



UNIVERSITY
of York

SCHOOL OF PHYSICS,
ENGINEERING AND TECHNOLOGY

Systems Programming for ARM Laboratory Scripts

Dr. Andy Pomfret

Laboratory 9

DocetOS, SVC, Stacks and Privilege

9.1	Aims	93
9.2	DocetOS	93
9.2.1	Structure	94
9.3	Starting Out	95
9.3.1	Initialising DocetOS	95
9.3.2	Fixing the Code	95
9.3.3	More Tasks	98
9.4	Reporting Privilege, Stack and Mode	98
9.5	Adding a Software Interrupt	100
9.5.1	A No-Argument Delegate	100
9.5.2	Adding Arguments	101
9.6	Summary and Conclusions	102

9.1 Aims

In this laboratory session, you will:

- Learn about DocetOS, the basic operating system designed for this module
- Suppress some code warnings
- Add software interrupt handlers to DocetOS
- Experiment with privileged and unprivileged code

9.2 DocetOS

There won't be much to do in this section, but it's important to read through it so that you understand the structure and operation of DocetOS. You'll be expected to work with its source code, so you will need to know it quite well.

⚠ *DocetOS is not a mature platform, and has only been developed for teaching on this module. It is quite possible that bugs and limitations will be found that necessitate changes to the functionality or API of the OS. The information in this section should be considered subject to change at any time.*⁸



Use **Project→Open Project...** to open `DocetOS\DocetOS_base\DocetOS.uvprojx`. The following subsections will make more sense if you have a project open in front of you.

You will be working with this same project for the whole of this lab session, and likely for the rest of the semester, so **keep your copy safe!**

First, **build the code**. You should find some warnings in the build console. Some of these relate to a feature you'll need to implement soon, whereas others are harmless — but your code should not build with warnings, so we'll look at some ways to suppress these soon.

9.2.1 Structure

DocetOS is a very simple operating system kernel. It essentially consists of a context switcher, and the necessary setup routines to establish an idle task, switch stacks, and successfully make the first switch. The scheduler itself is a simple round-robin type.

The entire operating system is contained in three source code files: `os.c` and `os_asm.s` provide the core functionality, and `scheduler.c` contains the scheduler. A small number of header files are also provided. When looking at the source code, note the naming conventions:

- Most externally-visible C functions, variables and types are prefixed with `os_`, for example `os_init()`
- Variables and types intended only for internal use within the OS, whether static or externally-visible, are prefixed with an underscore
- Types are postfixed with `_t`, for example `os_tcb_t`

Each running task has a task control block (`os_tcb_t`) that holds its stack pointer when it's not running, and various other bits of information about the task. You can find the structure definition in `scheduler.h`.

You'll get to know the internal workings of DocetOS better over the next couple of weeks. For now, we will have a look at the example 'round robin' scheduler.

⁸Maybe `volatile`? Sorry. Poor joke.

9.3 Starting Out

Turn your attention to `main.c`. There are two task functions defined. Two TCBs are defined, and aligned memory for two 128-word stacks is allocated too (remember, stack pointers must usually be 8-byte aligned on this platform).

Note that the stacks and TCBs are allocated statically. If they weren't, then the storage space for the two task stacks would be allocated on the main (handler mode) stack, which can get confusing!

9.3.1 Initialising DocetOS



Add code to initialise the OS and two TCBs, add them to the scheduler, and start the OS.

- Add the line

```
OS_initialiseTCB(&TCB1, stack1+128, task1, NULL);
```

to the code. Look at the prototype for `OS_initialiseTCB()` in `scheduler.h`, and make sure you understand the arguments that are passed and why they're passed in this way. In particular, why is the value 128 added to the `stack1` pointer? What is the final argument for?

