MSP430X port - small memory model version

Jean-Luc Béchennec, Mikaël Briday

October 25, 2019

This document describes the small version for the CPUX of the Trampoline port on MSP430, which assumes that the code is hosted in the first 64kB of memory and therefore the addresses are stored on 16-bit words. Instruction set of the MSP430X is available in [4] or [3].

1 Multitasking

1.1 ABI

In [2] a change has been made in GCC so that it conforms to the ABI defined in [1] and becomes compatible with the proprietary Texas Instruments compiler. So there are two GCC compilers for MSP430: the one that does not conform to the ABI defined by Texas Instruments, *MSPGCC*, and the one that does conform to the ABI, *GCC compiler for MSP*.

As it is difficult to support both ABIs simultaneously, it was decided to support both ABIs at compile time. A precompiled *MSPGCC* is available in the latest version of Energia[1]. Energia can be downloaded at `https://energia.nu`. A precompiled *GCC compiler for MSP* is available at `http://www.ti.com/tool/msp430-gcc-opensource`.

In both ABIs the registers used to pass arguments to functions are `r12`, `r13`, `r14` and `r15`. In the ABI of *MSPGCC*, `r15` is the first argument, `r14` the second and so on. If a function returns a value, it is placed in `r15`. In the ABI of *GCC compiler for MSP* `r12` is the first argument, `r13` the second and so on. If a function returns a value, it is placed in `r12`. No Trampoline service uses more than 3 arguments and therefore `r12`, for *MSPGCC* ABI, or `r15`, for *GCC compiler for MSP* ABI, is available to pass the service ID into the wrapper.

Adapting to both ABIs at compile time is not very complicated. This involves exchanging the use made of the registers `r12`, identifying the service, and `r15`, the return value of the service and the argument of `tpl_run_elected`. This can be done by defining an abstract register to pass the service identifier and an abstract register to return the return value of the service. The register selection can be made using the preprocessor and the macro

[1] GCC 4.6.3.

`__GXX_ABI_VERSION` as shown at Figure 1. This macro is `1002` for *MSPGCC* and `1011` for *GCC compiler for MSP*. 2 abstract registers are defined: `REG_SID` which is `r12` in *MSPGCC* ABI and `r15` in *GCC compiler for MSP* ABI, and `REG_RETARG` which is `r15` in *MSPGCC* ABI and `r12` in *GCC compiler for MSP* ABI.

Figure 1: ABI selection with C preprocessor macros

```
#if __GXX_ABI_VERSION == 1002
/* MSPGCC ABI */
#define MSPGCC_ABI
#define REG_SID r12
#define REG_RETARG r15
#define REG_RETARG_OFFSET 8
#elif __GXX_ABI_VERSION == 1011
/* GCC compiler for MSP ABI */
#define GCCFORMSP_ABI
#define REG_SID r15
#define REG_RETARG r12
#define REG_RETARG_OFFSET 2
#else
#error "Unsupported ABI"
#endif
```

The following table summarizes the use of the registers in both ABIs if we consider all arguments are small enough to be stored in one register. Although `r11` is volatile in one of them, for simplification purposes later on, `r11` is considered as non-volatile. A preserved register is noted P and a Volatile register is noted V.

Register	*MSPGCC*	*GCC compiler for MSP*
`r0`	Program Counter, saved on stack by cpu	
`r1`	Stack Pointer	
`r2`	Status Register	
`r3`	Constants Generator	
`r4-r10`	Not preserved by the callee	
`r11`	V	P
`r12`	V, argument 4	V, argument 1, return value
`r13`	V, argument 3	V, argument 2
`r14`	V, argument 2	V, argument 3
`r15`	V, argument 1, return value	V, argument 4

It can be noted that the arguments being passed through the low weight 16 bits of the registers, except perhaps for the far pointers, the arguments of the Trampoline services must fit on 16 bits. This limits the tick argument of the services related to alarms to 16 bits.

1.2 Stack

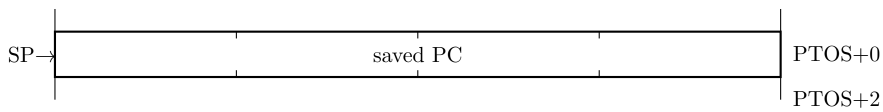
A service call is done using the `br` instruction in the service call wrapper to prevent 2 nested call and fold the `ret` instruction. The service identifier is passed to the service call handler through the `REG_SID` register. So a service call wrapper is as shown in listing at figure 2.

Figure 2: Service wrapper

```
mov   #<service_id>, REG_SID /* put the service id in the ad-hoc reg */
br    #tpl_sc_handler        /* branch to the service call handler   */
```

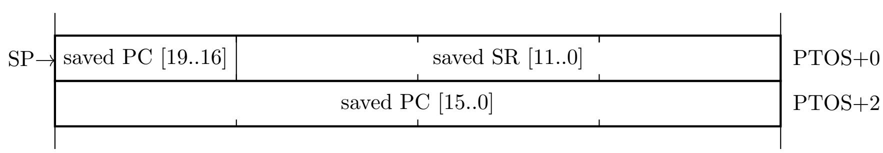
When in the `tpl_sc_handler` the stack is as shown at figure 3[2]. `PTOS` stands for *Process Top Of Stack*.

Figure 3: Stack at beginning of `tpl_sc_handler`



When an interrupt is taken into account, the `PC` and the `SR` are pushed on the stack. To save space, the `SR` is stored in the same 16-bit word as bits 19..16 of `PC`. For an obscure reason, words are in reverse order and bits 19..16 of `PC` are in high bits. Since all the code is in the first 64kb of the memory, bits 19 to 16 of the PC are always 0. The stack is shown at figure 4.

