Application Note

# Boot Loader for ZNEO-Based MCUs

AN032501-0411

## Abstract

This Application Note describes a boot loader program for the on-chip memory functions of Zilog's Z16F Series of Microcontrollers based on the ZNEO CPU architecture. The boot loader is loaded using Zilog's ZDS II IDE and provides the functionality to program an Intel HEX 32-format file to ZNEO-based MCU Flash memory using the RS-232 port. It is designed to provide an alternative to using USB communication via Zilog's XTools firmware.

**Note:** The source code associated with this application note, AN0325-SC01, has been tested with ZDS II version 4.11.1.

## ZNEO-Based MCUs: A Flash Memory Overview

The products in Zilog's Z16F Series of Microcontrollers feature up to 128KB of nonvolatile Flash memory with read/write/erase capability. The Flash memory array is arranged in 2KB pages, the minimum Flash block size that is erased. Flash memory is also divided into eight sectors and is protected from programming and erase operations on a per-sector basis.

Figure 1 illustrates the Flash memory arrangement of the Z16F2811 MCU.

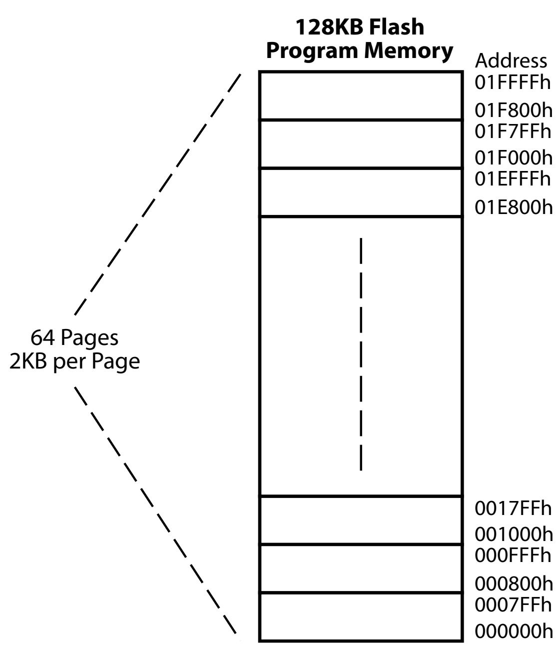



**Figure 1. Flash Memory Arrangement of Z16F2811 MCU**

For additional information regarding the Flash memory functions of the of Z16F2811 MCU, please see the ZNEO Z16F Series Product Specification (PS0220).

# 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 start 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 performs an error check on the received data using the checksum method.
- 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 code to prevent any inadvertent programming of the user code into its own memory space.

## Developing the ZNEO Boot Loader Application

A ZNEO CPU-based MCU can write to its own program memory space. It features an on-chip Flash controller that erases and programs on-chip Flash memory. Figure 2 shows a block diagram of the boot loader.

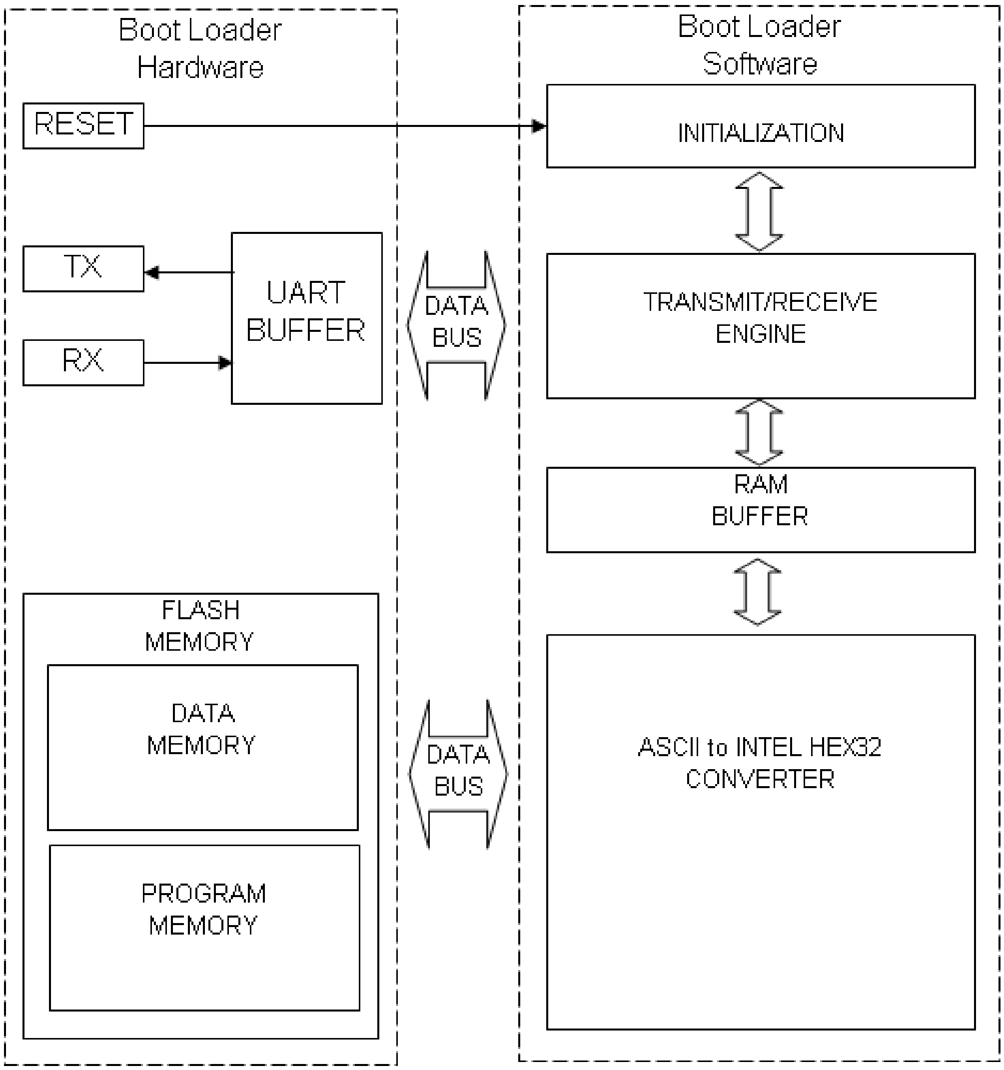



**Figure 2. Block Diagram of the ZNEO Boot Loader**

The boot loader program uses the Reset pin, the Flash controller and the on-chip UART to function; each is described below.

**Reset Pin.** The Reset pin is used to restart the Boot Loader firmware. If character `0x20` is received, the program counter redirects the program to the Flash Loader function; otherwise it routes directly to the application code.

**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 5).

**Flash Controller.** The Flash Controller provides the appropriate Flash controls and timing for the byte programming, Page Erase, and Mass Erase of 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 from32 KHz to 20 MHz.

For complete details about the on-chip Flash memory and Flash controller functions of the Z16F2811 MCU, refer to the ZNEO Z16F Series Product Specification (PS0220).

## ZNEO Boot Loader Features and Application

The boot loader program operates in the following sequence; refer to Figure 3 for corresponding address ranges in the Flash memory space.

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 `0x00008h–0x1F7FFh`.
2. The boot loader program selectively erases Flash memory before programming 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:** A brief description of the Intel Hex File Format is provided in the Appendix B. Intel Hex 32 Format section on page 43.

