Supervisor calls and passing arguments to the kernel

Dr. Larry Hughes

15 February 2012
Revised: 19 October 2016, 10 October 2017, 8 October 2018

1 Introduction

The SVC or supervisor call instruction allows a kernel function or user process to trap to the kernel. It is probably the most common means of allowing a user process to access kernel functions. Although the Cortex manual describes the instruction as an "exception", by definition it is not an exception since exceptions, like interrupts, are involuntary whereas a trap is voluntary.

This document describes a method of using the supervisor call instruction in a multiprocessing environment and shows how it can be used to pass arguments to the kernel.

2 The SVC instruction

The Cortex supervisor call instruction has the same effect as an interrupt or an exception: control passes from the trapping function (a user process or a kernel function) to the SVC handler entry point. The address of the entry point is specified in vector table location 12. As with an interrupt or exception, eight of the CPU's registers are pushed onto the trapping function's stack (either PSP or MSP, depending on which was used when the interrupt occurred), as shown in Figure 1.

Low memory	unsigned long R0	**Stack pointer (PSP)**
	unsigned long R1	
	unsigned long R2	
	unsigned long R3	
	unsigned long R12	
	unsigned long LR	
	unsigned long PC	
High memory	unsigned long PSR	

Figure 1: Stack frame after execution of SVC (assuming PSP used when interrupt occurred)

The link register (LR) indicates the source stack of the trapping process (i.e., in Thread mode) with a value of the exception return, *EXC_RETURN*: PSP (0xFFFF.FFFD) or MSP (0xFFFF.FFF9). Regardless of the source of the trap or the stack used, the MSP becomes the active stack pointer.

The SVC instruction requires a numeric value; for the purposes of this document, the value will be ignored by the SVC handler. In the Cortex C compiler, the SVC call is written as:

```
__asm("      SVC     #0");
```

Returning from the SVC handler requires the use of an instruction that puts the link register value into the PC; for example:

- If the LR has not been pushed onto the stack or changed, the BX instruction can be used:

```
__asm("   BX     LR");
```

- Alternatively, if the LR has been pushed onto the stack, the POP instruction can be used (the braces, '{' and '}', denote a register list and are required, even with a single register):

```
__asm("   POP    {PC}");
```

3 Saving registers R4 through R11

In addition to the PSR, PC, and LR, the Cortex only saves registers R0 through R3 and R12 on the trapping function's stack, the remainder remain unsaved in the CPU. Since the SVC handler could very easily use registers R4 through R11, it is advisable to save them upon entry to the handler and restore them upon exit. In order to save the registers, it is necessary to determine onto which stack they should be saved; although the MSP is the active stack, the trapping stack could be either the PSP or the MSP. Due to the limitations of the Cortex C compiler, it is necessary to use the assembler to save and restore the registers.

The basic algorithm is as follows:

```
If the trapping stack is MSP then
        Save R4 through R11 on the MSP
        Perform required SVC actions
        Restore R4 through R11 back to the MSP
Else (the trapping stack is PSP)
        Save R4 through R11 on the PSP
        Perform required SVC actions
        Restore R4 through R11 back to the PSP
EndIf
```

After saving registers R4 through R11, the trapping stack contents are as shown in Figure 2.

Low memory	unsigned long R4	**Stack pointer (MSP or PSP)**
	unsigned long R5	
	unsigned long R6	
	unsigned long R7	
	unsigned long R8	
	unsigned long R9	
	unsigned long R10	
	unsigned long R11	
	unsigned long R0	
	unsigned long R1	
	unsigned long R2	
	unsigned long R3	
	unsigned long R12	
	unsigned long LR	
	unsigned long PC	
High memory	unsigned long PSR	

Figure 2: Trapping stack (MSP or PSP) after explicit pushes of R4 through R11

The first step is to save the LR on the main stack (MSP) for returning to the trapping function:[1]

```
__asm("     PUSH {LR}");
```

Next, bit 3 of the LR (the least-significant bit is bit number 1) is checked (with the TST instruction and'ing #4 with the LR register) to determine the trapping source stack (see Table 1); the value is either '0' indicating the MSP stack (0xFFFF.FFF1 [the least significant nibble is 0001] or 0xFFFF.FFF9 [the least significant nibble is 1001]) or '1', indicating the PSP stack (0xFFFF.FFFD [the least significant nibble 1101). In the sample code, if the bit is set (i.e., not equal to zero), control passes to *RtnViaPSP*:

```
__asm("     TST   LR,#4");
__asm("     BNE   RtnViaPSP");
```

Table 1: Exception return – least-significant nibble values

Bit 4 - Mode		Bit 3 - Stack		Bits 2 and 1	EXC_RETURN
0	Handler	0	MSP	01	0xFFFF.FFF1
1	Thread	0	MSP	01	0xFFFF.FFF9
1	Thread	1	PSP	01	0xFFFF.FFFD

If bit 3 is clear (i.e., equal to zero), the trapping stack is pointed to by MSP. Since this is also the active stack, the registers R4 through R11 can be pushed directly onto the stack:

```
__asm("     PUSH {r4-r11}");
```

Since this is pushing onto the active stack (i.e., the MSP), there is no need to use *STMDB* (see below); however, the operations are effectively the same, with R11 being pushed onto the highest location and R4, the lowest on the MSP stack.

