Module 9

Design and Optimization of Embedded and Real-time Systems

1 GPIO Input

1.1 Concepts

GPIO (General Purpose Input Output) is probably the simplest yet ubiquitous hardware peripheral in any embedded systems. GPIO refers a collection of I/O pins grouped together into *ports*. On STM32L475 microcontroller, there are 6 ports named as PA, PB, PC, PD, PE and PH.

From PA to PE, each port has 16 pins whereas PH has 2 pins. Individual pins are named with its port name followed by its pin number, such as PA0 (PA.0) and PE13 (PE.13).

When a GPIO pin is used as an output pin, software controls its logic level which can be *low* (voltage < V_{OL}) or *high* (voltage > V_{OH}). It can be used to control external peripherals such as turning on/off an LED, switching on/off a relay or controlling the direction of a DC motor via an H-bridge.

When a GPIO pin is used as an input pin, it allows software to detect the logic state of an external peripheral such as a push button. It is the simplest form of a *sensor* – a touch sensor. It also allows a more complicated external sensor (such as an IMU – Inertial Measurement Unit) to inform software when a new data sample is ready to be acquired.

To work with an input pin, a simple way is to have software read its logic level periodically (*polling*). If the polling frequency is high enough (e.g. > 20Hz for a button) it would yield a pretty responsive system.

However in many cases this is not sufficient. In order to have a truly responsive system, we want software to detect and handle any input level changes as soon as possible. In those cases, we need to use *interrupts*.

1.2 GPIO Interrupts

1.2.1 Low-level Handling

The *lowest* level of interrupt handling in software is *exception vectors* – a table of code addresses for the CPU to jump to automatically when an unmasked (enabled) interrupt occurs. It is suitably called an *interrupt* since it interrupts the main flow of program. It is only after the interrupt handling has completed would the main flow of program resumes.

The interrupt vector table is defined in an assembly file named **Startup/startup_stm32l475vgtx.s**.

```
g_pfnVectors:
       .word  _estack
       .word  Reset_Handler
       .word  NMI_Handler
       .word  HardFault_Handler
       .word  MemManage_Handler
       .word  BusFault_Handler
       .word  UsageFault_Handler
       .word  0
       .word  0
       .word  0
       .word  0
       .word  SVC_Handler
       .word  DebugMon_Handler
       .word  0
       .word  PendSV_Handler
       .word  SysTick_Handler
       .word  WWDG_IRQHandler
       .word  PVD_PVM_IRQHandler
       .word  TAMP_STAMP_IRQHandler
       .word  RTC_WKUP_IRQHandler
       .word  FLASH_IRQHandler
       .word  RCC_IRQHandler
       .word  EXTI0_IRQHandler
       .word  EXTI1_IRQHandler
       .word  EXTI2_IRQHandler
       .word  EXTI3_IRQHandler
       .word  EXTI4_IRQHandler
       ...
       .word  EXTI15_10_IRQHandler


       .weak  EXTI15_10_IRQHandler
       .thumb_set EXTI15_10_IRQHandler,Default_Handler
```

The definition of the first 16 entries in the interrupt vector table is governed by the ARM Cortex-M architecture. The rest are determined by the microcontroller vendor (ST Micro). The interrupt vectors for GPIO store addresses labeled as *EXTIx_IRQHandler*. By default, they all refer to *Default_Handler()* which contains a dead-loop to avoid any accidental/undefined behaviors. Since it is a weak label, we can *override* it with our own definition, which are defined in **Src/stm32l4xx_it.cpp**:

```
// GPIO IN
// Must be declared as extern "C" in header.
extern "C" void EXTI15_10_IRQHandler(void)
{
    QXK_ISR_ENTRY();
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_15);
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_14);
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_12);
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_11);
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_10);
    QXK_ISR_EXIT();
}
```

```
extern "C" void EXTI9_5_IRQHandler(void)
{
    QXK_ISR_ENTRY();
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_9);
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_8);
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_7);
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_6);
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_5);
    QXK_ISR_EXIT();
}
extern "C" void EXTI4_IRQHandler(void)
{
    QXK_ISR_ENTRY();
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_4);
    QXK_ISR_EXIT();
}
...

extern "C" void EXTI0_IRQHandler(void)
{
    QXK_ISR_ENTRY();
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
    QXK_ISR_EXIT();
}

// Callback by STM32Cube library.
void HAL_GPIO_EXTI_Callback(uint16_t pin) {
    if (pin == GPIO_PIN_1) {
        Wifi::SignalCmdDataRdySem();
    } else {
        GpioIn::GpioIntCallback(pin);
    }
}
```

Key points:

1. The 'x' in the handler function *EXTIx_IRQHandler()* refers to the pin number that function is handling. Pins 0 to 4 each has a dedicated handler function named as *EXTI0_IRQHandler()* to *EXTI4_IRQHandler()*.

   Pins 5-9 share the handler *EXTI9_5_IRQHandler()*. Likewise pins 10-15 share the handler *EXTI15_10_IRQHandler()*.

   Sharing a common handler is less efficient as the handler needs to call the HAL function *HAL_GPIO_EXTI_IRQHandler()* multiple times to find out exactly which pins have triggered the interrupt (by reading GPIO control registers).

2. The HAL function *HAL_GPIO_EXTI_IRQHandler()* is provided by the STM32Cube library. It performs some low level processing and calls back to the application via a callback function named *HAL_GPIO_EXTI_Callback()*.

   Since it is a weak function, an application can override it to perform application-specific handling; otherwise the default version is called which does nothing.

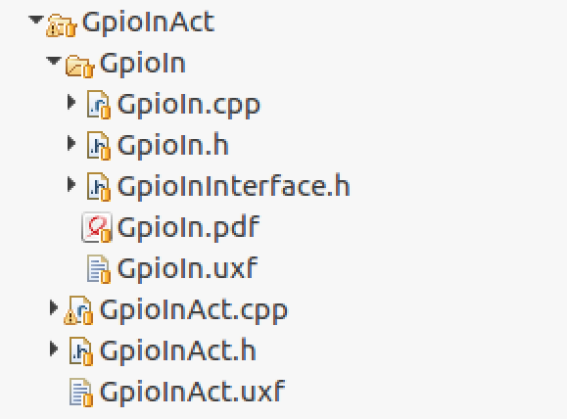
3. Let's consider the user button which is connected to PC.13. The application-specific callback *HAL_GPIO_EXTI_Callback()* calls *GpioIn::GpioIntCallback(pin)* with *pin* set to *GPIO_PIN_13*.

