EMBSYS 110 Module 9

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 (including unit testing.)

   (How would you fit a synchronous unit test framework into an event-driven system?)

1.3 GPIO Profiling

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.

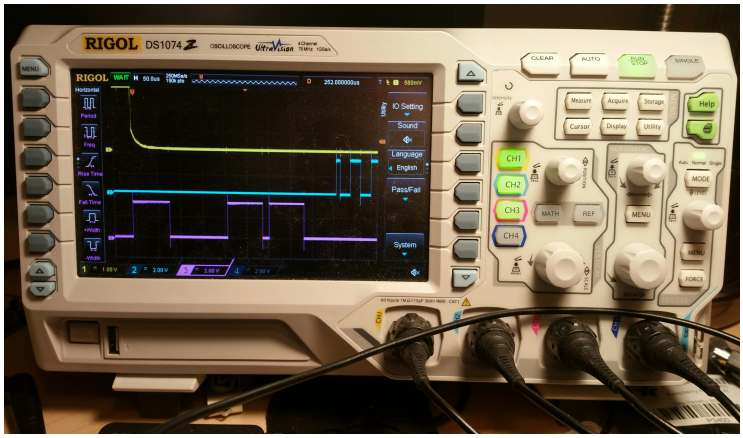
For this purpose we can call the STM32 HAL functions directly, such as the following code using PB.6:

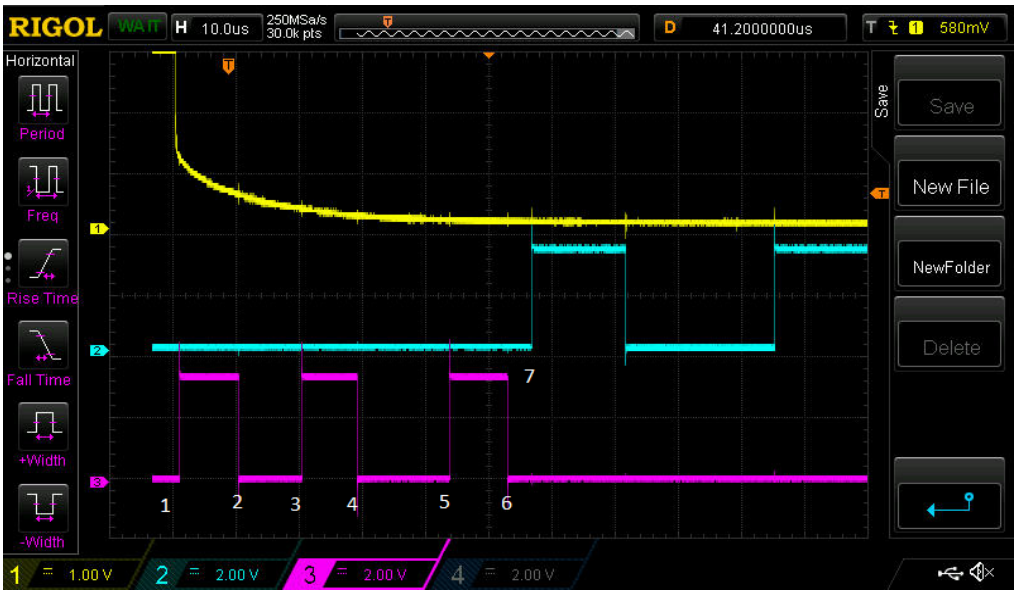
```
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);      // Sets pin high.
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);    // Sets pin low.
HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_6);                   // Toggles pin.
```

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.

