```
Learning Activity 15
=-=-=-=-=-=-=-=-=-=-

* Scaling Up:
   - Adding a third (or fourth...) thread is TRIVIAL:

       static thread_t threadTable[] = {
         thread1,
         thread2,
         thread3,
         thread1
       };
       // This is a cool idiom that doesn't need to change
       #define NUM_THREADS (sizeof(threadTable)/sizeof(threadTable[0]))

   - None of the code in the other threads was affected
   - This is one benefit of the "scheduler abstraction"

* Checking up:
   - Easiest place to do this is in threadStarter():

       void threadStarter(void)
       {
         fillStackWithSentinel();

         (*(threadTable[currThread]))(); // Run the thread

         reportFreeStackSpace();

         threads[currThread].active = 0;

         yield();
       }

* Gearing up:
   - Where would you FIRST switch from privileged mode to
     unprivileged mode? threadStarter() again is probably easiest.

   - Where would you NEXT switch from privileged mode to
     unprivileged mode? In scheduler() at about line 108:

         if (threads[currThread].active) {
           REDUCE_PRIVILEGE();
           longjmp(threads[currThread].state, 1);
         } else {
           i--;
         }

   - How is privilege level reduced?
```

Implementations can support an IMPLEMENTATION DEFINED number of priorities in powers of 2. Where fewer than 256 priorities are implemented, the low-order bits of the BASEPRI field corresponding to the unimplemented priority bits are RAZ/WI.

These registers can be accessed using the `MSR/MRS` instructions. The `MSR` instruction includes an additional register mask value BASEPRI_MAX, which updates BASEPRI only where the new value increases the priority level (decreases BASEPRI to a non-zero value). See *MSR (register)* on page B4-8 for details.

In addition:

- FAULTMASK is set by the execution of the instruction: CPSID f
- FAULTMASK is cleared by the execution of the instruction: CPSIE f
- PRIMASK is set by the execution of the instruction: CPSID i
- PRIMASK is cleared by the execution of the instruction: CPSIE i.

B1.4.4 The special-purpose ~~control~~ register

The special-purpose CONTROL register is a 2-bit register defined as follows:

- bit [0] defines the Thread mode privilege (Handler mode is always privileged)
  - 0: Thread mode has privileged access
  - 1: Thread mode has unprivileged access.
- bit [1] defines the stack to be used
  - 0: SP_main is used as the current stack
  - 1: For Thread mode, SP_process is used for the current stack. For Handler mode, this value is reserved.
  - Software can update bit [1] in Thread mode. Explicit writes from Handler mode are ignored.
  - The bit is updated on exception entry and exception return. See the pseudocode in *Exception entry behavior* on page B1-21 and *Exception return behavior* on page B1-25 for more details.
- bits [31:2] reserved.

The CONTROL register is cleared on reset. The `MRS` instruction is used to read the register, and the `MSR` instruction is used to write the register. Unprivileged write accesses are ignored.

An ISB barrier instruction is required to ensure a CONTROL register write access takes effect before the next instruction is executed.

B1.4.5 Reserved special-purpose register bits

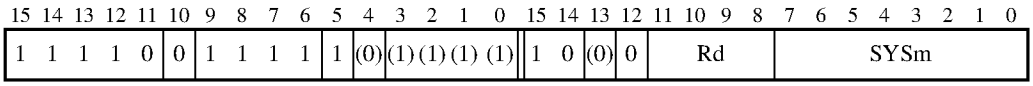
All unused bits in special-purpose registers are reserved. `MRS` and `MSR` instructions that access reserved bits treat them as RAZ/WI. For future software compatibility, the bits are UNK/SBZP. Software should write them to zero when initializing the register for a new process, otherwise software should restore reserved bits when updating or restoring a special-purpose register.

B4.1.2 MRS

Move to Register from Special Register moves the value from the selected special-purpose register into a general-purpose register.