1.2.2 High-level Handling

Our goal is to handle a GPIO interrupt (a user button or a data-ready pin from an IMU sensor) in a state machine (HSM). At first glance, this may look like an overkill. One may wonder what behaviors there could be for a single pin that warrants a state machine.

It could be partially true that this more sophisticated approach is used as a teaching example. We could have implemented all the handing directly in the low level function *HAL_GPIO_EXTI_Callback()*. However I have also seen many apparently simple low-level drivers evolve into complicated (implicit/pseudo) state machines using flags and if-else statements.

In our model-driven approach, GPIO interrupts are handled by an HSM named GpioIn (GPIO Input). Since there can be multiple GpioIn state machines running concurrently (one per pin), we have one active object called GpioInAct owning an array of GpioIn regions. The file structure is shown below:



GpioInAct only serves as a container to provide a thread context for all the regions. The actual behaviors are defined in the GpioIn region.

Next we look at how an interrupt can be converted into an event for a state machine to process. This is done by a static method of GpioIn:

```
void GpioIn::GpioIntCallback(uint16_t pin) {
    static Sequence counter = 0;
    Hsmn hsmn = GpioIn::GetHsmn(pin);
    Evt *evt = new Evt(GpioIn::TRIGGER, hsmn, HSM_UNDEF, counter++);
    Fw::Post(evt);
    DisableGpioInt(pin);
}
```

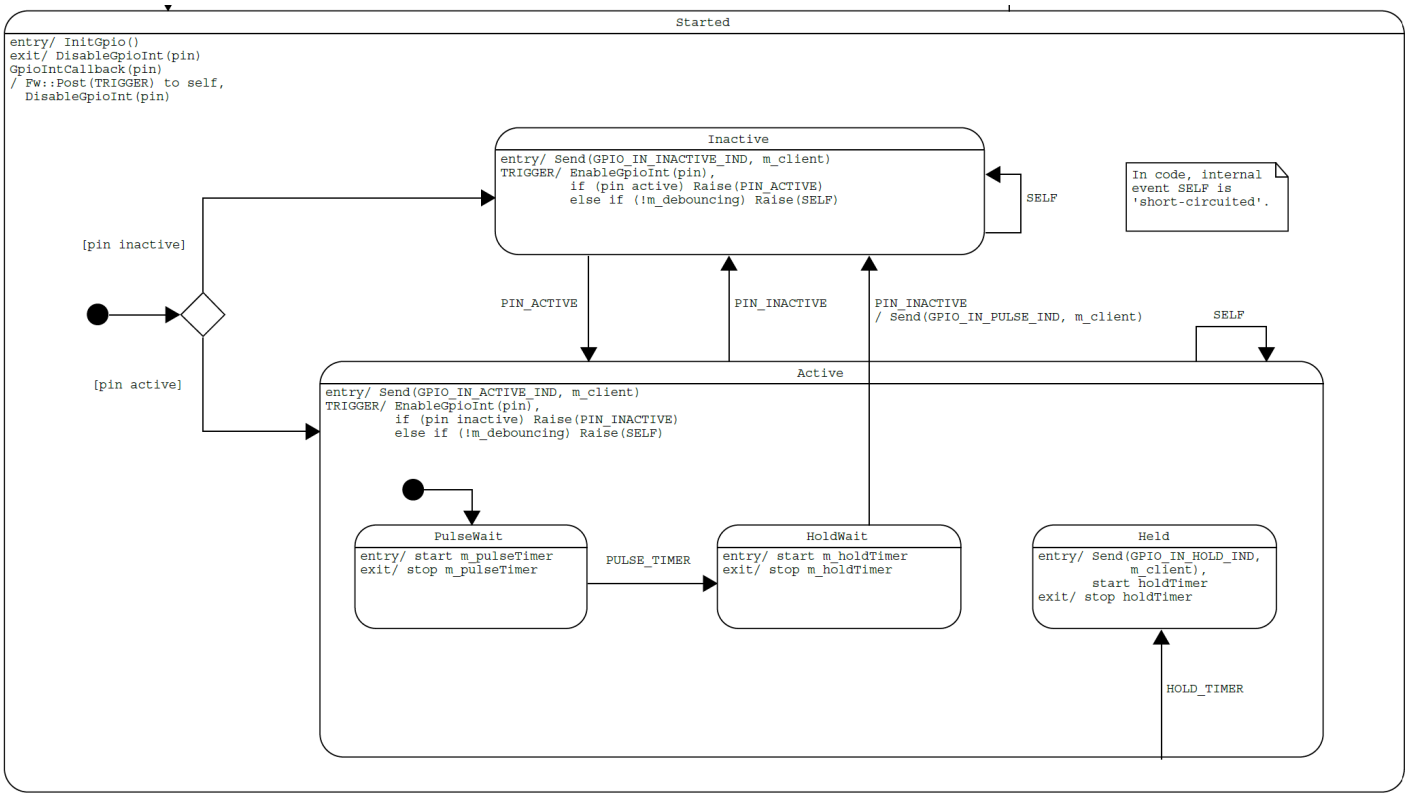
The helper function *GetHsmn(pin)* maps the pin number (0-15) to the ID (HSMN) of the GpioIn instance corresponding to this pin. Using the HSMN, it posts the event *TRIGGER* to the state machine to indicate that an edge transition of the GPIO pin has triggered.

This is a pattern of *deferred interrupt handling*. It is very important to disable the interrupt until it is completely handled in the state machine. What would happen if the interrupt were not disabled here? Since there can be a lot of glitches accompanying a single button press, tons of events may be generated to flood the event queue of the active object.