5. The boot loader program displays a progress indicator in HyperTerminal to indicate the status of 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: Change Constant Data(ROM)=0000-7FFF and
Program(EROM)=08000-1F7FF.
```

   If this error is received, *Data(ROM)* addressing must be changed to `0x0000–0x7FFF` and *Program(EROM)* addressing must be changed to `0x08000–0x1F7FF` because the boot loader code already occupies addresses in the range `0x1F800–0x1FFFF`.

| Address | Data |
|---|---|
| 1FFFFh ... 1F800h | Boot Loader Code (Restricted Address) |
| 1F7FFh ... 1F7F8h | Application Code Start Address |
| 1F7F7h ... 00008h | User Application Code |
| 00007h ... 00004h | Boot Loader Start Address (0001–F800h) |
| 00003h ... 00000h | Flash Option Bit (FFFFh) |

**Figure 3. Flash Memory Address of Application Code and Boot Loader Code**
**Legend: Green represents user-rewritable addresses; blue represents reserved addresses**

# Theory of Operation

Generally, a boot loader's sole function is to download a hex file created in ZDS II to MCU Flash memory. This application is designed to provide this hex file via the 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 hierarchy of the Main Function is diagrammed below; each line of this code is described in this section.

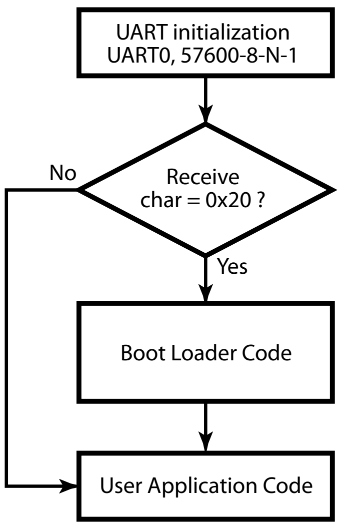
```
MAIN FUNCTION
Main Function Hierarchy
1.  Boot Loader Code
```

```
    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 Application Address
         1.3.1. Page Unlock
         1.3.2. Write 0001 F800h to Address 0004h-0007h
         1.3.3. Lock Flash
    1.4. Get Hex File
         1.4.1. Receive Character
         1.4.2. ASCII to INTEL HEX 32 Converter
                1.4.2.1 Receive Character
         1.4.3. Page Write
                1.4.3.1 ASCII to INTEL HEX 32 Converter
                1.4.3.2 Page Unlock
                1.4.3.3 Write Data Byte to Address Byte
    1.5. Lock Flash
2.  User Application Code
```

Figure 4 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. 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

The Main Function program enters the boot loader code when the space bar (ASCII character code `0x20`) and the MCU's reset button are simultaneously pressed. The boot loader code then downloads the hex file to the MCU's Flash memory (for details about this function, see the next section). The program then jumps to the start address of the user's downloaded application code, which executes in Flash memory.




```
Assembly Code
ld      sp,#%FFC000         ; stack pointer
                            ; initialized UART
ld      R1,#0               ; R1 = UART0
ld      R2,#_SYS_CLK_FREQ   ; R2=Clk Freq.
ld      R3,#57600           ; Baud Rate = 57600

ld      R0,R3               ; UART0 initialize
sll     R0,#3
add     R0,R2
sll     R3,#4
udiv    R0,R3
ld.w    U0BR,R0
ld      R0,#%30             ; UART GPIO init.
or.b    PAAFL,R0            ; PAAFL = 0x30
clr.b   U0CTL1              ; U0CTL1= 0x00
ld      R0,#%C0
ld.b    U0CTL0,R0           ; U0CTL0= 0xC0

ld.b    R0,U0RXD            ; Read U0RXD buff reg
cp      R0,#%20             ; if (U0RXD=0x20)
jp      z,bootloadercode    ; go boot loader; else
; APPLICATION CODE
ld      R4,%1F7FC
call    delay
call    delay
jp      (R4)                ; go to
application_code();
```

**Figure 4. ZNEO Boot Loader Main Function: Flow Diagram and Assembly Code**

## 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 user application code memory address range `00008h` to `1F7FFh`. The remaining portion of the memory, in the `1F800h` to `1FFFFh` address range, is boot loader code memory in which the boot loader program resides.

When the boot loader code starts, it displays `ZNEO Boot Loader` in the HyperTerminal window, followed by a sequence of tasks, as noted below and illustrated in Figure 5.

1. Flash memory initialization, during which the clock frequency is set for correct operation of MCU Flash memory.
2. Flash memory erasure, in which Flash memory is reset within the address range `00008h` to `1F7FFFh`. This address range contains the user's application code and the start address of the boot loader (`00000h–00007h`). Flash memory is erased so that new data can be written to Flash memory.

3. Boot loader address rewrite, in which the start of the boot loader program is restored to the start address of Flash memory. The data string `FFFF 0001 F800` is written to addresses in the range `00000h` to `00007h`.
4. `LOAD HEX FILE` is displayed in the HyperTerminal window to indicate that the MCU is ready to load the application code.
5. When the hex file is sent, the `get hex file` function writes the data to Flash memory.
6. After the data is completely written into Flash memory, Flash memory is locked to prevent the MCU from overwriting existing application code.
7. HyperTerminal displays `COMPLETED!`, as shown in Figure 5, to indicate that the application code hex file has successfully downloaded to the MCU.

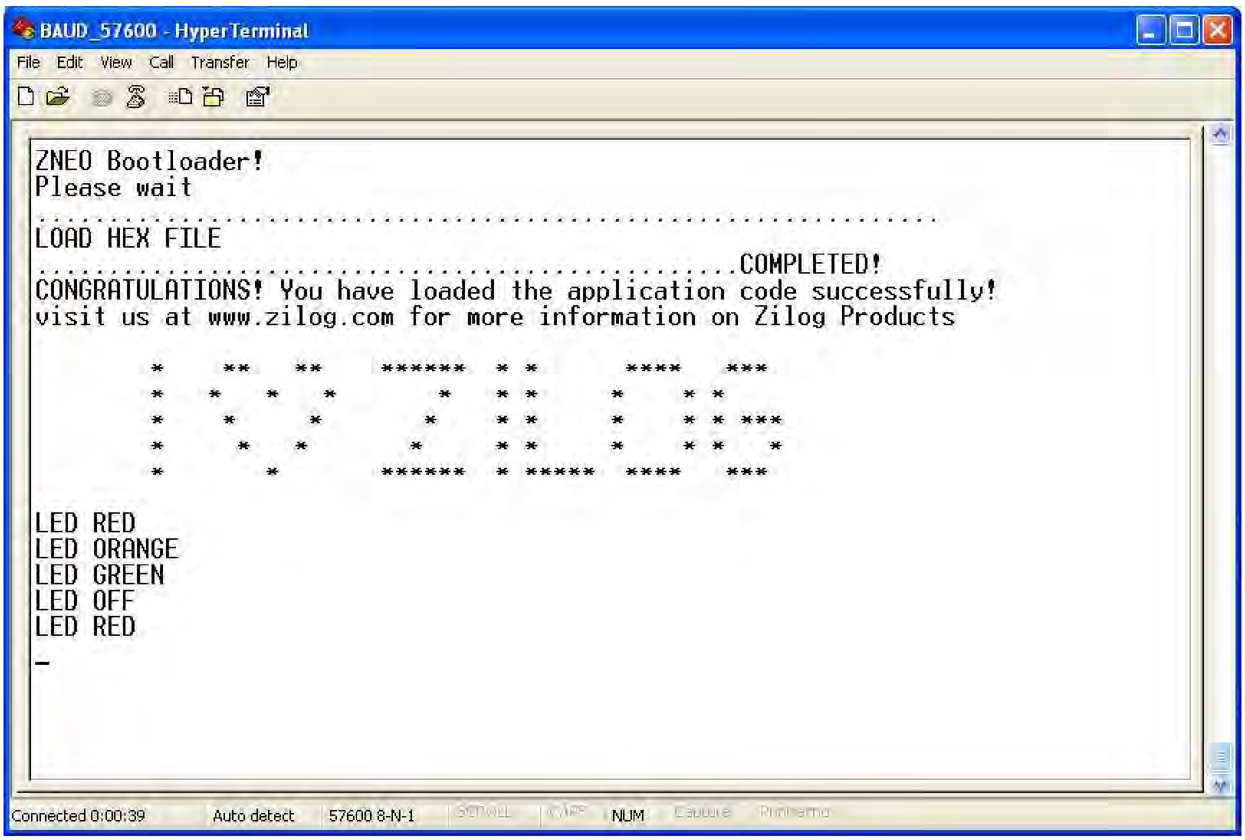
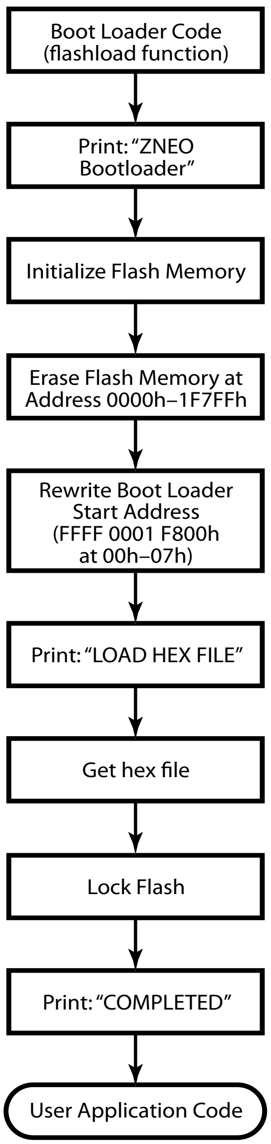



**Figure 5. HyperTerminal Displays the Loaded Application Code**

8. Finally, the program counter shifts to the start address of the user application code to execute the downloaded application code see Figure 6.




