Application Note

# Boot Loader for the Z8F082A and Z8F1680 MCUs

AN032801-0611

## Abstract

This Application Note describes a boot loader program for the on-chip memory functions of Zilog's Z8F082A and Z8F1680 Series of Microcontrollers, which are based on the Z8 Encore! XP® architecture.[1] The boot loader is loaded using Zilog's ZDSII IDE and provides the functionality to program an Intel HEX 32-format file to Z8 Encore! XP MCU Flash memory via the RS-232 port. It is designed to provide an alternative method for downloading the firmware to the MCU instead of using the Debug Interface and Zilog's Smart Cable.

This boot loader program associated with this application note supports the following Z8 Encore! XP MCUs:

- 4K Flash: Z8F042A
- 8K Flash: Z8F082A
- 16K Flash: Z8F1680

**Note:** The source code file associated with this application note, AN0328-SC01.zip, is available for download on zilog.com. This source code has been tested with version 5.0.0 of ZDSII for Z8 Encore! XP-powered MCUs. Subsequent releases of ZDSII may require you to modify the code supplied with this application note.

## Z8 Encore XP-Based MCUs: A Flash Memory Overview

The products in Zilog's Z8 Encore! XP Z8F082A and Z8F1680 Series of Microcontrollers feature 4KB to 16KB of nonvolatile Flash memory with read/write/erase capability. This Flash memory array is arranged in 512 bytes per page, the minimum Flash block size that is erased. Flash memory is also divided into 8 to 32 sectors and is protected from programming and erase operations on a per-sector basis.

Figures 1 and 2 illustrate the Flash memory arrangements of each of the Z8 Encore! XP MCUs listed in the Abstract above.

1. For the Z8F082A device, this boot loader program operates on all packaging types, from 8 pins to 28 pins.

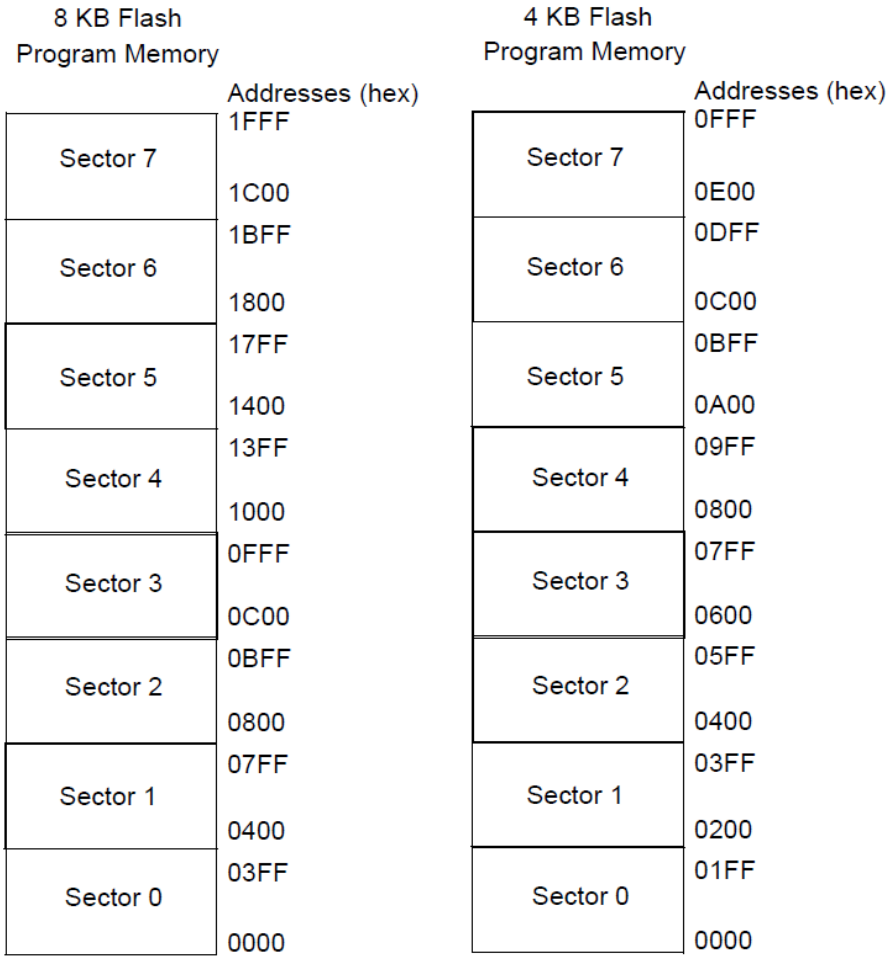



**Figure 1. Flash Memory Arrangement of the Z8F042A MCU (4KB Flash) and Z8F082A MCU (8KB Flash)**

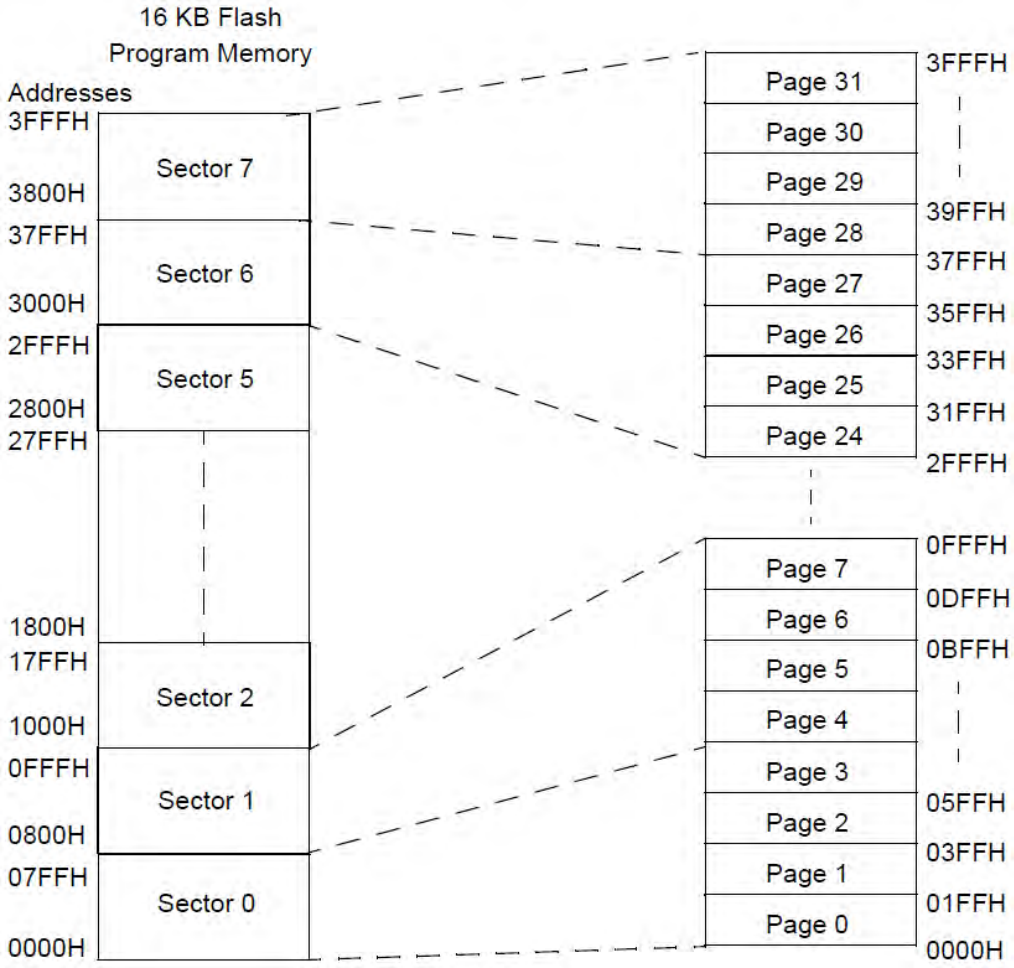



**Figure 2. Flash Memory Arrangement of the Z8F1680 MCU (16KB Flash)**

For additional information regarding the Flash memory functions of the Z8F082A and Z8F1680 families, please see the Z8 Encore! XP F082A Series Product Specification (PS0228) and the Z8 Encore! XP F1680 Series Product Specification (PS0250).

# Discussion

A boot loader is typically a program that permanently resides in the nonvolatile memory area of the target processor and is the first block of code to execute at Power-On Reset (POR).

A typical boot loader possesses the following functional characteristics:

- The reset address of the target CPU points to the starting address of the boot loader code.
- The boot loader polls the UART port to receive a specific character.

- When a specific character input is received on the polled UART port, the boot loader is invoked to load Flash memory, then to program new user code into Flash memory. When the boot loader is executing in Flash loading mode, it typically receives data through a COM port to program user data into Flash memory. In the absence of any other indications, the boot loader code branches to the existing user application program and begins execution.
- The boot loader issues commands to the Flash controller to program the data into Flash memory.
- The boot loader checks the destination address of the user application code to prevent any inadvertent programming of this application code into its own memory space.
- The boot loader performs an error check on the received data using the checksum method.

## Developing the Z8F082A and Z8F1680 Boot Loader Application

The Z8F082A and Z8F1680 boot loader is firmware that enables the Z8F082A and Z8F1680 MCUs to write to their own program memory spaces via the UART RS-232 interface. It features an on-chip Flash controller that erases and programs on-chip Flash memory. The boot loader program uses the Flash controller and the on-chip UART to function; each is described below.

**Flash Controller.** The Flash controller provides the appropriate Flash controls and timing for the byte programming, Page Erase and Mass Erase operations conducted in Flash memory. The Flash controller contains a protection mechanism, via the Flash Control Register (FCTL), to prevent accidental programming or erasure. Before performing either a programming or erase operation on Flash memory, the Flash Frequency High and Low Byte registers must be configured. These Flash Frequency registers allow the programming and erasure of Flash with system clock frequencies that can range from 32.8kHz to 20MHz.

**UART.** The UART0 is used to communicate with the HyperTerminal emulation program running on a PC; it is initialized to a required baud rate by writing appropriate values to the UART baud rate registers (these values are provided in the Software Implementation section on page 9).

**Reset Pin.** The Reset pin is used to restart the boot loader firmware; therefore, if the character `0x20` (ASCII for a space character) is received, the program counter redirects the program to the Flash Loader function; otherwise it goes directly to user application code.

Figure 3 shows a block diagram of the boot loader.

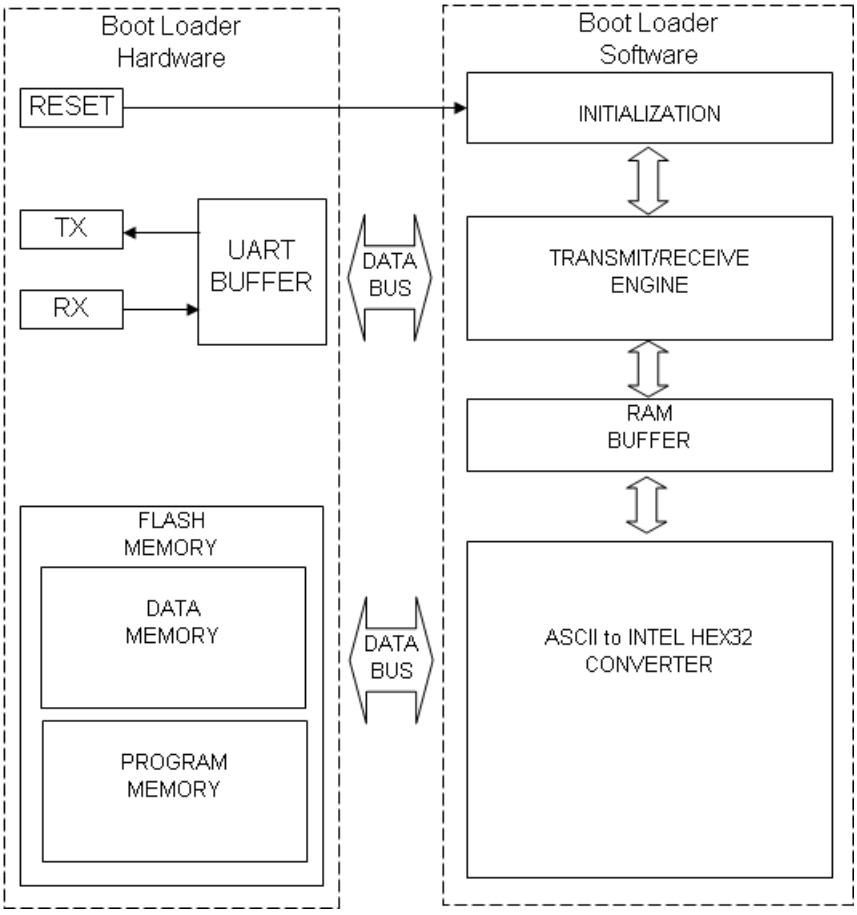



**Figure 3. Block Diagram of the Z8 Encore! XP Boot Loader**

For complete details about the on-chip Flash memory and Flash controller functions of the Z8F082A and Z8F1680 MCUs, refer to the following product specifications, available from the Zilog website at www.zilog.com:

- Z8 Encore! XP F082A Series Product Specification (PS0228)
- Z8 Encore! XP F1680 Series with Extended Peripherals Product Specification (PS0250)

## Z8F082A and Z8F1680 Boot Loader Features and Application

The boot loader program operates in the following sequence:

1. Flash loading mode is invoked upon polling the serial port for a specific character within a specified period of time. After this invocation, the boot loader program transfers control to the user application, which then begins to execute. The address of the application code can be found in the range 0008h–XBFFh (the value of the X varies depending on MCU Flash memory size).