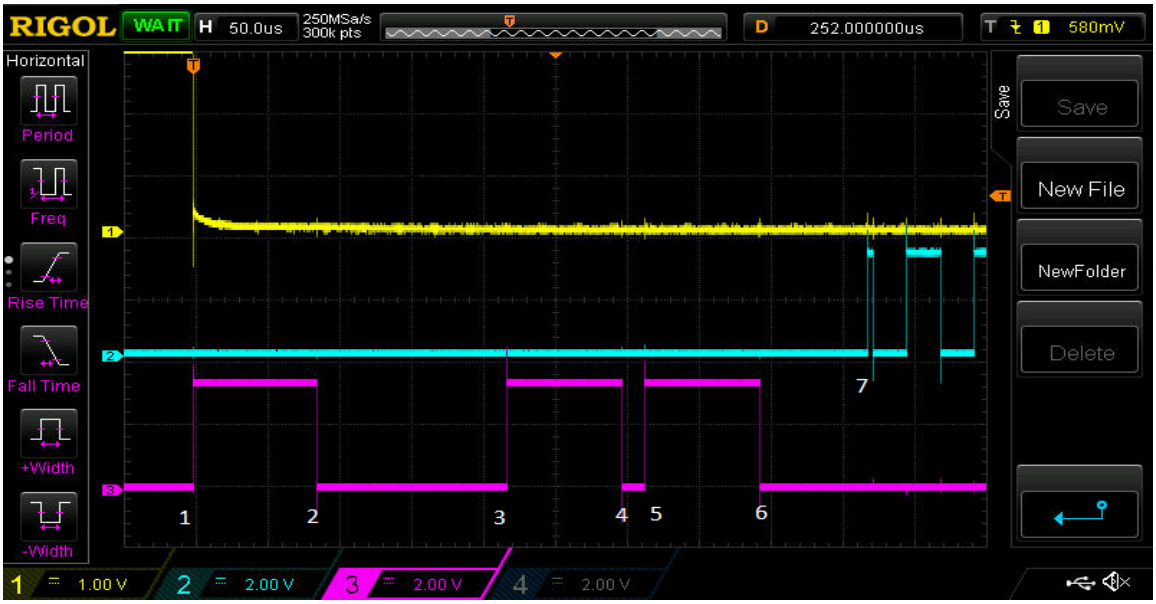




Ch 1 User Button Ch 2 User LED (PWM) Ch 3 Debug Output

(1) GPIO input (PC.13) ISR (EXTI15_10_IRQHandler() in *stm32f4xx_it.cpp*)
(2) TRIGGER event in GpioIn::InActive state
(3) Publishing GPIO_IN_ACTIVE_IND
(4) System publishing USER_LED_PATTERN_REQ
(5) UserLed publishing USER_LED_PATTERN_CFM
(6) System got USER_LED_PATTERN_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.

Discussion: Though it's quick, toggling a GPIO pin takes some time.

(1) How would you find out how much time it take to toggle a GPIO pin?