SVCHandler() can then be called; in order to allow the handler to access the active stack, R0 contains the value of the MSP and can be used as the argument to *SVCHandler()*:

```
__asm("     MRS   r0,msp");
__asm("     BL    SVCHandler");          /* r0 is MSP */
```

Upon return, registers R4 through R11 can be restored by pulling (popping) them from the active stack (MSP); the lowest address is popped into R4 and so on up to the highest address, which is popped into R11. The value on the stack is the LR (pushed above); if pulled into the PC, the Cortex will restore registers R0 through R3, R12, LR, PC, and PSR (note that the two POPs could be rolled into one); execution will resume with the trapping function and the MSP:

```
__asm("     POP   {r4-r11}");
__asm("     POP   {PC}");
```

The label *RtnViaPSP* indicates the start of the code to handle the trapping functions that use the PSP. In this case, since the trapping stack is the PSP, it is necessary to save registers R4 through R11 explicitly onto the PSP (the stack pointer cannot be used since it is pointing to the

[1] All code references refer to SVC_example.c.

MSP at this moment); this is done by copying the PSP to R0 (using the *MRS* instruction) and then copying each register to the address pointed by R0 and decrementing R0:

```
__asm("RtnViaPSP:");
__asm("      mrs    r0,psp");
__asm("      stmdb  r0!,{r4-r11}");        /* Store multiple, decrement before */
```

Note the instruction *STMDB* (STore Multiple Decrement Before) pushes the register with the highest register number in the highest address through to the register with the lowest register number to the lowest address. In this example, registers R4 through R11 are pushed, meaning that R11 is stored in the highest location on the stack and R4 is in the lowest.

The PSP is then updated to reflect the new top of stack and R0 has the address of PSP for use in the *SVCHandler* (as its first argument); note that the BL (branch with link) instruction uses LR, which is why it was saved on the stack at the outset (i.e., it is a volatile register):

```
__asm("      msr    psp,r0");
__asm("      BL     SVCHandler");              /* r0 Is PSP */
```

Upon return, registers R4 through R11 must be removed from the stack explicitly, in this example using *LDMIA* and R0, starting with the R4 in the lowest address through R11 in the highest. R0 now contains the new stack top, which means that it must be copied to the PSP:

```
__asm("      mrs    r0,psp");
__asm("      ldmia  r0!,{r4-r11}");        /* Load multiple, increment after */
__asm("      msr    psp,r0");
```

Finally, the return address (the *EXC_RETURN* value of 0xFFFF.FFFD pushed onto the stack at the outset) must be pulled from the stack and put into the PC; this will cause the trapping function registers to be pulled from the PSP:

```
__asm("      POP    {PC}");
```

4 Passing arguments to the kernel

Arguments can be passed to the kernel in a number of ways; for example, as globals, register values, or on the stack. Since globals can lead to other problems and explicitly accessing the Cortex registers can be a chore, this section explains how the trapping function's stack (either PSP or MSP) can be used to both pass arguments and take return values.

When an interrupt occurs, the kernel has access to the trapping function's stack and any data on it. The location of the data is known since it starts immediately after the PSR register on the trapping stack (see Figure 3).

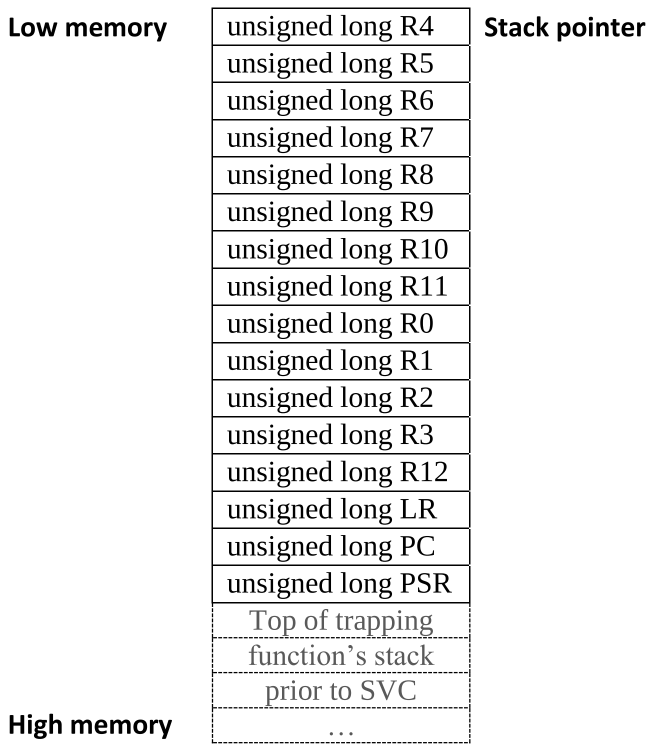

Figure 3: Stack image after call to *SVCHandler()* **(from** *SVCall()***)**

The address of the top of the trapping function's stack prior to the SVC trap is known to the kernel: it is the trapping function's saved stack pointer plus an offset of 16 long words (to bypass the saved registers).

