# Application Note

# Bootloader for the Z8 Encore! XP MCUs

**AN032802-1020**

## Abstract

This application note describes a boot loader program for the on-chip memory functions of Zilog's Encore! XP MCU Family (6). 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 offsite method for downloading/updating of firmware to the MCU instead of using the Debug Interface and Zilog's Smart Cable.

The bootloader program associated with this application note supports the following Z8 Encore! XP MCUs:

- Z8F0822 Series MCU
- Z8F0823 Series MCU
- Z8F082A Series MCU
- Z8F1680 Series MCU
- Z8F64XX Series MCU
- Z8F6482 Series MCU

➢ **Note:** The source code files associated with this application note, AN0328-SC01.zip is available for download from the Zilog website. This source code has been tested with version 5.5.0 of ZDSII for the Z8 Encore! XP MCU. 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 Series of Microcontrollers feature 4KB to 64KB 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 128 sectors and is protected from programming and erase operations on a per sector basis.

Figures 1—3 illustrate the Flash memory arrangements of each of the Z8 Encore! XP MCUs listed in the Abstract above.

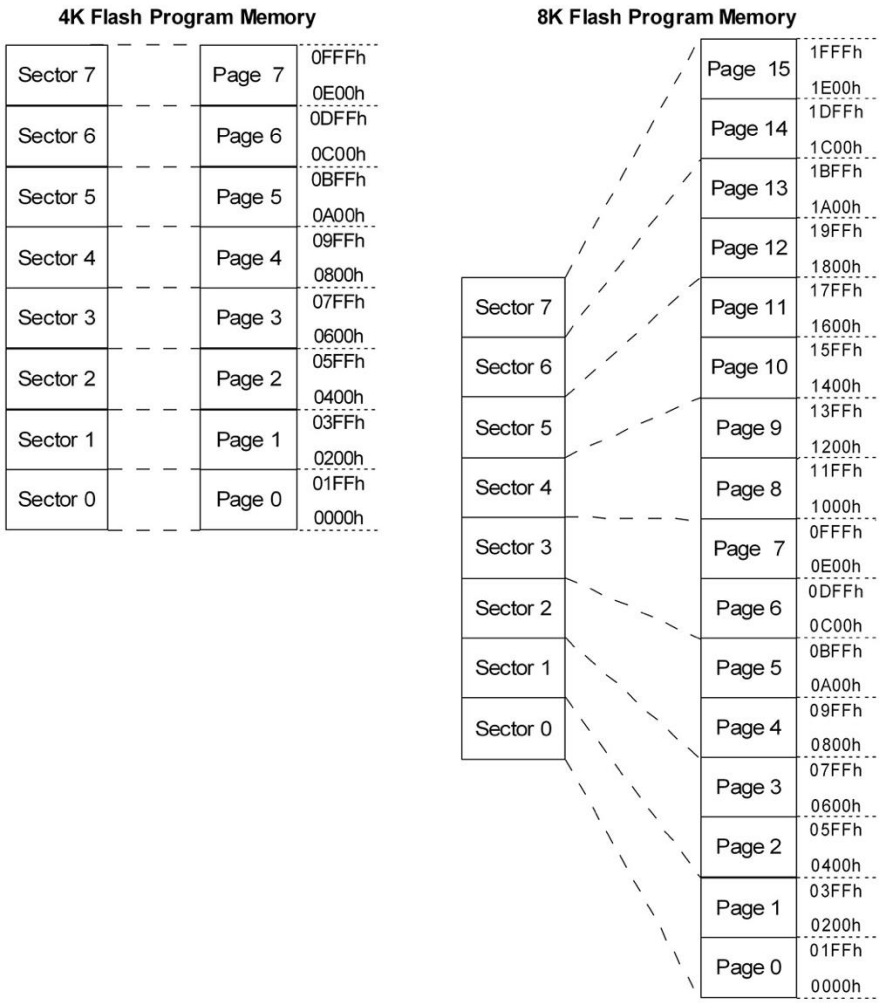


**Figure 1. Flash Memory Arrangement of 4K and 8K MCUs**

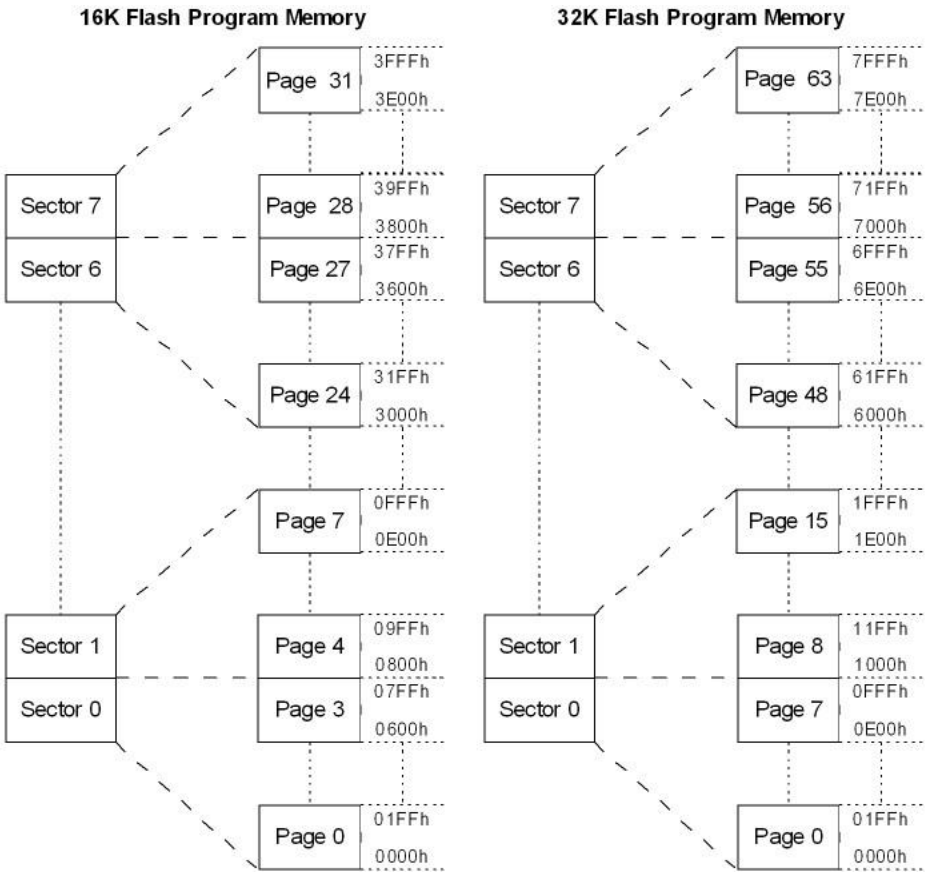


**Figure 2. Flash Memory Arrangement of 16K and 32K MCUs**

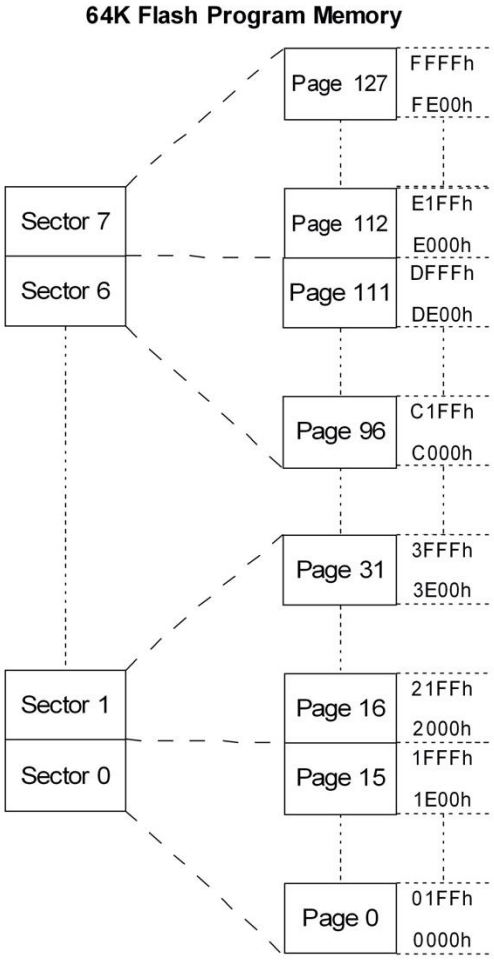


**Figure 3. Flash Memory Arrangement 64K MCUs**

For additional information regarding the Flash memory functions of each MCU, please see the MCU's respective product specifications.

