# EMBSYS 110 Module 8

Design and Optimization of Embedded and Real-time Systems

# 1 Debugging and Profiling

## 1.1 Logging

"Printf" is probably the most popular debugging tools. It is easy to use and does not seems to affect the normal flow of the system being debugging.

With some enhancements, "printf" can evolve into a much more powering logging service. It is non-trivial and is best done upfront before any real features are developed.

Our demo project presents a simple and practical logging system supporting:

1. Multiple verbosity level, including:

   INFO, LOG, CRITICAL, WARNING and ERROR

2. Enabling and disabling of the log output of individual components (HSMs).

The following macros are defined to output log messages to the console:

```
#define PRINT(format_, ...)      Log::Print(HSM_UNDEF, format_, ## __VA_ARGS__)
#define EVENT(e_)                Log::Event(Log::TYPE_LOG, me->GetHsm().GetHsmn(),
                                            e_, __FUNCTION__);
#define INFO(format_, ...)       Log::Debug(Log::TYPE_INFO, me->GetHsm().GetHsmn(),
                                            format_, ## __VA_ARGS__)
#define LOG(format_, ...)        Log::Debug(Log::TYPE_LOG, me->GetHsm().GetHsmn(),
                                            format_, ## __VA_ARGS__)
#define CRITICAL(format_, ...)   Log::Debug(Log::TYPE_CRITICAL, me->GetHsm().GetHsmn(),
                                            format_, ## __VA_ARGS__)
#define WARNING(format_, ...)    Log::Debug(Log::TYPE_WARNING, me->GetHsm().GetHsmn(),
                                            format_, ## __VA_ARGS__)
#define ERROR(format_, ...)      Log::Debug(Log::TYPE_ERROR, me->GetHsm().GetHsmn(),
                                            format_, ## __VA_ARGS__)
```

The global verbosity level and log output enabling/disabling for each HSM can be controlled at runtime through the "log" command.

Since logging needs to be used by multiple HSMs and it would be cumbersome to post an event for each log message, our framework provides a functional logging API in fw_log.h. It allows an HSM to be registered as an *output interface*. The registration API looks like this:

```
static void AddInterface(Hsmn infHsmn, Fifo *fifo, QP::QSignal sig,
                         bool isDefault);
static void RemoveInterface(Hsmn infHsmn);
```

The parameters to AddInterface() specifies the following properties of the HSM being registered as the output interface:

1. HSM number (ID)
2. Pointer to its output FIFO
3. Signal of the write request event

For example, upon initialization the Console active object registers its associated UART output region named UART2_OUT as the default logging interface:

```
Log::AddInterface(UART2_OUT, &me->m_outFifo, UART_OUT_WRITE_REQ, true);
```

From now on, any log messages written via the macros LOG(), EVENT(), PRINT(), etc will be sent to UART2_OUT.

Note – The macro EVENT(e) provides a convenient means to log any events processed by an HSM. This alone is sometimes enough to trace its operation.

## 1.2 Interactive Console

The design goal of the debug console is to make it convenient for developers to add their own debug commands. It has the following features:

1. It supports a hierarchy of command tables. The root command table is defined in Console/ConsoleCmd.cpp. Additional command tables can be added for individual HSMs and can be navigated from the root command table.
2. Each command handler is by itself an event handler – as you would expect from a fully event-driven system. It runs in the context of the Console active object. As a result, not only can it perform certain actions when a command is invoked, it can also respond to subsequent events posted to Console. This allows a command handler to perform complex interaction with other system components for integration testing or debugging. (E.g. Sending request A and upon receiving confirmation B then sending request C, etc.)

   Note - The current command handler will be deactivated when it returns CMD_DONE or when the user hits ENTER. You can try it out with the "timer" command.