```
Assembly Code
bootloadercode:
ld      R1,#print_Bootloader
call    puts
call    init_flash
ld      R1,#print_pleasewait
call    puts

clr     R2
ld      R3,#%1F800
call    erase_flash_address
call    write_bootloader_start_address

ld      R1,#print_Load_HEX_file
call    puts

clr     R12; init. extended address

call    get_hex_line
call    lock_flash

call    delay
ld      R1,#print_completed
call    puts

ld      R4,%1F7FC
call    delay
call    delay
jp      (R4)                 ; Go to Application
code
ret
```

**Figure 6. ZNEO Boot Loader Code: Flow Diagram and Assembly Code**

## Initialize Flash Memory

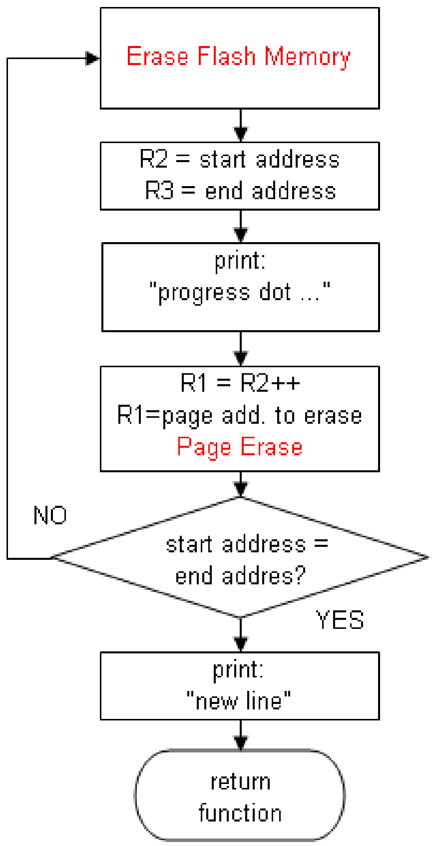
The Initialize Flash Memory function, exemplified in the following assembly code, is used to initialize the settings for Flash memory.

```
init_flash:
  ld      R1,#_SYS_CLK_FREQ     ; initialized clock frequency
  ld      R0,#%3E8
  udiv    R1,R0
  ld.w    FFREQ,R1
```

```
ret
```

## Erase Flash Memory

The Erase Flash Memory function, shown in Figure 7, is responsible for erasing the user application code in the address range `0000h–1F7FFh`, excluding the boot loader code in the address range (`1F800h–1FFFFh`). Register R2 is the start address, while R3 is the end address of the Flash memory to be erased.

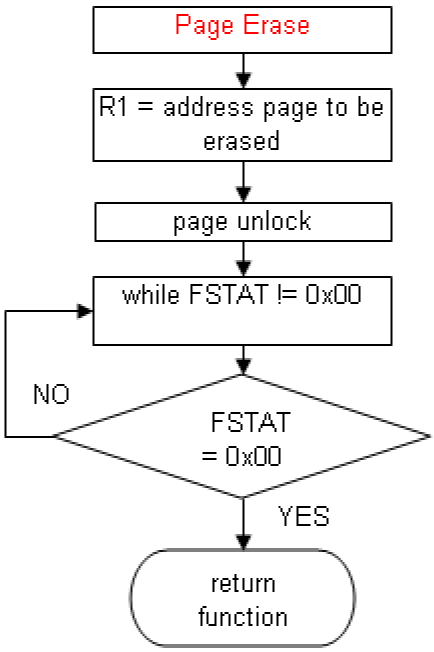



```
Assembly Code
erase_flash_address:

ld   R1,#%2E      ; print progress
call putch

ld   R1,R2        ; start address to
                  ; be erase to R1
callpage_erase    ; erase content
                  ; of page address
add  R2,#%800     ; inc. page erase
                  ; 1page = 0x800 bytes
cp   R2,R3        ; if current erase
                  ; page = end address page
jp   c, erase_flash_address   ; repeat
                  ;if current erase < end address

ld   R1,#%0A      ; print prog.new line
call putch

ret
```

**Figure 7. Erase Flash Memory: Flow Diagram and Assembly Code**

## Page Erase

The Page Erase function, shown in Figure 8, is used to erase a page of Flash memory at a given address. ZNEO Flash memory contains 64 pages, each of which contains 2KB (`800h`). The boot loader can only erase the lower 63 pages (in the address range `00000h`–`1F7FFh`) because the final page is allotted to boot loader code (in the address range `1F800h`–`1FFFFh`). Register R1 is used as the page address of the portion of Flash memory to be erased.




```
Assembly Code
page_erase:

call  page_unlock
ld    R0,#%95
ld.b  FCMD,R0
flash_stat:; while FSTAT is busy
ld    R0,#%10

tm.b  FSTAT,R0
jp    nz,flash_stat

ret
```

**Figure 8. Page Erase: Flow Diagram and Assembly Code**

## Page Unlock

The Page Unlock function, exemplified in the following assembly code, is used to unlock Flash memory for a specified address page. This function is necessary for writing and erasing Flash memory to and from this specified address page. Register R1 is used as the Flash memory page address to be unlocked.

```
Assembly Code
page_unlock:
  srl     R1,#%B
  sll     R1,#%3

  ld.w    FPAGE,R1
  ld      R0,#%73
  ld.b    FCMD,R0
  ld      R0,#%8C
  ld.b    FCMD,R0
  ret
```

## Lock Flash

The Lock Flash function, exemplified in the following assembly code, is used to protect Flash memory from its contents being overwritten or erased.

```
Assembly Code
```

```
lock_flash:
  clr.b   FCMD
  ret
```

## Write Boot Loader Start Address

The Write Boot Loader Start Address function, exemplified in the following assembly code, is used to rewrite the reset address of the boot loader code. This rewrite occurs because after the Erase Flash Memory function is implemented, the value of the address range `00000h`–`1F7FFh` is reset to `FFh`. As a result, the reset vector, in the address range `0004h`–`0007h`, is reset to the values (`FF FF FF FFh`).

```
Assembly Code
write_bootloader_start_address:
  ld      R1,#%00               ; R1 = unlock page address (0x00)
  call    page_unlock           ; unlock Flash memory
  ld      R1,#%0004
  ld.w    (R1++),#%0001         ; R1 = load data (00 01) to
address (0x0004-0x0005)
  ld.w    (R1),#%FC00           ; R1 = load data (F8 00) to
address (0x0006-0x0007)
  call    lock_flash            ; lock Flash memory
  ret
```

## Get Hex File

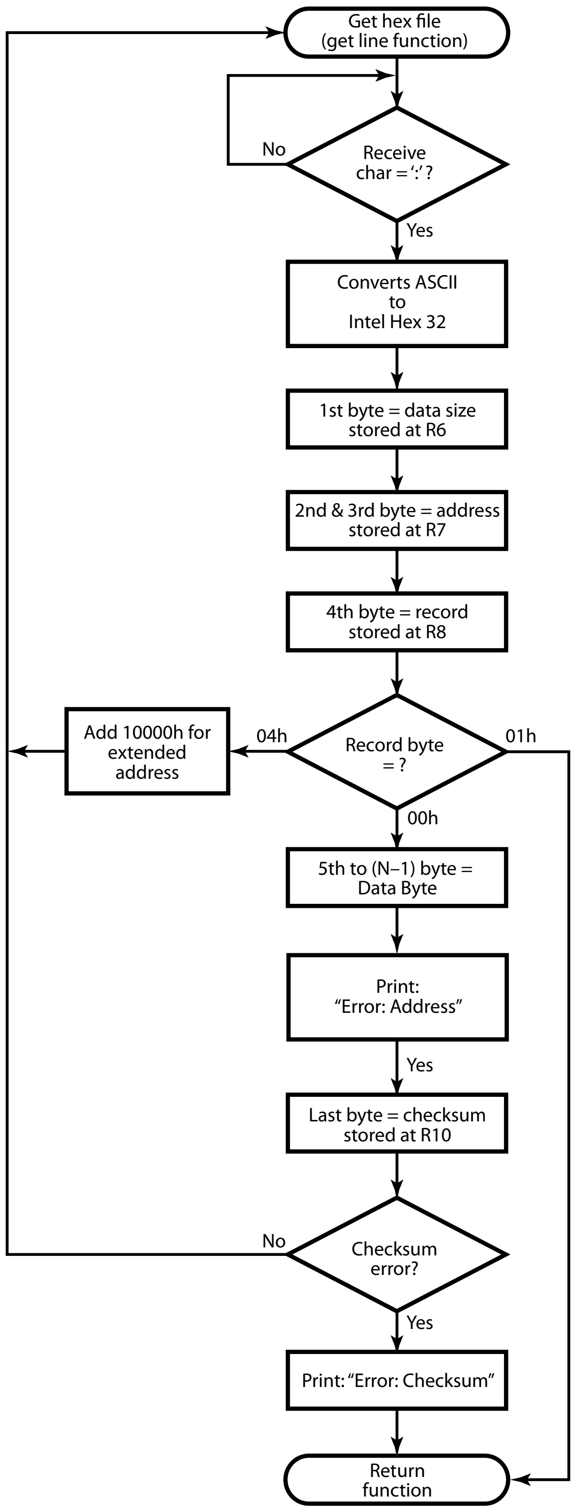
The Get Hex File function, shown in Figure 9, is responsible for reading the hex file and storing it in Flash memory pertinent to the following sequence.

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 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 (`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 6.
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 7. 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 a normal address, at an extended address, or at an end-of-file address.
   – Normal addressing is represented by the value `00h`, while extended addressing is indicated by the value `04h`. If extended addressing is detected, `1000h` is the next address to be read.

– 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 R6 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 1st byte to the (N–1) byte (see the equation below). Failure to satisfy this condition will result in program termination and will print `error: checksum` in HyperTerminal.