- Add a similar line to initialise the TCB for task 2. Remember to give it a different stack!
- Add two calls to `OS_addTask()` to add both your new tasks to the scheduler. `OS_addTask()` is actually a software interrupt handler (see the prototype with the `__svc` attribute in `os.h`, and the handler itself – called `_svc_OS_addTask()` – in `os.c`). This handler invokes a callback to the scheduler; see `simpleRoundRobin.c` to find the 'add task' callback for the simple round robin scheduler, and to find the function argument that you need.⁸
- Finally, call `OS_start()`. This call will start the scheduler and never return.

Build and test your code. Does it do what you expect?

9.3.2 Fixing the Code



Take a closer look at the two task functions in `main.c`. The first runs for 1000 iterations, while the second runs forever. And yet, when you run the code, *both* tasks stop running after a few seconds. This doesn't seem right — and it's not. Let's use the debugger to find out what's going on (and to find out why debugging this sort of code is hard!). Along the way you'll get some exposure to the structure of DocetOS.

⁸This means that the scheduler callback is run in handler mode, with privilege. The security implications of this should be obvious, and are among the many reasons why a production OS would not use dynamic linking of its scheduler!

1. To start with, run the code in the debugger until it halts. You should find that it's stopped at the line that says

```
__BKPT(0);
```

inside the `Hardfault_HandlerC()` function in `hardfault.c`. If you're thinking "that doesn't sound good", then you're right. Ending up in the hard fault handler is never great, but let's work out how we got here.

2. On entry to the hard fault handler, the CPU will have pushed `r0-r3`, `r12`, `lr`, `pc` and `psr` to the active stack. The handler works out which stack was active and then unpacks these pushed values for you to inspect.

Try hovering the mouse over `stacked_pc`. That's often a good first clue because that's the address of the instruction that the CPU was attempting to execute when the fault occurred. But in this case its value is `0x00000000`. Well, "there's your problem", as they say, but unfortunately we're no closer to knowing *why* the CPU was trying to execute code from that location.

3. The next best clue is often to be found from the stacked value of `lr`. This generally tells us *where the code would next have returned to* at the end of the problem function, though depending on the nature of the fault it can also tell us *where the code last returned to*. The value you see in the stacked link register may vary, but in my testing I observed a value of `0x08002169`. The LSB is the Thumb bit, so this corresponds to an address of `0x08002168`. Using the scrollbar on the disassembly window, find the instruction at the address that you observe in the stacked `lr` value. You should find a line like

```
POP    {r7, pc}
```

and looking in the source window you'll see that this is an implicit return at the end of the `_OS_task_end()` function.

4. The author of this code has been kind enough to add helpful comments above and within this function. It's clear that it's invoked when a task ends, as Task 1 does after 1000 iterations. For some reason when the function terminates the program counter value that it tries to pop off the stack is zero.

The comments in the function make it clear that a context switch should be triggered by the call to `_OS_task_exit()`, so maybe the clue we need is there. Recall that the context switch is contained within the handler for the PendSV software interrupt.

5. Set a breakpoint in `_OS_text_end()` on the line that calls `_OS_task_exit()`. Reset the debugger and run to the breakpoint. Now open `os_asm.s` and find the `PendSV_Handler` label; place a breakpoint just after it, and run to the breakpoint again. You should now be inside the context switch, shortly before the crash.
6. Step through the handler, and step into the scheduler when it's called by the line

```
BL    _OS_schedule
```

Once inside the scheduler, right-click the `task_list` variable and select Add ‘task_list’ to→Watch 1. This will let you examine the scheduler’s task list in the watch window.

In my case — your numbers may differ — the task at the head of the list has its TCB at address `0x20000224`. The other TCB, which is joined to the first by the `next` and `prev` pointers, is at address `0x2000234`. The call to `_OS_schedule()` will step through the list so the TCB at `0x2000234` will become the head, and this will then be returned as the task to be run next.

From this we can infer that **the TCB that was at the head of the list on entry (`0x20000224` in my case) is the TCB of the task that was running when the context switch was invoked**. It is therefore the TCB of the task that is in the process of ending. The task that is returned by the scheduler is the task that is *not* ending, so that should be safe.