Figure 4: Stack in an interrupt handler



Preemption cases

A preemption can be synchronous or asynchronous. A synchronous preemption (SP) happens when a service call is done, for instance when a task activates a higher priority task. An asynchronous preemption (AP) happens under interrupt, for instance when a higher

[2]stacks are drawn with the lower address up so they are growing upward, not downward. Each stack location is a 16 bits word.

priority task is activated by an alarm. A preempted task may resume its execution following a synchronous event (SR) : the running task calls `TerminateTask`, `ChainTask`, `WaitEvent` or `SetEvent` or following an asynchronous event (AR) : an alarm does a `SetEvent`. So there are 4 cases:.

SPSR Synchronous Preemption, Synchronous Resume. τ_1 is running, τ_2 is ready. $P(\tau_1) > P(\tau_2)$. τ_1 calls `WaitEvent` and is preempted synchronously, τ_2 becomes running and calls `SetEvent`. τ_2 is preempted and τ_1 is resumed synchronously.

SPAR Synchronous Preemption, Asynchronous Resume. τ_1 calls `WaitEvent` ans is synchronously preempted, An alarm does a `SetEvent` on τ_1 which is asynchronously resumed.

APSR Asynchronous Preemption, Synchronous Resume. τ_1 is running, τ_2 is suspended. $P(\tau_1) < P(\tau_2)$. An alarm activates τ_2, τ_1 is asynchronously preempted, τ_2 calls `TerminateTask`, τ_1 is synchronously resumed.

APAR Asynchronous Preemption, Asynchronous Resume. τ_1 is running, τ_2 is suspended. $P(\tau_1) < P(\tau_2)$. An alarm activates τ_2, τ_1 is asynchronously preempted. τ_2 is terminated by the OS because of protection fault, for instance a timing protection interrupt and τ_1 is asynchronously resumed.

So *the stack frame has to be normalized.* The normalized stack frame is the asynchronous one shown at figure 4 because it contains the Status Register. Normalization is done at the beginning of the `tpl_sc_handler`. The end of the `tpl_sc_handler`is done using the `reti` instruction, as at the end of an interrupt.

The normalized stack frame may be done only when a context is saved to prevent a normalization if there is no context switch. However, the load of the context is much complicated, as the restauration of `r2` (aka status register) in the `tpl_sc_handler`re-enable the interrupts before the end of the function.

1.3 The tpl_sc_handler

The background color of the code snippets depends on the current active stack:

green process stack

red kernel stack

yellow either kernel or process stack

The first thing to do is to compare the service id to the number of services to verify its validity.

```
    cmp   #SYSCALL_COUNT, REG_SID
    jlo   tpl_sc_handler_id_ok                /* continue if lower        */
    ret                                       /* return if higher or same */
tpl_sc_handler_id_ok:
```

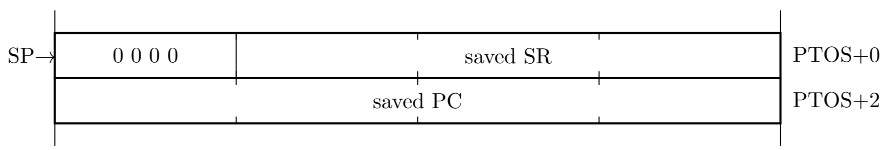
Disable interrupts so that the kernel cannot be interrupted. Check the reentrancy flag. If it is not zero, it means the service is called from a hook and has to be processed differently.

```
    dint
    tst.b &tpl_reentrancy_flag
    jnz   tpl_sc_handler_from_hook
```

We need to have the same stack pattern for both the `tpl_sc_handler` and an interrupt handler which calls the operating system. So we push the SR and we reset the 4 higher bits (high weight of PC, not sure it is needed) and set GIE in the saved SR.

```
    push         sr
    bic.b        #0xF0, 1(sp)     /* reset the 4 higher bits of saved SR */
    bis.b        #0x08, 0(sp)     /* set the GIE bit in the saved SR     */
```

The stack is then as follow:



Obviously volatile registers (r12 to r15 because we take into account both ABIs) are not saved in `tpl_sc_handler` since the caller does not expect their values to be preserved but we need to make room (8 bytes) on the stack for them because an interrupt handler will save these registers at this location. However register names appear in figures but are in italic. Either *r12* if MSPGCC ABI is used or *r15* if GCC for MSP ABI is used is for the `REG_RETARG` which is not saved yet.

```
    sub  #8, sp
```

The `tpl_sc_handler` needs one working register and we choose to use `r11` which has to be saved on the process stack before using it.

```
    push r11
```

At that stage the stack is shown in figure 5.

Before calling the service, we setup the kernel stack. The process stack pointer (PSP) is saved in r11, then SP is loaded to the kernel stack bottom and the PSP is saved on the kernel stack.

