Application Note
SLAA140 – March 2002

Mixing C and Assembler With the MSP430

Stefan Schauer *MSP430*

ABSTRACT

This application note describes how C and assembler code can be used together within an MSP430 application. The combination of C and assembler benefits the designer by providing the power of a high-level language as well as the speed, efficiency, and low-level control of assembler.

Contents

Figures

Tables

1 Definition of the IAR C-Compiler for Passing Variables Between Functions

1.1 Calling Convention - Register Usage With the IAR C-Compiler

The compiler uses two groups of processor registers.

- The scratch registers R12 to R15 are used for parameter passing and hence are not normally preserved across the call.
- The other general-purpose registers, R4 to R11, are mainly used for register variables and temporary results and must be preserved across a call. Within C this is handled automatically.

Note that the –ur45 option prevents the compiler from using registers R4 and/or R5.

1.2 Stack Frames and Parameter Passing

Each function call creates a stack frame as follows:

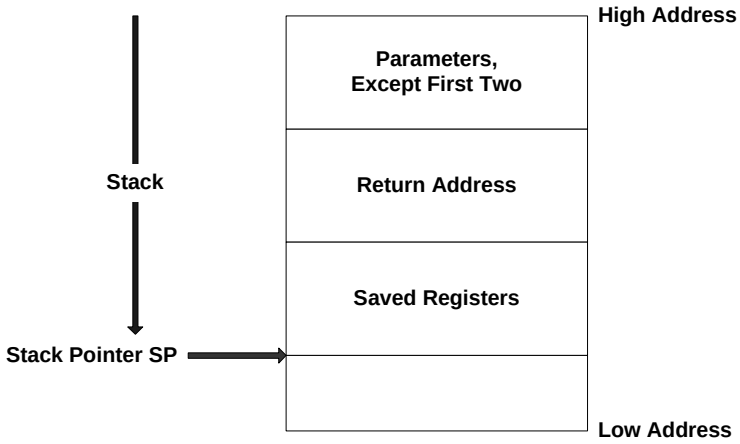

Figure 1. Parameter Passing From C

The parameters of a called function are passed to an assembler routine in a right to left order. The left most two parameters are passed in registers unless they are defined as a struct or union type, in which case they are also passed on the stack. The remaining parameters are always passed on the stack.

See the following example of a call.

f(w,x,y,z)

Since the arguments are dealt with in a right to left order, *z* is loaded onto the stack first, followed by *y,* and *x* is either in R14, R15:R14, or on the stack, depending on its type, as is *w*. The result is returned in R12 (or R13:R12 for a 32 bit type) and in a special area pointed to by R12 if it is a struct or union type.

Table 1. Location of Passed Parameters

Argument	<32 Bit Type	32 Bit Type	Struct/Union
4th (z)	On the stack	On the stack	On the stack
3rd (y)	On the stack	On the stack	On the stack
2nd (x)	R14	R15:R14	On the stack
1st (w)	R12	R13:R12	On the stack
Result	R12	R13:R12	Special area

1.3 Interrupt Functions

Interrupt functions written in C automatically preserve the scratch registers and SR (status register) as well as registers R4 to R11. The status register is saved as part of the interrupt calling process. Any registers used by the routine are then saved using push Rxx instructions. On exit, these registers are recovered using pop Rxx instructions and the RETI instruction is used to reload the status register and return from the interrupt.

Functions written in assembler have to take special care of this.

2 Requirements of Assembler Routines to Support Being Called From C

An assembler routine that is to be called from C must do the following:

- Conform to the calling convention described above.
- Have a PUBLIC entry-point label.
- Be declared as external before any call, to allow type checking and optional promotion of parameters, as in extern int foo() or extern int foo(int i, int j).

2.1 Local Storage Allocation

If the routine needs local storage, it allocates it in one or more of the following ways:

- On the hardware stack.
- In static workspace, provided of course that the routine is not required to be simultaneously reusable (re-entrant).

Functions can always use R12 to R15 without saving them and R6 to R11 provided they are pushed before use. R4 and R5 must not be used for ROM monitor compatible code.

If the C code is compiled with -ur45, but the application is not to run in the ROM monitor, then it is possible to use R4 and R5 in the assembler routine without saving them, since the C code never uses them.