7. Allow the debugger to run to the next breakpoint again. You should find yourself back in the context switch, which indicates that the last switch was indeed safe and did not trigger the bug. This time, however, if you step into the scheduler you should find that the TCB it returns *is the TCB of the task that has finished execution*. This should not happen. If you continue to step through the context switch you will find that at the end it returns to `_OS_task_end()`, and then the hard fault handler is triggered.

- ❗ So, what have we learned? We have learned that the crash is triggered by an attempt to switch context to a task that has ended, and that should no longer be runnable. For some reason the task that is ending is not removed from the scheduler as it should be.

A glance at the static `list_remove()` function in `scheduler.c` (just above the scheduler function itself) will reveal the answer: the removal of tasks from the task list is not yet implemented. That’s also the source of two of the warnings you see after a full build — the parameters `list` and `task` are unused.

- ⚙ Write this function. You’ve done this before; this is just a circular doubly-linked list, exactly like the one you wrote in Laboratory 4 (Section 4.5.1). Build and test the code, and confirm that it no longer crashes when one task finishes.


9.3.2.1 Other Warnings


If you perform a full build of the code, you should see two warnings reported in `main.c` about an “unused parameter” in each of the task functions. DocetOS has the facility to pass a parameter to a task function when it starts, but neither of the tasks uses this parameter, which is what generates the warning.

- ⚙ The idiomatic way to suppress these warnings if they are known to be safe, as they are here, is a “cast to `void`”. Try adding a line like


```
(void) args;
```

to the top of each task function. Try building the code and check that the warnings are gone.

 This works because the parameter is no longer “unused”, because it forms part of a simple statement (actually just an expression). That statement doesn’t do anything, but nor does it generate any errors or warnings because the cast to `void` effectively suppresses any result from the expression.

 Different versions of the ARM Compiler 6 may also report other warnings when you perform a full rebuild. If you get other warnings **you must remove them**, either by modifying the code to address the root cause of the warning, adding qualifiers or casts to suppress the warnings, or (as a last resort) disabling the warnings locally or at the file scope. Please ask for help if you are seeing warnings and would like to get rid of them. Your code should not build with warnings!


9.3.3 More Tasks

 Add a third task, that will print out a different letter of the alphabet. Allocate a stack and TCB for it, and add it to the scheduler as before.

Check that your modification does what you expect.

9.4 Reporting Privilege, Stack and Mode

Your next task will be to write some functions that allow you to discover what mode (thread or handler) the CPU is in, which stack (MSP or PSP) is in use, and whether the running code is privileged or unprivileged.

 Write a function to return the value of the Processor Status Register (PSR). You will need to use an `MRS` instruction for this. You could either write a standalone assembly language function, or you could use inline assembly language.

- Create two new files, called `mode_utils.c` and `mode_utils.h`, in the `src` and `inc` folders respectively. If you’ve chosen to write a standalone assembly language function, also create `mode_utils.s` in the `src` folder. Add `mode_utils.c` (and `mode_utils_asm.s` if appropriate) to a source file group – perhaps creating a new group for these would be a good idea.
- Add the lines

```
#ifndef MODE_UTILS_H
#define MODE_UTILS_H
```

to the top of `mode_utils.h` and

```
#endif /* MODE_UTILS_H */
```


to the end. You will learn more about these ‘guards’ soon if you don’t already know about them. Everything you add to this header file should go between the `#define` and the `#endif`.

- If writing a standalone function, create an assembly-language function in `utils_asm.s` that will use the `MRS` instruction to read the Processor Status Register (PSR) and return it in `r0`. Call your function `getPSR()`. Note that you will need to use an assembler directive like

```
AREA utils, CODE, READONLY
```

at the top of your file. Also remember to include the `END` directive at the end, and to export the `getPSR` symbol.