# 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 Boot Loader Application

The bootloader is firmware that enables the MCUs to write 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 programing and erasure of Flash with system clock frequencies that can range from 32.8kHz to 20 MHz.

**UART.** The UART0 is used to communicate with the RealTerm 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).

**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 4 shows a block diagram of the boot loader.

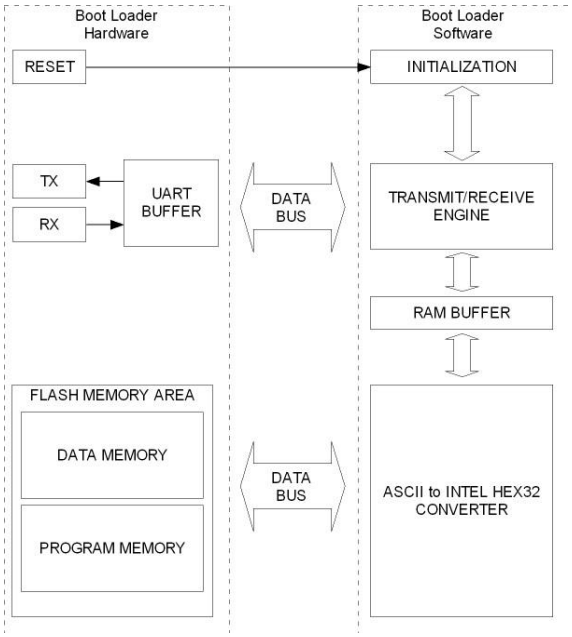


**Figure 4. Block Diagram of the Z8 Encore! XP Boot Loader**

For complete details about the on-chip Flash memory and Flash controller functions of each MCU, refer to the device's product specifications.

# Boot Loader Features and Applications

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; it calculates and verifies a checksum for error detection. If the loaded hex file contains checksum errors, it displays:

   ```
   Error: checksum
   ```

   in RealTerm 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 32 File Format in Appendix A.

5. The boot loader program displays a progress indicator in RealTerm to indicate the status of the data being loaded into Flash; it displays `COMPLETED!` in RealTerm 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, 16K and 64K products discussed herein, see Figure 5.