2.2 Interrupt Functions

The calling convention cannot be used for interrupt functions, since the interrupt may occur during the calling of a foreground function. Hence, the requirements for an interrupt function routine are different from those of a normal function routine as follows:

- The routine must preserve all used registers, including scratch registers R12–R15.
- The routine must exit using RETI.
- The routine must treat all flags of the status register as undefined (Carry / Neg. / Zero / Overflow).

2.3 Defining Interrupt Vectors

As an alternative to defining a C interrupt function in assembly language as described above, the user is free to assemble an interrupt routine and install it directly in the interrupt vector.

The interrupt vectors are located in the INTVEC segment.

3 Combining C and Assembler Functions

3.1 General Basics

The mechanics to combine C and assembler functions are fairly straightforward. Basically C code within .c files imports labels exported by the assembler files using the *extern* keyword. Assembler codes within .s43 files export labels to the C code using the *PUBLIC* keyword. Assembler code import labels exported by C code using the *EXTERN* keyword. No keyword is required to export C labels to assembler code.

Once the .c and .s43 files are written, they are added to the workbench project and the project is built. See the example .c, .s43, and project (.prj) files included with this application note. See the IAR documentation for a more complete description of this process.

3.2 Calling Assembler Functions Without Parameters to Pass

If no parameters have to be passed between the C code and the assembler function, a simple call instruction can be used. See example 1.

3.3 Calling Assembler Functions With Parameters to Pass

When it is required to pass parameters from C to the assembler function, the parameters must be located as described in the *Stack Frames and Parameter Passing* section.

Example 2 shows how parameters are passed between the C main program and an assembler function.

3.4 Defining a Interrupt Service Routine in Assembler

It is often required that the interrupt service routine be optimized for speed. This can best be achieved using assembler. See the *Interrupt Functions* section for the requirements of an assembler interrupt service routine.

Example 4 shows a watchdog interrupt service routine written in assembler that is used within a C program.

3.5 Defining Interrupt Service Routine for Special Interrupts (e.g. Timer A, ADC, …)

Some modules (e.g. TimerA / TimerB/ ADC12) use a special combination of hardware and software to detect the source of the individual interrupt. Also see:

- - *MSP430x1xx Family User's Guide,* literature number SLAU012
- - *MSP430x3xx Family User's Guide,* literature number SLAU049
- - *MSP430x4xx Family User's Guide*, literature number SLAU056

Again, the interrupt service routine for these modules is best written in assembler. Example 5 shows how this can be done.

3.6 Calling C Functions from Assembler

It is also possible to call C function from an assembler routine. The same restrictions described in section 1 apply.

In Example 3, the C function rand() is called from the assembler program.

3.7 Switching Off the Low-Power Mode While Within the Interrupt Service Routine

During an interrupt service routine, the status register together with the return address is stored on the stack (see the user's guide for more detailed information). To set the CPU into active mode after returning from the interrupt service routine, the value of the status register on the stack has to be modified (specifically, the bits that specify the low-power mode have to be cleared). From C, it is not possible to directly access the stack pointer, but with the intrinsic function *BIC_SR_IRQ(bits*), the save status register on the stack can be modified.

Note: The intrinsic function *BIC_SR_IRQ(bits)* is only available in the IAR workbench starting from version 1.25.

```
....

#include <msp430x11x1.h>

void main(void)
{
  ...

  _EINT();                              // Enable interrupts

  while(1)
  {
    ...

    _BIS_SR(LPM3_bits);                 // Enter LPM3

    ...
  }
}

interrupt[WDT_VECTOR] void watchdog_timer (void)
{
    _BIC_SR_IRQ(LPM3_bits);             // Clear LPM3 bits from saved SR on Stack
}
```

4 Building Libraries

4.1 Libraries

A library is a single file that contains a number of relocatable object modules, each of which can be loaded independently from other modules in the file, as it is needed.

Normally, the modules in a library file all have the *library* attribute, which means that the linker only loads them if they are actually needed in the program. This is referred to as demand loading of modules.

On the other hand, a module with the *program* attribute is always loaded when the linker processes the file in which it is contained.

A library file is no different from any other relocatable object file produced by the assembler or C compiler, except that it includes a number of modules of the *library* type.

4.2 Using Libraries With Assembler Programs

If programming small assembler programs, there is little need to use libraries. However, when writing medium and large-sized applications, libraries provide the following advantages:

- They allow combining utility modules used in more than one project into a simple library file. This simplifies the linking process by eliminating the need to include a list of input files for all the modules needed. Only the library module(s) needed for the programs are included in the output file.