Encoding T1 ARMv6-M, ARMv7-M Enhanced functionality in ARMv7-M.

```
MRS<c> <Rd>,<spec_reg>
```




```
d = UInt(Rd);
if d IN {13,15} || !(UInt(SYSm) IN {0..3,5..9,16..20}) then UNPREDICTABLE;
```

Assembler syntax

```
MRS<c><q>  <Rd>, <spec_reg>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<spec_reg>` Encoded in SYSm, specifies one of the following:

Special register	Contents	SYSm value
APSR	The flags from previous instructions	0
IAPSR	A composite of IPSR and APSR	1
EAPSR	A composite of EPSR and APSR	2
XPSR	A composite of all three PSR registers	3
IPSR	The Interrupt status register	5
EPSR	The execution status register	6
IEPSR	A composite of IPSR and EPSR	7
MSP	The Main Stack pointer	8
PSP	The Process Stack pointer	9
PRIMASK	Register to mask out configurable exceptions	16 [a]
BASEPRI	The base priority register	17 [b]

Add footnote:

The EPSR is RAZ

Special register	Contents	SYSm value
BASEPRI_MAX	This acts as an alias of BASEPRI on reads	18 [c]
FAULTMASK	Register to raise priority to the HardFault level	19 [d]
CONTROL	The special-purpose control register	20 [e]
RSVD	RESERVED	unused

a. Raises the ~~current~~ priority to 0 when set to 1. This is a 1-bit register.
b. Changes the current pre-emption priority mask to a value between 0 and N. 0 means the mask is disabled. The register only has an effect when the value (1 to N) is lower (higher priority) than the non-masked priority level of the executing instruction stream.
The register can have up to 8 bits (depending on the number of priorities supported), and it is formatted exactly the same as other priority registers.
The register is affected by the PRIGROUP (binary point) field. See *Exception priorities and pre-emption* on page B1-17 for more details. Only the pre-emption part of the priority is used by BASEPRI for masking.
c. When used with the MSR instruction, it performs a conditional write.
d. This register raises the ~~current~~ priority to -1 (the same as HardFault) when it is set to 1. This can only be set by privileged code with a priority below -1 (not NMI or HardFault), and self-clears on return from any exception other than NMI. This is a 1-bit register.
e. The control register is composed of the following bits:
[0] = Thread mode privilege: 0 means privileged, 1 means unprivileged (User). This bit resets to 0.
[1] = Current stack pointer: 0 is Main stack (MSP), 1 is alternate stack (PSP if Thread mode, RESERVED if Handler mode). This bit resets to 0.

Operation

```
if ConditionPassed() then
    R[d] = 0;
    case  SYSm<7:3> of
        when '00000'
            if SYSm<0> == '1' and CurrentModeIsPrivileged() then
                R[d]<8:0> = IPSR<8:0>;
            if SYSm<1> == '1' then
                R[d]<26:24> = '000';     /* EPSR reads as zero */
                R[d]<15:10> = '000000';
            if SYSm<2> == '0' then
                R[d]<31:27> = APSR<31:27>;
        when '00001'
            if CurrentModeIsPrivileged() then
                case SYSm<2:0> of
                    when '000'
                        R[d] = MSP;
                    when '001'
                        R[d] = PSP;
        when '00010'
            case SYSm<2:0> of
                when '000'
                    R[d]<0> = if CurrentModeIsPrivileged() then
                        PRIMASK<0> else '0';
                when '001'
                    R[d]<7:0> = if CurrentModeIsPrivileged() then
                        BASEPRI<7:0> else '00000000';
                when '010'
                    R[d]<7:0> = if CurrentModeIsPrivileged() then
                        BASEPRI<7:0> else '00000000';
                when '011'
                    R[d]<0> = if CurrentModeIsPrivileged() then
                        FAULTMASK<0> else '0';
                when '100'
                    R[d]<1:0> = CONTROL<1:0>;
```

Exceptions

None.

Notes

Privilege If User code attempts to read any stack pointer or the IPSR, it returns 0s.

EPSR None of the EPSR bits are readable during normal execution. They all read as 0 when read using MRS (Halting debug can read them via the register transfer mechanism).

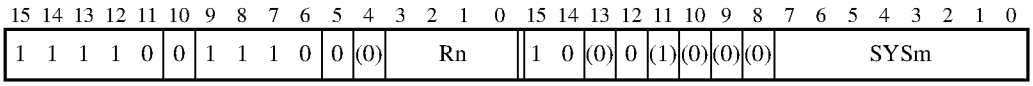
Bit positions The PSR bit positions are defined in *The special-purpose program status registers (xPSR)* on page B1-8.

B4.1.3 MSR (register)

Move to Special Register from ARM Register moves the value of a general-purpose register to the selected special-purpose register.

Encoding T1 ARMv6-M, ARMv7-M Enhanced functionality in ARMv7-M.

