Technical Note

Setting Interrupts for eZ80® Devices

TN001802-1203

Abstract

This Technical Note discusses how to set interrupts for devices in the eZ80® family of microprocessors. The eZ80 family features a number of options for handling interrupts. The eZ80 family includes the eZ80190, eZ80L92, eZ80F91, eZ80F92, and eZ80F93 devices.

eZ80190 MPU Interrupts

On power-up, the interrupt vector table starts at address 0000h. Each vector is two bytes and can jump within a 64KB range. The interrupt system is disabled upon power-up, and the PC starts running at address 0000h. Users can alter the code on power-up to jump around this interrupt vector range (00h–70h). Next, there are two options for setting up the final run-time vectors. One option is to hard-code the vectors into another address within the 64KB range by setting the I register to control the upper 8 bits of the vector address. To jump into an interrupt vector routine that is outside the 64KB range, build a second jump table comprising four bytes.

The other option for setting up the 64KB interrupt table is with on-chip RAM. The eZ80190 device features 8KB of on-chip RAM and can be mapped into the code space of the eZ80 processor. This RAM, when enabled, overlays any other memory that might also be enabled. The user's code should set up the RAM space and the I register, and load the RAM space with the short and long jump tables for the interrupt routine before enabling the master interrupt system of the eZ80 with the EI instruction. Below is a code listing showing the different segments required in the code.

```
_vector_uart0:
  .byte     %12           ; Address of first jump vector for
                          ; UART0
;-------------------------------
; on-chip RAM jump tables
_interrupt_table:
  .long     %0e000        ; This is the start address
                          ; of the first jump table

_interrupt_jump_table_1:  ; This is the start address of the
  .long     %00e100       ; Long jump table.

_nvectors:                ; Number of interrupts to init.
  .word     96
```

```
****** Start of code

  Org 0000h
  Di        ; Just to make sure
  Jp.lil  _Cint0          ; Do a long jump here to enable the full 24 bit
                          ; address range.

  **** You might want to block off this area for NMI and trap vectors

    Org 0100h

_Cint0:
    ld.sis  sp,TOSPS      ;Set up Little stack pointer
    ld.lil  sp,TOSPL      ;Set up Big stack pointer

;Set up internal RAM
    ld      a,80h         ;Enable on-chip SRAM at 00E000h - 00FFFFh 8K
    out     (RAM_CTL0),a
    ld      a,0h
    out     (RAM_CTL1),a

  **** Your other chip init code goes here

    Jp      _main


Void Main(void)
{
init_interrupts();        // Set up the I reg. And the jump
                          // tables with the null vector isr.

sethandler(isr_uart0, (unsigned char)vector_uart0);
                          //load on-chip RAM tables to point
                          // the UART0 ISR routine to "isr_uart0"
                          // function. Vector_uart0 is the vector
                          // number as defined by the 190 user
                          // manual.

  ei();                   // Make sure this goes after you do
                          // The above init. Code.

  do {
      Do_something_routine();
     } while(1);
}/** end of main();

************************
*
* These are interrupt routines for the eZ80
*
* Because the interrupt table and the interrupt routines must be within
```

```
* the first 64KB boundary, the following will allow interrupt
* routines to be anywhere within the 16MB address space.
*
* To enable this, two tables are setup in the on-board SRAM which
* is at 00E000 to 00FFFF
*
* The first table is the interrupt vector table. It lies
* at 00E000 to 00E0FF (or wherever 'interrupt_table' points to)
* and contains vectors into our next interrupt "jump" table.
*
* The second table lies from 00E100 to 00E37f (or wherever
* 'interrupt_jump_table_1' points to). It is a table of jp.lil
* instructions to the 24 bit address of the actual interrupt
* routine. Each entry is 5 bytes. The first two bytes are the opcode
* 0x5b, 0xc3 which is the jp.lil pneumonic. The next three bytes are
* the 24 bit address of the interrupt routine.
*
* The sethandler function will automatically place an isr routine
* in the interrupt "jump" table.
*
/*************************************************************
* This function will place an interrupt handler in the interrupt
* "jump" table.
*
* You only need to pass it the actual interrupt vector number. It
* Will compute the offset into our interrupt jump table and set
* it accordingly.
*
* Will return the old interrupt handler.
*************************************************************/
void* sethandler(void (*handler)(void), unsigned char vector)
{
  void* oldhandler;
  void** ptr;

  ptr = (void*)(interrupt_jump_table_1+vector/2*5);

  /* point vector to our jump table */
  *((unsigned short*)(interrupt_table+vector))= (unsigned short)ptr;

  /* set our jp.lil opcode in big endian format */
  *((unsigned short *)ptr) = 0xc35b;

  ptr=(void **)(interrupt_jump_table_1+vector/2*5+2);

  oldhandler = *ptr;
  /* put address of our isr handler in the jump table */
  *ptr = (void *)handler;

  return oldhandler;
}
```

```
/*************************************************************
* This function sets up the interrupts tables on the eZ80. It will
* initialize each vector in the interrupt vector table to point to
* its corresponding entry in the interrupt "jump" table.
* It will also initialize the interrupt jump table and point each entry
* to null_isr which is defined as
*
* _null_isr:
* ei
* reti
*
*************************************************************/

void init_interrupts(void)
{
  int i;

    RAM_CTL0=0x80;        //enable on-chip SRAM
    RAM_CTL1=0x00;

  //initialize all interrupt vectors to null isr
  for(i=0; i<nvectors; i+=2) {
    sethandler(&isr_null,i);
  }
_asm("\tld a,%e0\n\tld i,a") ; //Set the I reg to E0

}

/*************************************************************/

#pragma interrupt
void isr_null(void)
{
  return;
}

/*************************************************************
* ISR routine for UART 0
*
*/

#pragma interrupt
void isr_uart0(void)
{
  ** Your UART0 ISR routine goes here....
}
```