- If using inline assembly language, instead create a C function called `getPSR()`, use inline assembly language to get the PSR value using an `MRS` instruction, and return it. **Note that the inline assembler requires a different mnemonic for the PSR (`xPSR`).**
- Create a prototype for `getPSR()` in `mode_utils.h`. Your function does not take any parameters, and it returns a `uint32_t`. (Remember to include `stdint.h` in any files that use these types.)
- Include `mode_utils.h` in `mode_utils.c`.

Once you’re sure this works, you can write another similar function to return the value of the CONTROL register and then put these together to create a function that reports the mode of the CPU.

- Create another function that will use the `MRS` instruction to read and return the CONTROL register. Call your function `getCONTROL()`.⁸
- Write a C function in `mode_utils.c` called `reportState()`. It should:
 - Call `getPSR()` to obtain the PSR.
 - Check bits 8:0 (note, that’s 9 bits) to see whether the CPU is in thread mode or handler mode. This is the current ISR number, so all zeros here indicates thread mode, and anything else indicates handler mode.
 - Call `getCONTROL()` to obtain the CONTROL register.
 - Check bit 1 (`SPSEL`) to find out whether the MSP (0) or the PSP (1) is the active stack pointer.
 - Check bit 0 (`nPRIV`) to find out whether thread mode code is privileged (0 means yes, 1 means no).
 - Print out a status message, looking something like this:
Thread mode, privileged, PSP in use

⁸There are actually CMSIS compiler extensions that do these things already – but writing your own in assembly language is a good way to practice your coding.

Remember that handler-mode code is always privileged, but for thread-mode code you'll have to look at the `nPRIV` bit.

- Finally, add a prototype for `reportState()` to `mode_utils.h`.



Try calling your new function from inside `main()` before the scheduler is started, and again from inside one of the task functions.

Build and run the project. Does it do as you expect? You may wish to find a way to prevent the other task from interrupting the one that's calling `reportState()` – for example, by simply disabling it temporarily – so that you can read the text.

9.5 Adding a Software Interrupt

You will remember from the lectures that the software interrupt handler (`SVC_Handler`), defined in `os_asm.s`, does a number of things in order to take the correct action. These are:

- It uses the return code stored in `lr` to determine whether the MSP or the PSP was in use at the time of the interrupt;
- It places the appropriate stack pointer in `r0`;
- It ensures that the handler index in `r7` is within the bounds of the array of jump labels;
- It uses an `LDR pc` instruction to jump to the appropriate function, known as a *delegate*.

On entry to the delegate, `r0` contains a copy of either the MSP or the PSP, as appropriate, so that the delegate can extract data from the registers that were pushed to the stack on entry to the SVC handler.

- ① See `os_asm.s` for the software interrupt handler. Delegates can be found in `OS.c` and `scheduler.c`. None of the delegates actually uses the stack pointer that they're passed, *yet*. You'll soon have the joy of creating one that does.

9.5.1 A No-Argument Delegate



Try adding your own software interrupt delegate. To do this, you will need to:

- Write the delegate. You can call it anything you like, though it might be a good idea to stick to the naming convention that the other delegates use (ending the name with `_delegate`). You can also make it do anything you like – a call to your `reportState()` function might be a good idea, so that you can confirm that the delegate is executing in privileged handler mode and is using the MSP. Declare your delegate so that it takes no arguments. Put your delegate in `mode_utils.c`

along with the `reportState()` function. To suppress the warning, you should also prototype your delegate – but this can be done *inside the C file* because it is not intended that any C code will call the delegate directly, so it does not need to be prototyped in the header file.

- Choose and define an SVC number. The easiest way to do this is to add a new value to the `OS_SVC_e` enumeration in `os.h`. If you add a new constant name to the end, it will automatically be assigned the next available number. Name your constant starting `SVC_`, but don't start it with `OS_SVC_` because the new SVC delegate is not part of the OS proper.
- Add the symbol for your delegate to the end of the jump table in the SVC handler (in `OS_asm.s`). Make sure that the symbols in the jump table are in the same order as the enumeration constants in `os.h`. Remember to import the symbol.
- Include `OS/os.h` in `mode_utils.h`, and then declare an SVC prototype. For an example, see the following line in `os.h`:

```
#define OS_yield() _svc_0(OS_SVC_YIELD)
```

This causes the preprocessor to replace calls to `OS_yield()` with calls to `_svc_0()`, which places the value of the supplied constant into `r7` and then issues an `SVC` instruction. You can find the definition of this macro in `os.h`. Other functions, such as `_svc_1()`, and available which also ensure that the parameters they're passed are placed in the correct registers before the `SVC` instruction is executed. Write your own SVC prototype following this example.