The following is an example showing how a test command can be used to compare performance between the block memory allocator in QP and the standard C++ shared_ptr memory allocation.

```
static CmdStatus Perf(Console &console, Evt const *e) {
    switch (e->sig) {
        case Console::CONSOLE_CMD: {
            uint32_t startMs = GetSystemMs();
            const uint32_t TEST_CNT = 100000;
            const uint16_t TEST_SIZE = 32;
            for (uint32_t i = 0; i < TEST_CNT; i++) {
                QEvt *evt = QF::newX_(TEST_SIZE, 0, 0);
                evt->sig = i;
                QF::gc(evt);
            }
            console.Print("Elapsed time with QF = %d\n\r",
                          GetSystemMs() - startMs);

            startMs = GetSystemMs();
            for (uint32_t i = 0; i < TEST_CNT; i++) {
                auto evt = std::make_shared<QEvt>(0);
                evt->sig = i;
            }
            console.Print("Elapsed time with new = %d\n\r",
                          GetSystemMs() - startMs);
            break;
        }
    }
    return CMD_DONE;
}
```

The result shows:

```
1319 CONSOLE_UART2> perf
Elapsed time with QF = 267
Elapsed time with new = 386
```

For any serious projects, a versatile debug console should be the first thing to build before any customer features are implemented. Unfortunately its value is often not recognized and is postponed for too long until it has become difficult to fit it into the architecture.

Can you think of other uses of the console?

Here are some examples:

1. Monitoring CPU utilization.
2. Changing configuration parameters.
3. Memory dump.

4. Monitoring event queue and memory pool statistics (e.g. watermark, current usage.)
5. Instrumenting timing measurements (e.g. interrupt latencies, task wakeup time.)
6. Injecting simulation events or errors to the system for testing.

## 1.3 GPIO

While UART output is very useful for getting an insight into a running system, it is rather intrusive (even with DMA buffer write) as string formatting can be expensive and can affect the timing. For timing critical debugging or profiling it is much more efficient to use GPIO pins.

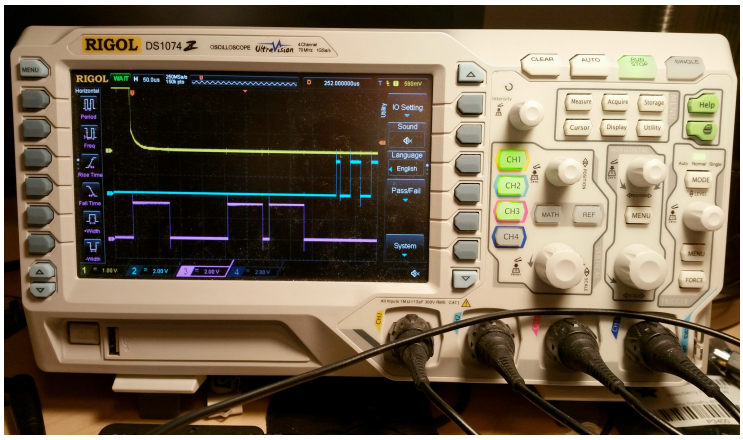
For this purpose we can call the STM32 HAL functions directly, such as the following code using PB.6:

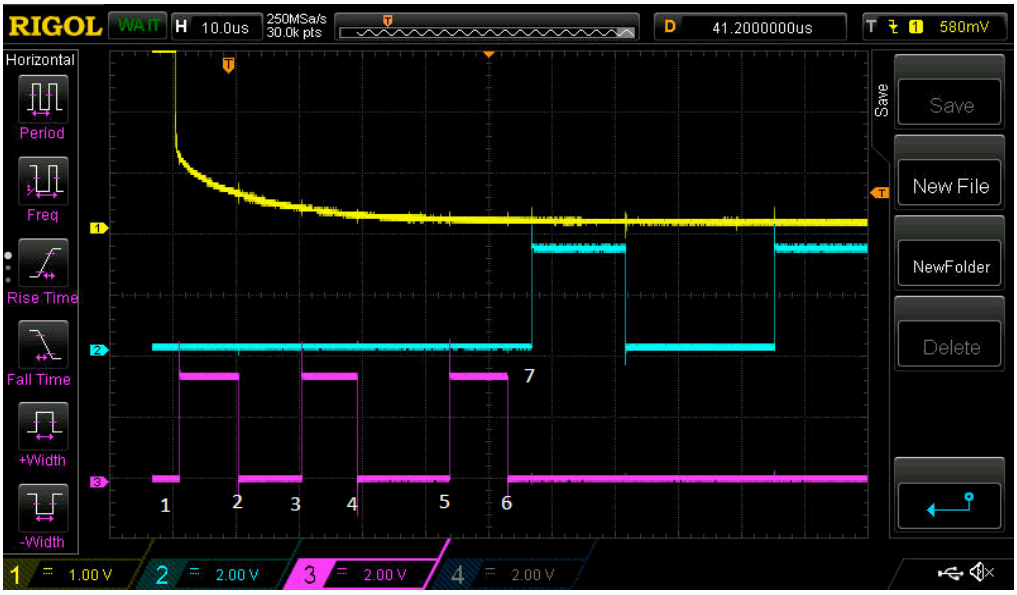
```
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);      // Sets pin high.
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);    // Sets pin low.
HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_6);                   // Toggles pin.
```