2. The boot loader program selectively erases Flash memory before programming the user code; the portion of memory in which the boot loader code resides remains unchanged.
3. The boot loader program receives the user application code via the RS-232 port (HyperTerminal); it calculates and verifies a checksum for error detection. If the loaded hex file contains checksum errors, it displays `Error: checksum` in HyperTerminal and terminates execution.
4. The boot loader program loads data in the Intel HEX 32 format into Flash memory one line at a time.

**Note:** See a brief description of the Intel Hex File Format in Appendix B. Intel Hex 32 Format on page 52.

5. The boot loader program displays a progress indicator in HyperTerminal to indicate the status of the data being loaded into Flash; it displays `COMPLETED!` in HyperTerminal after programming is completed.
6. The boot loader program protects its own memory space by preventing the user code from being programmed into the area occupied by the boot loader. If the loaded hex file contains the same address range as the boot loader code, it displays:

   ```
   Error: Address: ROM = 0000–XBFF
   ```

   In this error statement, the value of `X` varies depending on the size of MCU Flash memory.
7. If the above error is received, data (ROM) addressing must be changed to `0000h–XBFFh` because the boot loader code already occupies the upper byte addresses in the range `XC00h–XFFFh` (the value of `X` varies depending on the size of MCU Flash memory).

   To determine the Flash memory address ranges for the boot loader code and user application code spaces in each of the 4K, 8K and 16K products discussed herein, see Figures 4 through 6.

**Note:** In Figures 4 through 6, the color green represents rewritable addresses and blue represents nonrewritable addresses.

| MCU | 4 KB Flash Memory |
|---|---|
| ADDRESS | DATA |
| 0FFFh<br>...<br>0C00h | Boot Loader Code<br>(Restricted Address) |
| 0BFFh<br>...<br>0BFEh | User's Application<br>Code Start Address |
| 0BFDh<br>...<br>00004h | User's<br>Application Code |
| 00003h<br>...<br>00002h | Boot Loader<br>Start Address<br>(0001 F800h) |
| 00001h<br>...<br>00000h | Flash Option Bit<br>(FFFFh) |

**Figure 4. Flash Memory Addresses: Application Code and Boot Loader Code for the Z8F042A 4K MCU**

| MCU | 8 KB Flash Memory |
|---|---|
| ADDRESS | DATA |
| 1FFFh<br>...<br>1C00h | Boot Loader Code<br>(Restricted Address) |
| 1BFFh<br>...<br>1BFEh | User's Application<br>Code Start Address |
| 1BFDh<br>...<br>00004h | User's<br>Application Code |
| 00003h<br>...<br>00002h | Boot Loader<br>Start Address<br>(0001 F800h) |
| 00001h<br>...<br>00000h | Flash Option Bit<br>(FFFFh) |

**Figure 5. Flash Memory Addresses: Application Code and Boot Loader Code for the Z8F082A 8K MCU**

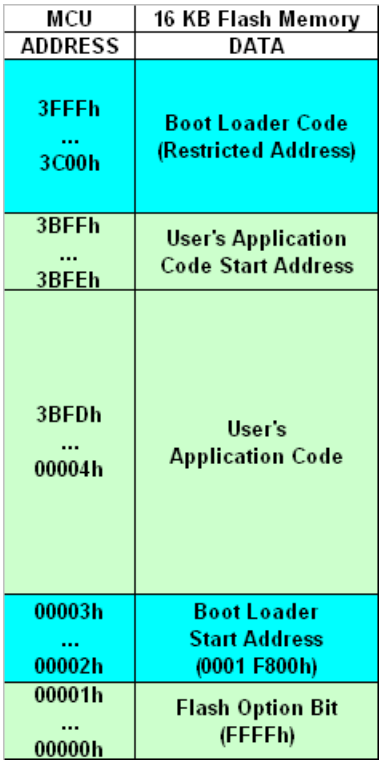



**Figure 6. Flash Memory Addresses: Application Code and Boot Loader Code for the Z8F1680 16K MCU**

# Theory of Operation

Generally, a boot loader's sole function is to download a hex file created in ZDSII to MCU Flash memory. This application is designed to provide this hex file via the MCU's UART function, which is an alternative to using Zilog's XTools firmware (which communicates via a USB port). The advantage of using the UART is that the user can update firmware via the RS-232 serial interface.

# Software Implementation

The following hierarchy represents the sequence of boot loader execution within the `main` function; each linked segment in this sequence references its description in this section.

## Main Function Hierarchy

1. Boot Loader Code
   1.1. Initialize Flash Memory

## Setting Communication Parameters

UART0 communication parameters are set to the following values in HyperTerminal (or similar terminal emulation program):

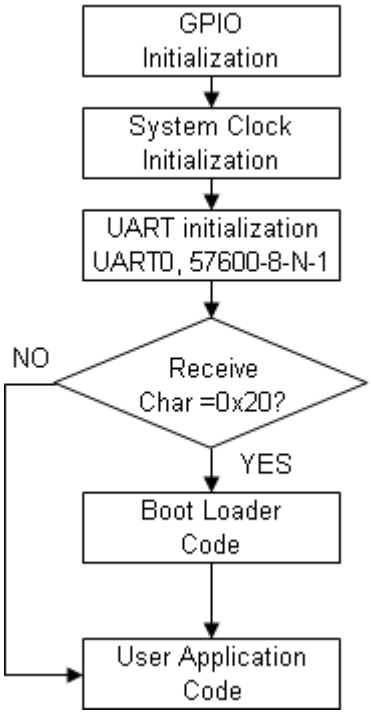
- 57600 baud rate
- No parity
- 8 data bits
- 1 stop bit
- No flow control

## Boot Loader Flow

Figure 7 shows the typical flow of a boot loader execution, which comprises UART initialization, the transfer of boot loader code and the transfer of user application code.

The `main` program enters the boot loader code when the space bar (ASCII character code `0x20`) and the MCU's reset button are simultaneously pressed and released. The boot loader code then downloads the hex file to the MCU's Flash memory (for details about this function, see the Boot Loader Code section on page 11). The program then jumps to

the starting address of the user's downloaded application code, which executes in Flash memory.




```
Assembly Code
main:
srp   #%E0
ldx   SPL, #LOW(STACK_POINTER)
ldx   SPH, #HIGH(STACK_POINTER)


call  init_gpio
call  init_sysclk
call  init_uart


call  delay
ldx   r0,U0RXD
cp    r0,#%20



jp    z,bootloadercode

application_code:
ld    r2,#HIGH(AppCodeAddress_Start)
ld    r3,#LOW(AppCodeAddress_Start)
ldc   r0,@rr2
inc   r3
ldc   r1,@rr2
jp    @rr0
```

**Figure 7. Main Function Flow Diagram of the Z8F082A and Z8F1680 Boot Loaders**

## Boot Loader Code

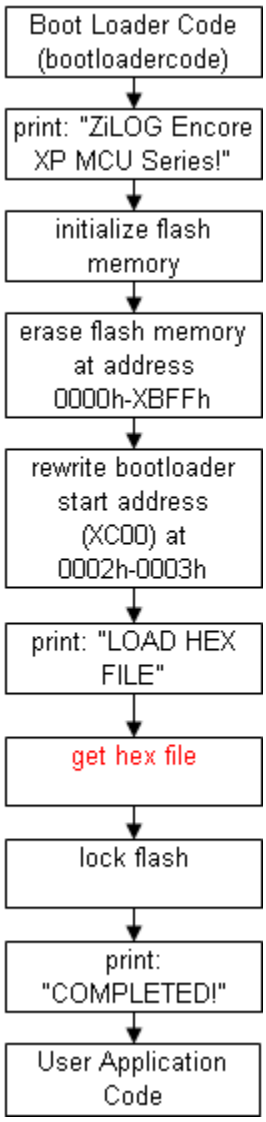
The boot loader code is responsible for reading the hex file coming from the UART and downloading it to Flash memory in the address ranges 0000h–0002h and 0004h–XBFFh; the latter is defined as user application code memory space. The remaining portion of the memory, in the XC00h–XFFFh address range, is boot loader code memory in which the boot loader program resides.

Boot loader code operation depends on the following sequence of functions, which is also shown in workflow format in Figure 8 on page 13.

1. The boot loader code starts; it displays Zilog Z8 Encore! XP MCU Series in the HyperTerminal window.
2. Flash memory initialization, during which the clock frequency is set for correct operation of MCU Flash memory.

3. Flash memory erasure, in which Flash memory is reset within the address range 0000h–XFFFh. This address range contains the user's application code and the reset address of the boot loader (0002h–0003h). Flash memory is erased so that new data can be written to Flash memory.
4. A boot loader address rewrite, in which the starting address of the program is restored to the starting address of Flash memory. The data string XC00h[2] is written to addresses in the range 0002h to 0003h.
5. LOAD HEX FILE is displayed in the HyperTerminal window to indicate that the MCU is ready to load the application code.
6. When the hex file is sent, the Get Hex File function writes the data to Flash memory, then performs a checksum routine.
7. After the data is completely written to Flash memory, Flash memory is locked to prevent the MCU from overwriting existing application code.
8. HyperTerminal displays COMPLETED!, indicating that the application code hex file has been successfully downloaded to the MCU.
9. Finally, the program counter shifts to the user application code starting address to implement the downloaded application code.

2. The value of *X* is dependent on the amount of MCU memory. The greater the amount of memory, the higher the value of *X* can be.




```
Assembly Code
bootloadercode:

ld    r0, #HIGH(print_Bootloader)
ld    r1, #LOW(print_Bootloader)
call  puts

call  init_flash
ld    r0, #HIGH(print_pleasewait)
ld    r1, #LOW(print_pleasewait)
call  puts

ld    r2,#HIGH(AppCodeErasePage)
ld    r3,#LOW(AppCodeErasePage)
call  erase_flash_address

call  write_bootloader_reset_vector

ld    r0, #HIGH(print_Load_HEX_file)
ld    r1, #LOW(print_Load_HEX_file)
call  puts

call  get_hex_line

call  lock_flash

ld    r0, #HIGH(print_completed)
ld    r1, #LOW(print_completed)
call  puts

jp    application_code
```

**Figure 8. Flow Diagram of Boot Loader Code**

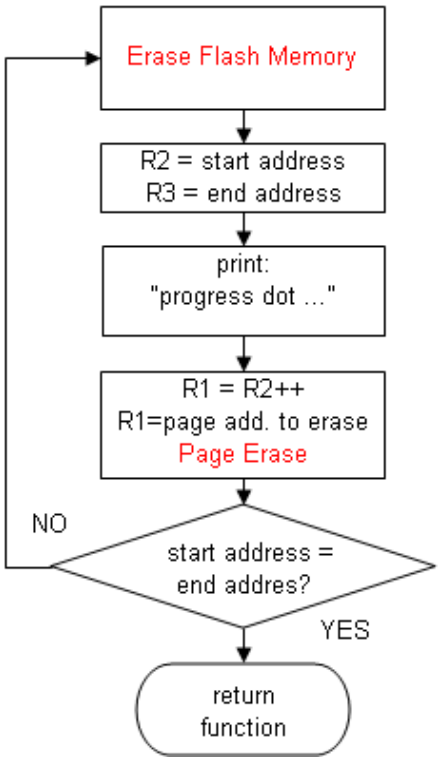
## Initialize Flash Memory

The Initialize Flash Memory function is used to configure the Flash memory settings, including the clock frequency, as indicated in the following code.

```
init_flash:
  ldx          FFREQH,#%15; initialized clock frequency of Flash
  ldx          FFREQL,#%99
  ret
```

## Erase Flash Memory

The Erase Flash Memory function shown in Figure 9 is responsible for erasing the user application code address (0000h–XBFFh) excluding the boot loader code address (XC00h–XFFFh). Register R2 is the starting address while R3 is the ending address of Flash memory to be erased.