```
MSR<c> <spec_reg>,<Rn>
```




```
n = UInt(Rn);
if n IN {13,15} || !(UInt(SYSm) IN {0..3,5..9,16..20}) then UNPREDICTABLE;
```

Assembler syntax

```
MSR<c><q>  <spec_reg>, <Rn>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` Is the general-purpose register to receive the special register contents.

`<spec_reg>` Encoded in SYSm, specifies one of the following:

Special register	Contents	SYSm value
APSR	The flags from previous instructions	0
IAPSR	A composite of IPSR and APSR	1
EAPSR	A composite of EPSR and APSR	2
XPSR	A composite of all three PSR registers	3
IPSR	The Interrupt status register	5
EPSR	The execution status register ~~(reads as zero, see Notes)~~	6
IEPSR	A composite of IPSR and EPSR	7
MSP	The Main Stack pointer	8
PSP	The Process Stack pointer	9
PRIMASK	Register to mask out configurable exceptions	16 [a]
BASEPRI	The base priority register	17 [b]

Insert footnote:

The EPSR ignores writes

Special register	Contents	SYSm value
BASEPRI_MAX	On writes, raises BASEPRI but does not lower it	18 [c]
FAULTMASK	Register to raise priority to the HardFault level	19 [d]
CONTROL	The special-purpose control register	20 [e]
RSVD	RESERVED	unused

a. Raises the ~~current~~ priority to 0 when set to 1. This is a 1-bit register.
b. Changes the current pre-emption priority mask to a value between 0 and N. 0 means the mask is disabled. The register only has an effect when the value (1 to N) is lower (higher priority) than the non-masked priority level of the executing instruction stream.
The register can have up to 8 bits (depending on the number of priorities supported), and it is formatted exactly the same as other priority registers.
The register is affected by the PRIGROUP (binary point) field. See *Exception priorities and pre-emption* on page B1-17 for more details. Only the pre-emption part of the priority is used by BASEPRI for masking.
c. When used with the MSR instruction, it performs a conditional write. The BASEPRI value is only updated if the new priority is higher (lower number) than the current BASEPRI value.
Zero is a special value for BASEPRI (it means disabled). If BASEPRI is 0, it always accepts the new value. If the new value is 0, it will never accept it. This means BASEPRI_MAX can always enable BASEPRI but never disable it. PRIGROUP has no effect on the values compared or written. All register bits are compared and conditionally written.
d. This register raises the ~~current~~ priority to -1 (the same as HardFault) when it is enabled set to 1. This can only be set by privileged code with a priority below -1 (not NMI or HardFault), and self-clears on return from any exception other than NMI. This is a 1-bit register.
The CPS instruction can also be used to update the FAULTMASK register.
e. The control register is composed of the following bits:
[0] = Thread mode privilege: 0 means privileged, 1 means unprivileged (User). This bit resets to 0.
[1] = Current stack pointer: 0 is Main stack (MSP), 1 is alternate stack (PSP if Thread mode, RESERVED if Handler mode). This bit resets to 0.