```
Checksum = FFh and [(FFh – (1st byte + 2nd Byte + … + (N–
1) byte)) + 1] or [00h – (1st byte + 2nd byte + … + (N-1)
byte)]
```




```
Assembly Code
get_hex_line:
call receive_char ; start line
ld   R0,R5         ; R5 = received data
cp   R0,#%3A    ; check for ':' or 0x34 ASCII
jp   nz,get_hex_line

clr  R11       ; initialized checksum = 0x00
;--------------DATA SIZE-------------------
call ascii_to_intelhex; data size
ld   R6,R0         ; R6 = data size

;--------------ADDRESS---------------------
call ascii_to_intelhex
sll  R0,#8         ; high byte address
ld   R7,R0         ; R7 = high byte address

call ascii_to_intelhex;low byte address = R0
add  R7,R0    ;R7=add high and low add. byte
;--------------RECORD BYTE-----------------
call ascii_to_intelhex; record byte
ld   R8,R0         ; R8 = record byte
cp   R8,#%01       ; check rec. type if EOF
jp   z,get_line_end

cp   R8,#%04   ; check rec. type if ext. add.
jp   nz,write_data_byte
ld   R12,#%10000
jp   get_hex_line
;--------------DATA BYTE-------------------
write_data_byte:
ld   R2,R7         ; R2 = start add.(R7)
add  R2,R12        ; R2 = start add.(R7) +
ext.
                   ; add.(R12)
ld   R3,R6         ; R3 = data size(R6)
add  R3,R2         ; R3= data size(R6) + start
           ; address(R2) = terminal address
call page_write    ; write the data to memo.
;--------------CHECKSUM------------------
ld   R2,R11        ; (R2) sum of all = data
size
                   ; + byte_address
                   ; + record_byte + data byte
call ascii_to_intelhex; checksum
and  R0,#%000000FF
ld   R10,R0

and  R2,#%000000FF ; checksum = 0xFFFFFFFF
                   ; - sum of all + 0x01
ld   R0,#%FFFFFFFF
sub  R0,R2
add  R0,#%01
and  R0,#%000000FF

cp   R0,R10
jp   nz,print_error_check_sum
ld   R1,#%2E       ; print progress
call putch

jp   get_hex_line
get_line_end:
ret
```

**Figure 9. Get Hex File: Flow Diagram and Assembly Code**

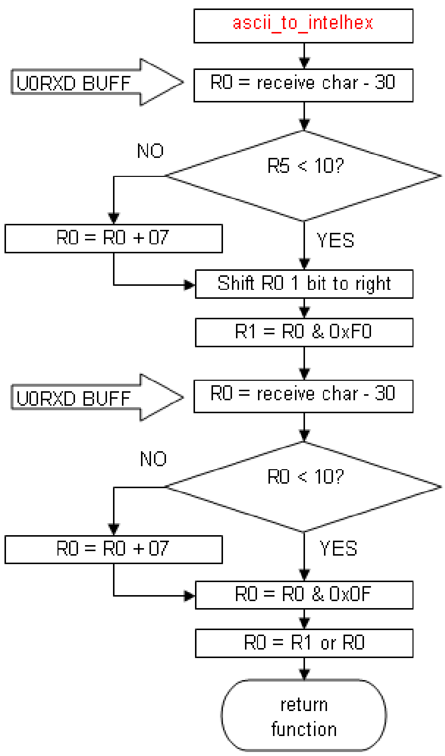
## Receive Character

The Receive Character function, exemplified in the following assembly code, is used to *get* a character from the U0RXD registers buffer to the R5 register. The U0STAT Register is used to indicate if the character is received from the U0RXD register buffer.

```
Assembly Code
receive_char:
  ld      R0,#%80
  tm.b    U0STAT0,R0            ; Read the UART0 status register
  jp      z,receive_char        ; Check if any character is
received
  ld.b    R5,U0RXD              ; R5 = Store the data from receive
register (U0RXD)
  ret
```

## ASCII to INTEL HEX 32

The ASCII to INTEL HEX 32 function, shown in Figure 10, is used to convert ASCII characters to INTEL 32 data byte format.




```
Assembly Code
ascii_to_intelhex:
call   receive_char
ld     R0,R5               ; R5 = received data
add    R0,#%FFFFFFBF
       ; check if received char > 0x40
jp     c,ascii_to_intelhex_H
add    R0,#%07

ascii_to_intelhex_H:
       ; converts ASCII char to
       ; Intel Hex High Byte
add    R0,#%0A
sll    R0,#4               ; shifted to the high
byte
and    R0,#%000000F0
ld     R1,R0               ; load the high byte
data

call   receive_char
ld     R0,R5               ; R5 = received data
add    R0,#%FFFFFFBF
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,#%0000000F
or     R0,R1;R0 = converted Intel Hex 32

add    R11,R0              ; checksum addition
ret
```

**Figure 10. ASCII to INTEL HEX 32: Flow Diagram and Assembly Code**

## Page Write

The Page Write function, shown in Figure 11, is responsible in writing the data bytes to the specified Flash address. Thus the following steps below are done to be able to write the correct data bytes to the Flash address.

1. The register R2 is used to store the start address value and the register R3 is used to store the end address.
2. The R2 start address (current address) is checked to determine it exceeds the restricted address, which is the boot loader address range `1F800h-1FFFFh`. Failure to satisfy this condition will result in program termination and will display the following error message in HyperTerminal:

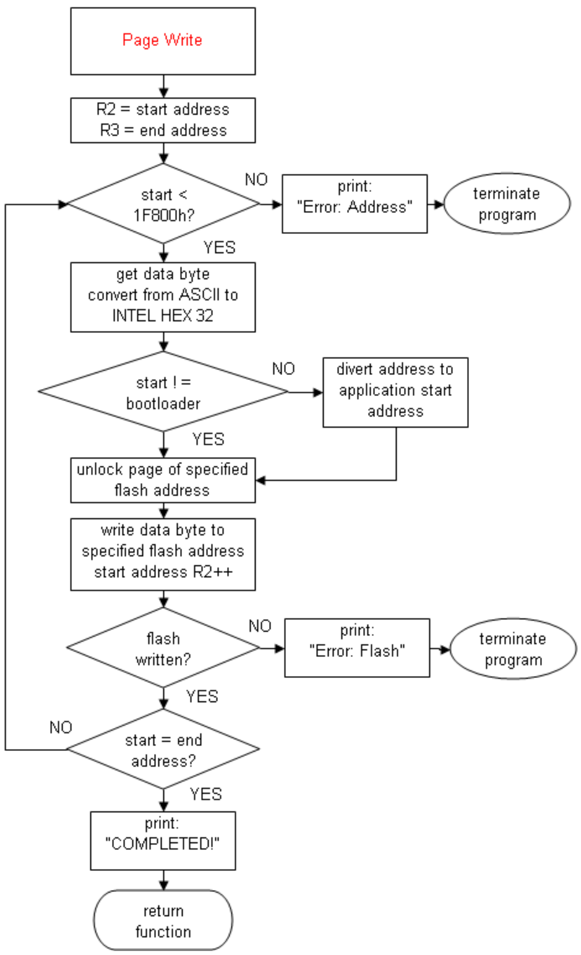
```
Error: Address: Change Constant Data (ROM) = 00000-XXXXX
and Program(EROM) = (XXXXX+1)-1F7FF
```

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 start address then the value of the current address is diverted to the application code start address.
6. Flash memory is checked to determine if it actually wrote the value of the data byte it received. This error-checking condition also prevents a corrupted program from being programmed in Flash memory. Failure to satisfy this condition will result in program termination and will display the following error message in HyperTerminal:

   ```
   Error: Flash Write
   ```

7. Finally, if it has reached the end of the hex file data, the current address is compared to the end address. If the end of the hex file has been reached, HyperTerminal will display:

   ```
   COMPLETED!
   ```