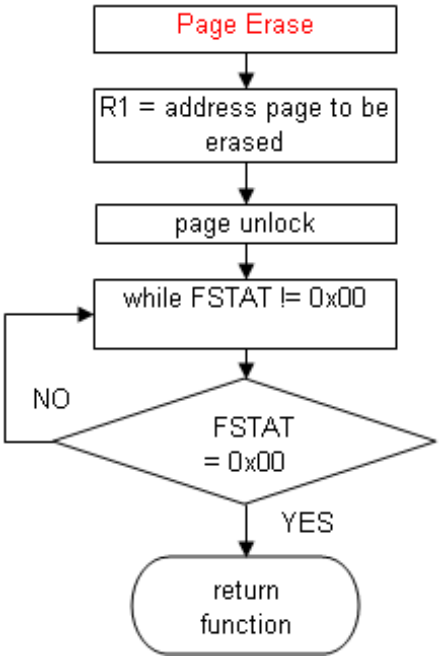



```
Assembly Code
erase_flash_address:

ld     r0,#%2E
call   putc

ld     r1,r2
call   page_erase
inc    r2

cp     r2,r3
jp     c, erase_flash_address

ret
```

**Figure 9. Flow Diagram of Erase Flash Memory**

## Page Erase

The Page Erase memory shown in Figure 11 is used to erase the page of Flash memory at the given address. The Z8 Encore! XP Flash memory has 8 to 48 pages (depending on MCU Flash memory) which each pages contains 512 bytes (200h). The boot loader can only erase the lower pages (address range 0000h–XBFFh) since the last two page is allotted to boot loader code (XC00h–XFFFh). Register R1 is used as the page address of Flash memory to be erased.




```
Assembly Code
page_erase:

call  page_unlock

ldx                         FCTL,#%95
flash_stat:
ldx                         r0,FSTAT

btjnz 4,r0,flash_stat

ret
```

**Figure 10. Flow Diagram of Page Erase**

## Page Unlock

The Page Unlock function is used to unlock Flash memory at a specified address page. This is necessary for writing and erasing Flash memory at a specified address page.

Register R1 is used as the page address of Flash memory to be unlocked.

```
Assembly Code
page_unlock:
ldxFPS,R1       ; page to be unlock FLASH Page Select
ldxFCTL,#%73    ; first unlock command to Flash control reg FCTL
ldxFCTL,#%8C    ; second unlock command to Flash control reg FCTL
ldxFPS,R1       ; page to be unlock FLASH Page Select
ret
```

## Lock Flash

The Lock Flash function is used to protect Flash memory from writing or erasing its content.

```
Assembly Code
lock_flash:
ldxFSTAT,#%00   ; clear to lock Flash memory
ret
```

### Write Boot Loader Start Address

The Write Boot Loader Start Address function is used to rewrite the reset address of the boot loader code after the Erase Flash function is implemented; as a result, the value of the address range 0000h–XBFFh is reset to FFh and the reset vector (0002h–0003h) is reset to FF FFh; however, the value of XC00h[3] must be rewritten.

```
Assembly Code
write_bootloader_start_address:
ld        R1,#%00                   ; R1 = unlock page address (0x0000-0x01FF)
call      page_unlock               ; unlock Flash memory

ld        R2,#HIGH(Reset_Vector_Address); high byte address of reset vector
ld        R3,#LOW(Reset_Vector_Address) ; low byte address of reset vector

ld        R0,#HIGH(BootCodeAddress_Start); Boot Loader Code Start Address high
byte
ldc       @rr2,R0                   ; load data (BootCodeAddress_Start) to
address (0x0002)

ldc       R1,@rr2                   ; Verify that the Flash memory space was
                                    ; written to correctly
cp        R0,R1
jp        nz,FLASH_verify_fail      ; No return terminate program

inc       R3
ld        R0,#LOW(BootCodeAddress_Start); Boot Loader Code Start Address low
byte
ldc       @rr2,R0                   ; load data (BootCodeAddress_Start) to
address (0x0003)

ldc       R1,@rr2                   ; Verify that the Flash memory space was
                                    ; written to correctly
cp        R0,R1
jp        nz,FLASH_verify_fail      ; No return terminate program
call      lock_flash                ; lock Flash memory (its an auto lock)
ret
```

### Get Hex File

The Get Hex File function shown in Figure 11 is responsible for reading the hex file and storing it in Flash memory pertinent to the following sequence.

1. The received data is checked. If the received character is ':', the starting line of the hex file is indicated.
2. All ASCII characters are converted to the Intel Hex 32 file format. Essentially, ASCII characters A to F (41h–46h) are converted to the numbers 10–15 (Ah–Fh) while

3. The value of *X* is dependent on the amount of MCU memory. The greater the amount of memory, the higher the value of *X* can be.

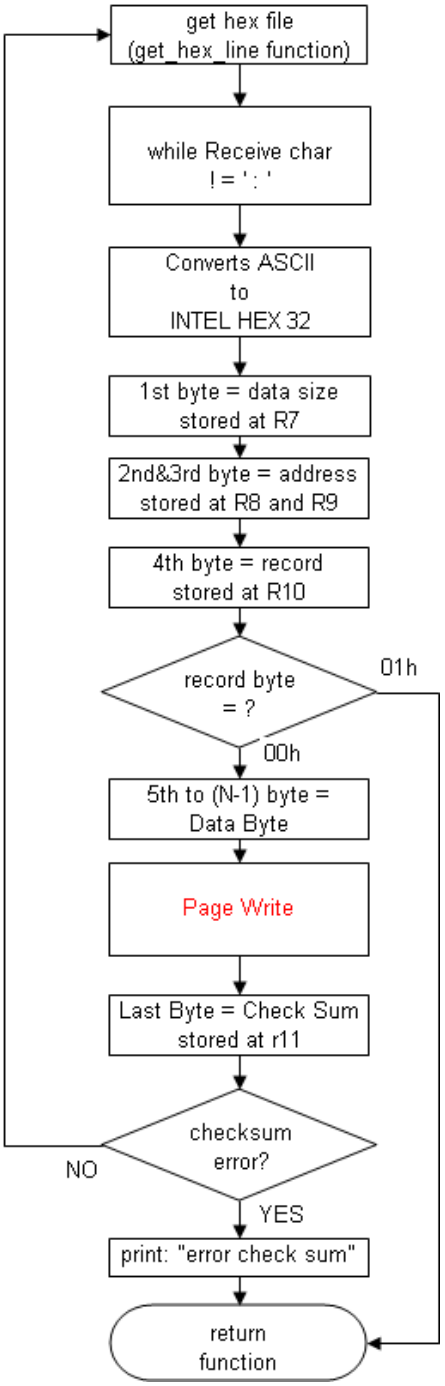
ASCII characters 0–9 are converted to the numbers 0 to 9 (30h–39h) remain the same.

3. The first byte indicates the amount of data in a line; this amount is stored as a value in register 7.
4. The second byte indicates the MSB of the address and the third byte is the LSB of the address; both are stored as values in register R8 for MSB and register R9 for LSB. The address indicates the location of the data to be stored in Flash memory.
5. The fourth byte indicates the record byte of the data. The record byte is used to determine whether the data should be stored at the normal address, at the extended address, or at the end-of-file address. Normal addressing is represented by the value 00h, while end-of-file addressing is represented by 01h. If end-of-file addressing is detected, the function defaults to the Return command.
6. The fifth to (N–1) byte indicates the data to be stored in Flash memory. For example, if the data size stored in R7 is X, then there are X number of data bytes in a line.
7. The Page write function is called to write the data bytes to its specified address.
8. The final byte (N) indicates a checksum, which is used to check for errors during communication. The checksum byte must be equal to the two's complement of the total value of the first byte to the (N–1) byte; see the following equation.

   Checksum = FFh and [ FFh – (1st byte + 2nd Byte + …. + (N–1) byte) + 01h ]

   Failure to satisfy the above condition will result in program termination and will display the following statement in HyperTerminal:

   ```
   error: checksum
   ```




```
Assembly Code
get_hex_line:
;-----------GET START LINE---------------------
call   receive_char
cp     r0,#%3A
jp     nz,get_hex_line

clr    r14                    ; clear check sum
accumulator

;-----------GET DATA SIZE----------------------
call   ascii_to_intelhex
ld     r7,r0

;-----------GET ADDRESS BYTE-------------------
call   ascii_to_intelhex
ld     r8,r0
call   ascii_to_intelhex
ld     r9,r0

;-----------GET RECORD BYTE--------------------
call   ascii_to_intelhex
ld     r10,r0
cp     r10,#%01
jp     z,get_line_end

;-----------GET DATA BYTE----------------------
ld     r6,r7
ld     r12,r8
ld     r13,r9
call   addition_16bit

ld     r2,r8
ld     r3,r9
ld     r4,r12
ld     r5,r13

call   page_write

;------------GET CHECK SUM---------------------
call   ascii_to_intelhex
ld     r11,r0
sub    r14,r0
ld     r0,#%FF
sub    r0,r14
inc    r0

cp     r0,r11
jp     z,get_hex_line

ld     r0, #HIGH(print_error)
ld     r1, #LOW(print_error)
call   puts
ld     r0, #HIGH(print_error_checksum)
ld     r1, #LOW(print_error_checksum)
call   puts
jp     $
;-----------END GET LINE----------------------
get_line_end:
call   lock_flash
ret
```

**Figure 11. Flow Diagram of Get Hex File**

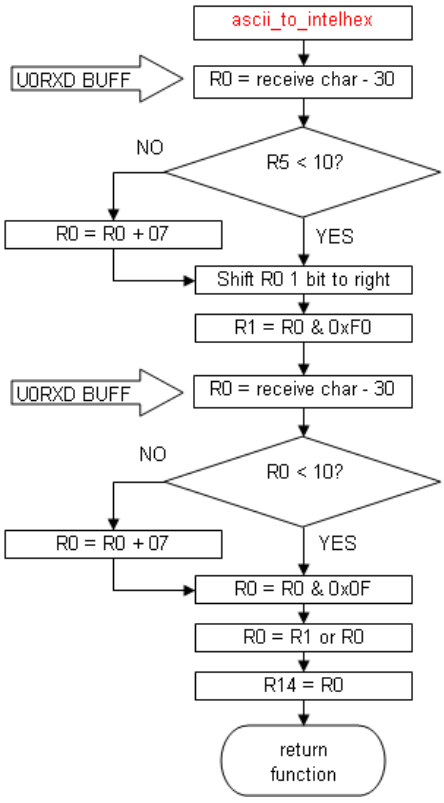
## Receive Character

The Receive Character function is used to get the character from the U0RXD registers buffer to the R5 register. The U0STAT register is used to indicate if character is received from the U0RXD register buffer.

```
Assembly Code
receive_char:
ldx R0,%F41             ; Read the UART0 status register
and R0,#%80             ; to check for the received character
jr  z,receive_char
ldx R0, U0RXD           ; store received byte
ret
```

## ASCII to INTEL HEX 32

The ASCII to INTEL HEX 32 function is used to convert the ASCII characters to INTEL 32 data byte. The Flow Diagram is shown below.




```
Assembly Code
ascii_to_intelhex:
call    receive_char
add     r0,#%BF
jp      c,ascii_to_intelhex_H
add     r0,#%07

ascii_to_intelhex_H:
add     r0,#%0A
ld      r1,r0
swap    r1
ld      r0,r1
and     r0,#%F0
ld      r1,r0
call    receive_char
add     r0,#%BF
jp      c,ascii_to_intelhex_L

add     r0,#%07
ascii_to_intelhex_L:
add     r0,#%0A
and     r0,#%0F
or      r0,r1

add     r14,r0                  ; add byte to
                                ; check sum accumulator
ret
```

**Figure 12. Flow Diagram of ASCII to INTEL HEX 32**

## Page Write

The Page Write function is responsible for writing the data bytes to the specified Flash address. Observe the following steps to write the correct data bytes to the Flash address.

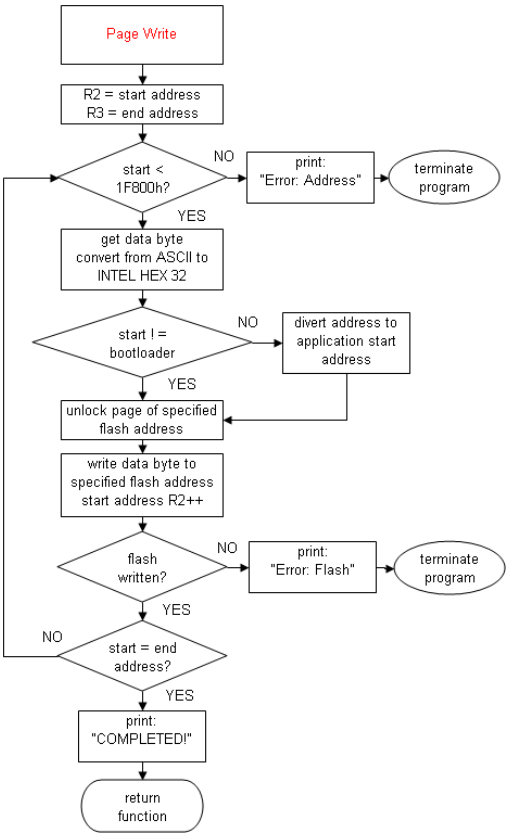
1. Use Register R2 to store the starting address value and Register R3 to store the ending address.
2. The R2 start (i.e., current) address is checked to determine if it exceeds the restricted address, which is the boot loader address (1F800h to 1FFFFh). Failure to satisfy this condition will result in program termination and will display the following error message in HyperTerminal:

```
Error:
Address: Change ROM=0000-0BFD(Z8F042A Series)
Address: Change ROM=0000-1BFD(Z8F082A Series)
Address: Change ROM=0000-3BFD(Z8F1680 Series)
```