- They simplify program maintenance by allowing multiple modules to be placed in a single assembler source file. Each of the modules can be loaded independently as a library module.

- They reduce the number of object files that make up an application, maintenance, and documentation.

Assembly language library files can be created using one of two basic methods:

- Assembling a single assembler source file that contains multiple library-type modules can create a library file. The resulting library file can then be modified using XLIB.

- Using the XLIB librarian to merge any number of existing modules together to form a user-created library can produce a library file.

The *name* and *module* assembler directives are used to declare modules as being of *program* or *library* type, respectively.

Additional information can be found in the *IAR MSP430 Assembler, Linker and Librarian Programming Guide* (a430.pdf) that is located in the *doc* folder of the IAR embedded workbench or in the help system of the embedded workbench.

4.3 Merging Modules Into a Library

If functions need to be added to new or existing libraries, the following steps are required.

- Build the object file of the library. Take care of the requirements listed above.
- Build a script file based on the example below, or insert the commands directly into the command window after starting the XLIB program.

build_lib.xlb

```
def-cpu msp430

cd debug
cd obj

make-lib port1
make-lib port2
fetch-mod port1 port
fetch-mod port2 port

list-entries port
list-entries port lstport

exit
```

If this script is stored with the name *build_lib.xlb*, it can be executed with *$PATH$\XLIB build_lib*.

The functions within the port1 and port2 objects are combined into a single library file port (port.r43). The port1 and port2 functions can now be used by any project by simply linking the library to the project.

5 Using Watch Windows With Assembler Variables

First, it is important to realize that C-SPY is a C debugger. With that in mind, here are two ways for watching variables in assembler.

If a variable in the watch window is not known in the current scope (i.e., function or global), then the message *Variable not active* is displayed.

The watch window is limited. An existing watched variable (other than the display format) cannot be modified. If the variables address changes, etc., delete the variable and create a new variable.

In C, select and then drag-n-drop the variable into the watch window. But as said before, C-SPY is not as easy to use when it comes to working with assembler as it is in C. However, it is possible to work to an assembler solution.

Note: Obviously, since the MSP430 peripherals are memory mapped, watching variables to watching peripherals must be able to be extended. Just be aware when the peripherals are read and written by C-SPY (this may have side effects).

5.1 Watching Registers in the Watch Window

Use the # special character to view the device registers that were referenced in the source (i.e., #PC, #SP, #Rn [if Rn was used in the source]).

5.2 Watching Variables

When debugging assembler code it is not possible to use the direct watch window directly. The variables have to be defined in the C nomenclature.

5.2.1 Method 1

Assuming that the variables are defined in RAM, say:

```
RSEG UDATA0
varword ds 2 ; two bytes per word
varchar ds 1 ; one byte per character
```

In C-SPY:

1) Open the eatch window: Window->Watch

2) Use Control->Quick Watch

3) To watch varword, first enter in the *Expression* box: (unsigned int *) #varword

3.1) To watch varchar, enter in the *Expression* box: #varbyte

4) Press the *Add Watch* button

5) Close the *Quick Watch* window

6) For the created entry in the watch window, click on the + symbol. This displays the contents (or value) of the watched variable.

6) To change the format of the displayed variable (binary, ascii, hex, int, oct), select the value and then click the right mouse button and select Properties. This brings up a window that permits the display format to change to whatever is desired. The value of the watched variable from this window can also be changed. Note, the value of the watched variable from the watch window (but it is not easy to do just the right mouse clicks) can also be changed.

5.2.2 Method 2

The variables are also defined in C and then used from the assembler as externals.

The label "?CL430_*x_xx*_L08" has to be defined and set as public in the assembler and where *x_xx* is the version number of the C compiler.

C-File:

```
unsigned int varword;   // varword      DS  2
char varbyte;           // varbyte      DS  1
```

Assembler:

```
; Define Label for C-Library
?CL430_1_24_L08
        PUBLIC  ?CL430_1_24_L08
…..
        EXTERN  varword ; define as external
        EXTERN  varbyte ; define as external
…..
        mov.b   #00011h,varbyte ; int varbyte
        mov.w   #01111h,varword ; int varword
```

This is demonstrated in Example 6.