```
Assembly Code
page_write:
sub     R3,#%01
jp      write_end
write_continue:
call    ascii_to_intelhex
sll     R0,#8
ld      R4,R0
call    ascii_to_intelhex
add     R4,R0
cp      R2,#%1F800
jp      nc,print_error_write_address
ld      R0,#%06
cp      R0,R2
jp      c,write_continue_data
cp      R2,#%04
jp      c,write_continue_data
add     R2,#%1F7F8
ld      R1,R2
call    page_unlock
ld.w    (R2),R4
ld.w    R0,(R2++)
cp      R0,R4
jp      nz,print_error_write_flash
sub     R2,#%1F7F8
jp      write_end

write_continue_data:
ld      R1,R2
call    page_unlock
ld.w    (R2),R4
ld.w    R0,(R2++)
cp      R0,R4
jp      nz,print_error_write_flash

write_end:
cp      R2,R3
jp      c,write_continue
ld      R1,R6
and     R1,#%01
cp      R1,#%00
jp      z,page_write_end
call    ascii_to_intelhex
sll     R0,#8
ld      R4,R0
add     R4,#%00FF
ld      R1,R2
call    page_unlock
ld.w    (R2),R4
page_write_end:
ret
```

**Figure 11. Page Write: Flow Diagram and Assembly Code**

## User Application Code

The user application space, exemplified in the following assembly code, contains the downloaded application code which resides in the address range `0000h–1F7FFh`. Within this range, the application start vector resides at address `1F7FCh`.

```
Assembly Code
ld    R4,%1F7FC         ; R4 = Application Code Start Address
(0x1F7FC)
call  delay
call  delay
jp    (R4)              ; go to application_code();
```

## Puts Function

The Puts function, exemplified in the following assembly code, is used to print a string of characters, starting with the 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, exemplified in the following assembly code, is used to print a 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, exemplified in the following assembly code, 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, exemplified in the following assembly code, is used to delay the next instruction.

```
Assembly Code
delay:
  ld      R2,#%FF
loop2:
  ld      R1,#%FF
loop1:
  dec     R1
  ld      R0,R1
  jp      nz,loop1
  dec     R2
  ld      R0,R2
  jp      nz,loop2
  ret
```

## Test 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

- ZNEO Series Development Kit, including development board, power supply, USB interface and Zilog XTools
- RS-232 cable

## Download the Boot Loader Program

Observe the following procedure to download the boot loader code to the Z16F Series MCU.

1. Extract the AN0325-SC01.zip file to a convenient location on your PC's hard drive.
2. Connect the power, USB Smart Cable and serial cable to the ZNEO Series Development Board.
3. Launch ZDSII 4.11.1 – ZNEO.
4. From the **File** menu in ZDSII, click **Open** and select the **ZNEO_bootloader** project file to display the **Boot Loader** dialog box.
5. 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 Figure 12.

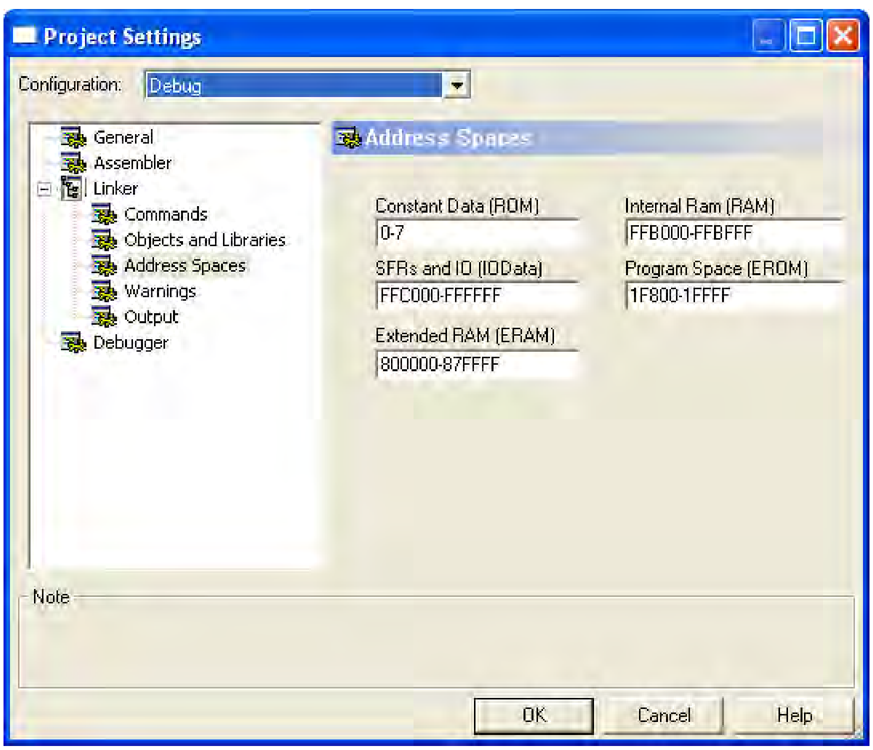



**Figure 12. Project Setting (Address Space) of the ZNEO Boot Loader**

6. Compile and download the program to the ZNEO development board.

## Load Hex Code

Observe the following procedure to establish serial communication and load the boot loader hex code to the Z16F Series MCU.

1. Unplug the ZDSII IDE from the MCU and connect the RS-232 cable to your PC and to the development board.
2. Double-click the `BAUD_57600` file included in the AN0325-SC01.zip source code file (available on [www.zilog.com](http://www.zilog.com)) to Launch HyperTerminal; the **Port Settings** parameters should already be configured as shown in Figure 13.

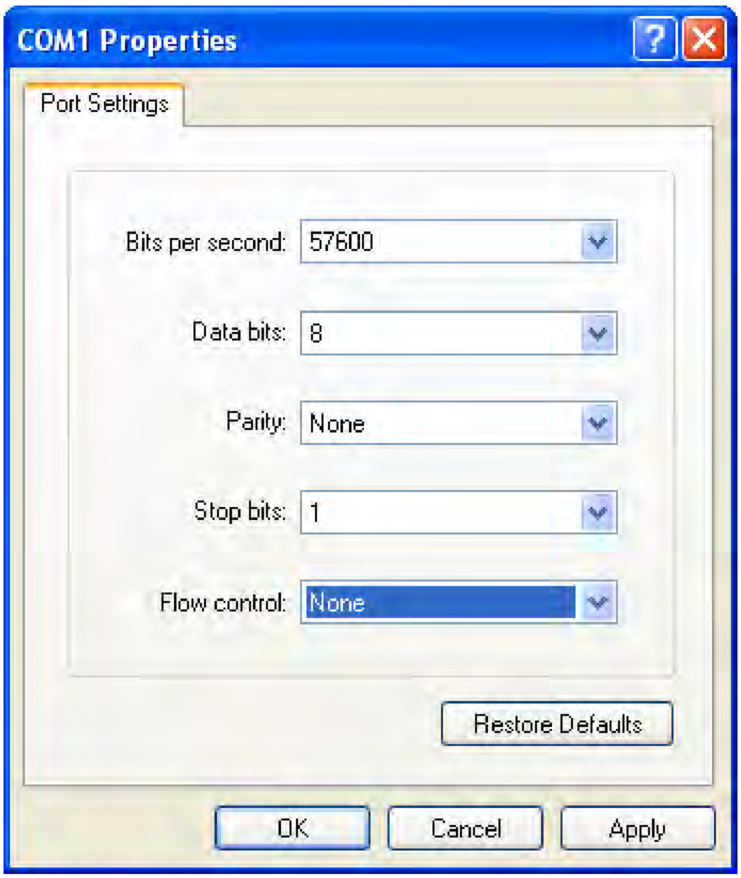



**Figure 13. Port Settings in HyperTerminal**

3. Press the space bar on your keyboard and, at the same time, press the reset button on the development board to reset the MCU.
4. Release the space bar of the PC, then release the reset button to start the user program. HyperTerminal should present a display similar to the result shown in Figure 14.

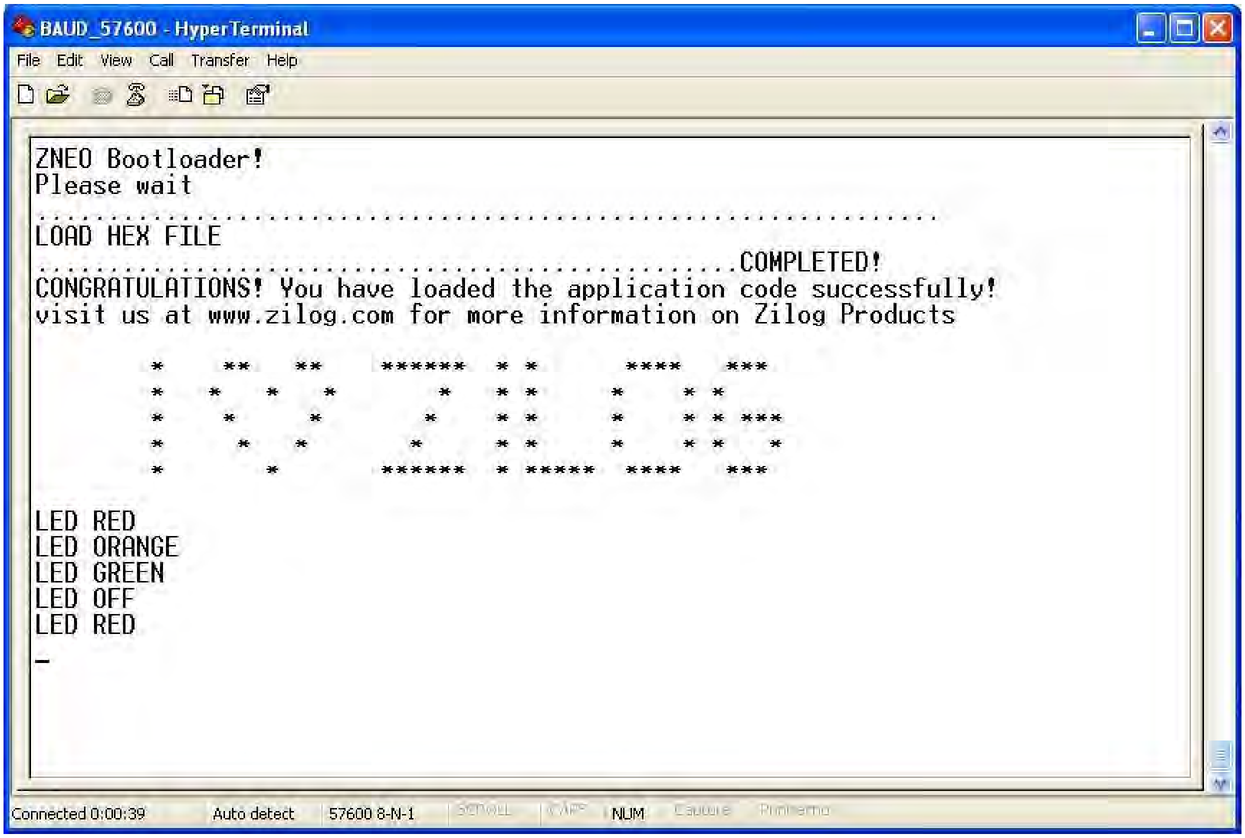



**Figure 14. HyperTerminal Displays the Boot Loader Initialization**

5. When you are prompted by the `LOAD HEX FILE` statement, click **Transfer**, then choose **Send Text File**. Search for and open the file labeled `application_code.hex`.
6. The HyperTerminal application should present the following information:

```
CONGRATULATIONS! You have loaded the application code suc-
cessfully!
Visit us at www.zilog.com for more information on Zilog
Products
```

After a few seconds, the HyperTerminal screen will scroll to slowly show:

```
I ♥ ZILOG
```

followed by the LED light sequence.

### Configure Application Code Address Space

Before compiling the user application code to create the user application hex file, the user should configure the address space for user application code by observing the following procedure.

1. In ZDSII, navigate to the **Project Settings** dialog box.
2. In the left pane, click **Linker**, then **Address Spaces**. The **Address Spaces** panel will appear, as shown in Figure 15.
3. In the **Constant Data (ROM)** field of the **Address Spaces** panel, enter an address range of `0x00000–0x07FFF`.
4. In the **Program Space (EROM)** field, enter an address range of `0x08000–1F7FF`.

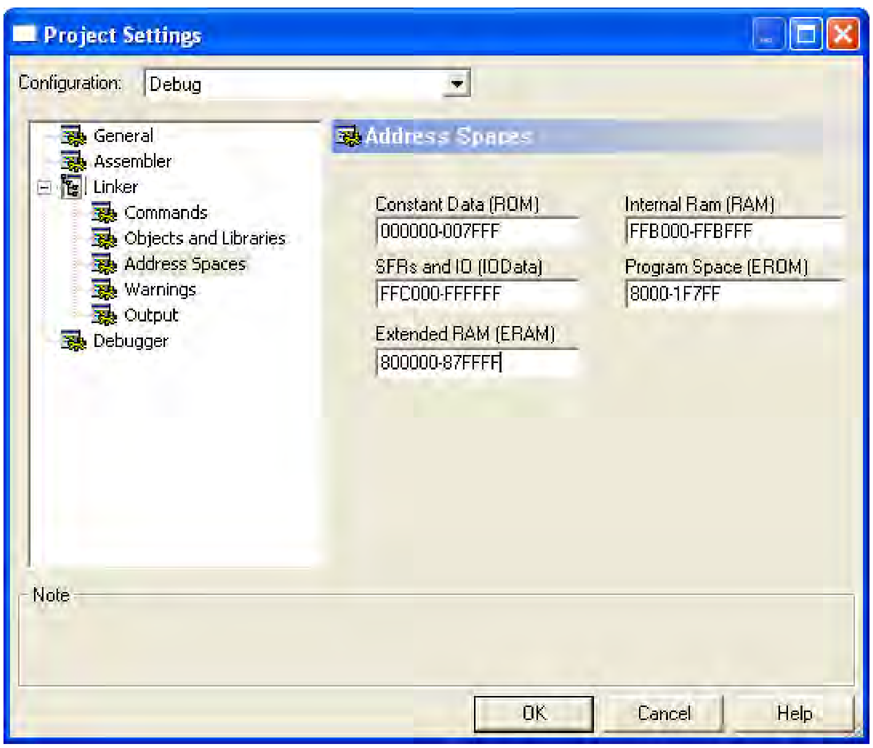



**Figure 15. Address Space for Application Code**

## Summary

This boot loader program for ZNEO CPU-based MCUs is designed to be used as a serial communication alternative to Zilog's XTools firmware downloader, which communicates via a USB port. One limitation observed is that the Flash memory settings located in the address range `00000h–00003h` are permanent and cannot be changed when using this serial boot loader. Another limitation is that Flash memory can only handle 63 pages (126 KB) because the last page of memory must be reserved for boot loader code.

# References

- ZNEO Z16F Series Product Specification (PS0220)
- ZNEO CPU User Manual (UM0188)

# Appendix A. Assembly Code for a ZNEO-Based Boot Loader

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

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

**Code**

```
main:
        ld      sp,#%FFC000         ; stack pointer = 0xFFC000
                                    ; initialized UART
        ld      R1,#0
        ld      R2,#_SYS_CLK_FREQ
        ld      R3,#57600

        ld      R0,R3               ; UART0 initialization
        sll     R0,#3
        add     R0,R2
        sll     R3,#4
        udiv    R0,R3
        ld.w    U0BR,R0             ; UART Baud Rate

        ld      R0,#%30             ; UART GPIO initialization
        or.b    PAAFL,R0            ; PAAFL = 0x30
        clr.b   U0CTL1              ; U0CTL1= 0x00
        ld      R0,#%C0
        ld.b    U0CTL0,R0           ; U0CTL0= 0xC0
                                    ; delay function
        call    delay
        call    delay

        ld.b    R0,U0RXD            ; Read the data from data receive register
        cp      R0,#%20             ; if (U0RXD=0x20)
        jp      z,bootloadercode    ; bootloadercode();
                                    ; else
                                    ; application_code();
```

```
ld      R4,%1F7FC       ; R4 = Application Code Start Address
                        (0x1F7FC)
call    delay
call    delay
jp      (R4)            ; go to application_code();
```

**Function** Bootloadercode

**Description** Contains the initialization code for Flash memory, including the unlock Flash, lock Flash, erase Flash, write Flash and get hex line functions; this latter function is used to read the application code hex file.

**Included Functions**

- Init_flash
- Puts
- Erase_flash_address
- Write_bootloader_start_address
- Get_hex_line
- Lock_flash

**Registers** R0, R1, R2, R3 and R4 are used as variable registers.

**R12 used for extended addressing**

**Code**

