Module 8

Design and Optimization of Embedded and Real-time Systems

1 Introduction

After looking at state-machine designs for sensors, LCD display and WiFi module using I2C and SPI buses, we now come back to UART which is probably the most ubiquitous hardware peripheral. We will design an optimized UART driver using a model-based approach.

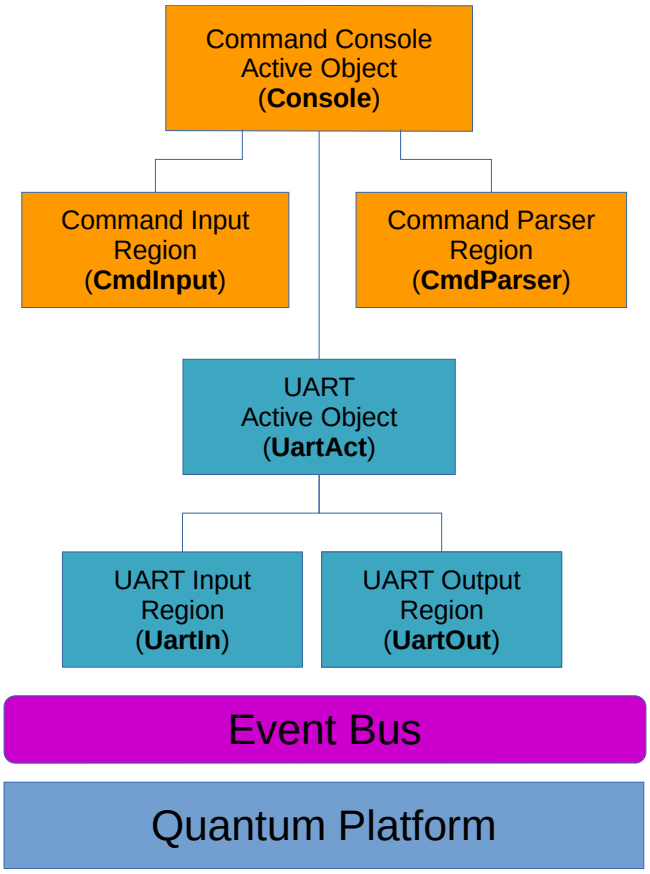
Though most laptops and PCs have long stripped out serial ports (or COM ports), the serial interface or the Universal Asynchronous Receiver-Transmitter (UART) interface remains fundamental in embedded systems development. Most development boards have built-in UART-to-USB converters to allow emulation of the serial interface over USB.

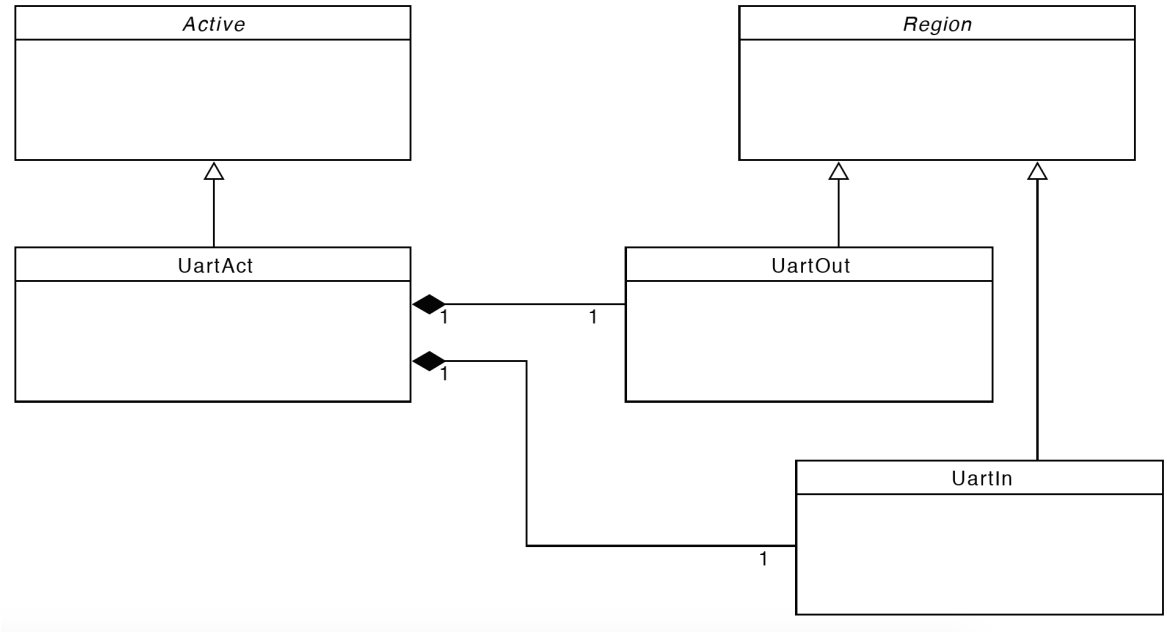
The term "*Asynchronous"* in UART means there is no separate clock signal (line) in the interface. The clock used to sample the received data (on the Rx line) is derived from the received data by the receiver hardware. "*Receiver-Transmitter"* in UART means the interface supports simultaneous transmission and reception (i.e. full-duplex). It means that our driver needs to handle data transmission and reception independently (or in parallel), which is best modeled by *orthogonal regions*.