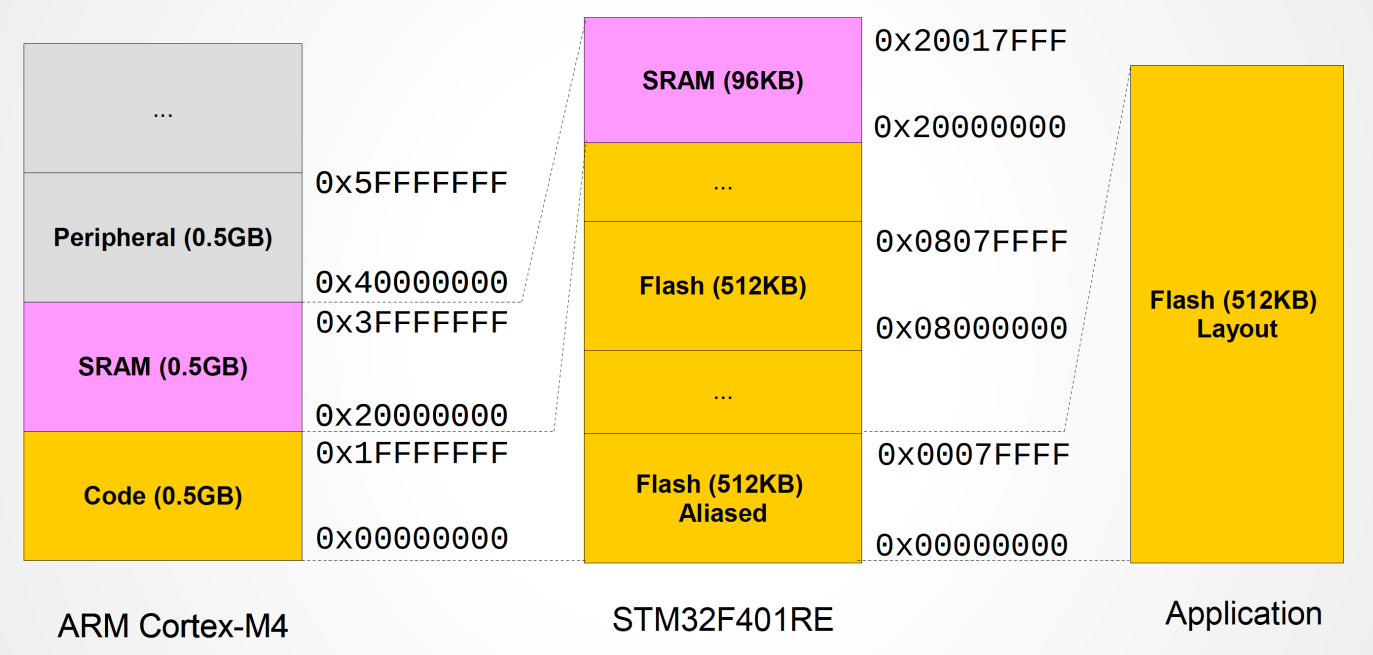
(2) How would you measure system timing even more accurately than using GPIO pins?

1.4 Exceptions

The processor triggers an exception when something *unexpected* has happened, such as an invalid memory access, a bus error or an undefined instruction. The processor interrupts the current execution flow and calls the corresponding exception handler to let the application handle the exception gracefully, such as logging register contents for diagnosis and rebooting the system.

First we look at the hard fault exception handler in our project, which dumps out the exception stack. We will see how it helps us identify the function causing the exception.

Before we start, let's remind ourselves of the memory map of STM32F401 processor:



1.4.1 Exception Handler

The hard fault exception handler is named **HardFault_Handler**. It is originally defined as a weak function in **system/src/cmsis/startup_stm32f401xe.S**. Our "overriding" version is defined in **system/src/cortexm/exception_handlers.c**. It is part of a thin layer of lower level support functions and startup files in the µOS++ IIIe Project. See http://micro-os-plus.github.io/ for details.

```
void __attribute__ ((section(".after_vectors"),weak,naked))
HardFault_Handler (void)
{
  asm volatile(
      " tst lr,#4       \n"
      " ite eq          \n"
      " mrseq r0,msp    \n"
      " mrsne r0,psp    \n"
      " mov r1,lr       \n"
      " ldr r2,=HardFault_Handler_C \n"
      " bx r2"
      : /* Outputs */
      : /* Inputs */
      : /* Clobbers */
  );
}

void __attribute__ ((section(".after_vectors"),weak,used))
HardFault_Handler_C (ExceptionStackFrame* frame __attribute__((unused)),
                     uint32_t lr __attribute__((unused)))
{
  uint32_t mmfar = SCB->MMFAR; // MemManage Fault Address
  uint32_t bfar = SCB->BFAR; // Bus Fault Address
  uint32_t cfsr = SCB->CFSR; // Configurable Fault Status Registers

  trace_initialize();
  trace_printf ("[HardFault]\n");
  dumpExceptionStack (frame, cfsr, mmfar, bfar, lr);

  __DEBUG_BKPT();
  while (1) {}
}
```

It dumps the exception stack automatically captured by the processor when an exception occurs. Below is the output of a test program with a *deliberate* hard fault added to the code:

```
HardFault]
Stack frame:
 R0 =  00000034
 R1 =  00000012
 R2 =  00000000
 R3 =  7FFFFFFF
 R12 = 0000000D
 LR =  08013E2B
 PC =  08013E38
 PSR = 01000000
FSR/FAR:
 CFSR =  00000400
 HFSR =  40000000
 DFSR =  00000008
 AFSR =  00000000
Misc
 LR/EXC_RETURN= FFFFFFF9
```

This *stack frame* shows the content of the registers at the time when an exception occurred. Of particular interest is the PC register which tells us the address of the instruction causing the exception (Note – prefetching may add 4 bytes to the address.)