In some cases, a peripheral interrupt is level-triggered (e.g. UART Tx FIFO empty) and will keep occurring until the interrupt condition has been cleared (e.g. data written to the Tx FIFO). If the interrupt handling is deferred to a state machine, the CPU will keep re-entering the IRQ handler (ISR) and essentially hang the system.

1.3 GpioIn State Logic

The statechart for GpioIn is extracted below (showing the *Started* state):



First let's take a look at the public event interface defined in GpioInInterface.h. This is how other components (HSMs) interacts with this HSM. With the help of our event naming convention, we can conveniently tell the direction (sending vs receiving) of these events.

```
#define GPIO_IN_INTERFACE_EVT \
    ADD_EVT(GPIO_IN_START_REQ) \
    ADD_EVT(GPIO_IN_START_CFM) \
    ADD_EVT(GPIO_IN_STOP_REQ) \
    ADD_EVT(GPIO_IN_STOP_CFM) \
    ADD_EVT(GPIO_IN_INACTIVE_IND) \
    ADD_EVT(GPIO_IN_ACTIVE_IND) \
    ADD_EVT(GPIO_IN_PULSE_IND) \
    ADD_EVT(GPIO_IN_HOLD_IND)
```

The code for the Active state is listed below to demonstrate the direct mapping between graphical statechart and textual code:

```
QState GpioIn::Active(GpioIn * const me, QEvt const * const e) {
    switch (e->sig) {
        case Q_ENTRY_SIG: {
            EVENT(e);
            me->Send(new GpioInActiveInd(), me->m_client);
            return Q_HANDLED();
        }
        case Q_EXIT_SIG: {
            EVENT(e);
            return Q_HANDLED();
        }
        case Q_INIT_SIG: {
            return Q_TRAN(&GpioIn::PulseWait);
        }
        case TRIGGER: {
            EVENT(e);
            EnableGpioInt(me->m_config->pin);
            if (!me->IsActive()) {
                me->Raise(new Evt(PIN_INACTIVE));
            } else if (!me->m_debouncing) {
                me->Raise(new Evt(SELF));
            }
            return Q_HANDLED();
        }
        case HOLD_TIMER: {
            return Q_TRAN(&GpioIn::Held);
        }
        case PIN_INACTIVE: {
            return Q_TRAN(&GpioIn::Inactive);
        }
        case SELF: {
            return Q_TRAN(&GpioIn::Active);
        }
    }
    return Q_SUPER(&GpioIn::Started);;
}
```

Key design ideas include:

1. The *TRIGGER* event indicates an edge transition (rising or falling) has occurred. It's up to the state machine to read the logic level of the pin and determine what actions to take. Why can't we read the pin level in the IRQ handler and carry this information along with the event (e.g. via an event parameter or encoding it with the event name)?

   Upon the *TRIGGER* event, the event handler re-enables the GPIO interrupt *before* reading the pin level. This ensures a new interrupt will be generated if the pin level changes after it is sampled. Do you see any problems with the following implementation?

   ```
   case TRIGGER: {
       EVENT(e);
       if (!me->IsActive()) {
           me->Raise(new Evt(PIN_INACTIVE));
       } else if (!me->m_debouncing) {
           me->Raise(new Evt(SELF));
       }
       // What is the pin level changes here?
       EnableGpioInt(me->m_config->pin);
       return Q_HANDLED();
   }
   ```

2. (For the moment, assume *m_debouncing* is true.) Once it has read the pin level, it either remains in the same state (if not changed) or transitions to the opposite state (if toggled). The entry action of the Active or Inactive state sends the respective indication event (GPIO_IN_ACTIVE_IND or GPIO_IN_INACTIVE_IND) to the client HSM (registered via the GPIO_IN_START_REQ). Such event informs the client the current pin level of the GPIO pin. We use the notion of *active* or *inactive* vs *high* or *low*, since a pin can be *active-low*.

3. The initial statechart only has the *Inactive* and *Active* states in *Started.* Later there is an additional requirement to add a *glitch filter* and *hold detection* for mechanical buttons. Thanks to the hierarchical nature of statechart, we could conveniently *refine* the design by adding the substates *PulseWait*, *HoldWait* and *Held* within the *Active* state.

4. The *PulseWait* state ensures the pin remains *active* for at least a minimal duration (e.g. 50ms) before it sends out the GPIO_IN_PULSE_IND event to the user. What would happen if the pin returns to *inactive* before timeout?

   The events GPIO_IN_PULSE_IND and GPIO_IN_HOLD_IND are independent of the basic events GPIO_IN_INACTIVE_IND and GPIO_IN_ACTIVE_IND events.

   The inherit delay caused by event queuing (a few us) provides some degree of glitch filtering. The *PulseWait* state provides additional filtering.

5. The *HoldWait* state checks if a pin is held at the active level for a prolonged duration (e.g. 1s), and if so a GPIO_IN_HOLD_IND event is sent to the client as it transitions to the *Held* state.

6. The *Held* state continues to send a GPIO_IN_HOLD_IND event periodically if the pin remains *active*.

   Both the *HoldWait* and *Held* states relies on the same timeout event HOLD_TIMER to send the GPIO_IN_HOLD_IND event. This is an example of code reuse (factoring) by using a composite-to-substate (*Active* to *Held*) transition.

   If we simplify the diagram, we simplify the design. (The diagram is the design itself).

7. [Advanced] The *m_debouncing* flag (passed in via GPIO_IN_START_REQ) allows a client to disable glitch filtering caused by event queuing, which is useful when interfacing with an interrupt pin from a hardware device (e.g. data-ready from an IMU sensor). Consider the following scenario:

   a) Data-ready interrupt goes *active* causing GpioIn to enter the *Active* state.

   b) GpioIn sends the event GPIO_IN_ACTIVE_IND to the client HSM *SensorAccelGyro*.

   c) *SensorAccelGyro* reads data from the IMU sensor causing the data-ready pin to return *inactive*. This is a common *auto-clearing* feature of many hardware interrupt sources.

      This *active-to-inactive* edge transition causes a TRIGGER event to be sent to GpioIn.

   d) However before GpioIn has a chance to process the TRIGGER event, the IMU sensor toggles the data-ready pin to *active* again as a new data sample is available.