Figure 5: Stack shape before calling the service

SP→	saved r11	PTOS+0
	saved r12 - will hold REG_RETARG in GCC for MSP ABI	PTOS+2
	saved r13	PTOS+4
	saved r14	PTOS+6
	saved r15 - will hold REG_RETARG in MSPGCC ABI	PTOS+8
	0 0 0 0 | saved SR	PTOS+10
	saved PC	PTOS+12

```
mov   r1,r11
mov   #tpl_kern_stack_bottom, r1
push  r11
```

The kernel stack is as follow (`KTOS` stands for *Kernel Top Of Stack*):

SP→	saved PSP	KTOS+0

Init the `NEED_SWITCH/SAVE` in `tpl_kern`.

```
mov   #tpl_kern, r11
mov.b #NO_NEED_SWITCH_NOR_SCHEDULE, TPL_KERN_OFFSET_NEED_SWITCH(r11)
mov.b #NO_NEED_SWITCH_NOR_SCHEDULE, TPL_KERN_OFFSET_NEED_SCHEDULE(r11)
```

Call the service. The reentrancy flag is incremented before and decremented after.

```
inc.b &tpl_reentrancy_flag           /* surround the call by inc ...  */
rla   REG_SID                        /* index -> offset               */
call  tpl_dispatch_table(REG_SID)
dec.b &tpl_reentrancy_flag           /* ... and dec of the flag.      */
```

From there, `REG_RETARG` holds the return value. It is put at its location in the process stack. Also `r13` and `r14` become usable whatever is the ABI.

```
pop  r13                                /* get back PSP => r13.      */
mov  REG_RETARG, REG_RETARG_OFFSET(r13) /* put in Process's stack    */
```

Check the context switch condition in `tpl_kern`.

```
    mov     #tpl_kern, r11
    tst.b   TPL_KERN_OFFSET_NEED_SWITCH(r11)
    jz      tpl_sc_handler_no_context_switch
```

1.3.1 Branch of context switching

Prepare the call to `tpl_run_elected` by setting `REG_RETARG` to 0, aka no save.

```
    mov     #0, REG_RETARG
```

Test the `NEED_SAVE` condition.

```
    bit.b   #NEED_SAVE, TPL_KERN_OFFSET_NEED_SWITCH(r11)
    jz      tpl_sc_handler_no_save_running_context
```

Save the context. The MSP430 have a "push multiple words", but no "move multiple word". So, we get back to process stack to benefit this instruction

```
    mov     r1, r14     /* get a copy of the KSP to restore it later */
    mov     r13, r1     /* change stack to process stack */
    pushm.w #7, r10     /* Push r4 to r10 on process stack (save) */
```

The whole context is now saved on process stack and the kernel stack has been cleaned. The saved context structure is shown at figure 6.

Now the stack pointer is saved in the dedicated location.

```
    mov     &tpl_kern, r11  /* Get the s_running slot of tpl_kern in r11 */
    mov     @r11, r11       /* Get the pointer to the context (SP alone) */
    mov     r1, @r11        /* Save the stack pointer                    */
```

Prepare the argument of `tpl_run_elected` : 1 (aka save) and call it after switching back to the kernel stack.

```
    mov     r14, r1     /* get back to kernel stack */
    mov     #1, REG_RETARG
tpl_sc_handler_no_save_running_context:
    call    tpl_run_elected
```

`tpl_run_elected` has copied the elected process slot of `tpl_kern` to the `running` slot. We load the stack pointer of the new running process.

```
    mov     &tpl_kern, r11  /* Get the s_running slot of tpl_kern in r11 */
    mov     @r11, r11       /* Get the pointer to the context (SP alone) */

    mov     @r11, r1        /* Get the stack pointer                     */
```

Now, the context of the new running process is loaded. At start it has the same pattern as the one shown at figure 6. Registers `r4` to `r15` are popped and we return.

Figure 6: Context saved on stack

SP→	saved r4	PTOS+0
	saved r5	PTOS+2
	saved r6	PTOS+4
	saved r7	PTOS+6
	saved r8	PTOS+8
	saved r9	PTOS+10
	saved r10	PTOS+12
	saved r11	PTOS+14
	saved r12 - REG_RETARG in GCC for MSP ABI	PTOS+16
	saved r13	PTOS+18
	saved r14	PTOS+20
	saved r15 - will hold REG_RETARG in MSPGCC ABI	PTOS+22
	0 0 0 0	saved SR	PTOS+24
	saved PC	PTOS+26

```
    popm.w #12,r15       /* Pop r4 to r15 at once              */
    reti                 /* and return with interrupts enabled */
```

1.3.2 Branch of No context switching

In case of no context switch, we have to get to the process stack, stored in r13

```
tpl_sc_handler_no_context_switch:
    mov  r13, r1         /* get back to process stack */
```

Here we have the stack shaped as shown at figure 5. `REG_RETARG` is restored, `r11` is restored, the stack is cleaned and we return. Interrupts are enabled at that time.

```
    mov  REG_RETARG_OFFSET(r1), REG_RETARG /* get back REG_RETARG     */
    pop  r11                               /* get back r11            */
    add  #8, r1                            /* clean the stack         */
    reti                                   /* return with int enabled */
```

1.3.3 Branch when the sc handler is called from hook

Here we are on the kernel stack already and the `pc` has been pushed on the stack by the `call`. `REG_SID` contains the identifier of the service and the 3 other registers contain the arguments if any. We do not need to do complicated stuff here because we have no context switch to do. We only call the service then return and that's it.

```
tpl_sc_handler_from_hook:
    rla   REG_SID                        /* index -> offset              */
    call  tpl_dispatch_table(REG_SID)
    ret
```

1.4 Context initialisation

The context that shoud be set during the task's initialisation (`tpl_init_context`) is the one of the figure 6, but with a call to either `CallTerminateTask` or `CallTerminateISR2` as return address of the task/ISR2 function, depending of the type of the process to init.