Appendix

Example 1

Calling an assembler function from a C program without passing parameters and return values. This routine uses an assembler function to toggle port pin P1.0.

Example C Code for Calling an Assembler Function

```
/*****************************************************************/
/*  example1.c                                          2001-02-18 */
/*                                                                 */
/*    Mixing C and Assembler Code                                  */
/*                                                                 */
/* This software demonstrates how C and Assembler Code could be    */
/* mixed to get the optimum of both programming languages          */
/*                                                                 */
/* Note 1: project must include assembly file "Port1.s43"          */
/*                                                                 */
/*                                                                 */
/* Texas Instruments Incorporated                                  */
/* Stefan Schauer                                                  */
/*****************************************************************/
#include  <MSP430x14x.h>       /* Processor specific definitions */

/* -------------------external Function Prototypes ------------------- */
extern void set_port(void);     /* Function Prototype for asm function */



/*****************************************************************/
/* main */
/*****************************************************************/
void main( void )
{

// === Initialize system ================================================
  IFG1=0;                     /* clear interrupt flag1 */
  WDTCTL=WDTPW+WDTHOLD;       /* stop WD */
  P1DIR = 0x01;

  while(1)
  {
     set_port();
  }
}
// === end of main ======================================================
```

Called Assembler Function

```
; ************************************************************************
; File:     Port1.s43
; Autor:    Stefan Schauer, Texas Instruments Deutschland
; Date:     18. Feb 2001
;
; Routines to get and set the Port 1
; accessed out of C as standard extern Calls (see "Example1.c"):
;
; ************************************************************************

        #include "msp430x14x.h"       ; Processor specific definitions

        NAME        Port1

        EXTERN rand

;=========================================================================
; set_port
;=========================================================================

        PUBLIC    set_port            ; Declare symbol to be exported
        RSEG      CODE                ; Code is relocatable
set_port;
          xor.b   #01h,&P1OUT         ; Toggle 0x01 bit Port 1 output
          ret


        END
```

Example 2 Calling an Assembler Function From a C Program and Passing Parameters and a Return Value

This program reads the input of Port1 using a mask that is passed to the assembler function. Depending upon the returned value, port pin P1.0 is toggled or not.

Example C Code for Calling an Assembler Function

```
/*********************************************************************/
/*  example2.c                                          2001-02-18 */
/*                                                                   */
/*    Mixing C and Assembler Code                                    */
/*                                                                   */
/* This software demonstrates how C and Assembler Code could be      */
/* mixed to get the optimum of both programming languages            */
/*                                                                   */
/* Note 1: project must include assembly file "Port1.s43"            */
/*                                                                   */
/*                                                                   */
/* Texas Instruments Incorporated                                    */
/* Stefan Schauer                                                    */
/*********************************************************************/
#include  <MSP430x14x.h>       /* Processor specific definitions */

/* -------------------- external Function Prototypes ---------------------- */
extern char get_port(char mask);    /* Function Prototype for asm function */
extern void set_port(char mask);    /* Function Prototype for asm function */
```

```
/*********************************************************************/
/* main */
/*********************************************************************/
void main( void )
{

// === Initialize system ================================================
  IFG1=0;                     /* clear interrupt flag1 */
  WDTCTL=WDTPW+WDTHOLD;       /* stop WD */
  P1DIR = 0x01;

  while(1) /* Infinite loop*/
  {
    char value, mask;        /* Declare local variables*/
    mask = 0x80;
    value = get_port(mask);  /* Call the assembler function */
    if(value==mask)
    {
    /* Do something if value is mask */
      set_port(0x01 ^ get_port(0xFF)) ; /* Toggle value on Port 1.0 */
    }
  }
}
// === end of main ======================================================
```

Called Assembler Function

```
; *************************************************************************
; File:     Port1.s43
; Autor:    Stefan Schauer, Texas Instruments Deutschland
; Date:     18. Feb 2001
;
; Routines to get and set the Port 1
; accessed out of C as standard extern Calls(see "Example2.c"):
;
; *************************************************************************

        #include "msp430x14x.h"       ; Processor specific definitions

        NAME        Port1


;=========================================================================
; set_port
;=========================================================================

        PUBLIC    set_port            ; Declare symbol to be exported
        RSEG      CODE                ; Code is relocatable
set_port;
          mov.b   R12,&P1OUT          ; Set R12 (First parameter) to Port 1 out
          ret

;=========================================================================
; get_port
;=========================================================================

        PUBLIC    get_port            ; Declare symbol to be exported
        RSEG      CODE                ; Code is relocatable
```

```
get_port;
          mov.b   &P1IN,R13           ; Store Port 1 into R13 (unused cause of byte)
          and.b   R12,R13             ; use mask
          mov.b   R13,R12             ; Store value into R12 (return parameter)
          ret


         END
```

Example 3

In this example, the assembler function calls the standard C library function rand(). This function returns a random number. The lower byte of the returned value is written to Port1.