**Note:** In Figure 5, the color green represents rewritable addresses and blue represents non-rewritable addresses.

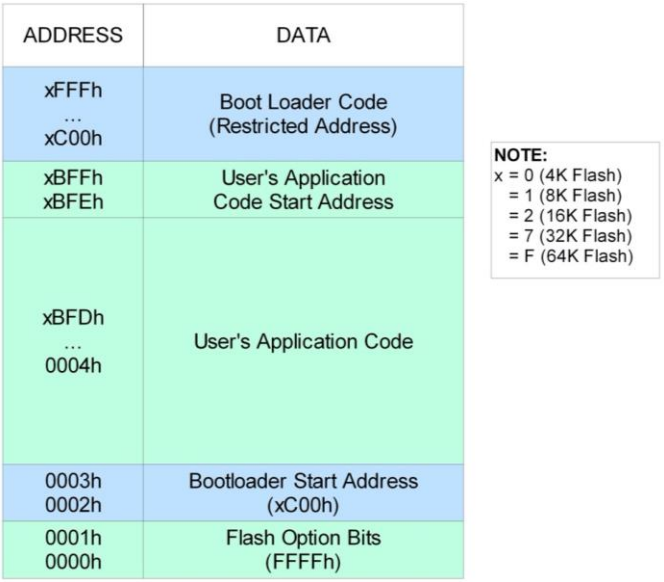


**Figure 5. Flash Memory Addresses: Application Code and Boot Loader Code**

# 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
   - 1.2. Erase Flash Memory
     - 1.2.1. Page Unlock
     - 1.2.2. Page Erase
   - 1.3. Write Boot Loader Start Address
     - 1.3.1. Page Unlock
     - 1.3.2. Write Boot Loader Start Address
     - 1.3.3. Lock Flash
   - 1.4. Get Hex File
     - 1.4.1. Receive Character
     - 1.4.2. ASCII to INTEL HEX 32
       - 1.4.2.1. Receive Character
     - 1.4.3. Page Write
       - 1.4.3.1. ASCII to INTEL HEX 32
       - 1.4.3.2. Page Unlock
       - 1.4.3.3. Write Boot Loader Start Address
   - 1.5. Lock Flash
2. User Application Code

### Setting Communication Parameters

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

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

### Boot Loader Flow

Figure 6 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). The program then jumps to the starting address of the user's downloaded application code, which executes in Flash memory.

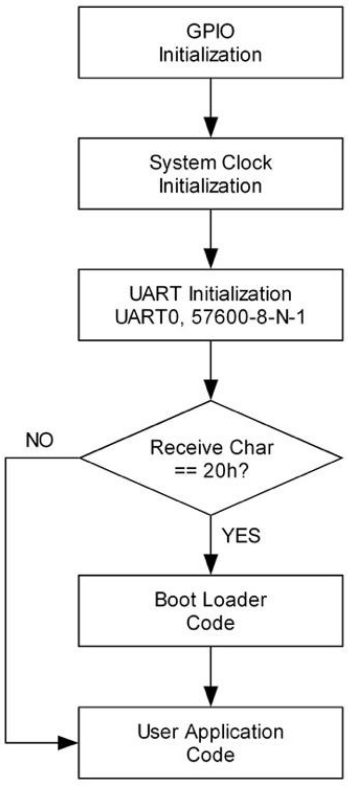


**Figure 6. Main Function Flow Diagram of the Bootloader Application**

## 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 7.

1. The boot loader code starts; it displays Zilog Z8 Encore! XP MCU Series in the RealTerm 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 is written to addresses in the range 0002h – 0003h.
5. LOAD HEX FILE is displayed in the RealTerm 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. RealTerm 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.

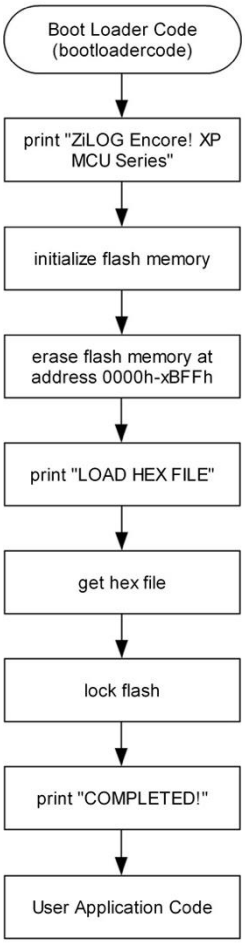


**Figure 7. Flow Diagram of Boot Loader Code**

## Initialize Flash Memory

The Initialize Flash Memory function is used to configure the Flash frequency register to allow programming and erasure of the Flash memory. The Flash frequency register is used to control the timing for Flash program and erase operations. This value is calculated using the following equation:

$$\text{FFREQ}[15:0] = \frac{\text{system clock frequency, Hz}}{1000}$$

This application note uses 5.529 MHz as system clock.

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

## Erase Flash Memory

The Erase Flash Memory function shown in Figure 8 illustrates the high level steps in 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.