Figure 7: Context initialization

SP→	r4		PTOS+0
	r5		PTOS+2
	r6		PTOS+4
	r7		PTOS+6
	r8		PTOS+8
	r9		PTOS+10
	r10		PTOS+12
	r11		PTOS+14
	r12 - REG_RETARG in GCC for MSP ABI		PTOS+16
	r13		PTOS+18
	r14		PTOS+20
	r15 - REG_RETARG in MSPGCC ABI		PTOS+22
	0 0 0 0	SR	PTOS+24
	PC		PTOS+26
	CallTerminateTask/CallTerminateISR2		PTOS+28

Beside that, registers from `r4` to `r15` may be initialized to 0 or left uninitialized to save both execution time and energy consumption. `PC` has to be initialized to the address of the task/ISR2 function. `SR` has to be initialized with:

- `V` at 0
- `SCG1`, `SCG0`, `OSCOFF` and `CPUOFF` control the low power mode and are all at 0. This correspond to the Active Mode.
- `GIE` at 1 so interrupts are enabled when the task runs.
- `N`, `Z` and `C` at 0.

So the initialization value of `SR` is `0x0008`.

2 Interrupt Handlers

Interrupt handlers are generated from the OIL description. There are 3 categories of interrupt handlers in Trampoline which are handlers that link an interrupt vector to:

- the increment of one or more counters
- the execution of a category 1 ISR
- the execution of a category 2 ISR

The incrementation of a counter or the execution of a category 2 ISR involves an interaction with the OS with possible rescheduling and context switch, while the execution of a category 1 ISR does not involve an interaction with the OS.

For ISR 1 the interrupt handler will only backup the volatile registers, call the function implementing ISR 1 and restore the volatile registers. For ISR2 and counters, the handler will be similar to the one of the service call.

In addition, the interrupt vectors related to the GPIO ports, one vector for each port, are shared among the I/O pins of the port.

Interrupt vectors are defined in `templates/config/msp430x/small/msp430fr5969/config.oil` file. The following vectors are available and can be used as `SOURCE` attribute is `ISR` and `COUNTER` objects:

- `AES256_VECTOR`,
- `RTC_VECTOR`,
- `PORT4_VECTOR`,
- `PORT3_VECTOR`,
- `TIMER3_A1_VECTOR`,
- `TIMER3_A0_VECTOR`,
- `PORT2_VECTOR`,
- `TIMER2_A1_VECTOR`,

- TIMER2_A0_VECTOR,
- PORT1_VECTOR,
- TIMER1_A1_VECTOR,
- TIMER1_A0_VECTOR,
- DMA_VECTOR,
- USCI_A1_VECTOR,
- TIMER0_A1_VECTOR,
- TIMER0_A0_VECTOR,
- ADC12_VECTOR,
- USCI_B0_VECTOR,
- USCI_A0_VECTOR,
- WDT_VECTOR,
- TIMER0_B1_VECTOR,
- TIMER0_B0_VECTOR,
- COMP_E_VECTOR,
- UNMI_VECTOR

Several `ISR` or `COUNTER` objects cannot share the same `SOURCE`.

The `SystemCounter` uses the `TIMER3_A0_VECTOR` and is defined as follow in `templates/config/msp430x/small/msp430fr5969/config.oil` file:

```
COUNTER SystemCounter {
  SOURCE = TIMER3_A0_VECTOR;
};
```

When a `PORTx_VECTOR` source is used, a `BIT` sub-attribute can be added to select which bit is used as interrupt source. In this case several `ISR` or `COUNTER` may share the same vector but shall be of the same type. In other words 2 counters may share the same port vector, each on its bit or 2 ISR 1or 2 ISR 2 but you can't have a counter and an ISR sharing the same port vector or an ISR 1 and an ISR 2.

Examples can be found in `examples/msp430x/small/msp430fr5969/launchpad`. In `readbutton_isr1`, an ISR 1 is linked to button S1 which is connected to bit `5` of `PORT4`:

```
ISR buttonS1 {
  CATEGORY = 1;
  PRIORITY = 1;
  SOURCE = PORT4_VECTOR {
    BIT = 5;
  }; /* Button S1 is on GPIO port 4, bit 5 */
};
```

`readbutton_isr2` is the same example but with an ISR 2 instead of the ISR 1.