3. The data byte received from the UART0 is converted from ASCII to INTEL HEX 32 format.
4. The current address of Flash memory is unlocked.
5. The data byte is written to the specified Flash address. If the current address is equal to the boot loader starting address then the value of the current address is diverted to the application code starting address.
6. Flash memory is checked to determine if it has actually written the value of the data byte received. This also prevents the corrupted program to be programmed in Flash memory. Failure to satisfy this condition will result in program termination and will display the following error message in HyperTerminal:

```
Error: Error verifying Flash Write
```

7. Finally, the current address is compared to the ending address to determine if it is the end of the hex file data. If it is the end of hex file the HyperTerminal will display: COMPLETED!




```
Assembly Code
page_write:
jp      write_compare_address
write_continue:
ld      r0,#HIGH(BootCodeAddress_Start)
cp      r2,r0
jp      nc,print_error_write_address

cp      r2,#%00
jp      nz,write_application_data
cp      r3,#%02                 ; if 0001h < address
                                ; < 0004h
jp      z,write_application_entry_vector
cp      r3,#%03
jp      nz,write_application_data; else

write_application_entry_vector:
add     r2,#HIGH(AppCodeAddress_Sub2)
add     r3,#LOW(AppCodeAddress_Sub2)
ld      r1,r2
srl     r1                      ; address of page to
be unlocked
call    page_unlock
call    ascii_to_intelhex
ldc     @rr2,r0
ldc     r6,@rr2                 ; Verify the written
                                ;data to flash
cp      r6,r0
jp      nz,FLASH_verify_fail

sub     r2,#HIGH(AppCodeAddress_Sub2)
sub     r3,#LOW(AppCodeAddress_Sub2)
jp      continue_write

write_application_data:
ld      r1,r2
srl     r1                      ; address of page to
be unlocked
call    page_unlock

call    ascii_to_intelhex
ldc     @rr2,r0
ldc     r6,@rr2
cp      r6,r0
jp      nz,FLASH_verify_fail
continue_write:
ld      r6,#%01
ld      r13,r3                  ; LSB of 16 bit
adder(r13) = r3
call    addition_16bit          ;
ld      r2,r12                  ; MSB result of 16
bit adder
ld      r3,r13                  ; LSB result of 16
bit adder

ld      r0,#%2E
call    putc                    ; print progress
"...."

write_compare_address:
cp      r2,r4
jp      c,write_continue        ; { end writing}
check_LSB:                      ; else compare LSB
cp      r3,r5
jp      c,write_continue
write_end:
        ret
```

**Figure 13. Flow Diagram of Page Write Function**

## User Application Code

The user application code contains the downloaded application code which resides in the address range 0000h–XBFFh, within which the application start vector resides in address XBFDh.

```
Assembly Code
application_code:   ; application code
  ld    R2,#HIGH(AppCodeAddress_Start); application code starts at
                    ; the value
  ld    R3,#LOW(AppCodeAddress_Start); stored in XBFB-XBFC
  ldc   R0,@rr2
  inc   R3
  ldc   R1,@rr2
  jp    @rr0        ; jump to application code
```

## Puts Function

The Puts function is used to print the string of characters starting with address stored in Register R1.

```
Assembly Code
puts:
  ld      R2,R1         ;R2 = address of the string
  cp      R2,#0         ;ifaddress is 0 return
  jp      eq,lputs3
  jp      lputs1
lputs2:
  ld.ub   R1,(R2)       ; R1 = character from string pointed by R2
  call    putch         ; Call _putch with R1 containing character
  add     R2,#1         ; Increment pointer R2
lputs1:
  cpz.b   (R2)          ; if character pointed by R2 is 0 return
  jp      ne,lputs2     ; else go back to loop
lputs3:
  ld      R0,#0
  ret
```

## Putch Function

The Putch function is used to print the character stored in Register R1.

```
Assembly Code
putch:
  ld      R0,R1
  ext.ub  R5,R0
  cp      R5,#10
  jp      ne,lputch1    ; If (character == \n)
  ld      R1,#13
```

```
  call    send          ; Call _send with character
lputch1:
  ld      R1,R0
  call    send          ; Call _send with character
  ld      R0,#0         ; Return R0 = 0
  ret                   ; Return
```

### Send Function

The Send function is used to transmit the data byte stored in Register R0 using the U0TXD Register Buffer.

```
Assembly Code
send:
  pushmlo <R0>          ; Save register
lsend1:
  ld      R0,#4         ; Send on UART0
  tm.b    U0STAT0,R0
  jp      eq,lsend1     ; while (!(U0STAT0 & %4))
  ld.b    U0TXD,R1      ; U0TXD = R1 (character)
  popmlo  <R0>          ; Restore registers
  ret                   ; Return
```

### Delay Function

The delay function is used to delay the next instruction.

```
Assembly Code
delay:
  ld    r2,#%FF
loop1:

  ld    r1,#%FF
  djnz  r1,$
  djnz  r2,loop1
  ret
```

# Testing/Demonstrating the Application

Testing this application involves downloading the boot loader program and loading the boot loader hex code pertinent to the following requirements and procedures.

## Equipment Used

- Z8 Encore! XP F1680 28-Pin Series Development Kit, including development board, power supply, USB interface and Zilog XTools

- Z8 Encore! XP F082A Series (28-pin) Development Kit, including development board, power supply, USB interface and Zilog XTools
- Z8 Encore! XP F042A Series (8-Pin) Development Kit, including development board, power supply, USB interface and Zilog XTools
- RS-232 cable

To download the Z8F082A or Z8F1680 boot loader to the Z8 Encore! XP MCU, observe the following instructions.

1. Launch ZDSII for Z8Encore! 5.0.0.
2. From the **File** menu in ZDSII, click **Open** and select **Z8EncoreXP_bootloader** from the Application Code Source. The **Boot Loader** dialog box is displayed.
3. From the **Project** menu in the **Boot Loader** dialog box, choose **Settings** to open the **Project Settings** dialog box. The address settings in this dialog must be the same as those shown in Figures 14 through 16, depending on the MCU used.
4. In the **Project Settings** dialog box, click the **Debugger** tab, then click the **Setup** button to open the **Configure Target** dialog box.
5. In the **Configure Target** dialog, set the Clock Frequency to 5.52960MHz, then click **OK**.