Before you can use a pin, you need to initialize it with the following code:

```
__HAL_RCC_GPIOB_CLK_ENABLE();
GPIO_InitTypeDef  GPIO_InitStruct;
GPIO_InitStruct.Mode  = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull  = GPIO_PULLUP;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
GPIO_InitStruct.Pin = GPIO_PIN_6;
HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
```

The following diagrams shows the sequence of events from a button press to an LED being turned on. In this example, the button press was first detected by an ISR which sent an event to the USER_BTN active object. USER_BTN then notified the System active object (coordinator) which sent a request to the USER_LED active object.

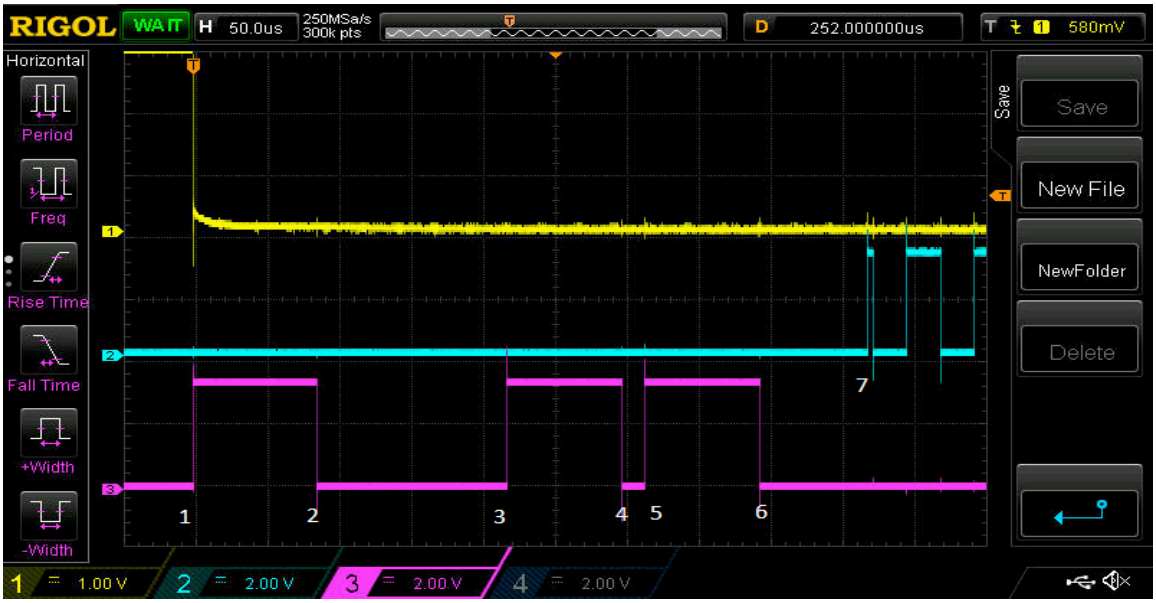





Ch 1 User Button  Ch 2 User LED (PWM)  Ch 3 Debug Output

(1) Button interrupt ISR
(2) USER_BTN_TRIG event in UserBtn::Up state
(3) Publishing USER_BTN_DOWN_IND
(4) System publishing USER_LED_ON_REQ
(5) UserLed  publishing USER_LED_ON_CFM
(6) System got USER_LED_ON_CFM
(7) UserLed turns on PWM

The next diagram shows the same setup but with logging enabled (event logs, entry/exit logs, etc.).