```
bootloadercode:
        ld      R1,#print_Bootloader            ; printf("ZNEO Boot Loader");
        call    puts
        call    init_flash                      ; initialized Flash memory
        ld      R1,#print_pleasewait            ; printf("Please wait...");
        call    puts

        clr     R2                              ; R2 = start address of flash to
                                                be erase
        ld      R3,#%1F800                      ; R3 = end address to flash be
                                                erase
        call    erase_flash_address             ; erase address (0x0000-
                                                0x1F7FF)
        call    write_bootloader_start_address  ; rewrite boot loader code
                                                address (0x04-0x07)

        ld      R1,#print_Load_HEX_file         ; print "LOAD HEX FILE"
        call    puts

        clr     R12                             ; initialized extended address
```

```
        call    get_hex_line            ; read the hex file of the appli-
                                        cation code
        call    lock_flash

        call    delay
        ld      R1,#print_completed     ; print "COMPLETED!"
        call    puts

        ld      R4,%1F7FC               ; R4 = Application Code Start
                                        Address (0x1F7FC)
        call    delay
        call    delay
        jp      (R4)                    ; Go to Application code
        ret
```

**Function** init_flash

**Description** Used to initialize the clock in Flash memory.

**Included Functions** None.

**Registers** R0 and R1 are temporary variables.

**Code**

```
init_flash:
        ld      R1,#_SYS_CLK_FREQ   ; initialized clock frequency
        ld      R0,#%3E8
        udiv    R1,R0
        ld.w    FFREQ,R1
        ret
```

**Function** erase_flash_address

**Description** Erase the Flash address range specified by R2 (the start address) to R3 (the end address).

**Registers** R2 = Start address of the Flash memory space to be erased.

R3 = End address of the Flash memory space to be erased.

**Code**

```
erase_flash_address:
        ld      R1,#%2E             ; print progress "...."
        call    putch
```

```
        ld      R1,R2                   ; loads the start address to be erase
                                        to R1
        call    page_erase
        add     R2,#%800                ; increment page erase since 1 page
                                        = 0x800 bytes
        cp      R2,R3                   ; compare if current erase page =
                                        end address page
        jp      c, erase_flash_address  ; repeat if current erase page < end
                                        address page

        ld      R1,#%0A                 ; print progress new line
        call    putch
        ret
```

**Function** Page_erase

**Description** Erase the page within Flash memory at an address specified by register R1.

**Included Functions**
- Page_unlock

**Registers** R1 = Address of Flash memory to be erased.

**Code**

```
page_erase:
        call    page_unlock
        ld      R0,#%95
        ld.b    FCMD,R0
flash_stat:                     ; wait until status register is clear
        ld      R0,#%10
        tm.b    FSTAT,R0
        jp      nz,flash_stat
        ret
```

**Function** Page_unlock

**Description** Unlock the page within 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:
        srl     R1,#%B
        sll     R1,#%3

        ld.w    FPAGE,R1
        ld      R0,#%73
        ld.b    FCMD,R0
        ld      R0,#%8C
        ld.b    FCMD,R0
        ret
```

| | |
|---|---|
| **Function** | lock_flash |
| **Description** | Lock the contents of Flash memory to protect them from being overwritten. |
| **Included Functions** | None. |
| **Registers** | N/A |

**Code**

```
lock_flash:
        clr.b   FCMD            ; FCM = 0x00;
        ret
```

| | |
|---|---|
| **Function** | write_bootloader_start_address |
| **Description** | Rewrite the start address of the boot loader code. |
| **Included Functions** | • Page_unlock<br>• Lock_flash |
| **Registers** | R1 = Start address of the boot loader code.<br>Write (0001 F800) to address (0x0004-0x0007). |

**Code**

```
write_bootloader_start_address:
        ld      R1,#%00         ; R1 = unlock page address (0x00)
        call    page_unlock     ; unlock Flash memory.

        ld      R1,#%0004
        ld.w    (R1++),#%0001   ; R1 = load data (00 01) to address (0x0004-
                                0x0005)
        ld.w    (R1),#%FC00     ; R1 = load data (F8 00) to address (0x0006-
                                0x0007)
```

```
call        lock_flash          ; lock Flash memory
ret
```

**Function** get_hex_line

**Description** Reads a line in a hex file. The following steps are involved in reading a hex file:

1. Check if (R5) receive char is ':' or char (0x34)
2. Store 1st Hex Byte to Data size (R6)
3. Store 2nd Hex Byte to High Byte Address (R7)
4. Store 3rd Hex Byte to Low Byte Address
5. Write 4th to (N-1) Hex Byte to the Address specified (depends on the data size)
6. Store Last (Nth) Hex Byte Data Size(R6)

Notes:
- In the write byte, if the address = (0x00000-0x00007), then divert to (0x1F7F8-1F7FF).
- If the address = (0x00008-0x1F7FF), go directly to the address.
- If address = (0x1F800-0x1FFFF), an illegal hex line can overlap the boot loader.

**Included Functions**
- Receive_char
- Ascii_to_hex_line
- Page_write
- Putch

**Registers** R5 = Hex byte.

R6 = Data size.

R7 = Address byte.

R8 = Record byte.

R9 = Data byte = # of byte (R6) is equal to data size store in address (R7).

R10= Checksum = hex byte(R6) + hexbyte_H(R7) + hexbyte_L(R7) + hexbyte(Nth) = R11.

**Code**

```
get_hex_line:
            call        receive_char        ; start line
            ld          R0,R5               ; R5 = received data
            cp          R0,#%3A             ; check for ':' or 0x34 ASCII
            jp          nz,get_hex_line

            clr         R11                 ; initialized checksum = 0x00
            ;--------------------------------DATA SIZE----------------------------------------------
            call        ascii_to_intelhex   ; data size
```

```
        ld      R6,R0               ; R6 = data size

;--------------------------------ADDRESS------------------------------------------------
        call    ascii_to_intelhex
        sll     R0,#8               ; high byte address
        ld      R7,R0               ; R7 = high byte address

        call    ascii_to_intelhex   ; low byte address = R0
        add     R7,R0               ; R7 = add high and low address byte
;--------------------------------RECORD BYTE-------------------------------------------
        call    ascii_to_intelhex   ; record byte
        ld      R8,R0               ; R8 = record byte
        cp      R8,#%01             ; check record type if end of file
        jp      z,get_line_end

        cp      R8,#%04             ; check record type if extended address
        jp      nz,write_data_byt
                e
        ld      R12,#%10000
        jp      get_hex_line
;--------------------------------DATA BYTE---------------------------------------------
write_data_byte:
        ld      R2,R7               ; R2 = start address(R7)
        add     R2,R12              ; R2 = start address(R7) + extended
                                    address(R12)
        ld      R3,R6               ; R3 = data size(R6)
        add     R3,R2               ; R3= data size(R6) + start address(R2) = ter-
                                    minal address
        call    page_write          ; write the data_bytes to memory
;--------------------------------CHECKSUM----------------------------------------------
        ld      R2,R11              ; (R2) sum of all = data size + byte_address +
                                    ; record_byte + data byte

        call    ascii_to_intelhex   ; checksum
        and     R0,#%000000FF
        ld      R10,R0
```

```
            and     R2,#%000000FF   ; checksum = 0xFFFFFFFF - sum of all +
                                    0x01
            ld      R0,#%FFFFFFFF
            sub     R0,R2
            add     R0,#%01
            and     R0,#%000000FF

            cp      R0,R10
            jp      nz,print_error_ch
                    eck_sum
            ld      R1,#%2E         ; print progress
            call    putch

            jp      get_hex_line
get_line_end:
            ret
```

**Function** receive_char

**Description** Get a character from U0RXD and store it to R5.

**Included Functions** Receive_char

**Registers** R5 = Hold the received character.

**Code**

```
receive_char:
            ld      R0,#%80
            tm.b    U0STAT0,R0      ; Read the UART0 status register.
            jp      z,receive_char  ; Check if any character is received.
            ld.b    R5,U0RXD        ; R5 = Store the data from data receive regis-
                                    ter (U0RXD).
            ret
```

**Function** ascii_to_intelhex

**Description** Get a character from U0RXD via R5 and convert it to INTEL HEX 32 format.

**Included Functions** • Receive_char

**Registers** R0 = Converted to Intel Hex 32 format.

R5 = Received ASCII character.