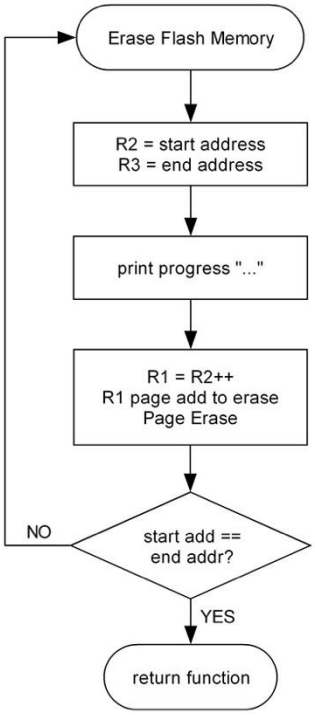


**Figure 8. Flow Diagram of Erase Flash Memory**

## Page Erase

The Page Erase memory shown in Figure 9 illustrates the low level steps used to erase a page of Flash memory where the given address resides. The Z8 Encore! XP Flash memory has 8 to 128 pages (depending on MCU Flash memory) where each page 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.

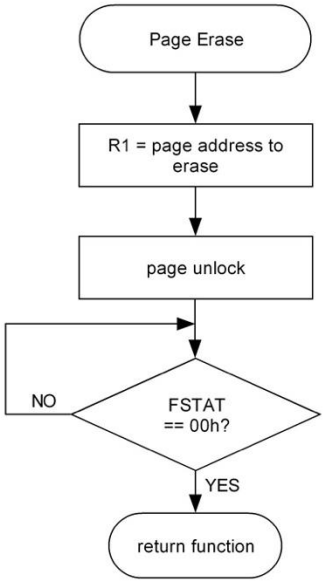


**Figure 9. Flow Diagram of Page Erase**

## 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:
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
```

## 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 FFFFh; however, the value of XC00h must be rewritten.

```
Assembly Code
write_bootloader_start_address:
ld R1,#%00                          ; R1 = unlock page address 0000h-01FFh
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 0002h
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 0003h
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 10 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 ASCII characters 0–9 are converted to the numbers 0 to 9 (0h–9h).
3. The first byte indicates the amount of data in a line; this amount is stored in register R7.

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 RealTerm:

   ```
   Error: checksum
   ```

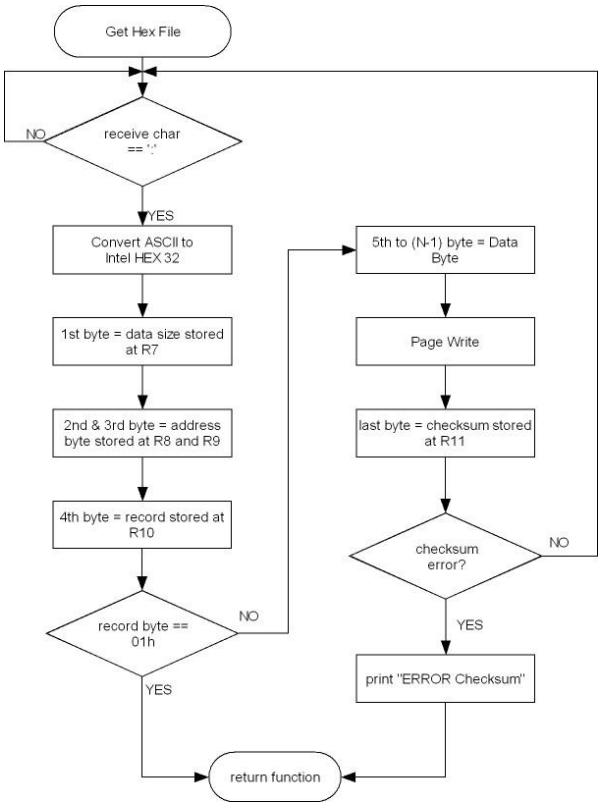


**Figure 10. Flow Diagram of Get Hex File**

### Receive Character

The Receive Character function is used to get the character from the U0RXD registers buffer to the R0 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 HEX 32 data byte. The Flow Diagram is shown in Figure 11 below.