This is an important piece of information. Now we know the code at address **0x08013E38** 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 (**Debug/platform-stm32f401-nucleo.map)** to narrow it down to the function that encompasses this fault address. The trick is to search for the greatest function address in the map file that is smaller than the exception PC (**0x08013E38**).

```
32904 .text._ZN3APP6GpioIn9PulseWaitEPS0_PKN2QP4QEvtE
32905                0x08013d58       0x68 ./src/GpioInAct/GpioIn/GpioIn.o
32906                0x08013d58                APP::GpioIn::PulseWait(APP::GpioIn*, QP::QEvt const*)
32907 .text._ZN3APP6GpioIn4HeldEPS0_PKN2QP4QEvtE
32908                0x08013dc0       0x88 ./src/GpioInAct/GpioIn/GpioIn.o
32909                0x08013dc0                APP::GpioIn::Held(APP::GpioIn*, QP::QEvt const*)
32910 .text._ZN3APP6GpioIn8HoldWaitEPS0_PKN2QP4QEvtE
32911                0x08013e48       0x8c ./src/GpioInAct/GpioIn/GpioIn.o
32912                0x08013e48                APP::GpioIn::HoldWait(APP::GpioIn*, QP::QEvt const*)
```

The function is found to be GpioIn::Held().

```
QState GpioIn::Held(GpioIn * const me, QEvt const * const e) {
    switch (e->sig) {
        case Q_ENTRY_SIG: {
            EVENT(e);
            Evt *evt = new GpioInHoldInd(me->m_client, GET_HSMN(), GEN_SEQ());
            Fw::Post(evt);
            me->m_holdTimer.Start(HOLD_TIMEOUT_MS);
            *(uint32_t *)(0x7fffffff) = 0x1234;
            return Q_HANDLED();
        }
        case Q_EXIT_SIG: {
            EVENT(e);
            me->m_holdTimer.Stop();
            return Q_HANDLED();
        }
    }
    return Q_SUPER(&GpioIn::Active);
}
```

Voilà, someone mistakenly adds a line that writes to an invalid address when the USER_BTN is held down!

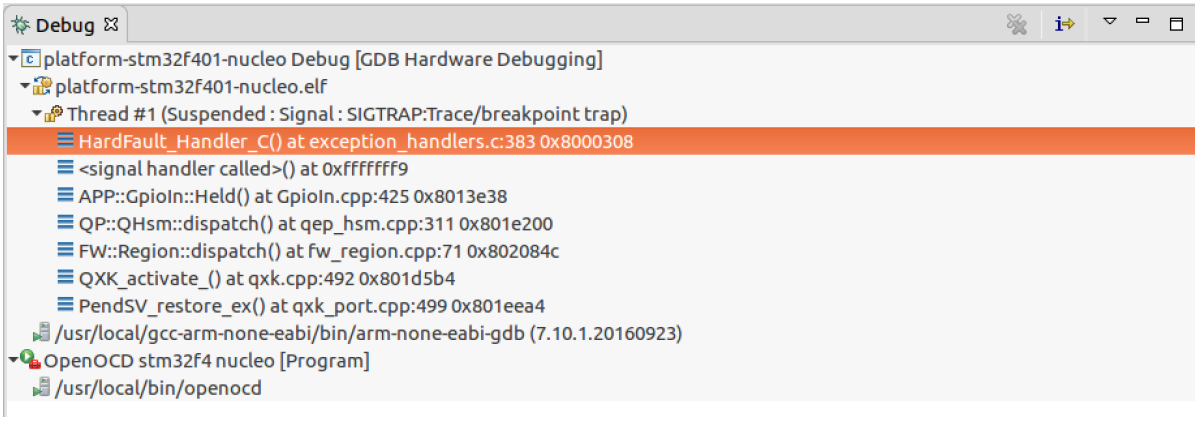
As we have seen, exceptions are more difficult to debug. What if the map file is not available? We should therefore try our best to avoid them from happening in the first place, e.g. using *assertion* (FW_ASSERT()) to catch them before they end up as hardware faults. (How is assertion easier to debug than exception?)