      When GpioIn finally processes the TRIGGER event in the *Active* state, it reads the current pin level and finds it remains *active*. As a result, GpioIn should stay in the *Active* state.

This is when the *m_debouncing* option is used to configure two different behaviors. If it is set to true (default), no self-transition takes place and no GPIO_IN_ACTIVE_IND events are generated. As a result the *active-inactive-active* glitch is filtered. If the option is set to false, a self transition takes place which sends a new GPIO_IN_ACTIVE_IND event to *SensorAccelGyro* causing it to read a fresh sample from the IMU sensor.

Note – the SELF event may be *short-circuited* in code.

2 TinyML

2.1 Hello-World

Machine learning is transforming embedded systems. It makes *perception* a much easier task than with traditional programmatic approaches. A useful architecture is to employ machine learning to build sophisticated sensors generating events to state machines governing deterministic system behaviors.

A popular library for deploying machine learning in embedded systems is TinyML. There is a good book on this subject: *TinyML. Pete Warden and Daniel Situnayake. O'Reilly Media. Dec 2019.*

We have included the *hello-world of TinyML* in our course projects. For basics of machine learning and details of the hello-world example, please read Chapter 3 to 6 of the TinyML book.

2.2 Model Training

In the hello-world example, we train a model about a *sine* waveform using a training data set containing pairs of an input value and the corresponding (noise-injected) output value of a sine function. That is, we feed into the model being training values of x and the expected output values of sin(x).

At the end of the training, the model would be able to recognize the sine waveform and be able to predict the output of sin(x) for some input x which it hasn't seen before (i.e. not in the original training data set). Of course for something like a sine waveform, it would be easier to use the sin() function in the math library to calculate the results directly. However for patterns that are less mathematical (or purely empirical), machine learning would be a great choice.

The hello-world example is described in this github page:

tflite-micro/train_hello_world_model.ipynb at main · tensorflow/tflite-micro · GitHub

For there we can click the *Colab* link to try it out ourselves using an online Jupyter notebook:

train_hello_world_model.ipynb - Colaboratory (google.com)

2.3 Deployment to STM32L475

Training a machine learning model is computational intensive, and therefore we would like to run it on a powerful host machine (or in the cloud as in the case of Colab). The training data set (e.g. images, audio samples or IMU measurements, etc) will still be collected by the embedded system, but they will be uploaded to the host machine to train the model.

Once a model is trained, it is just a C data structure. It can then be *embedded* in the firmware source and compiled into a binary image. Apart from the model itself, we also need to include the TinyML library into our project. Calling the TinyML API, we can then make inferences using the embedded model and make predictions or classifications.

In our project, both the TinyML library source and the trained model are located under **Src/tinyml**. In particular, the hello-world example files are located under **Src/tinyml/tensorflow/lite/micro/examples/helloworld**. It includes:

- **model.cc** – The trained model data structure (2488 bytes only) named

  ```
  unsigned char g_model[]
  ```

- **main_functions.cc** – Entry points of the hello-world example. It includes the following functions:
  - **setup()** – Initializes the TinyML library and hooks it up with our model (g_mode[]).
  - **loop()** – To be called periodically to infer the sin(x) values for incremental input values of x in the range of [0, 2π]. It calls the hook function **HandleOutput()** for our application to display the results.
- **output_handler.cc** – It implements the **HandleOutput()** function in which we plot the inferred values of y = sin(x) at locations (x, y) on the LCD. We simply post an event to Ili9341 HSM:

  ```
  Evt *evt = new DispDrawRectReq(ILI9341, CONSOLE, y_pos, x_pos, kSize, kSize,
                                 (testCnt)%2 ? COLOR24_YELLOW:COLOR24_WHITE);
  Fw::Post(evt);
  ```

The hook function (DebugLog) for logging debug messages within the TinyML library is defined in:

Src/tinyml/tensorflow/lite/micro/mbed/debug_log.cc

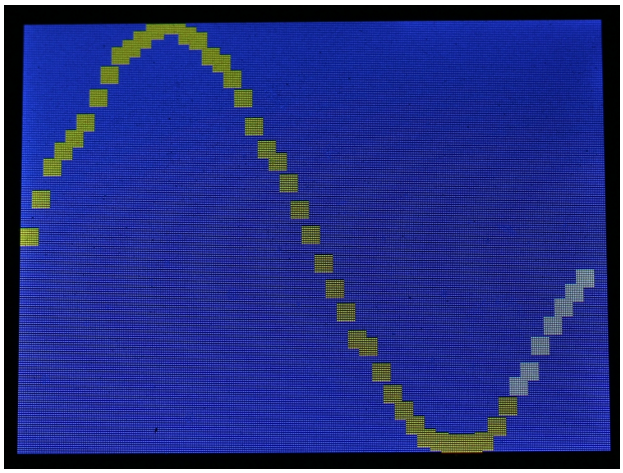
2.4 Testing

We have added the command "sys tensor" in **Src/app/System/SystemCmd.cpp** to test out the hello-world example. First we need to enable the **ENABLE_SENSOR** macro at the top of the file.

```
static CmdStatus Tensor(Console &console, Evt const *e) {
    switch (e->sig) {
        case Console::CONSOLE_CMD: {
            setup();
            console.Send(new LevelMeterStopReq(), LEVEL_METER);
            break;
        }
        case LEVEL_METER_STOP_CFM: {
            console.GetTimer().Start(100, Timer::PERIODIC);
            console.Send(new DispStartReq(), ILI9341);
            console.Send(new DispDrawBeginReq(), ILI9341);
            console.Send(new DispDrawRectReq(0, 0, 240, 320, COLOR24_BLUE), ILI9341);
            break;
        }
        case Console::CONSOLE_TIMER: {
            loop();
            break;
        }
    }
    return CMD_CONTINUE;
}
```

This is what it does:

1. Calls **setup()** to initialize and configure the TinyML library.
2. Stops the LevelMeter active object in case it is running in order to free up the LCD.
3. After the LevelMeter has been stopped, it starts a periodic timer (using a shared timer of the Console active object) with a timeout period of 100ms. It prepares the LCD and paints a blue background.
4. At each timeout, it calls **loop()** to plot a dot on the LCD. Overtime it forms a sine waveform.