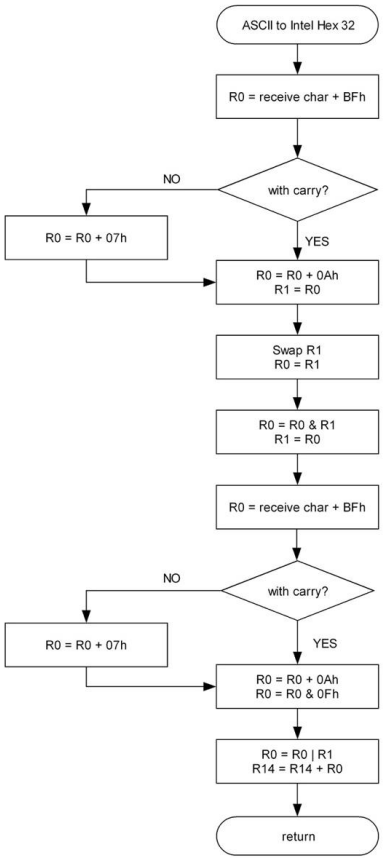


**Figure 11. Flow Diagram of ASCII to Intel HEX 32**

### 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 – 1FFFFh). Failure to satisfy this condition will result in program termination and will display the following error message in RealTerm:

   ```
   ERROR: Write to invalid address
   ```
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 RealTerm:

   ```
   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 RealTerm will display:

   ```
   COMPLETED!
   ```

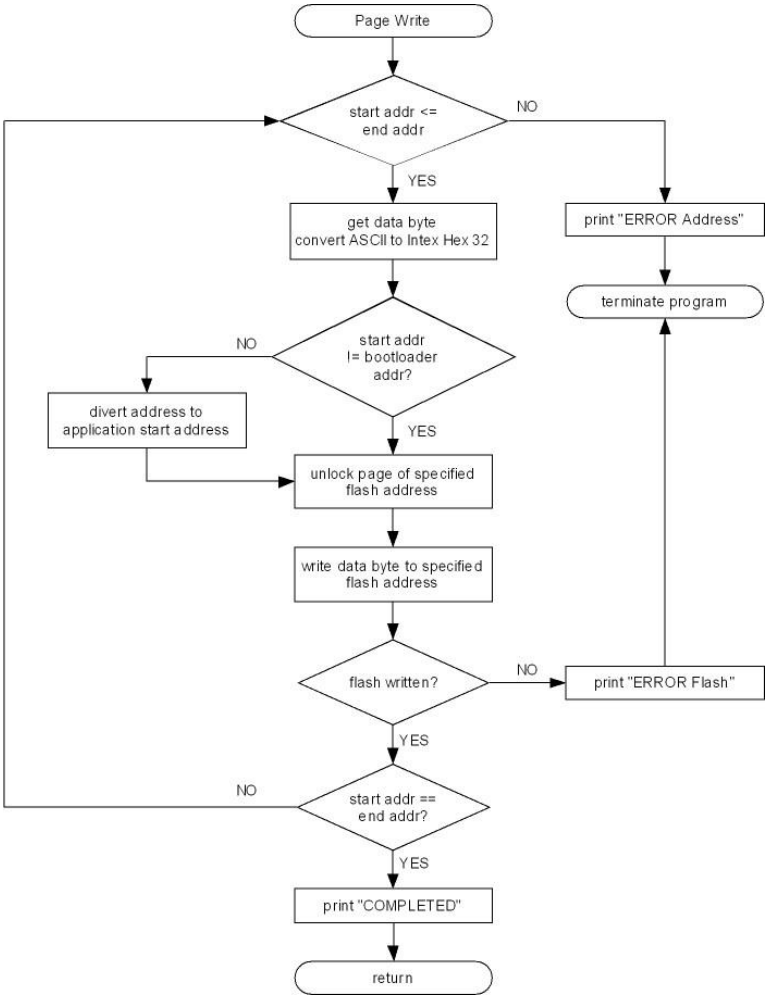


**Figure 12. Flow Diagram of Page Write Function**

### 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 XBFEh.

```
Assembly Code
application_code:
ld R2,#HIGH(AppCodeAddress_Start)   ; application code starts at the value
ld R3,#LOW(AppCodeAddress_Start)    ; stored in XBFEh-XBFFh
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:
 push r2
 push r3
 ld r2, r0
 ld r3, r1
$$:
 ldc r0, @rr2
 cp r0, #0
 jr z, $F
 call puts
 jr $B
$$:
 pop r3
 pop r4
 ret
```

### putc Function

The putc function is used to print the character stored in register R1.