- Call your SVC function from somewhere (perhaps from a task).

Build and run your code. Is your SVC delegate executed when the prototype is called? Does it run with the mode and privileges you expect?

- ① Note how (if your code is running properly!) the `printf()` strings from the two tasks interrupt each other – but the `printf()` string printed by the `reportState()` function from the SVC delegate is not interrupted. This shows that it is running with a higher priority than the thread-mode tasks.

9.5.2 Adding Arguments

- ① Adding arguments to a SVC function is a two-step process. Firstly the SVC prototype must be modified to declare the arguments. Secondly, the delegate must be modified to accept and return the arguments. But *this probably doesn't work like you expect*.

Remember that your SVC delegate is not 'called' like a normal function. When you call `OS_yield()`, for example, you are not directly calling the `_OS_yield_delegate()` delegate. Instead, when you use the preprocessor to emulate a function call and generate an SVC call, the SVC helper function (e.g. `_svc_2`) will put the arguments into `r0-r3`, put the

desired SVC delegate number into `r7`, and then generate an `SVC` instruction. This will automatically push `r0-r3`, `r12`, `pc`, `lr` and `PSR` to the stack, change modes and stack pointers as necessary, and invoke the SVC handler which in turn will call the delegate. The stack pointer will be passed to the delegate in `r0`.

So to get to the arguments that were passed to the SVC prototype, the delegate must go and find them on the stack. By treating the stack pointer as a pointer to an `_OS_SVC_StackFrame_t` structure type (declared in `os.h`), the delegate can access the stacked registers by name.

Immediately after the SVC delegate returns, the SVC handler will also return, causing the automatically-pushed registers to be popped. Anything that the delegate tried to return by putting it in `r0` would be overwritten by this automatic pop. But anything the delegate had placed in the stacked `r0` would be popped to `r0` – and so this is how a delegate can return a value.



Change your delegate so that it takes an argument of type `uint32_t` and returns another `uint32_t`. To do this:

- Change the SVC prototype so that the delegate takes and returns the correct arguments.
- Change the delegate itself:
 - Add the line `#define OS_INTERNAL` in `mode_utils.c`, above the import of `os.h`. This will give you access to the useful `_OS_SVC_StackFrame_t` structure type.
 - Change your delegate so that it takes a single argument called `stack`, of type `_OS_SVC_StackFrame_t * const`.
 - Add a line to the delegate so that it will print out the value of the `uint32_t` argument that it has been passed. Remember that this can be found on the stack, so you can use `stack->r0` to get at the value.
 - Add another line to *modify* the stacked value of `r0`, for example you could double it or increment it.
 - Modify your call to the SVC function so that it passes in a value. You will need to use the `_svc_1` helper because there is intended to be one argument. The SVC number should be supplied as the *last* parameter to the helper. Add another line of code to print out the returned value.

Build and run your code. Does it behave as you expect?

9.6 Summary and Conclusions

By now, you should:

- Know about the basic structure of DocetOS

- Understand how the simple round-robin scheduler works
- Understand how mode switches relate to privileged and unprivileged code, and to the use of the PSP and MSP
- Know how to write an SVC delegate for the DocetOS SVC handler, and to call it using an SVC prototype
- Know how arguments can be passed into and out of an SVC delegate via the stack
- See the impact of stack overflow on embedded systems
- Have an understanding of the difficulty of debugging at the systems level

If you have any questions about what you have learned, please ask.

Remember to copy your project files to a network drive before you log off. This is particularly important this week!