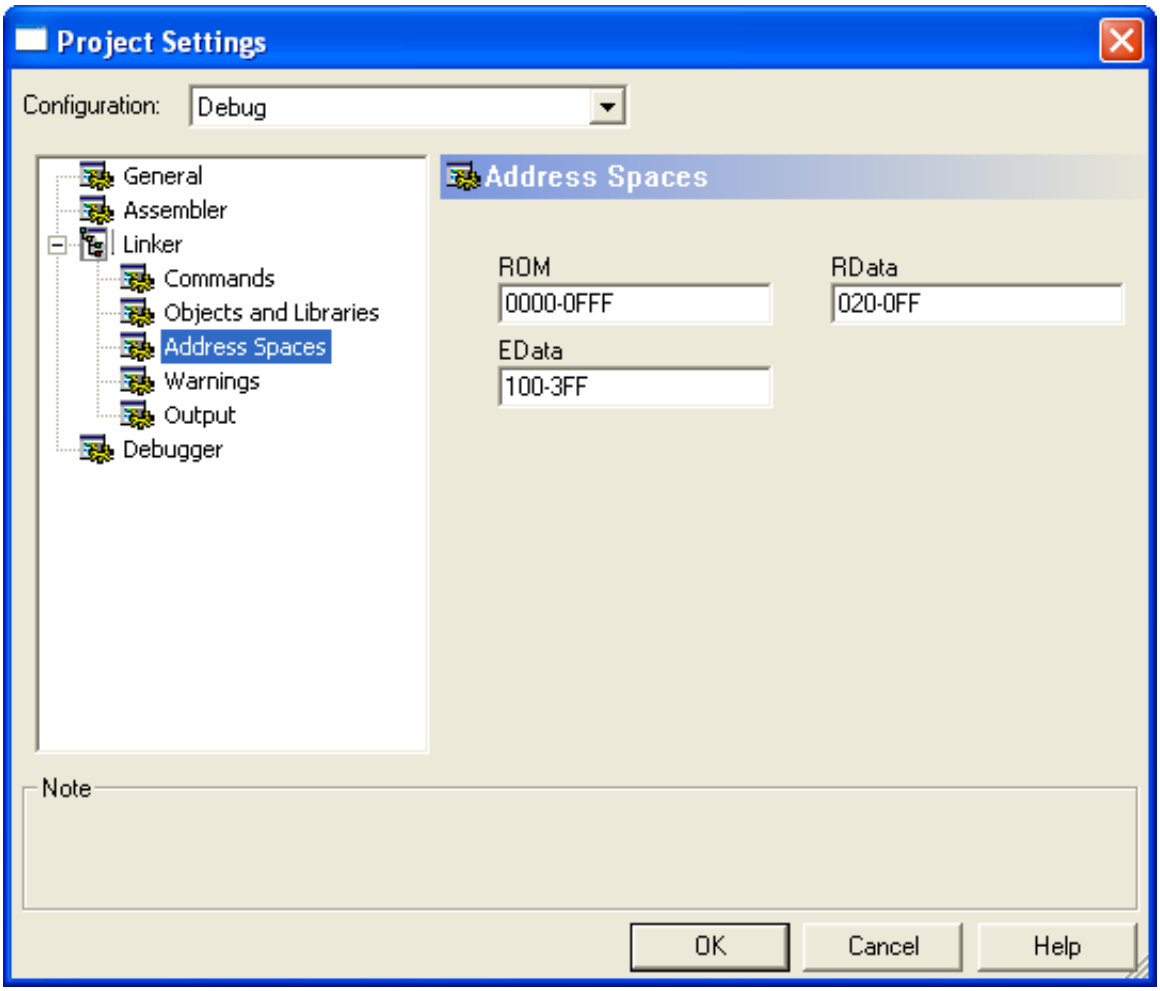



**Figure 14. Address Spaces for the Z8F042A 4K Boot Loader (Designed for the Z8F042A MCU)**

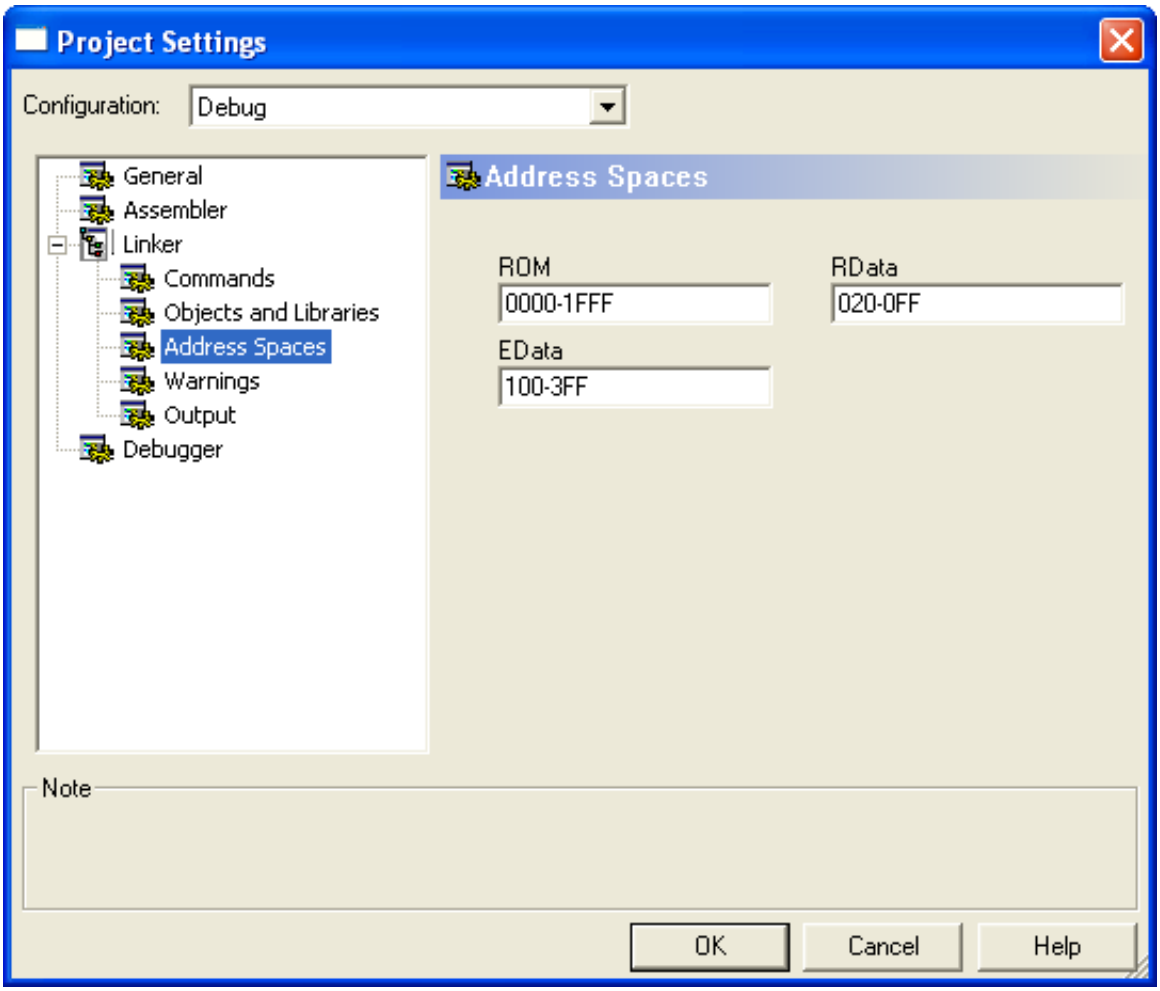



**Figure 15. Address Spaces for the Z8F082A 8K Boot Loader (Designed for Z8F082A Series MCUs)**

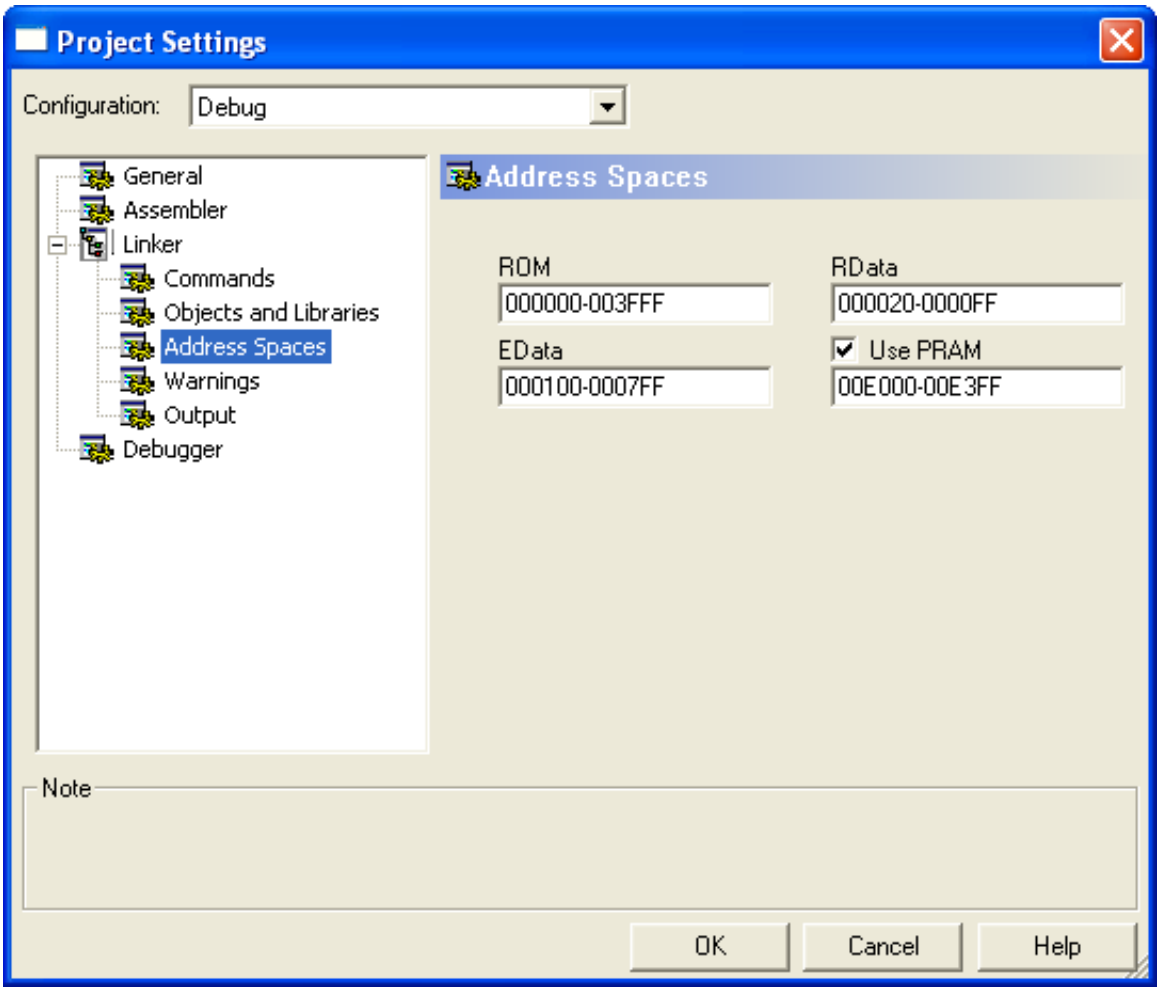



**Figure 16. Address Spaces for the Z8F1680 16K Boot Loader (Z8F1680 Series)**

6. Compile and download the program to the Z8 Encore! XP Development Board.

## Load Hex Code

After downloading the boot loader program to the Z8 Encore XP MCU, observe the following steps to load the new application code.

1. Unplug the ZDS II IDE from the MCU and connect the RS-232 cable to your PC
2. Launch HyperTerminal and set the parameters to:
   - 57600 baud rate
   - 8 data bits
   - No parity
   - 1 stop bit

– No flow control

3. Press the space bar on the keyboard of your PC, and at the same time press the reset button for the MCU.
4. Release the space bar on the keyboard, then release the reset button for the MCU. HyperTerminal should display a screen similar to the example shown in Figure 17.

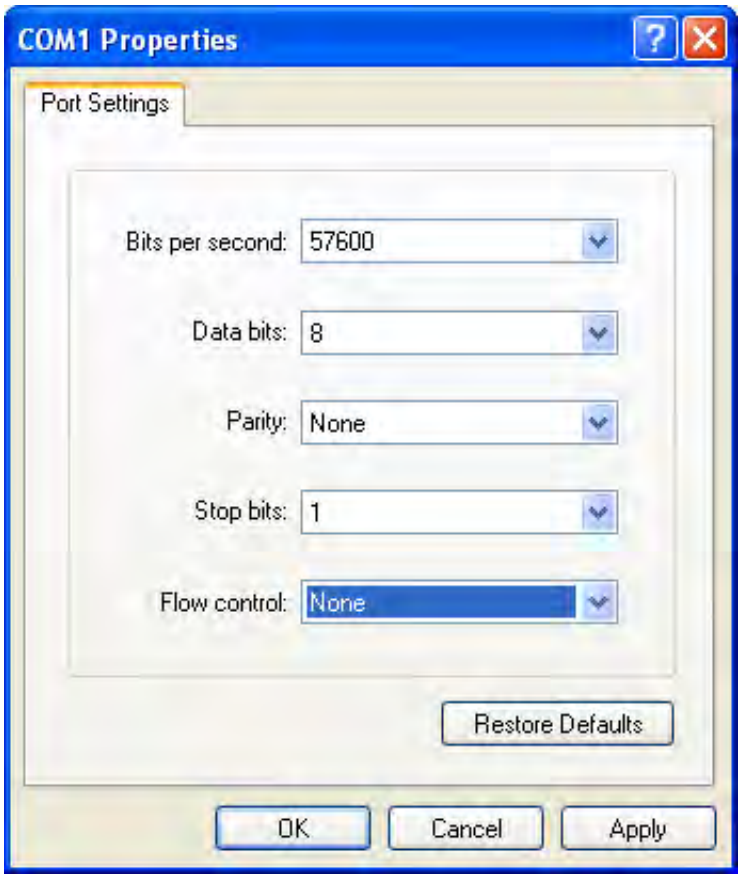



**Figure 17. HyperTerminal Port Settings**

5. Load the hex file to the MCU. In HyperTerminal, click **Transfer**, then choose **Send Text File**. Depending on which MCU you're working with, search for and open one of the following hex files:

| Device | Hex File |
|---|---|
| Z8F042A | `application_code_ZF042A.hex` |
| Z8F082A | `application_code_ZF082A.hex` |
| Z8F042A (8-pin) | `application_code_ZF042A_8pin.hex` |
| Z8F082A (8-pin) | `application_code_ZF082A_8pin.hex` |
| Z8F1680 | `application_code_ZF1680.hex` |

6. The hex file you selected will load into memory. After the hex file is loaded, HyperTerminal displays `"COMPLETE!"`.
7. The program executes the application code. If the application code does not execute, press the reset button of the Z8 Encore! XP MCU to restart the program.

**Note:** For the 8-pin versions of the Z8F042A and Z8F082A MCUs, power cycle the application board to reset the MCU.

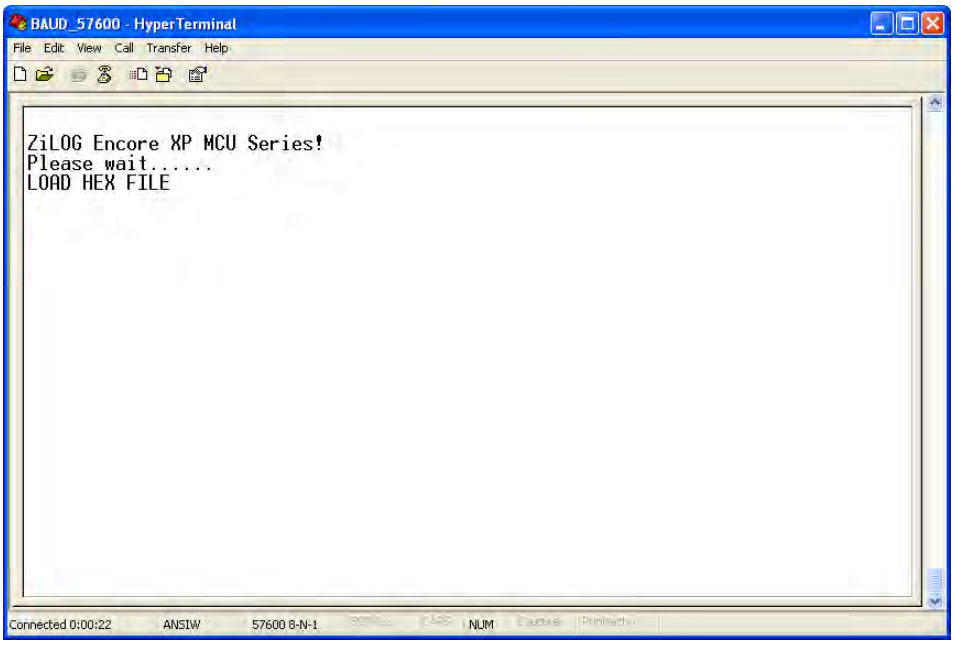



**Figure 18. Boot Loader Initialization in HyperTerminal**

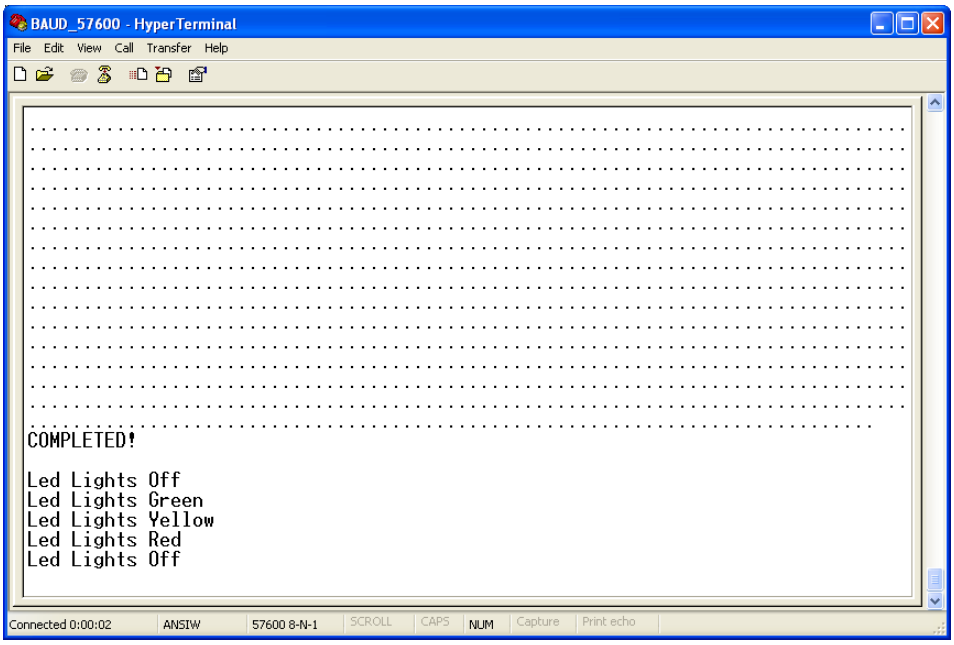



**Figure 19. Boot Loader Completion in HyperTerminal**

## Configure Address Space

Prior to creating and downloading your program via the Z8 Encore! XP boot loader, configure the memory address space for your ZDSII project by observing the following procedure.

1. Use the ZDSII project settings below depending on the MCU used; see Figures 20 through 22 for reference.
   - For Z8F042A 4K Series MCUs, ROM = `0x0000` to `0x0BFD`
   - For Z8F082A 8K Series MCUs, ROM = `0x0000` to `0x1BFD`
   - For Z8F1680 16K Series MCUs, ROM = `0x0000` to `0x3BFD`
2. Build and compile your application code.
3. Follow the Load Hex Code procedure (see page 27) to load your hex file to the Z8 Encore! XP development board.