Note: The NMI interrupt is one interrupt source that is not disabled on power-up and can also be active any time after power on reset. This vector should then be handled with a special `org` statement in your startup code if the NMI input pin is used.

eZ80L92 MPU Interrupts

The method of setting interrupts on the eZ80L92 device is the same as for the eZ80190 device, with the additional exceptioni of on-chip SRAM, which must be present in the system. Interrupt init and sethandler routines are the same as for the eZ80190 device, but a third jump table is required at the default vector address (address 00h—the final interrupt vector). This jump table jumps to another jump table that is within the first 64 KB range. This second jump table can then jump to a final ISR routine or to an off-chip RAM base vector table. Review ZiLOG's eZ80L92 Development Kit Flash Loader Installation Product User Guide (PUG0013) for more information on this subject. This document explains how the jump tables are controlled in the eZ80L92 Flash Loader.

eZ80F91, eZ80F92, and eZ80F93 Interrupts

The eZ80F92 and eZ80F93 MCUs operate much the same as the eZ80190 device as regards on-chip SRAM. If the user only targets code to run within on-chip Flash memory, then ISR routines must be maintained within the first 64 KB, and the default vector must point directly to addresses for the ISR routines.

The eZ80F91 device, however, features an added interrupt controller to facilitate interrupts. The default interrupt vectors are four-byte addresses, instead of two-byte addresses, to allow the user to point the default ISR vector directly to the ISR routine. Below is simple ISR setup code for the eZ80F91 device.

```
;*********************************
; Program entry point
;*********************************
  .org      %00
  di
  jp.lil    _c_int0       ; Jump around the ISR vectors.

;-------------------------------------------------------------
; ISR Vectors
;
// Note that the 'DL' define yields two words or 4 bytes. The upper
// byte is loaded with 00. We only need 24 bits.

  .org      %40
  dl        %000000       ;
  dl        %000000       ;
  dl        %000000       ;
  dl        %000000       ;
  dl        %000000       ;
  dl        _isr_timer0   ;PRT0_ISR
  dl        _isr_timer1   ;PRT1_ISR
  dl        %000000       ;
  dl        %000000       ;
  dl        %000000       ;
  dl        %000000       ;
  dl        %000000       ;
```

```
  dl        _isr_uart0    ;UART0_ISR
  dl        _isr_uart1    ;UART1_ISR
  dl        %000000       ;

  ** add all interrupt vectors that you are going to use.

;-------------------------------------------------------------------------
; Initialize Stack pointer
  extern    TOSPS
  extern    TOSPL

_c_int0:
  ld.sis    sp,TOSPS      ; Setup SPS
  ld.lil    sp,TOSPL      ; Setup SPL
;*************************************************************************

  ld a, 00h               ; Disable on-chip SRAM
  out (RAM_CTL0), a       ; depends on what you want to do with the
                          ; on-chip SRAM.

  **** do other chip init here.
  **** final jump to main

call_main                 ; main()

void main(void)
{
init_com1();              // Init com port - enable com1 ISR
  init_timer1();          // Init 100ms Timer - enable timer 1 ISR

_ei();                    // Turn on the master interrupt system.

do
{
  Your code goes here....
}while(1);

/*************************************************************
* This will initialize timer1 to interrupt every 10 ms
*
* 16 bit time constant is not big enough for 100 ms interrupts,
* so we will use additional intermediate counter to count
* every 10 ticks.
*/

void init_timer1(void)