3 Debugging and Profiling

3.1 Logging

"printf" is probably the most popular debugging tool. It is easy to use and does not seem to affect the normal flow of the system being debugging (vs break-points).

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 course 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 for 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__)
// The following macros can only be used within an HSM. Newline is automatically appended.
#define EVENT(e_)                Log::Event(Log::TYPE_LOG, me->GetHsmn(), e_, __FUNCTION__);
#define ERROR_EVENT(e_)          Log::ErrorEvent(Log::TYPE_LOG, me->GetHsmn(), e_,
                                                 __FUNCTION__);
#define INFO(format_, ...)       Log::Debug(Log::TYPE_INFO, me->GetHsmn(), format_,
                                            ## __VA_ARGS__)
#define LOG(format_, ...)        Log::Debug(Log::TYPE_LOG, me->GetHsmn(), format_,
                                            ## __VA_ARGS__)
#define CRITICAL(format_, ...)   Log::Debug(Log::TYPE_CRITICAL, me->GetHsmn(), format_,
                                            ## __VA_ARGS__)
#define WARNING(format_, ...)    Log::Debug(Log::TYPE_WARNING, me->GetHsmn(), format_,
                                            ## __VA_ARGS__)
#define ERROR(format_, ...)      Log::Debug(Log::TYPE_ERROR, me->GetHsmn(), format_,
                                            ## __VA_ARGS__)
```

The log verbosity level and 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 properties of the HSM being registered as the output interface:

1. infHsmn – HSMN (ID) of the interface state machine.

2. fifo – Pointer to the output FIFO.
3. sig – Signal of the write request event

Upon initialization the Console active object registers its associated UartOut region (UART1_OUT) as the default logging interface. It effectively calls the following function:

```
Log::AddInterface(UART1_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 UART1_OUT.

Note:

1. 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.
2. Since our UART driver (UartOut) is non-blocking, it minimizes *intrusion* to the system being debugged. Log data are pushed out in the background as fast as possible via DMA.
3. When log data is generated faster than the UART throughput, the software FIFO will overflow and a special character '#' will be printed to let us know some data are lost.
4. Despite our UART driver is optimized, logging still incurs overheads affecting system performance.

3.2 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 **Src/app/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 first invoked, it can also respond to subsequent events received by Console. This allows a command handler to perform complex interaction with other system components for testing or debugging. For example, a command handler first sends a request A and waits for the corresponding confirmation B. When it is received, it continues to send the next request C and so on.

   Note – The current command handler is deactivated when it returns CMD_DONE or when a 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:

```
409229 CONSOLE_UART1> perf
Elapsed time with QF = 292
Elapsed time with new = 238
```

Due to compiler optimization, the performance of the C++ shared_ptr allocation is faster than that of the QP block allocator.

For any serious projects, a versatile debug console should be the first thing to build before any customer features are implemented. 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 latency, task wake-up time.)
6. Injecting simulation events or errors for testing.
7. Automated unit and integration test regression.

4 GPIO Profiling

While log messages are very useful for gaining an insight into a running system, it is rather intrusive (even with DMA bufferred 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.

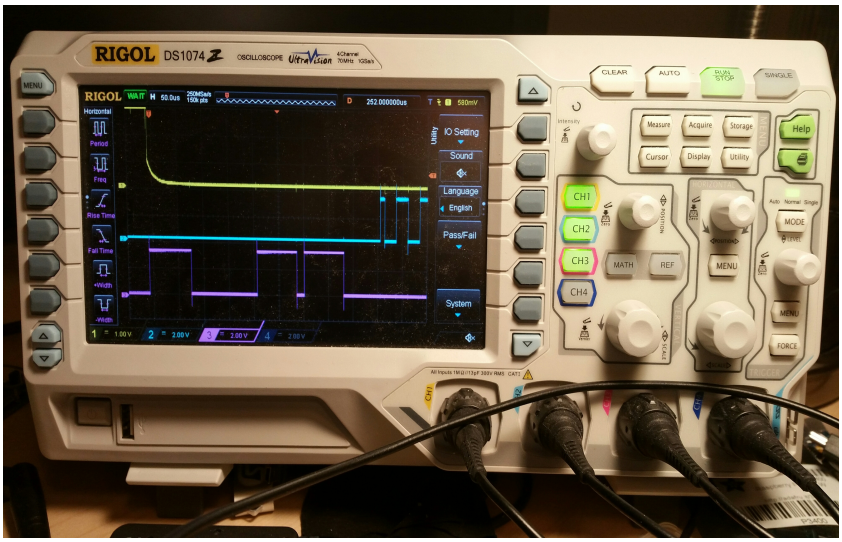
For this purpose we can call the STM32 HAL functions directly, such as the following code using PB.6:

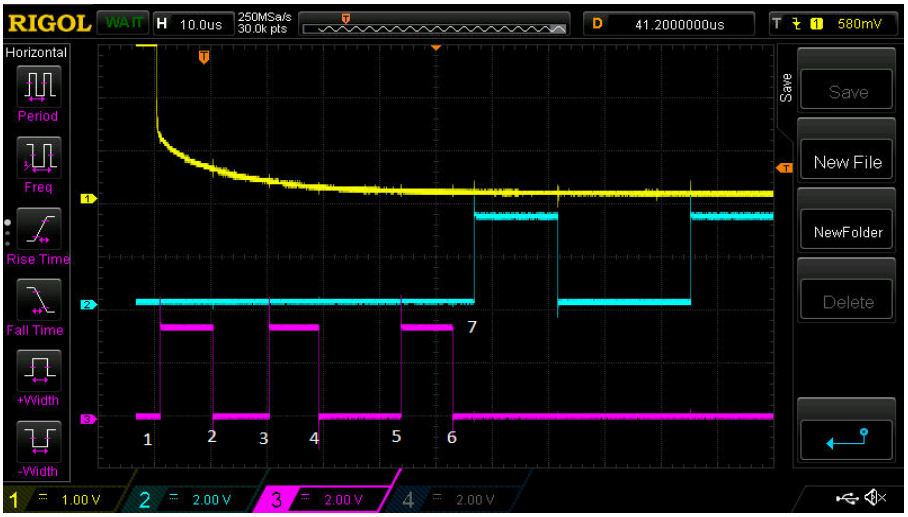
```
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);      // Sets pin high.
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);    // Sets pin low.
HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_6);                   // Toggles pin.
```

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 region. USER_BTN then notified the System active object (coordinator) which sent a request to the USER_LED region.

