    segment byte at: FDD8 'program'                ; AVD
                     jp    dummyisr
    segment byte at: FDDC 'program'                ; TIMER B
                     jp    dummyisr
    segment byte at: FDE0 'program'                ; MCC
                     jp    dummyisr
    segment byte at: FDE4 'program'                ; TIMER A
                     jp    dummyisr
    segment byte at: FDE8 'program'                ; SPI
                     jp    dummyisr
    segment byte at: FDEC 'program'                ; CSS
                     jp    dummyisr
    segment byte at: FDF0 'program'                ; ei1
                     jp    dummyisr
    segment byte at: FDF4 'program'                ; ei0
                     jp    dummyisr
    segment byte at: FDF8 'program'                ; TRAP
                     jp    dummyisr
```

```
    segment byte at: FDFC 'program'                 ; skip jump from BANK0
                    jp    start


;*************** SECTOR0 CODE ***********************************

    segment byte at: 80-FF 'bank0ram0'

b0byte0:                    ds.b 1
bytecount:                  ds.b 1
loadaddress:                ds.w 1
checksum:                   ds.b 1
flashbuffer:                ds.b 32
RAMCODE:                    ds.b 40


    segment byte at: FE00 'program'
BANK0:

       NOP
       NOP
       sim                                          ; GLOBALLY DISABLE INTERRUPTS


       ld a,#%11010010          ; set baud rate to 9600
       ld SCIBRR,a

       ld a,#%00001100          ; enable SCI transmitter and receiver
       ld SCICR2,a


; copy the loader routine from ROM to RAM
       clr x
ramtransfer:
       ld a,(FLASHPROGRAMMER,x)
       ld (RAMCODE,x),a
       inc x
       cp x,#40
       jrne ramtransfer


bootupwait:
       ld x,#55                 ; 5 seconds
       clr y
       clr a
```

```
charcheck:
       btjt SCISR,#5,doupdate          ; jump out and do update if character re-
ceived
       dec a
       jrne charcheck
       dec y
       jrne charcheck
       dec x
       jrne charcheck
       jp $FDFC                        ; timed out, escape to BANK1


doupdate:
; unlock flash control register
       ld a,#$56
       ld FCSR,a
       ld a,#$AE
       ld FCSR,a

       ld a,SCIDR
       ld SCIDR,a                            ; echo character
       jrt checkforS


b001:
       call waitforchar
checkforS:
       cp a,#'S'
       jrne b001      ; loop back until S is received
       call waitforchar
       cp a,#'1'
       jreq gots1                      ; jump if this is a S1 record
       cp a,#'9'
       jreq gots9                      ; jump if this is a S9 record
       jrt b001                        ; otherwise, loop back
gots1:
       call getbinary
       sub a,#3                              ; account for address and checksum
bytes
       ld bytecount,a                        ;byte count saved

       call getbinary
       ld loadaddress,a                      ;high byte of load address saved

       call getbinary
```

```
       ld {loadaddress+1},a                        ; low byte of load address
saved

       clr x
b002:
       cp x,bytecount
       jreq b003
       call getbinary
       ld (flashbuffer,x),a
       inc x
       jrt b002
b003:
       call getbinary
       ld checksum,a                          ; checksum for line saved

       call RAMCODE


waitforcr:
       call waitforchar
       cp a,#13
       jrne waitforcr

       jrt b001

gots9:
       call waitforchar
       cp a,#13
       jrne gots9

       bset WDGCR,#7                   ; force a cold reset by starting watchdog
       bres WDGCR,#6                   ; and then sitting in an endless loop
gots9A:
jrt gots9A; endless loop waiting for watchdog to
;       force a reset


;****************** bank zero subroutines ************************

waitforchar:
       btjf SCISR,#5,waitforchar
       ld a,SCIDR
       ld SCIDR,a                             ; echo character
       ret
```

```
; reads two successive characters from the character stream (hex-ascii)
; and returns them in a as a byte
getbinary:
       call getdigit
       swap a
       ld b0byte0,a
       call getdigit
       add a,b0byte0
       ret