Example C Code for Calling an Assembler Function

```
/*********************************************************************/
/*  example3.c                                              2001-02-18 */
/*                                                                     */
/*    Mixing C and Assembler Code                                      */
/*                                                                     */
/* This software demonstrates how C and Assembler Code could be        */
/* mixed to get the optimum of both programming languages              */
/*                                                                     */
/* Note 1: project must include assembly file "port1.s43"              */
/*                                                                     */
/*                                                                     */
/* Texas Instruments Incorporated                                      */
/* Stefan Schauer                                                      */
/*********************************************************************/
#include  <MSP430x14x.h>       /* Processor specific definitions */

/* -------------------external Function Prototypes ------------------- */
extern void set_port_rand(void);/* Function Prototype for asm function */


/*********************************************************************/
/* main */
/*********************************************************************/
void main( void )
{

// === Initialize system ================================================
  IFG1=0;                     /* clear interrupt flag1 */
  WDTCTL= WDTPW+WDTHOLD;      /* stop WD */
  P1DIR = 0xFF;               /* all ports of P1.x are output*/

  while(1) /* Infinite loop*/
  {
      set_port_rand() ;       /* Move rand() value to port 1 */
  }
}
// === end of main ======================================================


/*********************************************************************/
/* mult */
/*********************************************************************/
unsigned long mult(unsigned int x , unsigned int y)
```

```
{
  return ( x * y);           /* multiply x * y */
}

// === end of mult ====================================================
```

Assembler Function That Calls a C Function

```
; *************************************************************************
; File:     Port1.s43
; Autor:    Stefan Schauer, Texas Instruments Deutschland
; Date:     18. Feb 2001
;
; Routines to get and set the Port 1
; accessed out of C as standard extern Calls(see "Example3.c"):
;
; *************************************************************************

#include "msp430x14x.h"       ; Processor specific definitions

        NAME    Port1

        EXTERN  rand          ; Std C function

        EXTERN  mult          ; function in example3.c


;=========================================================================
; set_port_rand
;=========================================================================

        PUBLIC  set_port_rand         ; Declare symbol to be exported
        RSEG    CODE                  ; Code is relocatable
set_port_rand;
          call    #rand               ; Call rand() -> value stored at R12

          mov     #25,R14             ; 2nd operand -> R14
                                      ; 1st operand -> R12 from rand() function
          call    #mult               ; return in R12 / R13
          mov.b   R12,&P1OUT          ; Move low byte of R12 to Port 1 output
          ret


        END
```

Example 4

In this example, the interrupt service routine for the watchdog timer is handled by an assembler function. Interrupt functions cannot have parameter and return values. Because an interrupt can occur anywhere in the program execution, any registers used have to be saved and restored (typically on the stack).

Example C Code for Using Assembler Interrupt Service Routine

```
/*****************************************************************/
/*  example4.c                                          2001-02-18 */
/*                                                                 */
/*    Mixing C and Assembler Code                                  */
/*                                                                 */
/* This software demonstrates how C and Assembler Code could be    */
/* mixed to get the optimum of both programming languages          */
/*                                                                 */
/* Note 1: project must include assembly file " wdt_int.s43"       */
/*                                                                 */
/*                                                                 */
/* Texas Instruments Incorporated                                  */
/* Stefan Schauer                                                  */
/*****************************************************************/
#include  <MSP430x14x.h>            /* Processor specific definitions */


/*****************************************************************/
/* main */
/*****************************************************************/
void main( void )
{

// === Initialize system ================================================
  IFG1=0;                     /* clear interrupt flag1 */
  WDTCTL=WDT_MDLY_32;         /* WDT 32ms Interval Timer */
  P1DIR = 0x01;               /* P1.0 is output */

  IFG1 &= ~WDTIFG;            /* Clear pending WDT interrupts */
  IE1  |= WDTIE;              /* Enable WDT Interrupt */

  _EINT();

  while(1)
  {
  }
}
// === end of main ======================================================
```