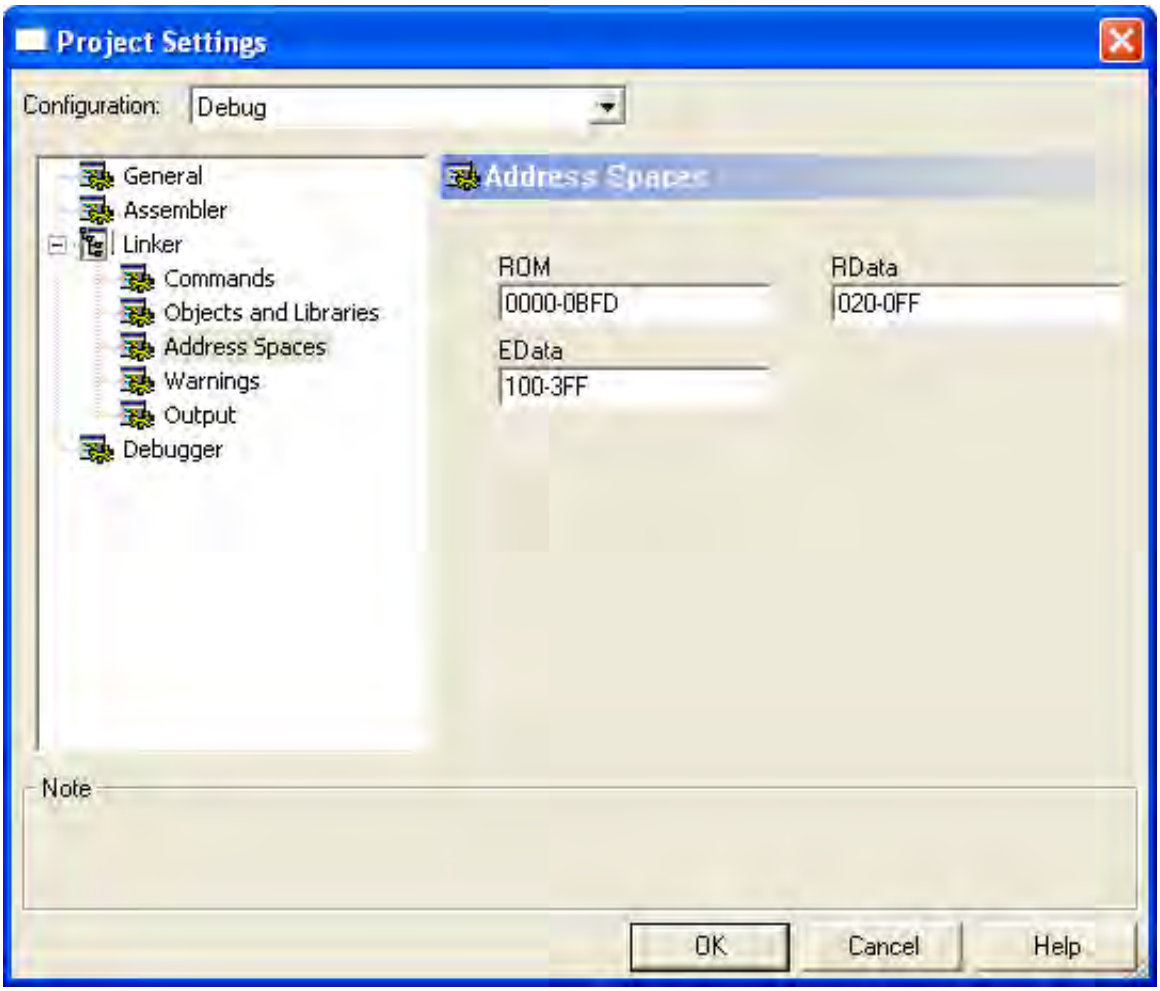



**Figure 20. Address Space for Z8F082A 4K Boot Loader User Application Code (Designed for the Z8F042A MCUs)**

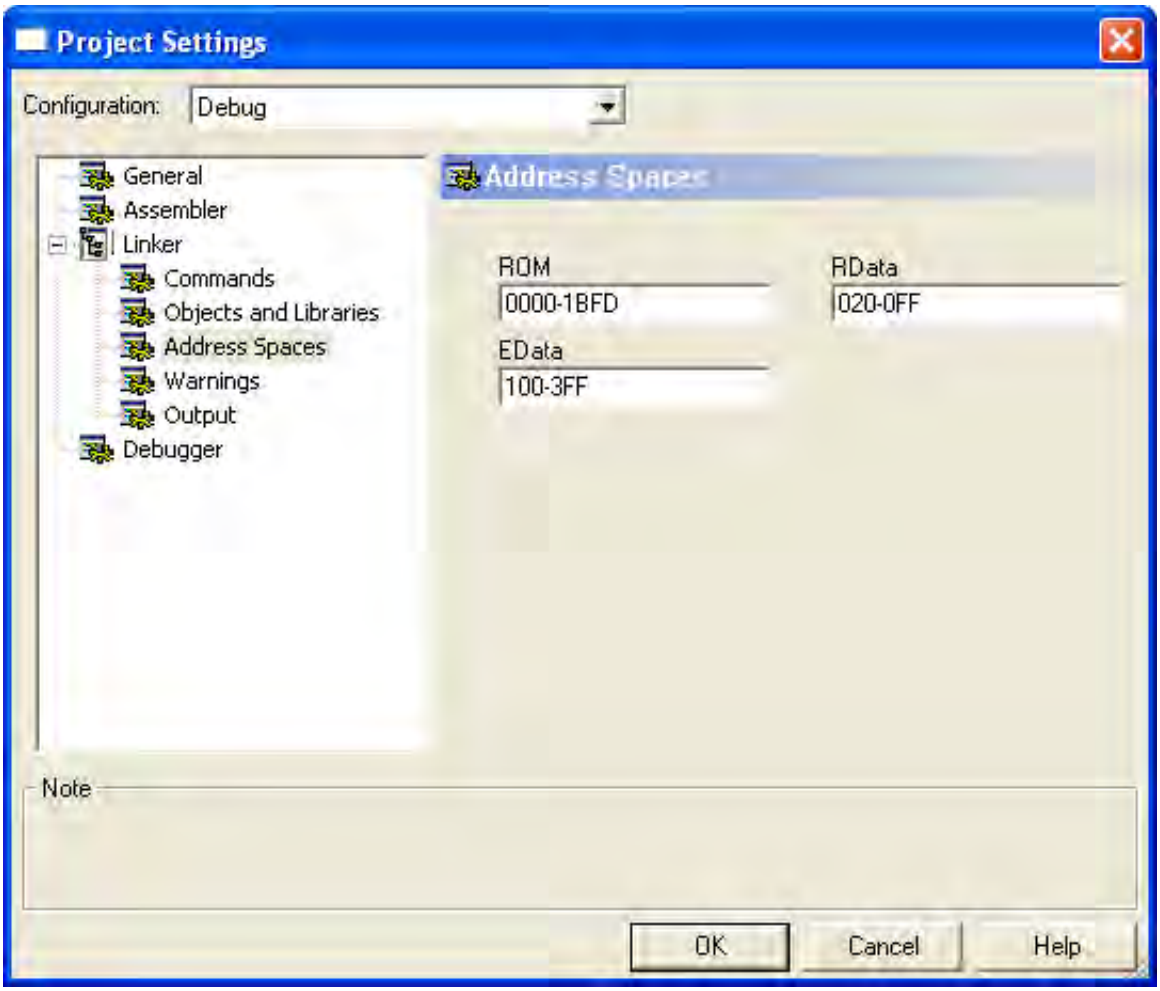



**Figure 21. Address Space for Z8F082A 8K Boot Loader User Application Code (Designed for the Z8F082A Series MCUs)**

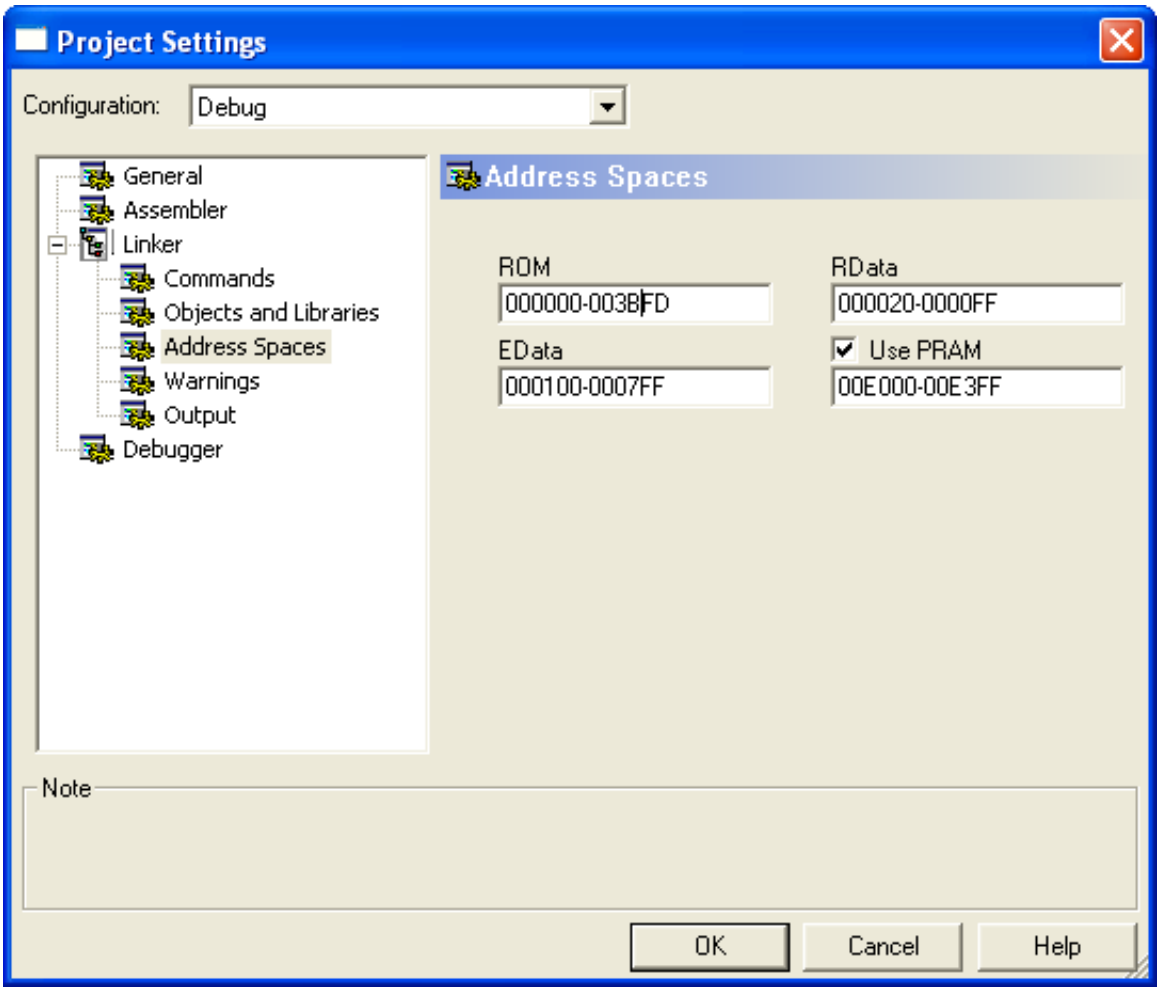



**Figure 22. Address Space for Z8F1680 16K Boot Loader User Application Code**

# Summary

The Z8F082A and Z8F1680 boot loader is used as an alternative firmware downloader to the MCU aside from using the Debug Interface with Zilog's Smart Cable. The only limitation is that MCU Flash memory was reduced by 1KB due to the boot loader code residing at the highest address space in the MCU.