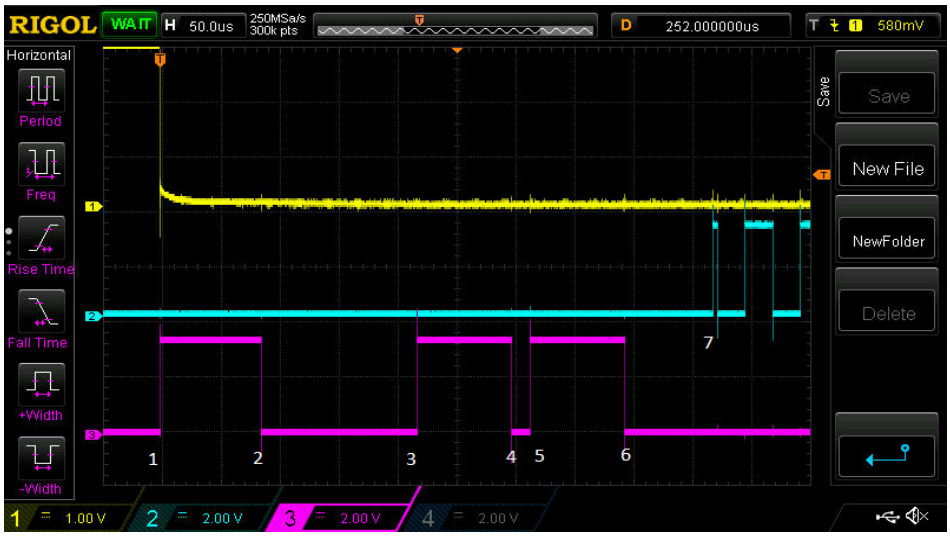




Ch 1 User Button Ch 2 User LED (PWM) Ch 3 Debug Output

(1) GPIO input (PC.13) ISR (*EXTI15_10_IRQHandler()* in stm32l4xx_it.cpp)
(2) TRIGGER event in GpioIn::InActive state
(3) GpioIn (USER_BTN) publishing GPIO_IN_ACTIVE_IND
(4) System (SYSTEM) publishing GPIO_OUT_PATTERN_REQ
(5) GpioOut (USER_LED) publishing GPIO_OUT_PATTERN_CFM
(6) System (SYSTEM) receiving GPIO_OUT_PATTERN_CFM
(7) GpioOut (USER_LED) turning on PWM

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



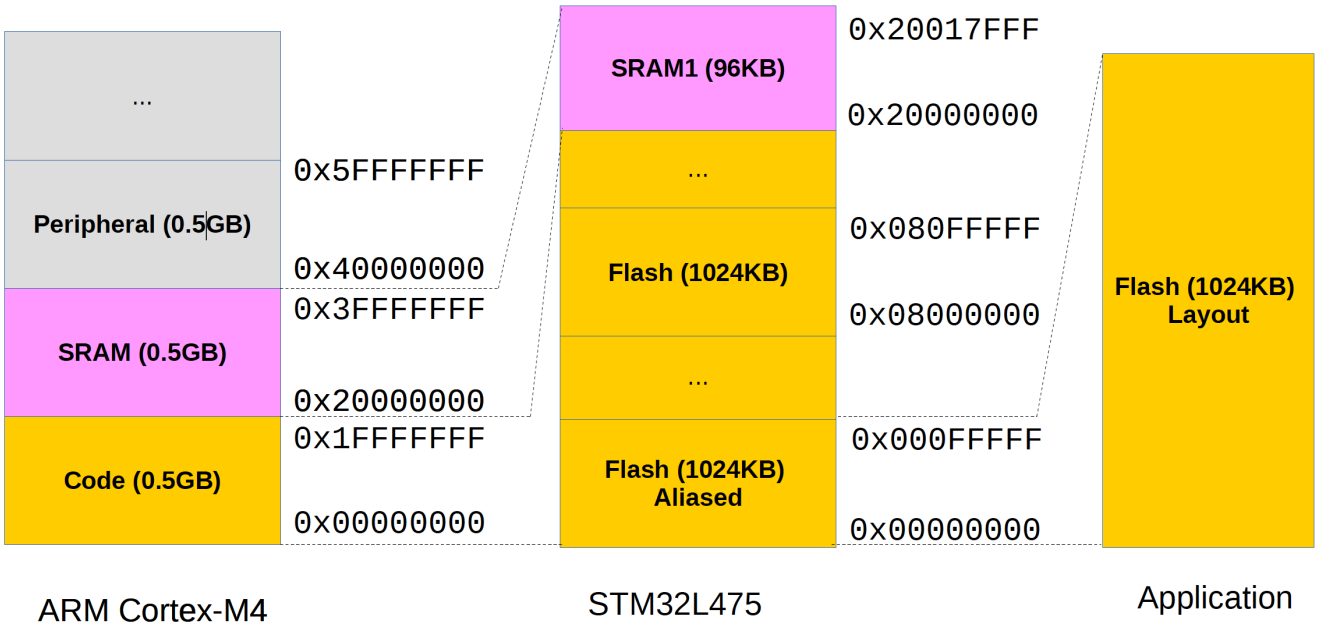
We can see how UART logging had slowed down the system. The response time from a button press to an LED lighting up changed from 55us to 475us.