Assembler Interrupt Service Routine

```
; ***************************************************************************
; File:     wdt_int.s43
; Autor:    Stefan Schauer, Texas Instruments Deutschland
; Date:     18. Feb 2001
;
; Routines to handle Watchdog Timer interrupt service
; accessed out of C as interrupt routine (see "Example4.c"):
;
; ***************************************************************************
```

```
        NAME        WDT_ISR

        #include "msp430x14x.h"       ; Processor specific definitions

;-------------------------------------------------------------------------
;=========================================================================
; WDT_isr        Watchdog Interrupt service
;=========================================================================
        PUBLIC  wdt_isr               ; Declare symbol to be exported
        RSEG    CODE                  ; Code is relocatable
wdt_isr
             xor.b   #001h,&P1OUT     ; Toggle P1.0
             reti

;=========================================================================
        COMMON  INTVEC(1)               ; Interrupt vectors
;=========================================================================

             ORG     WDT_VECTOR
             DW      wdt_isr


        END
```

Example 5

This example shows how the interrupt service routine for the Timer A, Timer B, and ADC12 is handled effectively within C Code.

Example C Code for Using Special Interrupt Handler

```
/*************************************************************************/
/*  example5.c                                                2001-02-18 */
/*                                                                       */
/*    Mixing C and Assembler Code                                        */
/*                                                                       */
/* This software demonstrates how C and Assembler Code could be          */
/* mixed to get the optimum of both programming languages                */
/*                                                                       */
/* Note 1: project must include assembly file "ta_int.s43"               */
/*                                                                       */
/*                                                                       */
/* Texas Instruments Incorporated                                        */
/* Stefan Schauer                                                        */
/*************************************************************************/
#include  <MSP430x14x.h>             /* Processor specific definitions */

int Count;

/*************************************************************************/
/* main */
/*************************************************************************/
void main( void )
{

// === Initialize system ================================================
  IFG1=0;                    /* clear interrupt flag1 */
```

```
  WDTCTL= WDTPW+WDTHOLD;     /* stop WD */
  P1DIR = 0x01;              /* P1.0 is output */

  TACTL =  TASSEL_2+ TACLR;  /* CLK = SMCLK ; clear counter*/
  CCR1  =  0x4000;           /* Capture value for CCR1 */
  CCTL1 =  CCIE;             /* enable CCR1 int */

  TACTL |= MC_2+ TAIE;       /* start cont. up ; enable TA int. */

  _EINT();

  while(1)
  {
  }
}
// === end of main ============================================================

/*****************************************************************************/
/* TIMOVH_C */
/*****************************************************************************/
interrupt void TIMOVH_C( void )
{
Count ++;
}
// === end of TIMOVH_C ========================================================


/*****************************************************************************/
/* TIMMOD1_C */
/*****************************************************************************/
interrupt void TIMMOD1_C( void )
{
P1OUT ^= 0x01;               /* Toggle P1.0 */
}
// === end of TIMMOD1_C =======================================================
```