1.4.2 Debugger Support

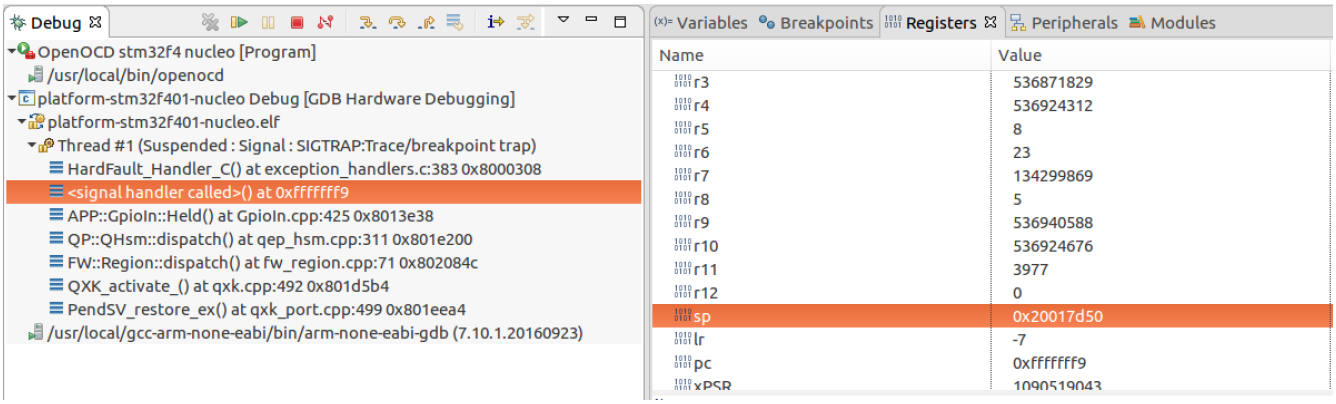
In case a serial port is not available to dump out the exception stack we may still be able to use JTAG to debug exceptions.

Eclipse and GDB works quite well in supporting the debug process. Here are some of the key notes:

1. The call "`__DEBUG_BKPT()`" in the exception handler triggers a breakpoint in the debugger. The call stack trace in the debugger window readily shows the function causing the exception. This is illustrated in the capture below:



2. Let's verify the call stack trace using first principles. First we find out where the exception stack is located in memory by inspecting the SP (stack-pointer) in the register window:



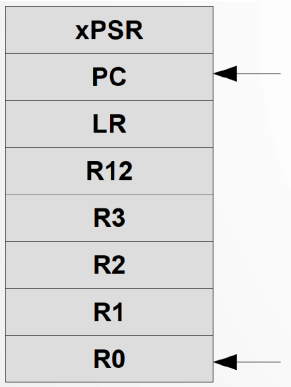
3. Knowing the SP is **0x20017d50**, we open the Memory window to inspect the exception stack starting at that location (Set rendering option to “Hex Integer” to show 32-bit in little endian).

Memory

0x20017d50 <Hex> | 0x20017d50 : 0x20017D50 <Hex Integer>

Address	0 - 3	4 - 7	8 - B	C - F
20017D50	00000034	00000012	00000000	7FFFFFFF
20017D60	0000000D	08013E2B	08013E38	01000000
20017D70	FFFFFFFF	2000D098	20017D7C	0801E201
20017D80	08013DC1	08013E49	080140DD	00000000
20017D90	00000000	00000000	00000000	00000000
20017DA0	00000000	2000D098	2000CDD4	2001102C
20017DB0	20011050	0802084D	00000010	0801D5B5
20017DC0	20001940	20011144	00000000	00FFFFFF
20017DD0	20011144	00000024	0000000A	0801EEA5

4. Next we need to find out the layout of the exception stack, which is shown below. From this we can see the offset from the base of the exception stack frame to the hard fault PC is 6 x 4 = 24 bytes. From the Memory window above we can see the memory at this offset contains **0x08013E38**, which is identical to that shown in the call stack trace (i.e. function **GpioIn::Held()**).