# References

| | |
|---|---|
| PS0228 | Z8 Encore! XP F082A Series Product Specification |
| PS0250 | Z8 Encore! XP F1680 Series Product Specification |
| UM0128 | eZ8 CPU User Manual |

# Appendix A. Assembly Code for Z8F082A and Z8F1680 Boot Loader

This appendix describes each of the assembly code functions of the Flash boot loader for MCUs based on the eZ8 CPU architecture.

| | |
|---|---|
| **Function** | **Pre Code Setting** |
| **Description** | Contains the pre main code settings |
| **Included Functions** | • Include file "ez8.inc".<br>• Code Segment address of main code.<br>• Variable equivalent. |
| **Registers** | None. |

**Code**

```
INCLUDE "ez8.inc"
;-------------------------------------------------------------------------
STACK_POINTER           EQU %00FF         ; Stack starting address
SYSFREQ                 EQU 5529600       ; System clock 5.5296 MHz
UART_BAUD               EQU 57600         ; UART baud rate selection
UART_BRG                EQU ((SYSFREQ + (UART_BAUD * 8)) / (UART_BAUD * 16))
;-------------------------------------------------------------------------
; Vectors assignments for used Vectors
VECTOR RESET = main

; CPU Family of Z8F042A MCU
IF (__Z8F042A=1) || (__Z8F042AXB=1)
  DEFINE CODESEG, SPACE = ROM, ORG = 7168 ; 07168 = 1C00h boot loader code
                                          ; address
  SEGMENT CODESEG
  Reset_Vector_Address  EQU %0002         ; reset vector address = 0002h
  BootCodeAddress_Start EQU %0C00         ; Boot Loader Code Start Address =
                                          ; 0C00h
  AppCodeAddress_Start  EQU %0BFE         ; Application Code Start Address =
                                          ; 0BFEh
  AppCodeAddress_Sub2   EQU %0BFC         ; 0BFCh = AppCodeAddress(0C00h) -
                                          ; reset vector
                                          ; Address (0002h)
  AppCodeErasePage      EQU %0006         ; number of erasable pages = 8 pages
                                          ; - 2
                                          ; boot loader code pages
ENDIF

; CPU Family of Z8F082A 8K MCU
IF (__Z8F082A=1) || (__Z8F082AXB=1)
  DEFINE CODESEG, SPACE = ROM, ORG = 7168 ; 07168 = 1C00h Boot Loader code
address
  SEGMENT CODESEG
```

```
  Reset_Vector_Address  EQU %0002          ; reset vector address = 0002h
  BootCodeAddress_Start EQU %1C00          ; Boot Loader Code Start Address =
                                           ; 1C00h
  AppCodeAddress_Start  EQU %1BFE          ; Application Code Start Address =
                                           ; 1BFEh
  AppCodeAddress_Sub2   EQU %1BFC          ; 1BFCh = AppCodeAddress (1C00h) -
                                           ; reset vector
                                           ; Address (0002h)
  AppCodeErasePage      EQU %000E          ; number of erasable pages = 16
                                           ; pages -
                                           ; 2 boot loader code pages
ENDIF
; CPU Family of 1680 16K MCU
IF(__Z8F1680XH=1) || (__Z8F1680XJ=1) || (__Z8F1680XM=1) || (__Z8F1680XN=1)
  DEFINE CODESEG, SPACE = ROM, ORG = 15360; 15360h = 3C00h Boot Loader code
                                           ; address
  SEGMENT CODESEG
  Reset_Vector_Address  EQU %0002          ; reset vector address = 0002h
  BootCodeAddress_Start EQU %3C00          ; Boot Loader Code Start Address =
                                           ; 3C00h
  AppCodeAddress_Start  EQU %3BFE          ; Application Code Start Address =
                                           ; 3BFEh
  AppCodeAddress_Sub2   EQU %3BFC          ; 3BFCh = AppCodeAddress (3C00h)
                                           ; - reset vector Address (0002h)
  AppCodeErasePage      EQU %001E          ; number of erasable pages = 32
                                           ; pages
                                           ; - 2 boot loader code pages
ENDIF
```

| | |
|---|---|
| **Function** | **Main** |
| **Description** | Contains the backbone of the program. |
| **Included Functions** | • Bootloadercode<br>• Applicationcode |
| **Registers** | R0, R1, R2, R3 and R4 are used as variable registers. |

**Code**

```
main:
    srp     #%E0                         ; Working registers E0-EF
    ldx     SPL, #LOW(STACK_POINTER)     ; Initialize stack pointer
    ldx     SPH, #HIGH(STACK_POINTER)
                                         ; else
    call    init_gpio                    ; initialization of GPIO
    call    init_sysclk                  ; initialization of system clock
    call    init_uart                    ; initialization of UART

    call    delay
    ldx     R0,U0RXD                     ; Receive char through UART0_RX
```

```
    cp      R0,#%20                         ; if U0RXD = 20h
    jp      z,bootloadercode                ; go to bootloadercode
                                            ; else
application_code:                           ; application code
    ld      R2,#HIGH(AppCodeAddress_Start)  ; application code starts at the
                                            ; value
                                            ; stored
    ld      R3,#LOW(AppCodeAddress_Start)   ; in XBFE-XBFF
    ldc     R0,@rr2
    inc     R3
    ldc     R1,@rr2

    jp      @rr0                            ; jump to application code
```

| | |
|---|---|
| **Function** | **Bootloadercode** |
| **Description** | Contains Flash memory initialization plus the unlock flash, lock flash, erase flash, write flash and get hex line functions, which are used to read the hex files in the application code. |
| **Included Functions** | • Init_flash<br>• Puts<br>• Erase_flash_address<br>• Write_bootloader_start_address<br>• Get_hex_line<br>• Lock_flash |
| **Registers** | R0, R1, R2 and R3 are used as variable registers. |

**Code**

```
bootloadercode:
    ld      R0, #HIGH(print_Bootloader)
    ld      R1, #LOW(print_Bootloader)
    call    puts
    call    init_flash                      ; initialized Flash memory
    ld      R0, #HIGH(print_pleasewait)
    ld      R1, #LOW(print_pleasewait)
    call    puts                            ; print "Please wait..."

    ld      R2,#HIGH(AppCodeErasePage)      ; R2 = starting address of Flash
                                            ; to be erased (0000h)
    ld      R3,#LOW(AppCodeErasePage)       ; R3 = ending address to flash be
                                            ; erased (XC00h)
    call    erase_flash_address             ; erase address (0000h-XBFFh)
    call    write_bootloader_start_address  ; rewrite boot loader code address
                                            ; (0002h-0003h)

    ld      R0, #HIGH(print_Load_HEX_file)
    ld      R1, #LOW(print_Load_HEX_file)
```

```
    call    puts                            ;print "LOAD HEX FILE"
    call    get_hex_line                    ; read the hex file of the
                                            ; application code
    call    lock_flash                      ; lock Flash

    ld      R0, #HIGH(print_completed)
    ld      R1, #LOW(print_completed)
    call    puts                            ; print "COMPLETED!"

    jp      application_code
```

| | |
|---|---|
| **Function** | **init_flash** |
| **Description** | Used to initialize the clock in Flash memory. |
| **Included Functions** | None. |
| **Registers** | R0 and R1 as temporary variables. |

**Code**

```
init_flash:
            ldx     FFREQH,#%15         ; initialized clock frequency of Flash
            ldx     FFREQL,#%99
            ret
```

| | |
|---|---|
| **Function** | **Erase_flash_address** |
| **Description** | Erase the Flash address range specified by the R2(starting address) to R3(ending address). |
| **Registers** | R2 = Starting address of the Flash memory space to be erased.<br>R3 = Ending address of the Flash memory space to be erased. |

**Code**

```
erase_flash_address:
            ld      R0,#%2E
            call    putc                    ; print progress "...."
            ld      R1,R2                   ; loads the starting address to be
                                            ; erase to R1
            call    page_erase
            inc     R2                      ; increment page erase since 1
                                            ; page = 0x200 bytes
            cp      R2,R3                   ; compare if current erase page =
                                            ; ending address page
            jp      c, erase_flash_address  ; repeat if current erase page <
                                            ; ending address page
            ret
```

| | |
|---|---|
| **Function** | **Page_unlock** |
| **Description** | Unlock a page of Flash memory at an address specified by register R1. |
| **Included Functions** | None. |
| **Registers** | R1 = Address of the Flash memory space to be unlocked. |

**Code**

```
page_unlock:
            ldx   FPS,R1    ; page to be unlock FLASH Page Select
            ldx   FCTL,#%73 ; first unlock command to Flash control reg FCTL
            ldx   FCTL,#%8C ; second unlock command to Flash control reg FCTL
            ldx   FPS,R1    ; page to be unlock Flash Page Select
            ret
```

| | |
|---|---|
| **Function** | **lock_flash** |
| **Description** | Locks Flash memory to protect it from being overwritten. |
| **Included Functions** | None. |
| **Registers** | N/A |

**Code**

```
lock_flash:
            ldx     FSTAT,#%00              ; clear to lock Flash memory
            ret
```

| | |
|---|---|
| **Function** | **write_bootloader_start_address** |
| **Description** | Rewrite the starting address of the boot loader code. |
| **Included Functions** | • Page_unlock<br>• Lock_flash |
| **Registers** | R1 = Starting address of the boot loader code.<br>Write (XC00) to address (0002h-0003h). |

**Code**

```
write_bootloader_start_address:
  ld    R1,#%00                     ; R1 = unlock page address (0x0000-0x01FF)
  call  page_unlock                 ; unlock Flash memory

  ld    R2,#HIGH(Reset_Vector_Address); high byte address of reset vector
  ld    R3,#LOW(Reset_Vector_Address) ; low byte address of reset vector

  ld    R0,#HIGH(BootCodeAddress_Start); Boot Loader Code Start Address high
                                    ; byte
  ldc   @rr2,R0                     ; load data (BootCodeAddress_Start) to
                                    ; address (0x0002)

  ldc   R1,@rr2                     ; Verify that the Flash was written
                                    ; correctly
  cp    R0,R1
  jp    nz,FLASH_verify_fail        ; No return

  inc   R3
  ld    R0,#LOW(BootCodeAddress_Start); Boot Loader Code Start Address low
                                    ; byte
  ldc   @rr2,R0                     ; load data (BootCodeAddress_Start) to
                                    ; address (0x0003)

  ldc   R1,@rr2                     ; Verify that the Flash was written
                                    ; correctly
  cp    R0,R1
  jp    nz,FLASH_verify_fail        ; No return terminate program
  call  lock_flash                  ; lock Flash memory (its an auto lock)
  ret
```

| | |
|---|---|
| **Function** | **get_hex_line** |
| **Description** | Read the line of the hex file. Below are the steps in reading the hex file:<br>1. Check if (R0) receive char is ':' or char (0x34)<br>2. Store 1st Hex Byte to Data size (R7)<br>3. Store 2nd Hex Byte to High Byte Address (R8)<br>4. Store 3rd Hex Byte to Low Byte Address (R9)<br>5. Write 4th to (N-1) Hex Byte to the Address specified (depends on the data size)<br>6. Store Last (Nth) Hex Byte Checksum (R11)<br>**Note:** In the write byte, if address = (0x0002-0x00003), then diverts to 0x0BFC-0x0BFD<br>If address = (0x0004-0x0BFB) goes to direct address.<br>If address = (0x0C00-0xFFF) then illegal hex line which can overlap the boot loader. |
| **Included Functions** | • Receive_char<br>• Ascii_to_hex_line<br>• Page_write<br>• Putch |
| **Registers** | R0 = hex byte.<br>R7 = data size.<br>r 8= address byte MSB.<br>r 9= address byte LSB.<br>R10 = record byte.<br>= data byte = # of byte (R6) is equal to data size store in address (R7).<br>R11= checksum.<br>R14= checksum = hex byte(R6) + hexbyte_H(R7) + hexbyte_L(R7) + hexbyte(Nth) = R11. |

**Code**

```
get_hex_line:
  ;--------------------------------GET START LINE--------------------------
  call  receive_char                ; received char is stored at R0
  cp    R0,#%3A
  jp    nz,get_hex_line

  clr   R14                         ; clear checksum accumulator
  ;--------------------------------GET DATA SIZE---------------------------
  ;get_data_size:
  call  ascii_to_intelhex           ; read hex byte
  ld    R7,R0                       ; R7 = data size

  ;--------------------------------GET ADDRESS BYTE------------------------
  ;get_address_byte:
  call  ascii_to_intelhex           ; read hex byte
  ld    R8,R0                       ; R8 = MSB address byte
```

```
  call  ascii_to_intelhex           ; read hex byte
  ld    R9,R0                       ; R9 = LSB address byte

  ;---------------------------------GET RECORD BYTE--------------------------
  ;get_record_byte:
  call  ascii_to_intelhex           ; read hex byte
  ld    R10,R0                      ; R10 = record byte

  cp    R10,#%01                    ; check record type if end of file
  jp    z,get_line_end
  ;---------------------------------GET DATA BYTE----------------------------
  ;get_data_byte
  ld    R6,R7                       ; rr12 = start address (rr8) + data size
                                    ; (R7)
  ld    R12,R8                      ; high byte addend
  ld    R13,R9                      ; low byte addend
  call  addition_16bit              ; addition_16bit (rr12 = rr8 + R6)

  ld    R2,R8                       ; R2 = starting address high byte (R8)
  ld    R3,R9                       ; R3 = starting address low byte (R9)

  ld    R4,R12                      ; R4 = end   address high byte (R12)
  ld    R5,R13                      ; R5 = end   address low byte (R13)
  call  page_write                  ; write the data (R7) to address (rr2)

  ;---------------------------------GET CHECKSUM-----------------------------
  ;get_check_sum:
  call  ascii_to_intelhex           ; read hex byte
  ld    R11,R0                      ; R11 = checksum
  sub   R14,R0                      ; sum of all = data size + byte address +
                                    ; record byte + data bytes
  ld    R0,#%FF
  sub   R0,R14
  inc   R0                          ; checksum calculated = (FFh - sum of
                                    ; all) + 1

  cp    R0,R11                      ; check if checksum = checksum
                                    ; calculated
  jp    z,get_hex_line              ; if checksum != checksum calculated then
go to checksum ok

  ld    R0, #HIGH(print_error)
  ld    R1, #LOW(print_error)
  call  puts                        ;print "print_error"
  ld    R0, #HIGH(print_error_checksum)
  ld    R1, #LOW(print_error_checksum)
  call  puts                        ;print "print_error_checksum"
  jp    $                           ; else terminate program (error on
                                    ; checksum)
  ;---------------------------------END GET LINE-----------------------------
```

```
get_line_end:
  call  lock_flash                  ; lock Flash memory (an auto lock)
ret
```

| | |
|---|---|
| **Function** | **receive_char** |
| **Description** | Get character from U0RXD and store it to R5. |
| **Included Functions** | • Receive_char |
| **Registers** | R0 = hold the received character. |