; this subroutine gets copied to and is executed out of RAM
FLASHPROGRAMMER:
       clr x
latchenable:
       bset          FCSR,#1                  ; enable Xflash latches
fp004:
       cp x,bytecount
       jreq fp005
       ld a,(flashbuffer,x)
       ld ([loadaddress.w],x),a
       inc x
       ld a,x
       add a,{loadaddress+1}
       and a,#$1F
       jrne fp004
; apply programming pulse
       bset FCSR,#0                           ; set PGM
fp006:
       btjt FCSR,#0,fp006
       jrt latchenable
fp005:
; apply programming pulse
       bset FCSR,#0                           ; set PGM
fp007:
       btjt FCSR,#0,fp007
       nop
       nop
       ret

; receives a hex-ascii character and converts to binary
getdigit:
       call waitforchar
       sub a,#48
       cp a,#9
       jrule getdigitA
```

```
      sub a,#7
getdigitA:
      ret


;*************** interrupt vectors ******************


    segment byte at: FFE4 'vector'                ; I^2C
    dc.w  $FDD0
    segment byte at: FFE6 'vector'                ; SCI
    dc.w                  $FDD4

    segment byte at: FFEC 'vector'                ; AVD
    dc.w  $FDD8
    segment byte at: FFEE 'vector'                ; TIMER B
    dc.w  $FDDC
    segment byte at: FFF0 'vector'                ; MCC
    dc.w  $FDE0
    segment byte at: FFF2 'vector'                ; TIMER A
    dc.w                  $FDE4
    segment byte at: FFF4 'vector'                ; SPI
    dc.w  $FDE8
    segment byte at: FFF6 'vector'                ; CSS
    dc.w  $FDEC
    segment byte at: FFF8 'vector'                ; ei1
    dc.w  $FDF0
    segment byte at: FFFA 'vector'                ; ei0
    dc.w  $FDF4
    segment byte at: FFFC 'vector'                ; TRAP
    dc.w  $FDF8
    segment byte at: FFFE 'vector'                ; RESET
    dc.w                  BANK0

