

# MSP430X port - small memory model version

Jean-Luc Béchenec

October 7, 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<sup>1</sup>. 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

---

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

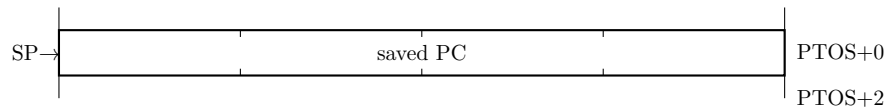
A service call is done using the `br` instruction in the service call wrapper to prevent 2 nested call and fold the `ret` instruction. The service identifier is passed to the service call handler through the `REG_SID` register. So a service call wrapper is as shown in listing at figure 2.

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 */
```

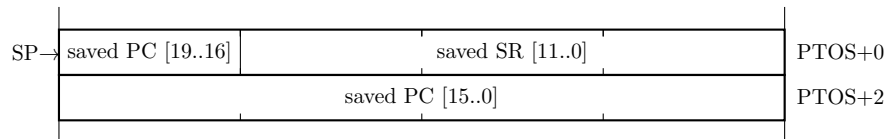
When in the `tpl_sc_handler` the stack is as shown at figure 3<sup>2</sup>. *PTOS* stands for *Process Top Of Stack*.

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

<sup>2</sup>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.  $\tau_1$  is running,  $\tau_2$  is ready.  $P(\tau_1) > P(\tau_2)$ .  $\tau_1$  calls `WaitEvent` and is preempted synchronously,  $\tau_2$  becomes running and calls `SetEvent`.  $\tau_2$  is preempted and  $\tau_1$  is resumed synchronously.

**SPAR** Synchronous Preemption, Asynchronous Resume.  $\tau_1$  calls `WaitEvent` and is synchronously preempted, An alarm does a `SetEvent` on  $\tau_1$  which is asynchronously resumed.

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

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

```

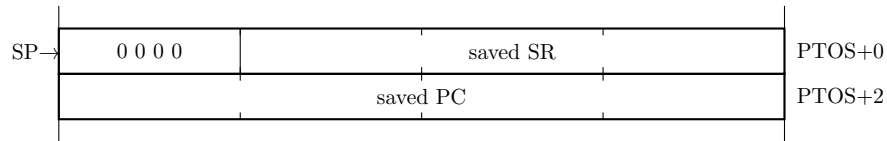
We need to have the same stack pattern for both the `tpl_sc_handler` and an interrupt handler which calls the operating system. So we push the SR and we reset the 4 higher bits (high weight of PC, not sure it is needed) and set GIE in the saved SR.

```

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.

```

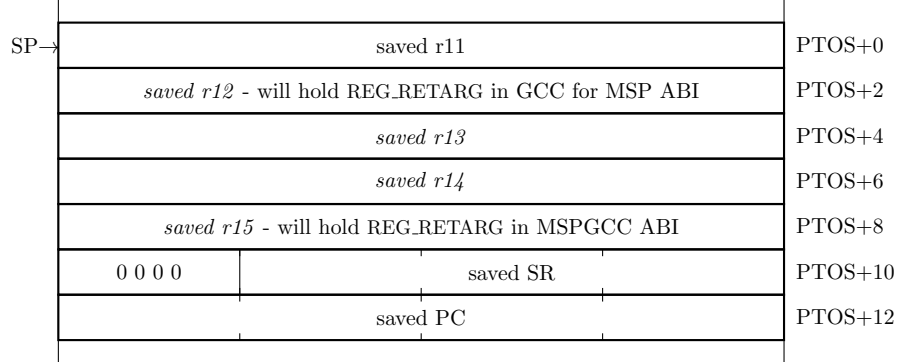
push  r11

```

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

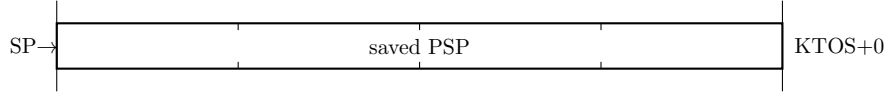


```

mov    r1,r11
mov    #tpl_kern_stack_bottom, r1
push   r11

```

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



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
	<i>saved r12 - REG_RETARG in GCC for MSP ABI</i>	PTOS+16
	<i>saved r13</i>	PTOS+18
	<i>saved r14</i>	PTOS+20
	<i>saved r15 - will hold REG_RETARG in MSPGCC ABI</i>	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 should 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`.

## 1.5 The SysTick Handler

The SysTick is generated by using a `TIMERA` register. On the `MSP430FR5969`, `TIMERA3` has been chosen. 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`<sup>3</sup> to `r15`, because they will not be saved when we will call the `tpl_counter_tick_SystemCounter` C function.

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

As a result the stack is as follow for *MSPGCC* ABI:

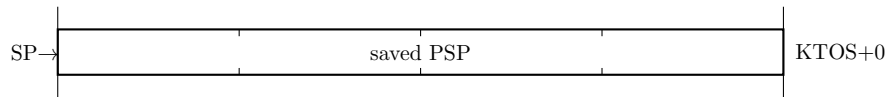
SP→	saved r11		PTOS+0
	saved r12		PTOS+2
	saved r13		PTOS+4
	saved r14		PTOS+6
	saved r15		PTOS+8
	saved PC [19..16]	saved SR [11..0]	PTOS+10
	saved PC [15..0]		PTOS+12

Switch to the kernel stack.