4.1 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 as a last resort 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 the STM32L475 processor:



4.1.1 Exception Handler

The hard fault exception handler is named *HardFault_Handler()*. It is defined as a weak function in **Startup/startup_stm32l475vgtx.s**. We override it with our own implementation in **Src/system/src/cortexm/exception_handlers.c**. It is part of a thin layer of lower level support functions and startup files provided by the µOS++ IIIe Project. See [The µOS++ IIIe Project (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 =  00000003
 R1 =  00000000
 R2 =  00001234
 R3 =  D0000000
 R12 = 20017C5D
 LR =  0800857F
 PC =  0800858A
 PSR = 01000000
FSR/FAR:
 CFSR =  00000400
 HFSR =  40000000
 DFSR =  0000000A
 AFSR =  00000000
Misc
 LR/EXC_RETURN= FFFFFFF9
```

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

Now we know the instruction at address **0x0800858A** causes the hard fault. The next step is to find out which function corresponds to this address so we can inspect it and hopefully fix the bug.

We can use the map file (**Debug/platform-stm32l475-nucleo.map)** to find out which function 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 (**0x0800858A**).

```
45790 .text._ZN3APP6GpioIn9PulseWaitEPS0_PKN2QP4QEvtE
45791                 0x080084c8       0x64 ./Src/app/GpioInAct/GpioIn/GpioIn.o
45792                 0x080084c8                APP::GpioIn::PulseWait(APP::GpioIn*, QP::QEvt const*)
45793 .text._ZN3APP6GpioIn4HeldEPS0_PKN2QP4QEvtE
45794                 0x0800852c       0x80 ./Src/app/GpioInAct/GpioIn/GpioIn.o
45795                 0x0800852c                APP::GpioIn::Held(APP::GpioIn*, QP::QEvt const*)
45796 .text._ZN3APP6GpioIn4RootEPS0_PKN2QP4QEvtE
45797                 0x080085ac       0x84 ./Src/app/GpioInAct/GpioIn/GpioIn.o
45798                 0x080085ac                APP::GpioIn::Root(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);
            me->Send(new GpioInHoldInd(), me->m_client);
            me->m_holdTimer.Start(HOLD_TIMEOUT_MS);
            *(uint32_t *)(0xD0000000) = 0x1234;
            return Q_HANDLED();
        }
        case Q_EXIT_SIG: {
            EVENT(e);
            me->m_holdTimer.Stop();
            return Q_HANDLED();
        }
    }
    return Q_SUPER(&GpioIn::Active);
}
```

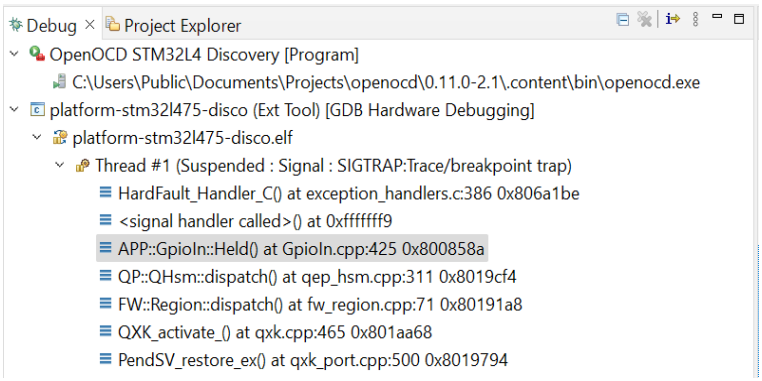
Can you spot the bug?

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 hard faults.

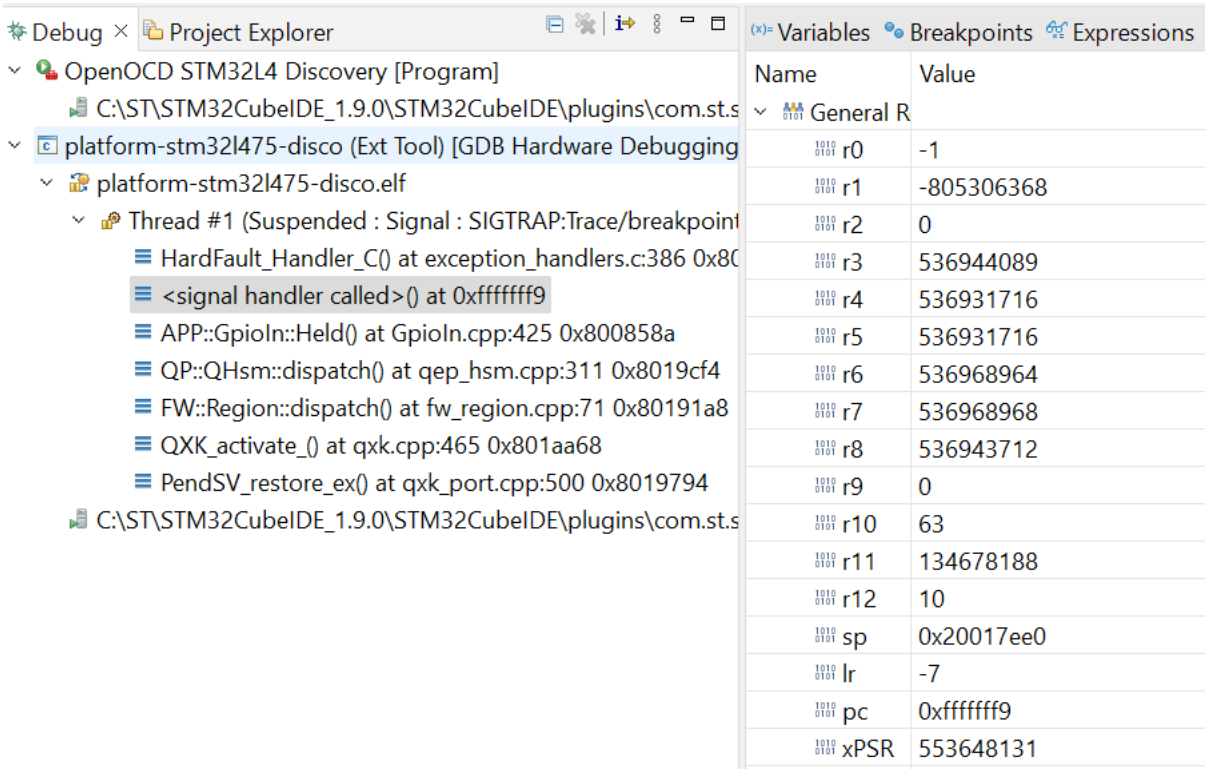
4.1.2 Debugger Support

If an ST-LINK connection is available, you may used Eclipse and GDB to debug exceptions.

1. The call "`__DEBUG_BKPT()`" in the exception handler triggers a break-point in the debugger. The call stack trace in the debugger window readily shows the function causing the exception.

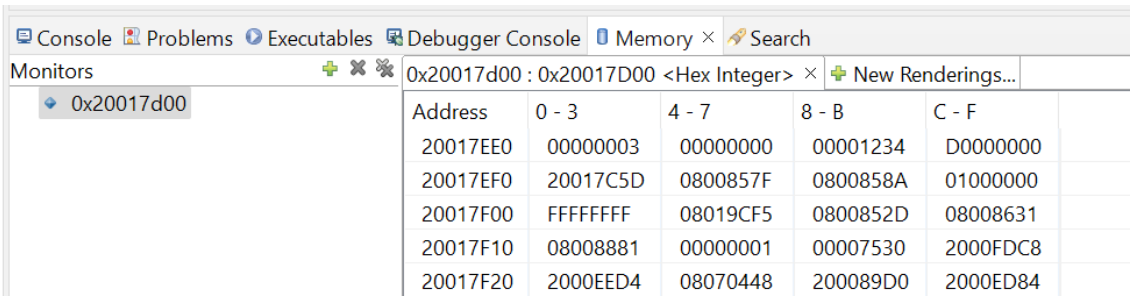


2. Let's verify the call stack trace using first principles. First we find out where the exception stack is located in memory by inspecting the SP (stack-pointer) in the register window:

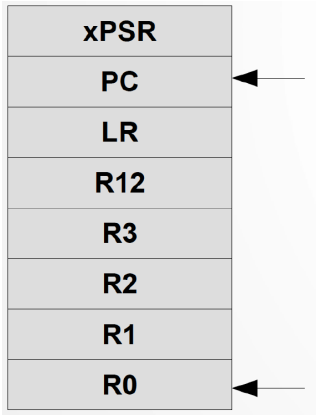


3. Knowing the stack pointer had the value of **0x20017EE0** when the exception occurred, we can now inspect the exception stack automatically pushed by the CPU upon exception. We open the *Memory* window and set the starting address to 0x20017EE0.

   Note – If you see all zeros in the Memory window, you may try to start at a smaller address such as 0x20017D00 and scroll down using the down-arrow button on the scroll bar. Since the stack frame is very close to the 96KB SRAM boundary (0x20018000), some debugger versions may have issues when they read beyond the boundary and show all zeros.



4. Recall the layout of the exception stack which is shown below. We can see the offset from the base of the exception stack frame to the PC causing the hard fault is 6 x 4 = 24 bytes. In the Memory window above the content at this offset is **0x0800858A**, which is identical to the address of the function **GpioIn::Held()** shown in the call stack trace.



5 Power Management

STM32L475 has many low-power modes, including:

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

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

     in **Src/bsp.cpp**.
   - Sleep mode is enter via the **WFI** (Wait-for-interrupt) instruction, or **__WFI()** function provided by CMSIS library.
2. **Stop 0/1/2 mode**
   - All clocks in the V_{CORE} domain are stopped.
   - SRAM1 (96KB), SRAM2 (32KB) and register contents are preserved.
   - Entered when system has been idle for some time, but fast wake-up is required.
3. **Standby mode**
   - All clocks in the V_{CORE} domain are stopped.

- *SRAM1 and register contents are lost* except for the backup domain. SRAM2 contents can be preserved.
- Entered when system is expected to be idle for a long period of time. Wake-up is slower since most SRAM and register contents are lost.

4. **Shutdown mode**
   - All clocks in the V_{CORE} domain are stopped.
   - It achieves the lowest power consumption.
   - *SRAM1, SRAM2 and register contents are lost* except for the 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.

For details, see Section 5.3 Low-power modes of STM32L47xxx, STM32L48xxx, STM32L49xxx and STM32L4Axxx advanced Arm®-based 32-bit MCUs - Reference manual.

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

1. Determines when the system should enter one of the low power modes (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.

With object-oriented design we try to encapsulate related hardware resources into separate classes. For example we have *UartAct* manage the UART, DMA and GPIO resources required by the serial port, and have *Sensor* manage the I2C and GPIO resources required by the sensors. However the hardware world is less object-oriented. A single GPIO port or DMA controller can be used for different purposes. As a result, we extract common hardware control into its own class named *Periph* in **Inc/periph.h**. The System active object calls its API to setup common clock and power settings for various power modes.