```
Assembly Code
putc:
 push r1
$$:
 ld r1, U0STAT0
 and r1, #%04
 jr z, $B
 ldx U0D, r0
 pop r1
 ret
```

### 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 procedure.

## Equipment Used

- Z8 Encore! XP F64xx Series Development Kit
- Z8 Encore! XP F6482 Series Development Kit
- Z8 Encore! XP F1680 (28-pin) Series Development Kit
- Z8 Encore! XP F0822 Series Development Kit
- Z8 Encore! XP F042A Series Development Kit
- Z8 Encore! XP F0423 Series Development Kit
- Zilog USB Smart Cable
- power supply
- RS-232 cable or USB to serial cable

To download the boot loader to the Z8 Encore! XP MCU, observe the following instructions:

1. Connect the USB Smart Cable to the DBG terminal of the development board and connect the USB side to the development PC's USB port.
2. Apply power to the development board.
3. Download AN0328-SC01.zip from zilog.com and extract the source code files.
4. Launch **ZDSII for Z8 Encore! Version 5.5.0**.
5. From ZDS **File** menu, select **Open Project…** to show the **Open File** dialog box.
6. Navigate to the extracted folder, select AN0328.zdsproj and click **Open** to open the project.
7. From the **Project** menu, select **Settings** to open the **Project Settings** dialog box.

8. In the **Project Settings** dialog box, do the following:
   8.1. Select **General**, then select the desired **CPU Family** and **CPU**.
   8.2. Select **Address Spaces**, the **ROM Address Space** must be set to the default value.
   - 0000-0FFF for 4K MCU
   - 0000-1FFF for 8K MCU
   - 0000-3FFF for 16K MCU
   - 0000-7FFF for 32K MCU
   - 0000-FFFF for 64K MCU

   8.3. Select **Debugger**, then click the **Target Setup** button to open the **Configure Target** dialog box. Set the clock frequency to match the code settings:
   - 5.5296 MHz for Z8F082A, Z8F1680, and Z8F0823 Series MCU
   - 5.5338 MHz for Z8F6482 Series MCU
   - 18.432 MHz for Z8F64XX and Z8F0822 Series MCU
9. Click **OK** to close the **Configure Target** dialog box, then click **OK** again to close the **Project Settings** dialog.
10. From the **Build** menu, select **Rebuild All** to rebuild the project.
11. From **Debug** menu, select **Download Code** to load the program to the MCU.
12. After programming is complete, from the **Debug** menu, select **Stop Debugging**.
13. Remove power from the board and disconnect the USB smart cable from the DBG terminals of the development board.

## Configure Address Space for the Application Code

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:

- For 4K MCUs, ROM = 0000h – 0BFDh
- For 8K MCUs, ROM = 0000h – 1BFDh
- For 16K MCUs, ROM = 0000h – 3BFDh
- For 32K MCUs, ROM = 0000h – 7BFDh
- For 64K MCUs ROM = 0000h - FBFDh

2. Build and compile your application code.
3. Follow the Load Hex Code via Bootloader procedure to load your hex file to the Z8 Encore! XP development board.

## Load Hex Code via Bootloader

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 ZDSII IDE from the MCU and connect the RS-232 (or USB-to-serial) cable to your PC.

2. Launch RealTerm 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 reset button for the MCU, then release the space bar on the keyboard. RealTerm should display a screen similar to the example shown in Figure 13.

**Note:** For Z8F6482 Series, wait for 10 seconds, before releasing the space bar on the keyboard.

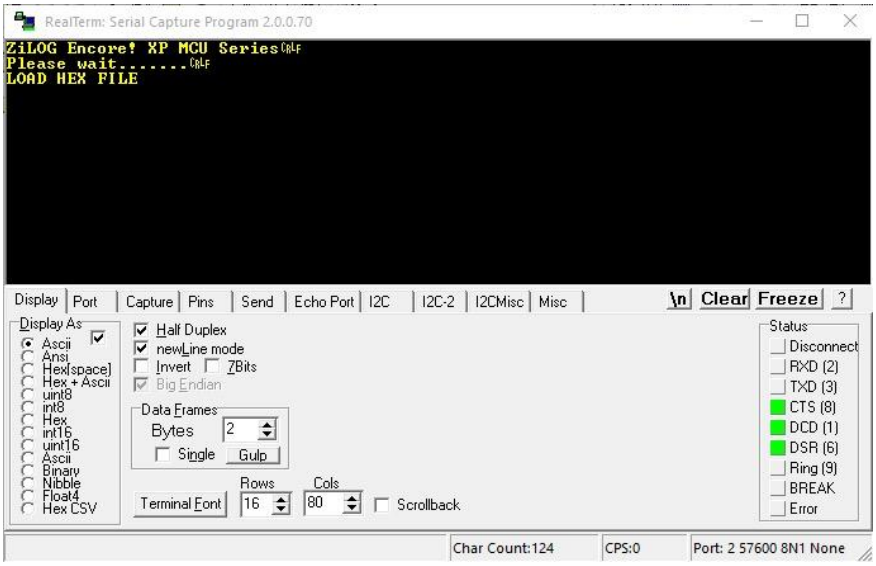


**Figure 13. Boot Loader Initialization in RealTerm**

5. In RealTerm, click the **Send** tab, click the … button to browse and open your application's hex file.
6. The hex file you selected will load into memory. After the hex file is loaded, RealTerm 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.

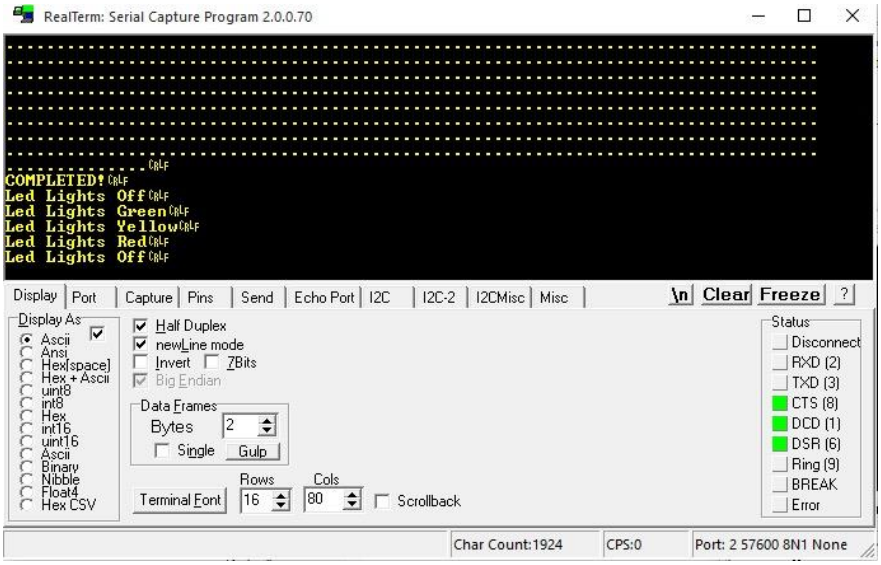


**Figure 14. Boot Loader Completion in RealTerm**

# Summary

The 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 1 KB due to the boot loader code residing at the highest address space in the MCU.

# References

Documents associated with this application note are listed below. Each of these documents can be obtained from the Zilog website by clicking the link associated with the document number where indicated.

**Reference Documents**

| Document Number | Document |
|---|---|
| PS0199 | Z8 Encore! XP F64XX Series Product Specification |
| PS0228 | Z8 Encore! XP F0822 Series Product Specification |
| PS0225 | Z8 Encore! XP F082A Series Product Specification |
| PS0243 | Z8 Encore! XP F0823 Series Product Specification |
| PS0250 | Z8 Encore! XP F1680 Series Product Specification |
| PS0294 | Z8 Encore! XP F6482 Series Product Specification |
| UM0128 | eZ8 CPU User Manual |

# Appendix A. 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 bytes | 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.

Checksum = 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.

To learn more about this product, find additional documentation, or to discover other facets about Zilog product offerings, please visit the Zilog Knowledge Base page.

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