Operation

```
if ConditionPassed() then
    case SYSm<7:3> of
        when '00000'
            if SYSm<2> == '0' then
                APSR<31:27> = R[n]<31:27>;
        when '00001'
            if CurrentModeIsPrivileged() then
                case SYSm<2:0> of
                    when '000'
                        MSP = R[n];
                    when '001'
                        PSP = R[n];
        when '00010'
            case SYSm<2:0> of
                when '000'
                    if CurrentModeIsPrivileged() then PRIMASK<0> = R[n]<0>;
                when '001'
                    if CurrentModeIsPrivileged() then BASEPRI<7:0> = R[n]<7:0>;
                when '010'
                    if CurrentModeIsPrivileged() &&
                    (R[n]<7:0> != '00000000') &&
                    (R[n]<7:0> < BASEPRI<7:0> || BASEPRI<7:0> == '00000000') then
                        BASEPRI<7:0> = R[n]<7:0>;
                when '011'
                    if CurrentModeIsPrivileged() &&
                    (ExecutionPriority > -1) then
                        FAULTMASK<0> = R[n]<0>;
                when `100`
                    if CurrentModeIsPrivileged() then
                        CONTROL<0> = R[n]<0>;
                        If CurrentMode == Mode_Thread then CONTROL<1> = R[n]<1>;
```

Exceptions

None.

Notes

Privilege Writes from unprivileged Thread mode to any stack pointer, the EPSR, the IPSR, the masks, or CONTROL, will be ignored. If privileged Thread mode software writes a 0 into CONTROL[0], the core will switch to unprivileged Thread mode (User) execution, and inhibit further writes to special-purpose registers.
An `ISB` instruction is required to ensure instruction fetch correctness following a Thread mode privileged => unprivileged transition.

IPSR The currently defined IPSR fields are not writable. Attempts to write them by Privileged code is write-ignored (has no effect).

EPSR The currently defined EPSR fields are not writable. Attempts to write them by Privileged code is write-ignored (has no effect).

Bit positions The PSR bits are positioned in each PSR according to their position in the larger xPSR composite. This is defined in *The special-purpose program status registers (xPSR)* on page B1-8.

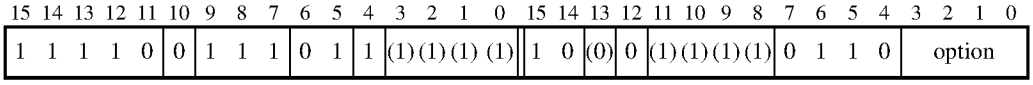
A6.7.36 ISB

Instruction Synchronization Barrier flushes the pipeline in the processor, so that all instructions following the `ISB` are fetched from cache or memory, after the instruction has been completed. It ensures that the effects of context altering operations, such as changing the ~~ASID, or completed TLB maintenance operations, or branch predictor maintenance operations, as well as all changes to the CP15~~ registers, executed before the `ISB` instruction are visible to the instructions fetched after the `ISB`.

In addition, the `ISB` instruction ensures that any branches that appear in program order after it are always written into the branch prediction logic with the context that is visible after the `ISB` instruction. This is required to ensure correct execution of the instruction stream.