{
  ticks1 = 0x00;
  intermediate_ticks1 = 0x00;

  TMR_CTL1 = 0x00;
```

```
  TMR_RRL1 = 0xFF;        // setup timer to interrupt every 10ms
  TMR_RRH1 = 0x1F;
  TMR_CTL1 = 0x0e;        // timer0 = multipass, /16, interrupt enable
  TMR_CTL1 |= 0x01;       // enable timer
  TMR_IER1 = 0x01;        // Enables timer 1 interrupt
}

void init_com1(void)
{

  PC_ALT1 &= 0xf0;        // PD0 = uart0_tx, PD1 = uart0_rx
  PC_ALT2 |= 0x0F;

  UART_LCTL1=0x80;        // select dlab to access baud rate generators
  BRG_DLRL1=0x45;         // 9600
  BRG_DLRH1=0x01;
  UART_LCTL1=0x00;        // disable dlab

  UART_FCTL1=0xc7;        // clear tx fifo, clear rx fifo, fifo enable
  UART_LCTL1=0x1B;        // 8-N-1
  UART_MCTL1=0x20;        // disable modem flow control
  UART_IER1=0x05;         // rx int enable, master int enable was 1
}

#pragma interrupt
void isr_timer1(void)
{
  unsigned char temp;
  unsigned int delay;
  temp = TMR_CTL1;        // read to clear pending int
  temp = TMR_IIR1;

  intermediate_ticks1++;
  if(intermediate_ticks1 >= 10)
  {
    intermediate_ticks1 = 0;
    ticks1++;
  }
}

/*****************************************************************
* All this ISR should do is put the data into our internal fifos
*
*/

#pragma interrupt
void isr_uart1(void)
{
     short temp;

     temp = UART_LSR1;
```

```
    if ( temp & 0x04 )
     {
    mdb_buff[byte_pos] = UART_RBR1;
     byte_pos++;
     done = 1;
     }

   if ( temp & 0x01)
     {
      mdb_buff[byte_pos] = UART_RBR1;
     byte_pos++;
     }


   while( UART_LSR1 & 0x20) {              //THRE int

     if( ! fifo_empty(uart1tx->fifo) ) {   // and we still have stuff to
send ...
      UART_THR1=fifo_get(uart1tx->fifo);   // send it.
     } else {                              // otherwise ...
      UART_IER1&=0xfd;                     // disable tx interrupts
      break;
     }
   }
}
}
```

Summary

Most of ZiLOG's tools feature a built-in macro function to help the user set up interrupts. A function can be defined as an interrupt routine using the `#pragma interrupt` keyword. Using `set_vector(ISR,Name of ISR function)`, the tool ensures the `org` of the correct jump address to the default ISR table. Because of the complex nature of the devices in the eZ80 family, there are a number of items to set up, such as the I register and the long and short jump tables.

In the future, ZiLOG intends to add support for the set vector macro function. Because the eZ80190, eZ80L92, eZ80F92, and eZ80F93 devices feature two-byte addresses for the default interrupt vectors, the user must set up other jump tables to bridge the gap to the 24-bit world. The other issue to keep in mind is the I register that controls the upper 8 bits of the default interrupt vector. This control allows the user to move the overall interrupt jump table anywhere within the 64KB range.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, or to request copies of publications, contact:

ZiLOG Worldwide Headquarters
532 Race Street
San Jose, CA 95126-3432
Telephone: 408.558.8500
Fax: 408.558.8300
www.zilog.com

Information Integrity

The information contained within this document has been verified according to the general principles of electrical and mechanical engineering. Any applicable source code illustrated in the document was either written by an authorized ZiLOG employee or licensed consultant. Permission to use these codes in any form, besides the intended application, must be approved through a license agreement between both parties. ZiLOG will not be responsible for any code(s) used beyond the intended application. Contact the local ZiLOG Sales Office to obtain necessary license agreements.

Document Disclaimer