2.1 Vector table generation

The OIL compiler generates the vector table according to what `SOURCE` are used in the OIL file. For instance here is the vector table generated for `readbutton_isr1` example:

```
__attribute__ ((section(".isr_vector")))
CONST(tpl_it_handler, AUTOMATIC) tpl_it_vectors[26] = {

  /* 0xFFCC, AES256_VECTOR    */ (tpl_it_handler)tpl_null_it,
  /* 0xFFCE, RTC_VECTOR       */ (tpl_it_handler)tpl_null_it,
  /* 0xFFD0, PORT4_VECTOR     */ (tpl_it_handler)tpl_direct_irq_handler_PORT4_VECTOR,
  /* 0xFFD2, PORT3_VECTOR     */ (tpl_it_handler)tpl_null_it,
  /* 0xFFD4, TIMER3_A1_VECTOR */ (tpl_it_handler)tpl_null_it,
  /* 0xFFD6, TIMER3_A0_VECTOR */ (tpl_it_handler)tpl_primary_irq_handler_TIMER3_A0_VECTOR,
  /* 0xFFD8, PORT2_VECTOR     */ (tpl_it_handler)tpl_null_it,
  /* 0xFFDA, TIMER2_A1_VECTOR */ (tpl_it_handler)tpl_null_it,
  /* 0xFFDC, TIMER2_A0_VECTOR */ (tpl_it_handler)tpl_null_it,
  /* 0xFFDE, PORT1_VECTOR     */ (tpl_it_handler)tpl_null_it,
  /* 0xFFE0, TIMER1_A1_VECTOR */ (tpl_it_handler)tpl_null_it,
  /* 0xFFE2, TIMER1_A0_VECTOR */ (tpl_it_handler)tpl_null_it,
  /* 0xFFE4, DMA_VECTOR       */ (tpl_it_handler)tpl_null_it,
  /* 0xFFE6, USCI_A1_VECTOR   */ (tpl_it_handler)tpl_null_it,
  /* 0xFFE8, TIMER0_A1_VECTOR */ (tpl_it_handler)tpl_null_it,
  /* 0xFFEA, TIMER0_A0_VECTOR */ (tpl_it_handler)tpl_null_it,
  /* 0xFFEC, ADC12_VECTOR     */ (tpl_it_handler)tpl_null_it,
  /* 0xFFEE, USCI_B0_VECTOR   */ (tpl_it_handler)tpl_null_it,
  /* 0xFFF0, USCI_A0_VECTOR   */ (tpl_it_handler)tpl_null_it,
  /* 0xFFF2, WDT_VECTOR       */ (tpl_it_handler)tpl_null_it,
  /* 0xFFF4, TIMER0_B1_VECTOR */ (tpl_it_handler)tpl_null_it,
  /* 0xFFF6, TIMER0_B0_VECTOR */ (tpl_it_handler)tpl_null_it,
  /* 0xFFF8, COMP_E_VECTOR    */ (tpl_it_handler)tpl_null_it,
  /* 0xFFFA, UNMI_VECTOR      */ (tpl_it_handler)tpl_null_it,
  /* 0xFFFC, SYSNMI_VECTOR    */ (tpl_it_handler)tpl_MPU_violation,
  /* 0xFFFE, RESET_VECTOR     */ (tpl_it_handler)tpl_reset_handler
};
```

Obviously the last 2 vectors, `SYSNMI_VECTOR` and `RESET_VECTOR`, are not usable by the application and are reserved to Trampoline.

2.2 ISR 1 interrupt handler

An ISR 1 handler has a name formed from the concatenation of `tpl_direct_irq_handler_` and the name of the source. For instance an ISR 1 handler for the `PORT4_VECTOR` has the name `tpl_direct_irq_handler_PORT4_VECTOR`.

When entering the ISR, the stack is as shown at figure 4 and `PC` (`r0`) and `SR` (`r2`) have been saved. Before doing anything we have to save the volatile registers, which are `r11`[3] to `r15`.

```
tpl_direct_irq_handler_PORT4_VECTOR:
  pushm.w #5, r15  /* Push r11, r12, r13, r14 and r15 */
```

As a result the stack is as follow:

[3] r11 is not volatile in the *MSPGCC* ABI but is volatile in *GCC compiler for MSP* ABI. Anyway, in order to limit variabilility, `r11` is saved for both ABIs.

SP→	saved r11	PTOS+0
	saved r12	PTOS+2
	saved r13	PTOS+4
	saved r14	PTOS+6
	saved r15	PTOS+8
	saved PC [19..16] | saved SR [11..0]	PTOS+10
	saved PC [15..0]	PTOS+12

If the vector is not a port vector, the code is straightforward.

```
  call      #buttonS1_function
```

If the vector is a port vector but no bit is specified, the ack of the interrupt is added.

```
  call      #buttonS1_function
  mov       #0,__P4IV
```

If the vector is a port vector and a bit is specified, the generated code follows the Texas Instruments recommendations as outlined in section 12.2.6.1 of [3].

```
  add       &__P4IV, pc
  jmp       tpl_direct_irq_handler_exit_PORT4_VECTOR
  jmp       tpl_direct_irq_handler_exit_PORT4_VECTOR      /* bit 0 */
  jmp       tpl_direct_irq_handler_exit_PORT4_VECTOR      /* bit 1 */
  jmp       tpl_direct_irq_handler_exit_PORT4_VECTOR      /* bit 2 */
  jmp       tpl_direct_irq_handler_exit_PORT4_VECTOR      /* bit 3 */
  jmp       tpl_direct_irq_handler_exit_PORT4_VECTOR      /* bit 4 */
  jmp       tpl_p4_5_handler                              /* bit 5 */
  jmp       tpl_direct_irq_handler_exit_PORT4_VECTOR      /* bit 6 */
  jmp       tpl_direct_irq_handler_exit_PORT4_VECTOR      /* bit 7 */
tpl_p4_5_handler:
  call      #buttonS1_function
tpl_direct_irq_handler_exit_PORT4_VECTOR:
```

Then the volatile registers are restored and we return.

```
  popm.w    #5, r15
  reti
```

2.3 ISR 2 interrupt handler

An ISR 2 handler has a name formed from the concatenation of `tpl_primary_irq_handler_` and the name of the source. For instance an ISR 2 handler for the `PORT4_VECTOR` has the name `tpl_primary_irq_handler_PORT4_VECTOR`.