Encoding T1 ARMv6-M, ARMv7-M

```
ISB<c> #<option>
```




```
// No additional decoding required
```

```
Learning Activity 13
=-=-=-=-=-=-=-=-=-=-=-

* Gearing up:
   - Where would you FIRST switch from privileged mode to
     unprivileged mode? threadStarter() again is probably easiest.

   - Where would you NEXT switch from privileged mode to
     unprivileged mode? In scheduler() at about line 108:

        if (threads[currThread].active) {
          REDUCE_PRIVILEGE();
          longjmp(threads[currThread].state, 1);
        } else {
          i--;
        }

   - How is privilege level reduced?

   - Where would you switch back to a privileged mode so the
     scheduler can run privileged? Also in scheduler():

      void scheduler(void) {
        unsigned i;

        currThread = -1;

        do {
          if (setjmp(scheduler_buf)==0) {

            i = NUM_THREADS;
            do {
              if (++currThread == NUM_THREADS) {
                currThread = 0;
              }

              if (threads[currThread].active) {
                REDUCE_PRIVILEGE();
                longjmp(threads[currThread].state, 1);
              } else {
                i--;
              }
            } while (i > 0);

            return;
          } else {
            INCREASE_PRIVILEGE();
            if (! threads[currThread].active) {
              free(threads[currThread].stack - STACK_SIZE);
            }
          }
        } while (1);

    - How is privilege level increased? OH-OH!! Thread was running
      in unprivileged mode so now we cannot write to the CONTROL register!

    - Only way to enter privileged mode now is through an exception.
```

A6.7.136 SVC (formerly SWI)

Generates a supervisor call. See *Exceptions* in the *ARM Architecture Reference Manual*.

Use it as a call to an operating system to provide a service.

Encoding T1 All versions of the Thumb ISA.

```
SVC<c> #<imm8>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	1	imm8							

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly, and is ignored by hardware. SVC handlers in some
// systems interpret imm8 in software, for example to determine the required service.
```

Assembler syntax

```
SVC<c><q>  #<imm>
```

where:

<c><q> See *Standard assembler syntax fields* on page A6-7.

<imm> Specifies an 8-bit immediate constant.

The pre-UAL syntax `SWI<c>` is equivalent to `SVC<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CallSupervisor();
```

Exceptions

SVCall.

```
SVC
=-=
* Gateway from unprivileged thread code to access protected system resources
  (or interact with the kernel/scheduler) in a controlled manner (and without
  the need for "funny business" like making a timer interrupt go off just for
  this reason)

* SVC generates an exception, hence SVC handler is in the privileged mode.
  Upon return from exception, code resumes in unprivileged mode.

* SVC instruction encodes an 8-bit integer which is completely ignored by the
  Cortex-M3. The exception handler, however, can inspect it and do different
  things depending on its value.

* Recall exception handling process:
   a) Push R0-R3, R12, R14, AddressToReturnTo, xPSR

      +-------------------+
      |       xPSR        |     (R13+28)
      +-------------------+
      | AddressToReturnTo |     (R13+24)
      +-------------------+
      |        R14        |     (R13+20)
      +-------------------+
      |        R12        |     (R13+16)
      +-------------------+
      |        R3         |     (R13+12)
      +-------------------+
      |        R2         |     (R13+8)
      +-------------------+
      |        R1         |     (R13+4)
      +-------------------+
      |        R0         | <-- R13
      +-------------------+

   b) What is AddressToReturnTo? Address of instruction AFTER the
      SVC instruction that caused the exception.

              SVC    #123     @ Causes SVC exception
              MOV    R0,R1    @ <-- AddressToReturnTo

   c) SVC Exception handler can inspect the SVC instruction that invoked this
      handler by going to AddressToReturnTo and backing up by 2 bytes:

              LDR  R0, [R13, #24]   @ R0 <-- AddressToReturnTo
              SUB  R0, R0, #2       @ R0 <-- Address of SVC instruction
              LDRH R0, [R0]         @ R0 <-- 16-bit encoding of SVC instruction

   d) Now lower 8 bits of R0 should contain 123. The exception handler
      can then do different things depending on this number, such as:

         - disable interrupts ---\____ Probably most common use
         - enable interrupts  ---/
         - change thread-mode privilege level (what we currently want)
           by writing to CONTROL register bit 0
         - reconfigure exception vector table
         - change memory region protections, etc.