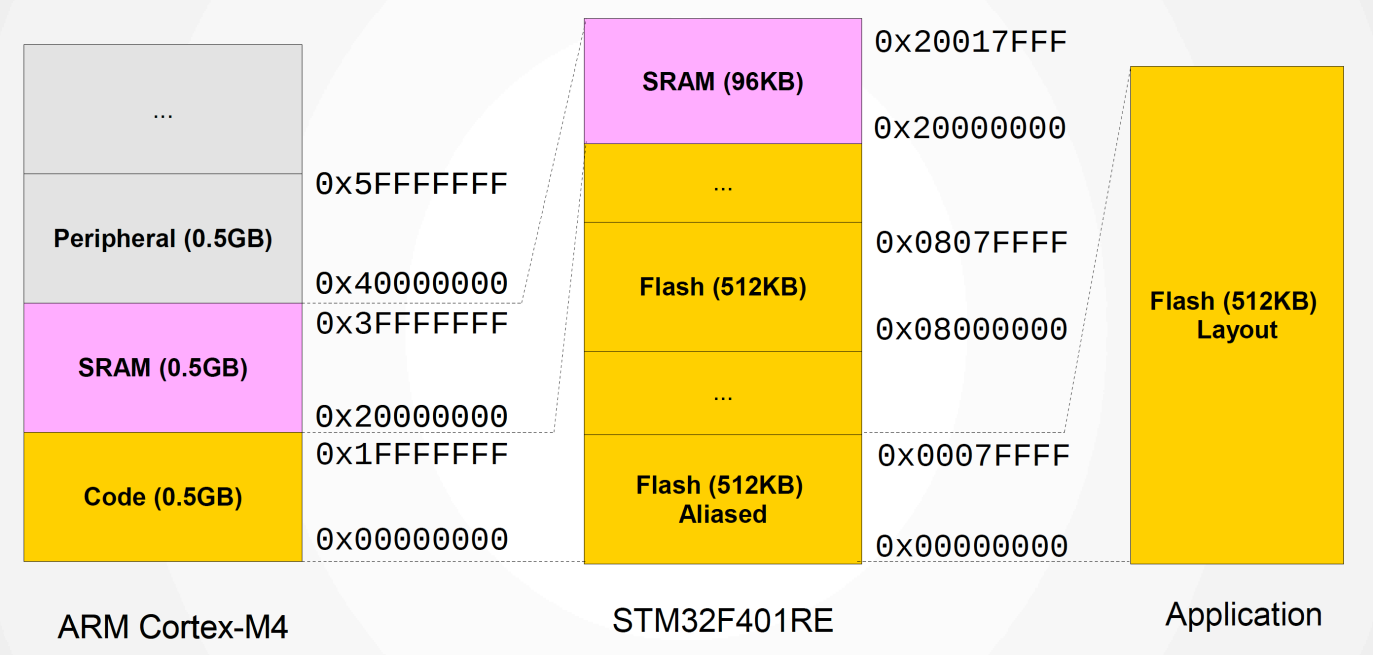


Now you can see how UART logging can slow down the system. The response time from a button press to an LED turning on changes from 55us to 475us.

## 1.4 JTAG

At this point we should all be familiar with the Eclipse IDE along with OpenOCD. Let walk through an example of how to use JTAG to track down a hardware fault or exception. These kind of problems are very difficult to debug, so we should try our best to avoid them from happening in the first place, e.g. using *assertion* (FW_ASSERT()) to catch them before they end up as hardware faults.

First let's remind ourselves of the memory map of STM32F401 processor:

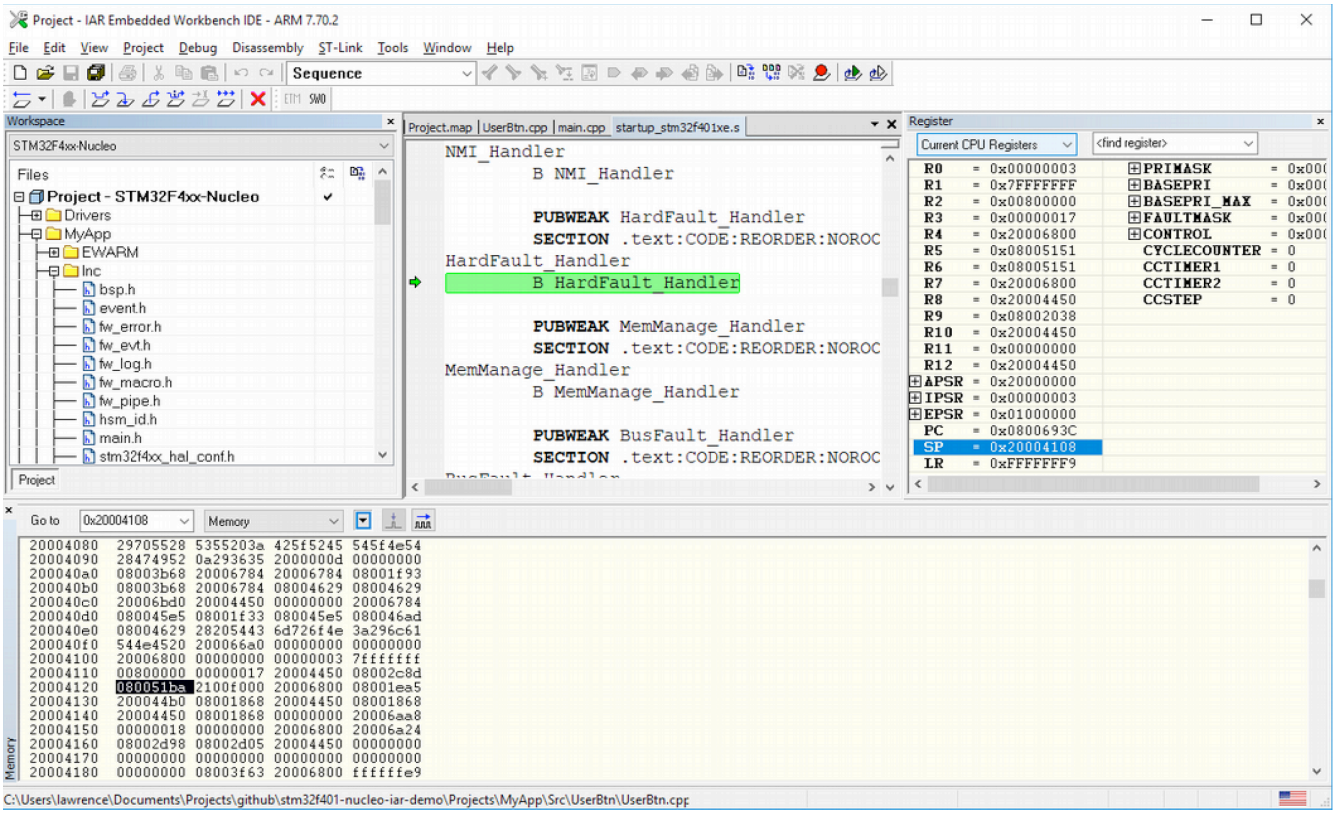



Now let's modify our code to add a deliberate error that would trigger an hardware fault:

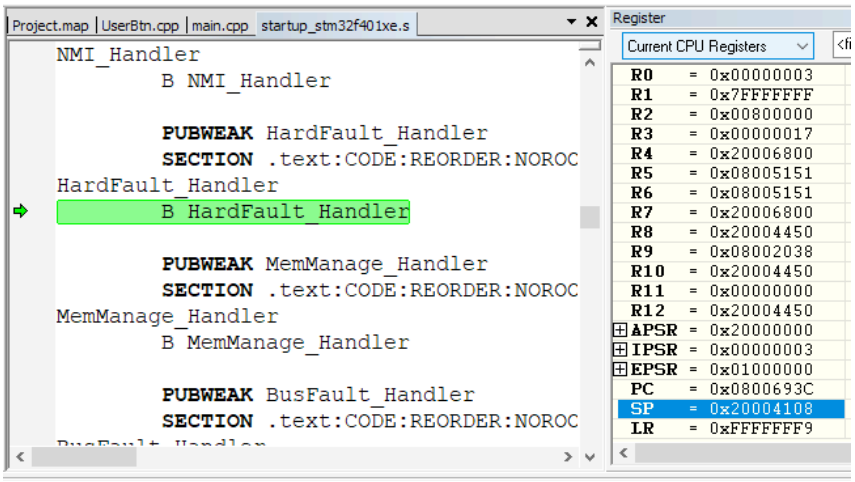
```
QState UserBtn::Up(UserBtn * const me, QEvt const * const e) {
    switch (e->sig) {
        case USER_BTN_TRIG: {
            EVENT(e);
            EnableGpioInt();
            if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_RESET) {
                ...
                *(uint32_t *)(0x7fffffff) = 0x1234;
            }
            return Q_HANDLED();
        }
```