When entering the ISR, the stack is as shown at figure 4 and `PC` (`r0`) and `SR` (`r2`) have been saved. Before doing anything we have to save the volatile registers, which are `r11` to `r15`.

```
tpl_primary_irq_handler_PORT4_VECTOR:
  pushm.w #5, r15  /* Push r11, r12, r13, r14 and r15 */
```

Then we switch to the kernel stack and init `tpl_kern`.

```
  mov       r1, r11                           /* Copy the PSP in r11      */
  mov       #tpl_kern_stack + TPL_KERNEL_STACK_SIZE, r1 /* kernel stack */
  push      r11                               /* Save PSP to kernel stack */
  mov       #tpl_kern, r11
  mov.b     #NO_NEED_SWITCH_NOR_SCHEDULE, TPL_KERN_OFFSET_NEED_SWITCH(r11)
  mov.b     #NO_NEED_SWITCH_NOR_SCHEDULE, TPL_KERN_OFFSET_NEED_SCHEDULE(r11)
```

Activate the ISR 2. Here the `#1` in `mov #1, REG_RETARG` is the identifier of the ISR 2. Only the most complex generated code is shown.

```
  add       &__P4IV, pc
  jmp       tpl_direct_irq_handler_exit_PORT4_VECTOR
  jmp       tpl_direct_irq_handler_exit_PORT4_VECTOR       /* bit 0 */
  jmp       tpl_direct_irq_handler_exit_PORT4_VECTOR       /* bit 1 */
  jmp       tpl_direct_irq_handler_exit_PORT4_VECTOR       /* bit 2 */
  jmp       tpl_direct_irq_handler_exit_PORT4_VECTOR       /* bit 3 */
  jmp       tpl_direct_irq_handler_exit_PORT4_VECTOR       /* bit 4 */
  jmp       tpl_p4_5_handler                               /* bit 5 */
  jmp       tpl_direct_irq_handler_exit_PORT4_VECTOR       /* bit 6 */
  jmp       tpl_direct_irq_handler_exit_PORT4_VECTOR       /* bit 7 */
tpl_p4_5_handler:
  mov       #1, REG_RETARG
  call      #tpl_fast_central_interrupt_handler
```

The remaining code is similar to the one of the `tpl_sc_handler`.

```
tpl_direct_irq_handler_exit_PORT4_VECTOR:
  mov       r1, r13 /* get a copy of the KSP to restore it later        */
  add       #2, r13 /* and forget the pushed PSP (not useful anymore).  */
  pop       r1      /* get the saved process stack pointer back         */
  mov       #tpl_kern, r11
  tst.b     TPL_KERN_OFFSET_NEED_SWITCH(r11)
  jz        tpl_PORT4_VECTOR_no_context_switch
  pushm.w   #7, r10  /* Push r4 to r10 */
  mov       &tpl_kern, r11 /* Get the s_running slot of tpl_kern in r11 */
  mov       @r11, r11      /* Get the pointer to the context (SP alone) */
  mov       r1, @r11       /* Save the stack pointer                    */
  mov       r13, r1        /* Switch back to the kernel stack           */
  mov       #1, REG_RETARG
  call      #tpl_run_elected
  mov       &tpl_kern, r11 /* Get the s_running slot of tpl_kern in r11 */
  mov       @r11, r11      /* Get the pointer to the context (SP alone) */
```

```
  mov       @r11, r1       /* Get the stack pointer                    */
  popm.w    #12,r15        /* Pop r4 to r15  */
  reti
tpl_PORT4_VECTOR_no_context_switch:
  popm.w    #5, r15
  reti
```

2.4 Counter interrupt handler

A counter interruption handler has the same structure as that of an ISR 2. The only difference is the function called. For instance for the SystemCounter the code is straightforward.

```
tpl_primary_irq_handler_TIMER3_A0_VECTOR:

/*-----------------------------------------------------------------------------
 * -1- Before doing anything we have to save the volatile registers, which
 * are r11 (r11 is not volatile in the MSPGCC ABI but is volatile in GCC
 * compiler for MSP ABI. Anyway, in order to limit variabilility, r11 is
 * saved for both ABIs) to r15, because they will not be saved when we will
 * call the underlying C function.
 */
  pushm.w   #5, r15  /* Push r11, r12, r13, r14 and r15 */
/*-----------------------------------------------------------------------------
 * -2- Switch to the kernel stack.
 */
  mov       r1, r11                          /* Copy the PSP in r11      */
  mov       #tpl_kern_stack + TPL_KERNEL_STACK_SIZE, r1 /* kernel stack  */
  push      r11                              /* Save PSP to kernel stack */
/*-----------------------------------------------------------------------------
 * -3- Init the NEED_SWITCH/SAVE in tpl_kern.
 */
  mov       #tpl_kern, r11
  mov.b     #NO_NEED_SWITCH_NOR_SCHEDULE, TPL_KERN_OFFSET_NEED_SWITCH(r11)
  mov.b     #NO_NEED_SWITCH_NOR_SCHEDULE, TPL_KERN_OFFSET_NEED_SCHEDULE(r11)
/*-----------------------------------------------------------------------------
 * -4- Call the underlying C function.
 */
  call      #tpl_tick_TIMER3_A0_VECTOR
/*-----------------------------------------------------------------------------
 * -5- Switch back to the process stack
 */
tpl_direct_irq_handler_exit_TIMER3_A0_VECTOR:
  mov       r1, r13 /* get a copy of the KSP to restore it later        */
  add       #2, r13 /* and forget the pushed PSP (not useful anymore).  */
  pop       r1      /* get the saved process stack pointer back         */
/*-----------------------------------------------------------------------------
 * -6- Check the context switch condition in tpl_kern.
 */
  mov       #tpl_kern, r11
```

```
  tst.b      TPL_KERN_OFFSET_NEED_SWITCH(r11)
  jz         tpl_TIMER3_A0_VECTOR_no_context_switch
/*----------------------------------------------------------------------------
 * -7- Save the rest of the context.
 */
  pushm.w    #7, r10  /* Push r4 to r10 */
/*----------------------------------------------------------------------------
 * -8- Now the stack pointer is saved in the dedicated location.
 */
  mov        &tpl_kern, r11 /* Get the s_running slot of tpl_kern in r11 */
  mov        @r11, r11      /* Get the pointer to the context (SP alone) */
  mov        r1, @r11       /* Save the stack pointer                    */
/*----------------------------------------------------------------------------
 * -9- Call tpl_run_elected with argument 1 (aka save) after switching back
 * to the kernel stack.
 */
  mov        r13, r1        /* Switch back to the kernel stack           */
  mov        #1, REG_RETARG
  call       #tpl_run_elected
/*----------------------------------------------------------------------------
 * -10- tpl_run_elected has copied the elected process slot of tpl_kern to
 * the running slot. We load the stack pointer of the new running process.
 */
  mov        &tpl_kern, r11 /* Get the s_running slot of tpl_kern in r11 */
  mov        @r11, r11      /* Get the pointer to the context (SP alone) */
  mov        @r11, r1       /* Get the stack pointer                     */
/*----------------------------------------------------------------------------
 * -11- Now, the context of the new running process is loaded. All registers
 * are popped.
 */
  popm.w     #12,r15        /* Pop r4 to r15  */
  reti
/*----------------------------------------------------------------------------
 * -12- We get here from stage 6. Restore the volatile registers and return
 * from the interrupt handler.
 */
tpl_TIMER3_A0_VECTOR_no_context_switch:
  popm.w     #5, r15
  reti
```