      The exception handler can accept/ignore the request based upon your
      system design. Cortex-M3 has hard protections on what can be done in
      privileged/unprivileged mode. The SVC call allows your own software to
      make those choices based upon your system design (e.g., certain threads
      have certain privileges....)
```

```
64
65  The Next Step: Pre-Emption
66  =-=-=-=-=-=-=-=-=-=-=-=-=-
67
68  * What needs to be done to change from a coroutine-based approach to a
69    fully pre-emptive approach?
70
71    - SysTick timer generates periodic interrupts (could really use any
72      timer but a) SysTick is guaranteed on all Cortex-M3 parts thus
73      your code will remain portable, and b) it was really meant for this)
74
75    - SysTick exception handler IS the scheduler:
76       --> save the state of the running thread
77       --> determine which thread to run next
78       --> restore the state of the next thread
79       --> return from exception handler (not to the interrupted thread!
80           return to the "next thread")
81
82      This is known as a "context switch".
83
84    - createThread() and threadStarter() functions modified appropriately
85
86    - yield() modified appropriately
87
88  * Saving and restoring state is similar in principle to setjmp/longjmp
89    but must save EVERYTHING:
90
91     --> setjmp saves (according to AAPCS):
92         R4, R5, R6, R7, R8, R9, R10, R11, R13, R14
93
94     --> context switch must save:
95         R0-R14, AddressToReturnTo (like R15), LR, xPSR
96         NOTE: LR != R14
97               --> LR is "special code" indicating what mode
98                   "BX LR" should return to (see Table B1-8
99                   page B1-26 of ARM-ARM) on exception exit
100              --> R14 is value of LR in user thread when
101                  exception arrived
102
103        ** R0-R3, R12, R14, AddressToReturnTo, xPSR
104           are already on the stack
105
106        ** R4-R11, R13, LR must be saved in addition
107
108 * Where to save this additional context?
109
110    a) On the stack, below the 8 registers already stacked by
111       exception handler
112
113    b) In the existing structure for thread state:
114
115       typedef struct {
116         int active;       // non-zero means thread is allowed to run
117         char *stack;      // pointer to TOP of stack
118
119         // No longer using setjmp, hence no longer needed
120         //     jmp_buf state;    // saved state for longjmp()
121
122         // Save R4-R11, R13, LR here while in exception handler
123         unsigned savedregs[40];  // Room for 10 registers
124       } threadStruct_t;
125
126
```

```
Details, Details, Details
=-=-=-=-=-=-=-=-=-=-=-=-=

* Which stack is state saved on? Process or Main stack?

  --> For Thread-Mode Process Stack exception, registers will
      be pushed onto Process Stack (see Section B1.5.6 of ARM-ARM),
      R13 then automatically switches to Main Stack

  --> For Handler-Mode Main Stack exception (suppose an exception
      happens during an exception), registers will be pushed onto
      Main Stack

* This means that scheduler (in exception handler) must explicitly
  get/change the Process Stack pointer for context/save restore.

                  R13(process) != R13(main) !!!

  How? Use MRS/MSR instructions:

            MRS   R0, MSP    @ R0 <-- Main Stack R13
            MSR   PSP, R1    @ Process Stack R13 <-- R1

  Of course, this only works in privileged mode, which is OK since
  context switch is an exception handler.

* How must createThread() change?

  --> Must set up initial values of R0-R14, AddressToReturnTo, xPSR, R13
      INTELLIGENTLY! (Left as an exercise for the student.....)

* How must threadStarter() change?

  --> Probably not much at all. Think about it.....

* How must yield() change?

  --> Could just simulate a SysTick interrupt? See Table B3-6 on
      page B3-12 of ARM-ARM. Writing PENDSTSET bit generates a
      SysTick. Sadly....writing to this register is for privileged
      modes only....can't be done directly by thread code.

      Same goes for NVIC (maybe you were thinking of simulating an
      interrupt that way).

  --> Maybe thread code could call an SVC handler to do this???

* How to kick off this entire process from main()? What will the
  very first context switch do given that there is no thread state
  to save?

  --> Left as an exercise for the student.......
```