When the user button is pressed, the system hangs. Break the execution in the debugger (IAR in this example) and it will show that it is looping at HardFault_Handler. It means that a hard fault exception has just occurred.

Exception stack frame is automatically pushed to the current stack, the top of which is pointed to by the Stack Pointer (SP). In this example SP is at **0x20004108** as shown in the Register window.

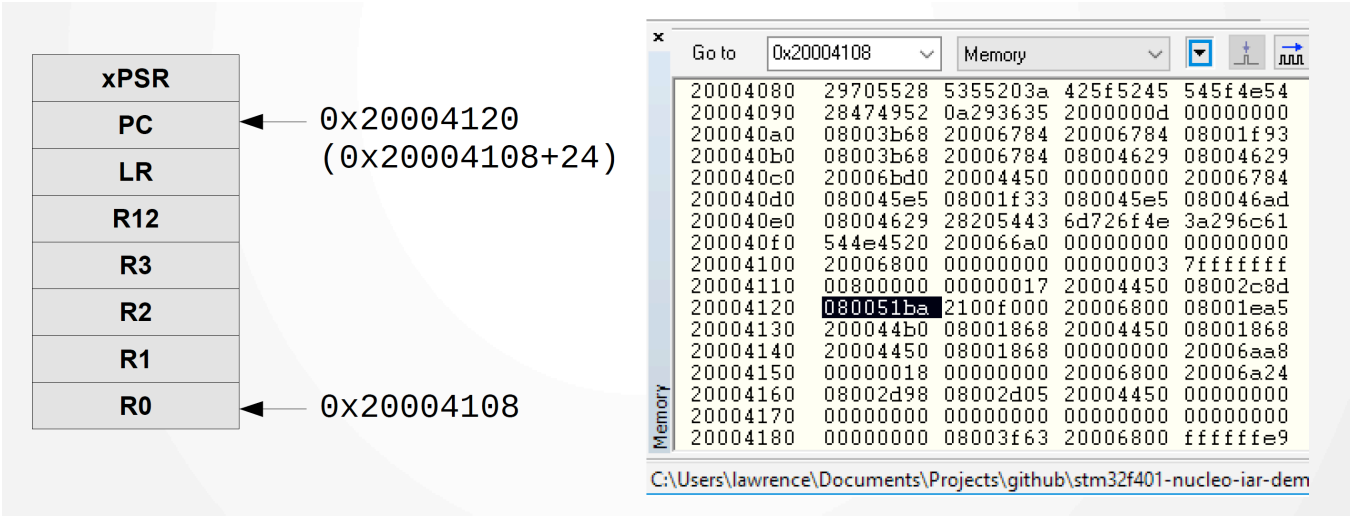



This is a magnified view of the register window:



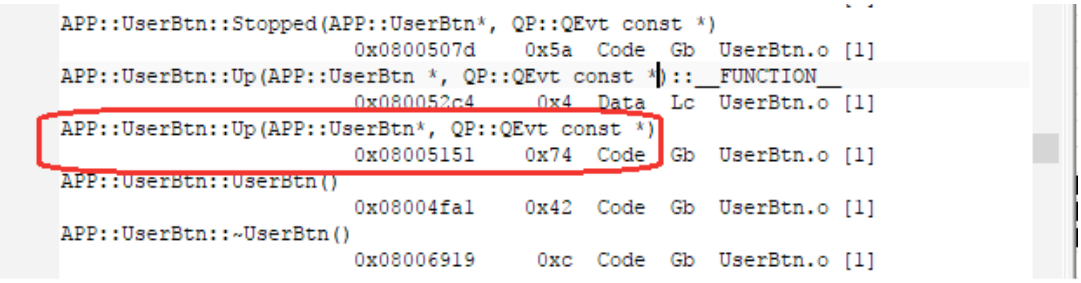
The *exception stack frame* captures the register contents at the time when hard fault occurs. Its layout follows a specific format as shown in the diagram on the left. Using the SP address obtained in the previous step, we should be able to inspect the exception stack frame in the Memory window.