In order that the trapping function and kernel refer to the same memory structure, it is advisable to create a common structure used by both. For example, the following structure allows the trapper to pass a code and two arguments to the kernel:

```
struct kcallargs
{
unsigned int code;          /* Command to kernel */
unsigned int rtnvalue;      /* Return value from kernel */
unsigned int arg1;          /* First argument (if any) */
unsigned int arg2;          /* Other arguments required by command */
};
```

If the trapping function creates an automatic variable of type *struct kcallargs*, it will appear on the trapping stack; for example (note that to ensure the compiler actually creates the space on the stack for the variable, it is necessary to indicate that it is *volatile*):

```
int trapperkernelcall()
{
volatile struct kcallargs tkcargs;
tkc . code = TKC_CODE;
__asm("     SVC    #0");
…
}
```

Note that although the compiler attempts to maximize its use of registers (and not the stack) when dealing with automatics, a preponderance of automatic variables may mean that the address of the argument structure could be in a memory location other than the one expected (i.e., at the top of the caller's stack).

The trapping stack immediately after registers R4 through R11 are placed on it is shown in Figure 4; note that the stack pointer is 16 long words "higher" than the stack pointer (pointed to by R0 by *SVCall()* prior to calling *SVCHandler()*).

Low memory	*unsigned long R4*	**Stack pointer**
	unsigned long R5	
	...	
	unsigned long PC	
	unsigned long PSR	
	...	**Stack pointer + 16 words**
	unsigned int code	**Base address of tkcargs**
	unsigned int rtnval	
	unsigned int arg1	
High memory	*unsigned int arg2*	

Figure 4: Trapped stack frame prior to call to *SVCHandler()*

Since the value(s) stored on the stack starting at address "Stack pointer + 16 words" may or may not coincide with the base address of *tkcargs***,** it is necessary to find a structure that will not change when the call takes place.

Fortunately, this is quite simple – the one structure (or set of structures) that remains constant after the *SVC()* are the process's registers since they are pushed onto the stack. By assigning the address of the structure being passed to the kernel to a seldom-used register, the kernel can access the register from the stack and, from that, the contents of the structure.

In the following example, R7 is used to pass the address of the structure to the kernel. Assigning to a register requires the use of assembler code and, most easily, a separate function to do the operation (passing the address of the structure in R0, the first argument to any function; note the use of *volatile* to ensure that R0 isn't optimized out by the compiler)):

```
void assignR7(volatile unsigned long data)
{
/* Assign 'data' to R7; since the first argument is R0, this is
 * simply a MOV from R0 to R7
 */
 __asm("        mov r7,r0");
}
```

Next, the code calling the kernel must be modified; for example:

```
int nice(unsigned priority)
{
volatile struct kcallargs getidarg;
getidarg . code = NICE;
getidarg . arg1 = priority;
assignR7((unsigned long) &getidarg)
SVC();
}
```

The code inside the SVC handler must also be modified to allow access to R7. Since the *SVCall()* entry point already pushes all registers onto the process stack (in addition to the registers pushed by the hardware because of the interrupt), the PSP points to the top of stack. At present, the call to *SVCHandler()* is supplied with the address of the top of the process stack; by changing the type of the argument *(*argptr)* to *struct stack_frame*, the code can the access R7 (and hence, the address of the structure):

```
void SVCHandler(struct stack_frame *argptr)
{
/*
 - Service call entry point
 - Should be using the MSP
 - Handle startup
*/
static int firstSVCcall = TRUE;
struct kcallargs *kcaptr;

if (firstSVCcall)
{
        …
}
else /* Subsequent SVCs */
{
        /* Get address of process structure from r7: */
        kcaptr = (struct kcallargs *) argptr -> r7;
        /* With this, the different fields can be accessed */
        switch (kcaptr -> code)
        {
        …
        case NICE:
                new_priority = kcaptr -> arg1;
                …
                kcaptr -> rtnvalue = result;
        break;
        …
        }
}
}
```

5 Returning values from the kernel

Any changes the kernel makes to the data on the trapping stack will be visible to the trapping function when control returns to the function. For example, if the *rtnvalue* in *kcallargs* was changed as follows:

```
kcaptr -> rtnvalue = result;
```

It could be retrieved in the trapping function by accessing the field *rtnvalue*:

```
int trapperkernelcall()
{
volatile struct kcallargs tkcargs;
tkc . code = TKC_CODE;
__asm("         SVC     #0");
return tkc . rtnvalue;
}
```

Finally, note that if a pointer to another data structure, such as a string, is passed as an argument, the kernel can access the structure by copying the argument into a pointer and using it directly.