```
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();
    ...
};
```

6 Performance Metrics

Before any optimization is made, we need to first establish system performance metrics. One simple and 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 (100 - % of CPU idle time).
2. During system startup, establishes 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. Subtracting that from 100 yields 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 learn how the idle counter is incremented. See the **Prestarting** state in System.cpp to learn how the baseline is established when the system starts up.

In our Level Meter application, if we set the pitch/roll update frequency to 3Hz, the CPU utilization during normal operation is measured to be about 8%:

```
56403 CONSOLE_UART1> sys cpu on
60178 SYSTEM(1): CPU util enabled
60178 CONSOLE_UART1> Utilization = 8
```

This tells us the CPU is rather lightly loaded in performing the following tasks:

1. Acquiring acelerometer data from IMU sensor at 50Hz via I2C.
2. Averaging the acquired data and computing the pitch and roll angles at 3Hz.
3. Presenting pitch and roll data at 3Hz on the LCD via SPI.
4. Updating pitch and roll data at 3Hz to the server via SPI (WiFi).

The low CPU utilization suggests that we could increase the pitch and roll update rate from 3Hz to 10Hz. We modified the following line in LevelMeter.h:

```
enum {
    REPORT_TIMEOUT_MS = 100 //333
};
```

and measured the CPU utilization again. As expected, it was higher than before but remained at a comfortable level of about 21%:

```
97328 CONSOLE_UART1> Utilization = 21
```