R11= Add the Intel Hex 32-converted character to the checksum.

**Code**

```
ascii_to_intelhex:
          call    receive_char
          ld      R0,R5                 ; R5 = received data.
          add     R0,#%FFFFFFBF
          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
          sll     R0,#4                 ; shifted to the high byte
          and     R0,#%000000F0
          ld      R1,R0                 ; load the high byte data

          call    receive_char
          ld      R0,R5                 ; R5 = received data
          add     R0,#%FFFFFFBF
          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,#%0000000F
          or      R0,R1                 ; R0 = converted Intel Hex 32

          add     R11,R0                ; checksum addition
          ret
```

**Function** page_write

**Description** Write the data to the specified address.

**Included Functions**
- Ascii_to_intelhex
- Page_unlock

**Registers** R2 = Start write address of the page.
R3 = End write address of the page.

**Code**

```
page_write:
          sub     R3,#%01
          jp      write_end
```

```
write_continue:
        call    ascii_to_intelhex            ; get the data byte
        sll     R0,#8                        ; R0 = high data byte
        ld      R4,R0                        ; R4 = high data byte

        call    ascii_to_intelhex            ; R0 = low data byte
        add     R4,R0                        ; R4 = add high and low data byte

        cp      R2,#%1F800
        jp      nc,print_error_write_address ; if address R2 > 0x1F7FF

        ld      R0,#%06
        cp      R0,R2                        ; if R0 reset address (0x06) <
                                             data_address (R2)
        jp      c,write_continue_data        ; then write data to specified data
                                             address
        cp      R2,#%04                      ; if data_address (R2) < reset
                                             address (0x06)
        jp      c,write_continue_data        ; then write data to specified data
                                             address
                                             ; else
        add     R2,#%1F7F8                   ; divert the address to app. start
                                             address (0x1F7FC)
        ld      R1,R2                        ; page address to be unlock
        call    page_unlock                  ; unlock Flash memory at 0x10000
        ld.w    (R2),R4                      ; write the data byte (R4) to
                                             address (R13)
        ld.w    R0,(R2++)                    ; check the flash if the data is writ-
                                             ten
        cp      R0,R4
        jp      nz,print_error_write_flash   ;if data is not written then print
                                             error write flash
        sub     R2,#%1F7F8
        jp      write_end

write_continue_data:
        ld      R1,R2
        call    page_unlock                  ; unlock Flash memory at 0x10000
        ld.w    (R2),R4                      ; write the data byte (R4) to
                                             address (R2)
```

```
        ld.w    R0,(R2++)               ; check the flash if the data is writ-
                                        ten
        cp      R0,R4
        jp      nz,print_error_write_flash  ; if data is not written then print
                                        error write flash

write_end:
        cp      R2,R3                   ; check if start write address = end
                                        write address
        jp      c,write_continue        ; else write continue

        ld      R1,R6                   ; R1 = data size (R6)
        and     R1,#%01                 ; check for odd or even
        cp      R1,#%00                 ; if even data byte
        jp      z,page_write_end        ; then go to page write end
                                        ; else
        call    ascii_to_intelhex       ; get the data byte
        sll     R0,#8                   ; R0 = high data byte
        ld      R4,R0                   ; R4 = high data byte
        add     R4,#%00FF               ; initialized LSB

        ld      R1,R2
        call    page_unlock             ; unlock Flash memory at
                                        0x10000.
        ld.w    (R2),R4                 ; write the data byte (R4) to
                                        address (R2)

page_write_end:
        ret
```

**Function** Puts

**Description** Print the string of character.

**Included Functions**
- Putch

**Registers** R1 = Register used to hold the character.

**Code**

```
puts:
        ld      R2,R1           ;R2 = address of the string
        cp      R2,#0           ;if address 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
```

**Function** Putch

**Description** Print the character.

**Included Functions** • send

**Registers** R1 = Register used to hold the character.

**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
```

**Function** Send

**Description** Used to send the character.

**Included Functions** None.

**Registers** R1 = Register used to transmit the data to TXD register.

**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
```

**Function** Delay

**Description** Used to delay the next instruction.

**Included Functions** None.

**Registers** R0, R1 and R2 used as variable registers.

**Code**

```
delay:
                ld      R2,#%FF
loop2:
                ld      R1,#%FF
loop1:
                dec     R1
                ld      R0,R1
                jp      nz,loop1
                dec     R2
                ld      R0,R2
                jp      nz,loop2
                ret
```

**Function** print_error_check_sum

**Description** Prints "error checksum" in HyperTerminal.

**Included Functions** • puts

**Registers** R1, which stores the character array.

**Code**

```
print_error_check_sum:                       ; computed checksum(R11) !=
                                             Hex Byte checksum(R10)
                ld        R1,#print_error    ; print "error"
```

```
            call    puts
            ld      R1,#print_error_checksum     ; print "checksum"
            call    puts
            jp      $                            ; terminate program
```

**Function** print_error_write_address

**Included Functions** • puts

**Description** Prints "error address" in HyperTerminal.

**Registers** R1, which stores the character array.

**Code**

```
print_error_write_address:                     ; address > 0x1F7FF boot loader
                                               code
            ld      R1,#print_error            ; print "error"
            call    puts
            ld      R1,#print_error_address    ; print "Address: Change Constant
                                               Data(ROM)=0000-77FF..."
            call    puts
            jp      $                          ; terminate program
```

**Function** print_error_write_flash

**Included Functions** • puts

**Description** Prints "error address" in HyperTerminal.

**Registers** R1, which stores the character array.

**Code**

```
print_error_write_flash:
            ld      R1,#print_error      ; print "error"
            call    puts
            ld      R1,#print_write_fl   ; print "flash"
                    ash
            call    puts
            jp      $                    ; terminate program
```

**Function** print_Load_HEX_file

**Included Functions** None.

**Description** Prints "LOAD HEX FILE" in HyperTerminal.

**Registers** R1, which stores the character array.

**Code**

```
print_Load_HEX_file:
                DB      "LOAD HEX FILE"
                DB      10,0
```

**Function** print_Bootloader

**Included Functions** None.

**Description** Prints "ZNEO Boot Loader!" in HyperTerminal.

**Registers** R1, which stores the character array.

**Code**

```
print_Bootloader:
                DB      "ZNEO Boot Loader!"
                DB      10,0
```

**Function** print_error

**Included Functions** None.

**Description** Prints "Error:" in HyperTerminal.

**Registers** R1, which stores the character array.

**Code**

```
print_error:
                DB      "Error:"
                DB      10,0
```

**Function** print_error_address

**Included Functions** None.

**Description** Prints "Address: Change Constant Data(ROM) = 0000-(XXXX-1) and Program(EROM)=XXXX-1F7FB" in HyperTerminal.

**Registers** R1, which stores the character array.

**Code**

```
print_error_address:
                DB      "Address: Change Constant Data(ROM) = 0000-(XXXX-1) and Pro-
                        gram(EROM) = XXXX-1F7FB"
                DB      10,0
```

**Function** print_error_checksum

**Included Functions** None.

**Description** Prints "checksum" in HyperTerminal.

**Registers** R1, which stores the character array.

**Code**

```
print_error_checksum:
                DB      "checksum"
                DB      10,0
```

**Function** print_write_flash

**Included Functions** None.

**Description** Prints "flash" in HyperTerminal

**Registers** R1, which stores the character array.

**Code**

```
print_write_flash:
                DB      "flash write"
                DB      10,0
```

**Function** print_pleasewait

**Included Functions** None.

**Description** Prints "Please wait" in HyperTerminal.

**Registers** R1, which stores the character array.

**Code**

```
print_pleasewait:
                DB      "Please wait"
                DB      10,0
```

**Function** print_completed

**Included Functions** None.

**Description** Prints "COMPLETED!" in HyperTerminal.

**Registers** R1, which stores the character array.

**Code**

```
print_completed:
```

```
DB      "COMPLETED!"
DB      10,0
```

# Appendix B. Intel Hex 32 Format

The boot loader application can program a standard file format into the ZNEO-based MCU's Flash memory. The Intel Standard Hex32 file format is one of the popular and commonly-used file formats. An Intel Standard Hex 32-formatted file is an ASCII file that contains one record per line, as described 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 in 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, including the following data types:

- Data Record or normal addressing (00)
- End Of File Record (01)
- Extended Linear Address Record

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

**Checksum.** This field is used to determine if the received data is correct. The checksum must be equal to the two's complement of the sum of data size, MSB address, LSB address, record type and the data bytes.

```
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 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