Our calculation shows that the PC (Program Counter) triggering the hard fault is stored at address 0x20004120. From the Memory window it is found to be **0x080051ba**.




This is an important piece of information. Now we know the code at address **0x080051ba** causes the hard fault. The last step is to find out which function this address corresponds to so we can inspect it and fix the bug.

Here we use the map file (stm32f401xe_flash.icf under IAR, or platform-stm32f401-nucleo.map under Eclipse) to narrow it down to the function that encloses this fault address, which is found to be UserBtn::Up().



# 2 Power Management

STM32F401 has three low-power modes:

1. Sleep mode
   - Cortex-M4 with FPU core stopped.
   - Peripherals kept running.
   - Entered whenever no tasks/active object are running (RTOS/QP idle loop). See

     ```
     void QXK::onIdle(void)
     ```

     in bsp.cpp
2. Stop mode
   - All clocks in the 1.2V domain stopped.
   - SRAM and register contents preserved.
   - Entered when system has been idle for some time, but fast wake-up is required.
3. Standby mode
   - Voltage regulator disabled.
   - 1.2V domain powered off.
   - SRAM and register contents lost (except in backup domain).
   - Entered when system is expected to be idle for a long period of time. Wake-up is slow since it requires a full reboot.

Sleep mode is handled *automatically* at the OS level, i.e. in the idle loop when no tasks are ready to run. Stop mode and standby mode are managed by the application which:

1. Determines when the system should enter the low power mode (based on system states.)
2. Ensures the peripherals it controls are properly shutdown (e.g. pending operations are complete) before the internal clocks or voltage regulators are disabled.

A high level coordinator, such as the System active object, is a good candidate to manage low power modes.

Note that with object-oriented design we *try* to encapsulate all hardware resources required by a component to its class. For example we *try* to have UartAct manage all required hardware resources such as GPIO port, UART port and DMA. However the hardware world is less object-oriented, and you will see a single GPIO port or DMA controller being shared by multiple components (e.g. UART and

I2C). As a result, in our project we extract these shared or common hardware control into its own class named Periph in periph.h/cpp.

```
// This is a static class to setup shared peripherals such as TIM, GPIO, etc.
// For dedicated peripherals, they should be setup in the corresponding
// HSM's.
class Periph {
public:
    // The following Setup/Reset functions MUST only be called from one
    // active object.
    // No critical sections are enforced inside them.
    static void SetupNormal();
    static void SetupLowPower();
    // Add more low power modes here if needed.
    static void Reset();
...
```

A high-level coordinator (e.g. System) can then use Periph's API to setup the common clock and power settings for various power modes.

# 3 Optimization

There is not a single absolute best optimization. It depends on what the measurement criterion is:

- Speed.
- Code size.
- Data size.
- Power.
- Development time.

Optimization for different criteria may contradict each other. Ultimately it boils down to meeting requirements. Despite the title of this course, *design* and *optimization* sometimes do not agree.

The goal of software design is not necessarily for optimal performance. It takes into account maintainability and robustness. For example, a common design principle is decoupling (separation of concerns). It divide the system into components (e.g. active objects).

It has the advantages of ease of reasoning, smaller problem to solve, isolation of errors, etc. However it incurs the cost of communication overhead (e.g. event creation and queuing), context switch (if multiple threads are used), etc.

On the contrary, if the goal is for optimal speed performance, it may be faster to sample all sensors in a

tight loop which has the advantages of reducing overhead (e.g. no interrupt, context switching or communication overhead) at the expense of having a less scalable design (i.e. adding more sensors may make the loop too long), wasting system resources in polling and having tight coupling among different types of sensors all in the same loop.

Here are some guidelines:

1. Need trade-off between an elegant architecture and performance. Avoid going to extreme to either side.
2. Optimize critical path.
3. Avoid being  wasteful, but also avoid over-optimizing.
4. Meet the requirements.