```
mov r1,r11 /* Copy the PSP in r11 */
mov #tpl_kern_stack_bottom, r1 /* Switch to the kernel stack */
push r11 /* Save PSP to kernel stack */
```

<sup>3</sup>`r11` is not volatile in the *MSPGCC* ABI but is volatile in *GCC compiler for MSP* ABI. Anyway, in order to limit variability, `r11` is saved for both ABIs.

The kernel stack is as follow:



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 function that increments the SystemCounter and manages alarm.

```
call   tpl_counter_tick_SystemCounter
```

Switch back to the process stack

```
mov    r1, r13 /* get a copy of the KSP to restore it later */
add    #2, r13 /* and forget the pushed PSP that is not useful anymore */
pop    r1      /* get the saved process stack pointer back */
```

Check the context switch condition in `tpl_kern`.

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

Save the rest of the context.

```
pushm.w #7, r10 /* Push r4 to r10 */
```

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 */
```

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
```

`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. All registers are popped.

```

    popm.w #12,r15      /* Pop r4 to r15 */
    reti

```

Restore the volatile registers and return from the interrupt handler.

```

tpl_systick_handler_no_context_switch:
    popm.w #5, r15
    reti

```

## 2 Memory mapping and memory protection

Memory organization of MSP430FR5969 is shown at figure 8.

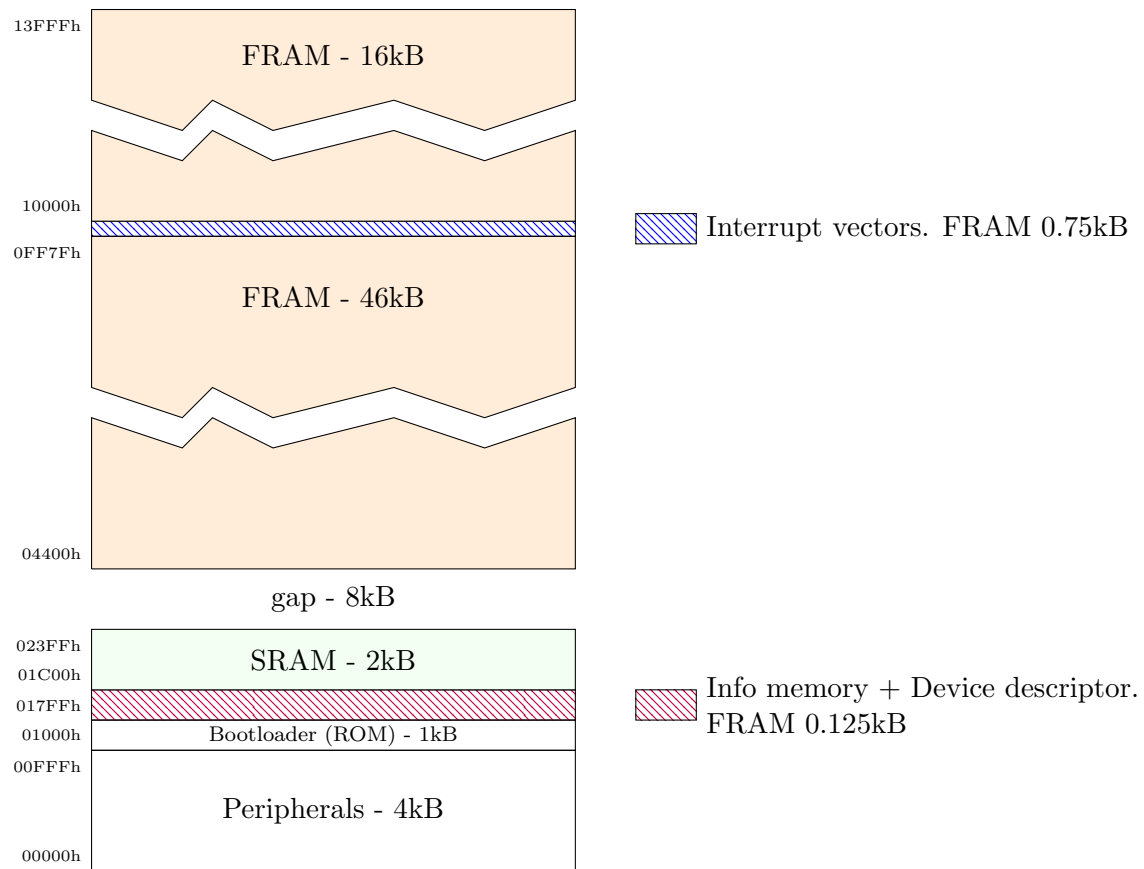


Figure 8: Memory organization of MSP430FR5969

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).

## 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.