**Code**

```
receive_char:
  ldx   R0,%F41                     ; Read the UART0 status register
  and   R0,#%80                     ; to check for the received character
  jr    z,receive_char
  ldx   R0, U0RXD                   ; store received byte
  ret
```

| | |
|---|---|
| **Function** | **ascii_to_intelhex** |
| **Description** | Get character from U0RXD through R0 and convert it to INTEL HEX 32. |
| **Included Functions** | • Receive_char |
| **Registers** | R0 = received ascii character.<br>R0 = converted intel hex 32.<br>R14= add the intel hex to the checksum. |

**Code**

```
ascii_to_intelhex:
  call  receive_char            ; received char is stored at R0
  add   R0,#%BF
  jp    c,ascii_to_intelhex_H   ; check if received char > 0x40
  add   R0,#%07

ascii_to_intelhex_H:            ; converts ASCII char to Intel Hex High Byte
  add   R0,#%0A
  ld    R1,R0
  swap  R1
  ld    R0,R1
  and   R0,#%F0
  ld    R1,R0                   ; load the high byte data

  call  receive_char            ; received char is stored at R0
  add   R0,#%BF
  jp    c,ascii_to_intelhex_L
  add   R0,#%07
ascii_to_intelhex_L:            ; converts ASCII char to Intel Hex Low Byte
  add   R0,#%0A
  and   R0,#%0F
  or    R0,R1                   ; R0 = converted Intel Hex 32

  add   R14,R0                  ; add byte to checksum accumulator
ret
```

| | |
|---|---|
| **Function** | **page_write** |
| **Description** | Write the data to the specified address. |
| **Included Functions** | Ascii_to_intelhex<br>Page_unlock |
| **Registers** | R2 = Starting write address of the page.<br>R3 = Ending write address of the page. |

**Code**

```
page_write:
  jp    write_compare_address
write_continue:
  ld    R0,#HIGH(BootCodeAddress_Start)
  cp    R2,R0                       ; if address >= (boot loader address)
  jp    nc,print_error_write_address; print error write

  cp    R2,#%00
  jp    nz,write_application_data
  cp    R3,#%02                     ; if 0001h < address < 0004h
  jp    z,write_application_entry_vector ; go to write application entry
                                    ; vector
  cp    R3,#%03
  jp    nz,write_application_data   ; else

write_application_entry_vector:
  add   R2,#HIGH(AppCodeAddress_Sub2)
  add   R3,#LOW(AppCodeAddress_Sub2)

  ld    R1,R2
  srl   R1                          ; address of page to be unlocked
  call  page_unlock                 ; unlock page specified in address R1

  call  ascii_to_intelhex           ; read hex byte and stored at R0
  ldc   @rr2,R0                     ; write the data byte (R0) to address
(rr2)

  ldc   R6,@rr2                     ; Verify the written data to Flash
  cp    R6,R0
  jp    nz,FLASH_verify_fail

  sub   R2,#HIGH(AppCodeAddress_Sub2)
  sub   R3,#LOW(AppCodeAddress_Sub2)
  jp    continue_write


write_application_data:
  ld    R1,R2
  srl   R1                          ; address of page to be unlocked
```

```
  call  page_unlock                ; unlock page specified in address R1

  call  ascii_to_intelhex          ; read hex byte and stored at R0
  ldc   @rr2,R0                    ; write the data byte (R0) to address
                                   ; (rr2)

  ldc   R6,@rr2                    ; Verify the written data to Flash
  cp    R6,R0
  jp    nz,FLASH_verify_fail

continue_write:
  ld    R6,#%01                    ; addend on the 16 bit adder
  ld    R12,R2                     ; MSB of 16 bit adder(R12) = R2
  ld    R13,R3                     ; LSB of 16 bit adder(R13) = R3
  call  addition_16bit             ; 16 bit adder (rr12 = rr12 + R6)
  ld    R2,R12                     ; MSB result of 16 bit adder
  ld    R3,R13                     ; LSB result of 16 bit adder

  ld    R0,#%2E
  call  putc                       ; print progress "...."

write_compare_address:             ; compare MSB_start_write_address vs.
                                   ; MSB_end_write_address
  cp    R2,R4                      ; if MSB_start_write_address >
                                   ; MSB_end_write_address
  jp    c,write_continue           ; { end writing }
check_LSB:                         ; else compare LSB
  cp    R3,R5                      ; compare LSB_start_write_address vs.
                                   ; LSB_end_write_address
  jp    c,write_continue           ; if LSB_start_write_address >
                                   ; LSB_end_write_address
                                   ; {end writing}
write_end:
ret
```

| | |
|---|---|
| **Function** | **Putc** |
| **Description** | Print the character. |
| **Included Functions** | • send |
| **Registers** | R0 = Holds the character to send. |

**Code**

```
putc:
    PUSH R1
$$:
    LDX R1, U0STAT0
    AND R1, #%04
    JR Z, $B
    LDX U0D, R0
    POP R1
    RET
```

| | |
|---|---|
| **Function** | **Puts** |
| **Description** | Print the string of characters. |
| **Included Functions** | • Putch |
| **Registers** | RR0 = Holds the address of the string. |

**Code**

```
puts:
    PUSH R2
    PUSH R3
    LD R2, R0
    LD R3, R1
$$:
    LDC R0, @RR2
    CP R0, #0
    JR Z, $F
    CALL putc
    INCW RR2
    JR $B
$$:
    POP R3
    POP R2
    RET

dummy_isr:
RET
```

| **Function** | **Delay** |
|---|---|
| Description | Used to delay the next instruction. |
| Included Functions | None. |
| Registers | R0, R1 and R2 are used as variable registers. |

**Code**

```
delay:
  ld    r2,#%FF
loop1:

  ld    r1,#%FF
  djnz  r1,$
  djnz  r2,loop1
  ret
```

| **Function** | **FLASH_verify_fail** |
|---|---|
| **Description** | Report error during Flash verify, restart boot loader. |
| **Included Functions** | • puts |
| **Registers** | R0 and R1 store the array of character addresses. |

**Code**

```
FLASH_verify_fail:
  ld    R0, #HIGH(print_error_FLASH_Write)
  ld    R1, #LOW(print_error_FLASH_Write)
  call  puts                        ; Print Flash error message
  jp    $                           ; Hang until reset
```

| | |
|---|---|
| **Function** | **print_error_write_address** |
| **Included Functions** | • puts |
| **Description** | Print "error address" in HyperTerminal. |
| **Registers** | R0 and R1 store the array of characters. |

**Code**

```
print_error_write_address:
  ld    R0, #HIGH(print_error)
  ld    R1, #LOW(print_error)
  call  puts
  ld    R0, #HIGH(print_error_address1)
  ld    R1, #LOW(print_error_address1)
  call  puts                        ; Print error Address
  ld    R0, #HIGH(print_error_address2)
  ld    R1, #LOW(print_error_address2)
  call  puts                        ; Print error Address
  ld    R0, #HIGH(print_error_address3)
  ld    R1, #LOW(print_error_address3)
  call  puts                        ; Print error Address
  jp
```

| | |
|---|---|
| **Function** | **print_Load_HEX_file** |
| **Included Functions** | None. |
| **Description** | Print "LOAD HEX FILE" in HyperTerminal. |
| **Registers** | R1 stores the array of characters. |

**Code**

```
print_Load_HEX_file:
  ASCIZ "\r\nLOAD HEX FILE"
```

| | |
|---|---|
| **Function** | **print_Bootloader** |
| **Included Functions** | None. |
| **Description** | Print "Zilog Encore! XP MCU Series" in HyperTerminal. |
| **Registers** | R1 stores the array of characters. |

**Code**

```
print_Bootloader:
  ASCIZ "\r\nZilog Encore XP! MCU Series"
```

| | |
|---|---|
| **Function** | **print_error** |
| **Included Functions** | None. |
| **Description** | Print "Error:" in HyperTerminal. |
| **Registers** | R1 stores the array of characters. |

**Code**

```
print_error:
  ASCIZ "\r\nError:"
```

| | |
|---|---|
| **Function** | **print_error_address** |
| **Included Functions** | None. |
| **Description** | Print "Address: Change Constant Data(ROM)=0000-(XXXX-1) and Program(EROM)=XXXX-1F7FB" in HyperTerminal. |
| **Registers** | R1 stores the array of characters. |

**Code**

```
print_error_address1:
  ASCIZ "\r\nAddress: Change ROM=0000-0BFD(Z8F042A Series)"
print_error_address2:
  ASCIZ "\r\nAddress: Change ROM=0000-1BFD(Z8F082A or Z8F0880 Series)"
print_error_address3:
  ASCIZ "\r\nAddress: Change ROM=0000-3BFD(Z8F1680 Series)"
```

| | |
|---|---|
| **Function** | **print_error_checksum** |
| **Included Functions** | None. |
| **Description** | Print "checksum" in HyperTerminal. |
| **Registers** | R1 stores the array of characters. |

**Code**

```
print_error_checksum:
  ASCIZ "\r\nchecksum"
```

| | |
|---|---|
| **Function** | **print_write_flash** |
| **Included Functions** | None. |
| **Description** | Print "Flash" in HyperTerminal. |
| **Registers** | R1 stores the array of characters. |

**Code**

```
print_error_FLASH_Write:
  ASCIZ "\r\nError verifying Flash Write"
```

| | |
|---|---|
| **Function** | **print_pleasewait** |
| **Included Functions** | None. |
| **Description** | Print "Please wait" in HyperTerminal. |
| **Registers** | R1 stores the array of characters. |

**Code**

```
print_pleasewait:
  ASCIZ "\r\nPlease wait"
```

| | |
|---|---|
| **Function** | **print_completed** |
| **Included Functions** | None. |
| **Description** | Print "COMPLETED!" in HyperTerminal. |
| **Registers** | R1 stores the array of characters. |

**Code**

```
print_completed:
  ASCIZ "\r\nCOMPLETED!"
```

# Appendix B. Intel Hex 32 Format

The boot loader application can program a standard file format into the Z8 Encore! XP MCU's Flash memory. The Intel Standard HEX 32 format file is one of the popular and commonly used file formats. An Intel Standard HEX 32-formatted file is an ASCII file with one record per line, as defined below.

| Record Mark | Data Size | Address MSB | Address LSB | Record Type | Data Byte | Checksum |
|---|---|---|---|---|---|---|
| 1 byte | 1 byte | 1 byte | 1 byte | 1 byte | *n* byte | 1 byte |

**Record Mark.** This field indicates the start of the hex line. It contains the char 3Ah or ":"

**Data Size.** This field indicates the size of the data of the hex line.

**Address.** This field indicates the address of the data to be stored in Flash memory which follows the big endian.

**Record Type.** This field indicates the type of the data. This types includes Data Record or normal addressing (00), End Of File Record (01) and Extended Linear Address Record.

**Data Byte.** This field contains the information which is written to Flash memory. The number of bytes depends on the data size

**Checksum.** This field is used to check if the received data is correct or not. The checksum must be equal to the two's complement of the sum of Data Size, MSB Address, LSB Address, Record Type and the number of Data Bytes, as exemplified in the equation below.

Checks sum = FFh and [ FFh-(1st byte + 2nd Byte + …. + (N-1) byte) + 01h ]

# Customer Support

To share comments, get your technical questions answered, or report issues you may be experiencing with our products, please visit Zilog's Technical Support page at http://support.zilog.com.

To learn more about this product, find additional documentation, or to discover other facets about Zilog product offerings, please visit the Zilog Knowledge Base at http://zilog.com/kb or consider participating in the Zilog Forum at http://zilog.com/forum.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, please visit the Zilog website at http://www.zilog.com.

**Warning:** DO NOT USE THIS PRODUCT IN LIFE SUPPORT SYSTEMS.

### LIFE SUPPORT POLICY

ZILOG'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS PRIOR WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF ZILOG CORPORATION.

### As used herein

Life support devices or systems are devices which (a) are intended for surgical implant into the body, or (b) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury to the user. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

### Document Disclaimer

 Information in this publication concerning the devices, applications, or technology described is intended to suggest possible uses and may be superseded. ZILOG, INC. DOES NOT ASSUME LIABILITY FOR OR PROVIDE A REPRESENTATION OF ACCURACY OF THE INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED IN THIS DOCUMENT. ZILOG ALSO DOES NOT ASSUME LIABILITY FOR INTELLECTUAL PROPERTY INFRINGEMENT RELATED IN ANY MANNER TO USE OF INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED HEREIN OR OTHERWISE. The information contained within this document has been verified according to the general principles of electrical and mechanical engineering.