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 slower 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

3.1 Concepts

There is not a single *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) which divides the system into components (e.g. active objects).

It has the advantages of ease of reasoning, having 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. Identify and optimize critical paths.

3. Avoid being wasteful, but also avoid over-optimizing.
4. Meet the requirements.

3.2 Case Study

In this section we will take our level meter project as an example to illustrate how we can identify and optimize critical paths.

The demo code is available in the final project baseline which can be downloaded the course website as **Files/Assignments/Final Project/platform-stm32f401-nucleo_finalproject.tgz**.

The level meter is definitely functional. It acquires accelerometer samples at 50Hz, averages them, computes the pitch and roll degrees and displays them on the LCD at about 10Hz. In addition, if a TCP server is connected (via the "wifi conn <ip addr> <port>" command) it will send the averaged acceleration readings to the server (in the format "<x> <y> <z>", e.g. "2 -15 1009").

Note: Sampling rate is configured in function:

LSM6DSL_X_Init(DrvContextTypeDef *handle)

in file src/Sensor/Iks01a1/BSP/Components/lsm6dsl/LSM6DSL_ACC_GYRO_driver_HL.c

at line:

```
/* Select default output data rate. */
pComponentData->Previous_ODR = 104.0f;
```

3.2.1 Performance Metrics

Before any optimization is made, we first need to establish system performance metrics. One simple yet useful metrics is CPU utilization.

The main ideas are:

1. Increments a counter in the idle loop. The number of increments per second is an effective measurement of the CPU idle time. CPU utilization is then equal to 1 - % of cpu idle time.
2. During system startup, establish the baseline by measuring how many times the idle counter is incremented per second when the system is idle.
3. At any time when the system is loaded, the idle counter should be incremented less often and hence the ratio ***(idle counter increments per second) / baseline * 100*** is a measurement of CPU idle percentage. 100 minus it will yield the CPU utilization percentage.
4. The command "**sys cpu on**" enables CPU utilization reporting once every 2 seconds. "**sys cpu off**" disables it.
5. See **src/bsp.cpp** to see how the idle counter is incremented. See the startup state in System.cpp

named **Prestarting()** to see how the baseline is established when the system starts up.

3.2.2 Identifying Critical Path

With the current level meter application, the CPU utilization stands at ~33%.

How could we identify the critical path? The easier way is to check the CPU utilization when a particular operation is disabled. For example:

1. Sensor data acquisition.
2. Pitch and roll calculation and display.

3.2.3 Optimizing Critical Path

See the following files with details explained in class:

1. (Non-critical 33% → 32%) LSM6DSL_ACC_GYRO_GetRawAccData() in src/Sensor/Iks01a1/BSP/Components/lsm6dsl/LSM6DSL_ACC_GYRO_driver.c
2. DrawChar() in src/Disp/Disp.cpp.

With the critical path optimized, the CPU utilization is now less than 1/3 of the original one (~9%). This leaves comfortable room for additional features, e.g. sensor fusion, or simply allowing a higher LCD report rate (e.g. 10Hz rather than 3Hz as defined in REPORT_TIMEOUT_MS in Disp.h).

Note – It's still worth optimizing the function LSM6DSL_ACC_GYRO_GetRawAccData() to reduce I2C bus traffic. While it will not be for CPU cycles, it optimizes I2C bandwidth usage which makes it possible to share the bus with other devices.

Note – The use of a shared FIFO between LevelMeter and Iks01a1AccelGyro is another optimization to reduce the number of events between the two components.