3 MCU Clocks

The MCU clocks uses the DCO as input clock. The CPU is limited to 16MHz.

3.1 Startup

The MCU clocks can be defined in the .oil file directly in `CPU->OS->CPU_FREQ_MHZ`. Value should be in the set *1,2,4,6,8,12,16,21 and 24* MHz.

By default, the frequency is set to 1MHz.

3.2 Dynamic update

The MCU clocks can be updated with a user fonction to update the frequency in `tpl_clocks.h`:

```
/* configure the frequency in MHz: 1,2,4,6,8,12,16,(21,24 overclock)
 * set to 1MHz in case of bad input frequency.
 **/

FUNC(void,OS_CODE) tpl_set_mcu_clock(uint16_t freq);
```

When the CPU clock is updated, the Wait States for the FRAM access are set accordingly: 1 wait state above 8MHz, and 2 wait states above 16MHz.

Note that the 21MHz and 24 MHz frequencies overclock the CPU capabilities and may not work.

A callback can be added each time the CPU clock is updated. This is done through the function:

```
void tpl_add_freq_update_callback(tpl_freq_update_item *freqObs);
```

Where `freqObs` is an item of a single linked list of function calls. This functionnality is implemented in the serial line driver (for debug purpose) of the launchpad like this:

```
#include "tpl_clocks.h"

tpl_freq_update_item tpl_serial_callback = {&tpl_serial_update_freq,NULL};

void tpl_serial_begin()
{
        /* make sure we are informed of a clock update. */
        tpl_add_freq_update_callback(&tpl_serial_callback);
        ..
}

void tpl_serial_update_freq()
{
        //callback that updates the serial baudrate configuration
        //in function of the new input clock.
}
```

The frequency of the MCU is defined using the DCO. The function `uint32_t tpl_getDCOFrequency();` returns the DCO output frequency in Hz. This can be used in the function callback.

4 Low power in idle

When Trampoline runs the idle task, the MCU can be put in low power mode. This is done by setting the attribute `IDLE_POWER_MODE` in the `OS` object. Possible values are `ACTIVE`, `LPM0`, `LPM1`, `LPM2` and `LPM3`. Default value, that is without setting this attribute, is `ACTIVE`.

5 Stack size estimation

In the following, tasks and ISR2s are collectively referred to as processes. Stack size of a process depends on what function the process calls, the size of the local variables, the optimization level of the compiler and if at least one ISR1 is used by the application[4]. So it is not something that is easy to compute. The minimum stack size of a process is the size needed to store an initialized context as shown at figure 7, i.e. 30 bytes.

When the process runs, the context is popped and the only element left on the stack is the return address to `CallTerminateTask` or `CallTerminateISR2`, which leaves 28 bytes for local variables and function calls.

Calling an OS service requires 14 bytes if the service does not lead to a context save and 28 bytes if it does not. Since the kernel runs on a dedicated stack, the stack depth required to run a service has no impact on the stack size of a process.

Example of a trivial basic task

Let's take the blink task from the example `readbutton_isr2`. Once compiled in `-O0`, the generated code is as follow:

```
00006238 <blink_function>:
    6238:       04 12           push    r4
    623a:       04 41           mov     r1,     r4
    623c:       24 53           incd    r4
    623e:       5f 42 02 02     mov.b   &0x0202,r15     /* read port  */
    6242:       6f e3           xor.b   #2,     r15     /* toggle     */
    6244:       c2 4f 02 02     mov.b   r15,    &0x0202 /* write port */
    6248:       b0 12 54 61     call    #0x6154 /* Call TerminateTask */
    624c:       34 41           pop     r4
    624e:       30 41           ret
```

By pushing `r4` on the stack and calling `TerminateTask`, the amount of stack consumed is 16 bytes out of the 28 bytes available. The minimum stack size is therefore sufficient. However, pressing the button at the time the task is executed leads to its pre-emption and the execution of ISR2. In this case, the stack must be able to contain the register `r4` that has been pushed and the context of the task, namely 30 bytes. By adding the two bytes of the `CallTerminateTask` address, the minimum stack is 32 bytes.