Assembler Interrupt Handler

```
; ***************************************************************************
; File:     ta_int.s43
; Autor:    Stefan Schauer, Texas Instruments Deutschland
; Date:     18. Feb 2001
;
; Routines to handle Timer A interrupt service
; accessed out of C as interrupt routine (see "Example5.h"):
;
; ***************************************************************************


        NAME        TA_ISR

        #include "msp430x14x.h"       ; Processor specific definitions

; ------------------- external Function Prototypes --------------------
        EXTERN TIMOVH_C
        EXTERN TIMMOD1_C

;============================================================================
; ta_isr        Watchdog Interrupt service
;============================================================================
```

```
        PUBLIC  ta_isr                  ; Declare symbol to be exported
        RSEG    CODE                    ; Code is relocatable

; Interrupt handler for Capture/Compare Modules 1 to 4.
; The interrupt flags CCIFGx and TAIFG are reset by
; hardware. Only the flag with the highest priority
; responsible for the interrupt vector word is reset.
ta_isr                                  ; Interrupt latency
             ADD &TAIV,PC               ; Add offset to Jump table
             RETI                       ; Vector 0: No interrupt

             JMP TIMMOD1_C              ; Vector 2: Module 1
             JMP TIMMOD2                ; Vector 4: Module 2
             JMP TIMMOD3                ; Vector 6: Module 3
             JMP TIMMOD4                ; Vector 8: Module 4

        ; Module 5. Timer Overflow Handler
        ; fall through

TIMOVH  ; Vector 10: TIMOV Flag
             JMP TIMOVH_C               ; Handle Timer Overflow in C
             ; RETI ; is handled by the C function


TIMMOD1 ; Vector 2: Module 1
        ; If JMP could not be executed through the limited jump width
        ; first jump to this position and then branch to the C function
             BR #TIMMOD1_C              ; Handle CCR1 interrupt in C
             ; RETI ; is handled by the C function

TIMMOD2
        ; If all five CCR registers are not implemented on a
        ; device, the interrupt vectors for the register that are
        ; present must still be handled.

TIMMOD3
TIMMOD4
             RETI                       ; Simply return

;------------------------------------------------------------------------------
        COMMON  INTVEC(1)               ; Interrupt vectors
;------------------------------------------------------------------------------

             ORG     TIMERA1_VECTOR     ; Timer A CC1-2, TA
             DW      ta_isr

        END
```

Example 6

This example shows how the assembler uses variables, which are defined in C so that the watch window is much easier to use.

Example C Code to Define Variables for the Assembler

```
/* This file only contains definitions of the variables
   which are used by the assembler Program */

//             RSEG     UDATA0
```

```
unsigned int Varword1;  // Varword1      DS  2
unsigned int Varword2;  // Varword2      DS  2
unsigned int Varword3;  // Varword3      DS  2


char Varbyte1;          // Varbyte1      DS  1
char Varbyte2;          // Varbyte2      DS  1
char Varbyte3;          // Varbyte3      DS  1
```

Example Assembler Code That Uses In C Defined Variables

```
#include  "msp430x11x1.h"

;-------------------------------------------------------------------------------
;    MSP430F1121 FET Demonstration Program - Software Wait
;
;    Description; This program will toggle P1.0. A software Wait is used, based
;    simply upon decrementing R15. Default Basic Clock settings.
;
;           MSP430F1121
;         _________________
;     /|\|              XIN|-
;      | |                 |
;      --|RST          XOUT|-
;        |                 |
;        |             P1.0|-->LED
;
;    Texas Instruments, Inc
;    November 1999
;-------------------------------------------------------------------------------
?CL430_1_24_L08
        PUBLIC  ?CL430_1_24_L08


             RSEG    CSTACK                  ; System stack
             DS      0


;             RSEG     UDATA0

        EXTERN  Varword1 ;
        EXTERN  Varword2 ;
        EXTERN  Varword3 ;
        EXTERN  Varbyte1 ;
        EXTERN  Varbyte2 ;
        EXTERN  Varbyte3 ;


;------------------------------------------------------------------------------
             RSEG    CODE                    ; Program code
Reset        mov     #SFE(CSTACK),SP         ; Initialize stackpointer
SetupWDT     mov     #WDTPW+WDTHOLD,&WDTCTL  ; Stop WDT

             mov.b   #00011h,Varbyte1
             mov     #01111h,Varword1
             mov.b   #00012h,Varbyte2
             mov     #01112h,Varword2
             mov.b   #00013h,Varbyte3
             mov     #01113h,Varword3

SetupP1      bis.b   #001h,&P1DIR            ; P1.0 output
```

```
Mainloop     xor.b   #001h,&P1OUT            ; Toggle P1.0
             mov     #065000,R15             ; Delay to R15
L1           dec     R15                     ; Decrement R15
             jnz     L1                      ; Delay over?
             jmp     Mainloop                ; Again

;-------------------------------------------------------------------------
             COMMON  INTVEC                  ; Interrupt vectors

             ORG     RESET_VECTOR
             DW      Reset

;-------------------------------------------------------------------------
             END
```

References

1. *IAR MSP430 C Compiler* Programming Guide
2. *IAR MSP430 Assembler, Linker and Librarian* Programming Guide
3. *MSP430x3xx Family User's Guide,* literature number SLAU012
4. *MSP430x1xx Family User's Guide,* literature number SLAU049
5. *MSP430x4xx Family User's Guide,* literature number SLAU056

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265