       end
```

6 APPENDEX2: BLOW BY BLOW SOFTWARE DESCRIPTION

6.1 POWER-UP INITIALIZATION

At power up reset the vector at $FFFE is fetched, which branches execution to the code label BANK0. At BANK0 interrupts are disabled, the UART baud rate is set to 9600 and the UART transmitter and receiver are enabled. Next the 40 byte routine at label FLASHPROGRAMMER is copied from its location in flash to a block in RAM. This is the code which actually writes to the flash and it is necessary that this code be executed out of RAM since, once the write sequence to flash is started, data cannot be read out of flash until the write is completed.

6.2 TO UPDATE OR NOT TO UPDATE

At label BOOTUPWAIT a 24 bit counter made up of concatenated registers x:y:a is load with $550000 and then the polling loop starting at label CHARCHECK is cycled through a total of 5,570,560 times, each time checking to see if a UART character has been received. With an 8 MHZ effective clock, this polling will take about five seconds. If no character is detected then the “jp $FDFC” instruction branches execution to the normal sector1 program via the skip-jump table. At label CHARCHECK, if a character is detected then execution branches to label DOUPDATE.

6.3 UPDATE

To begin the flash update process a sequence of two unique “keys” are written to control register FCSR. This exact data in this exact sequence must be written to this register in order to remove write protection from the flash. If any other data is written, the protection will “lock up” until the next reset of the MCU. Next, whatever character had been received in order to activate the update sequence is echoed back to the operator terminal as a confirmation.

6.4 INTERPRETING THE S-RECORDS FILE

At label B001 the subroutine WAITFORCHAR is called. This routine will poll indefinitely waiting for a character to be received by the UART and then echo back the character to the terminal and return it in the a register. Since an S-Records line must start with an upper case S, the program will loop back through label B001 until an S is seen. Next the program looks for either a 1 or a 9 to indicate an S1 or S9 record. Anything else is discarded as an error and execution goes back to label B001 to look for an S again. At label GOTS1 subroutine GETBINARY is called. This routine will read two characters from the character stream, convert them from hex/ascii to binary format, and return the number in register a. This number is the nn record (see appendix 3) which is the total number of bytes in the record. To get the number of actual data bytes in the record, 3 is subtracted from the value to account for the two address and one checksum byte that are included in the total count. This is saved in the variable

BYTECOUNT. GETBINARY is called twice more to retrieve the high and low bytes of the 16 bit load address, which is stored in variable LOADADDRESS. At label B002 a code loop is formed that will read all of the data bytes from the character stream and store them in the linear buffer FLASHBUFFER.

At label B003 the checksum is read form the character stream and stored as variable CHECKSUM. This program does not make use of the checksum but it is made available here for any future enhancements. Now the subroutine RAMCODE is called. The label RAMCODE is actually the 40 byte block of RAM where the flash writing routine was copied from flash at power-up initialization. We will now examine the code at label FLASHPROGRAMMER. First, the x pointer is cleared and the flash programming latches are enabled. Label FP004 is the top of the program loop. The test for the normal exit condition from the loop is at the top of the loop. When the pointer is equal to the byte count then execution transfers to label FP005. Before the loop is exited, each byte in the linear FLASHBUFFER is written to its proper load address. The low byte of the destination address is also checked to see if it has crossed any 32 byte sector boundary (any change in the upper 11 address bits). This check is necessary because of the hardware of the flash programming latch. The latch only has five address lines and thus the upper 11 address lines must not change during a single write sequence or writing will “wrap around” and start overwriting earlier data. If the load address is about to cross a 32 byte sector line then the program must fall out of the loop and activate the flash programming sequence with the bytes it has transferred so far. At label FP006 the program waits until the hardware programming of the flash is complete and then transfers back to label LATCHENABLE where transfer continues until the pointer is equal to the BYTECOUNT, at which point a programming pulse is applied to complete the programming of the last of the bytes from the S-Records line. The subroutine will then return back to the main program at label WAITFORCR where the program will wait to receive the carriage return which terminates the S-Records line and then jump back to label B001.

Back at B001 we are ready to receive and handle any additional S1 records until, eventually, an S9 record is received. This will cause execution to transfer to label GOTS9 where the program will read and discard characters until the carriage return that terminates the S9 record is recognized. At that point the program forces a cold reset to occur by starting the watchdog timer and then falling into an endless loop at label GOTS9A until the watchdog times out and forces a hardware reset. The cold reset is needed to “lock up” the flash memory array against any further writes. The ST7 does not include a “reset” instruction so the watchdog was used to force the situation.

7 APPENDEX3: S-RECORDS FORMAT DESCRIPTION

Data Record: 'Stnnaaaaddddddddddddddddddddddddddddddd...cc<cr><lf>'
Field Contents:
S -----ASCII 'S' indicates start of record.
t ------Record type, '1','2','3'=data, '9'=end of file
nn ----Number of bytes in record (including address and checksum). (in ASCII/HEX)
aaaa --Load address of data record. (in ASCII/HEX)
dd ----Actual data bytes in record (two ASCII characters per data byte). (in ASCII/HEX)
cc ----Checksum of count, address, and data. (in ASCII/HEX)
<cr> -ASCII carriage return character. (0D hex, 13 decimal)
<lf> -ASCII line feed character. (0A hex, 10 decimal)

Note 1: The checksum is computed as the one's complement of the eight bit sum of all values from 'nn' to the end of data (last 'dd' byte).

Note 2: Count 'nn' is three greater then the number of data bytes in the record, since two bytes of address and one byte of checksum are included in the count.

Note 3: The end of file record (S9) contains a count of 3 and address of 0000h.

“THE PRESENT NOTE WHICH IS FOR GUIDANCE ONLY AIMS AT PROVIDING CUSTOMERS WITH INFORMATION REGARDING THEIR PRODUCTS IN ORDER FOR THEM TO SAVE TIME. AS A RESULT, STMICROELECTRONICS SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT OR CONSEQUENTIAL DAMAGES WITH RESPECT TO ANY CLAIMS ARISING FROM THE CONTENT OF SUCH A NOTE AND/OR THE USE MADE BY CUSTOMERS OF THE INFORMATION CONTAINED HEREIN IN CONNECTION WITH THEIR PRODUCTS.”