[4] ISR1 executes on the stack of the running task/ISR2

When the application is compiled in `-O3`, the blink task code is as follows:

```
00005d60 <blink_function>:
    5d60:       e2 e3 02 02     xor.b   #2,     &0x0202 ;r3 As==10
    5d64:       b0 12 7a 5c     call    #0x5c7a
    5d68:       30 41           ret
```

This time `r4` is not pushed on the stack and its size can be 30 bytes. Given the time taken to complete this task, it is also questionable whether it would not be worthwhile to make it non-preemptible.

6 Memory mapping and memory protection

Memory organization of MSP430FR5969 is shown at figure 8.

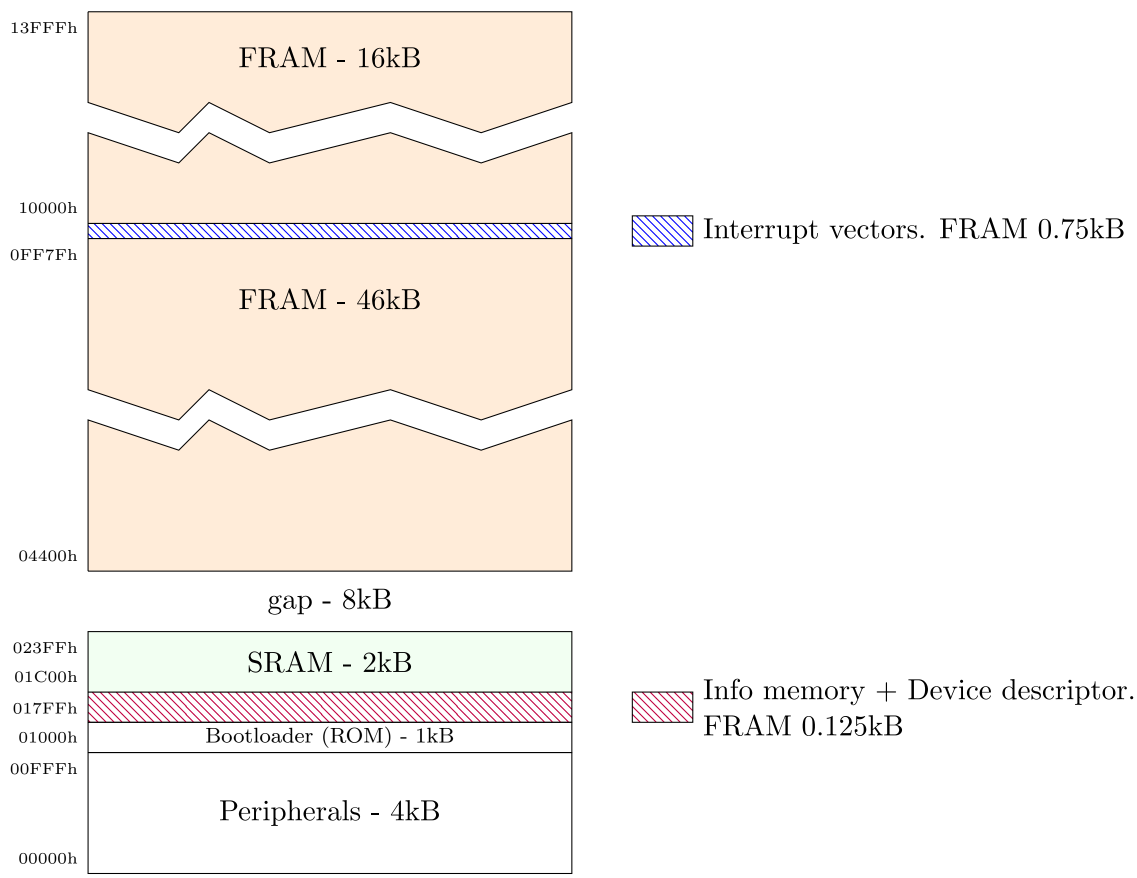


Figure 8: Memory organization of MSP430FR5969

The TI MSP430 uses a very simple memory protection scheme. The Memory Protection

Unit allows to define 2 boundaries, SEGB1 and SEGB2 and the access right corresponding to 3 regions, the one below SEGB1 (excluded), the one between SEGB1 (included) and SEGB2 (excluded) and the one above SEGB2 (included). Some addresses locations, the 16 bytes starting à `0xFF80` contain the JTAG password. Writing random values at theses addresses bricks the MCU. To prevent that, Trampoline initialize the MPU so that addresses below the start of FRAM (peripherals and SRAM) may be read and written, addresses from start of FRAM to `0x10000` may be read and executed and addresses from `0x10000` to the end of the FRAM may be read and written.

7 Libraries

7.1 Serial line

The launchpad kits use a serial line over USB that can be used for debugging purpose. The configuration is 115200 bauds, 8N1.

The library should be declared in the .oil file (so that dedicated files are included in the build process)

```
BUILD = TRUE {
  LIBRARY = serial;
};
```

The library is quite limited at this date, and it can only send characters (no reception). There is no ring buffer, and there is a waiting loop if the previous character is not yet sent.

An example is given for the msp430fr5969 launchpad.

The library supports various MCU change frequencies, and the output frequency is updated (the current message may be corrupted!).

References

[1] Texas Instruments. MSP430 Embedded Application Binary Interface. Technical report, Texas Instruments incorporated, June 2013.

[2] Texas Instruments. Calling Convention and ABI Changes in MSP GCC. Technical report, Texas Instruments Incorporated, February 2015.

[3] Texas Instruments. MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide. Technical report, Texas Instruments Incorporated, December 2017.

[4] Texas Instruments. MSP430x5xx and MSP430x6xx Family User's Guide. Technical report, Texas Instruments Incorporated, March 2018.