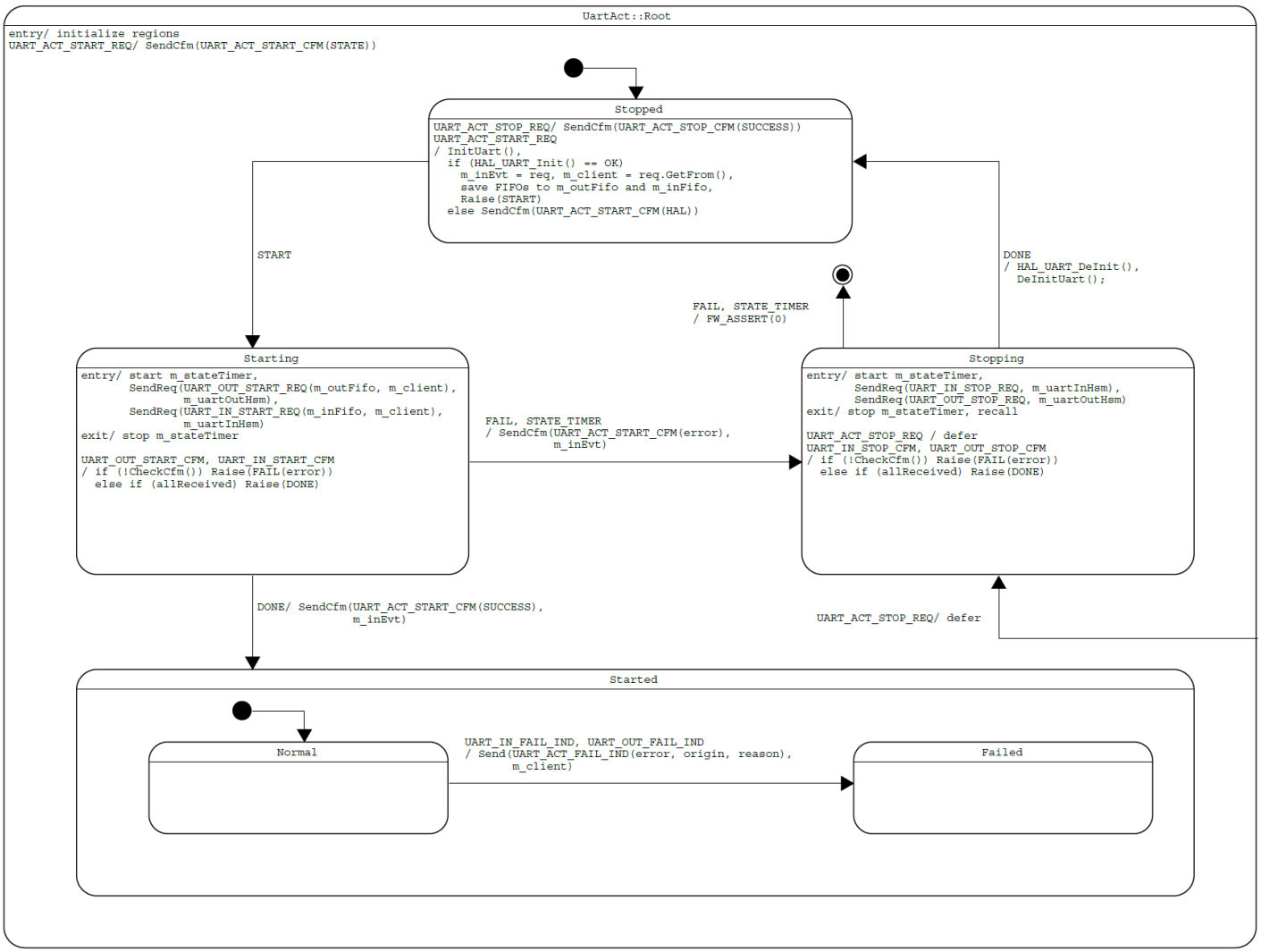
2 UartAct Active Object

At the top level we model our UART driver with an active object class named **UartAct**. We can instantiate multiple objects from the class to support different UART ports on the microcontroller. We will use one to build an interactive command console and reserve others for communicating with external modules (e.g. a secondary WiFi module).

The block and class diagrams below show the high-level design of a state-based UART driver. The detailed design of *UartAct* is shown in the following statechart.







2.1 Hierarchical Control Pattern

The design of the UART driver follows the architectural pattern named *Hierarchical Control Pattern*. See Chapter 11.3.4 of Real-Time Software Design for Embedded Systems by Gomaa. With this pattern, the system is organized into layers of components. At the top layer there is a single component serving as the *master coordinator* which controls a lower layer of controller components. Each controller component can in turn controls a lower layer of subordinate components, until it reaches the lowest layer which contains basic input and output components.

In our example, the Command Console (Console) is a higher level controller which controls the following subordinate components:

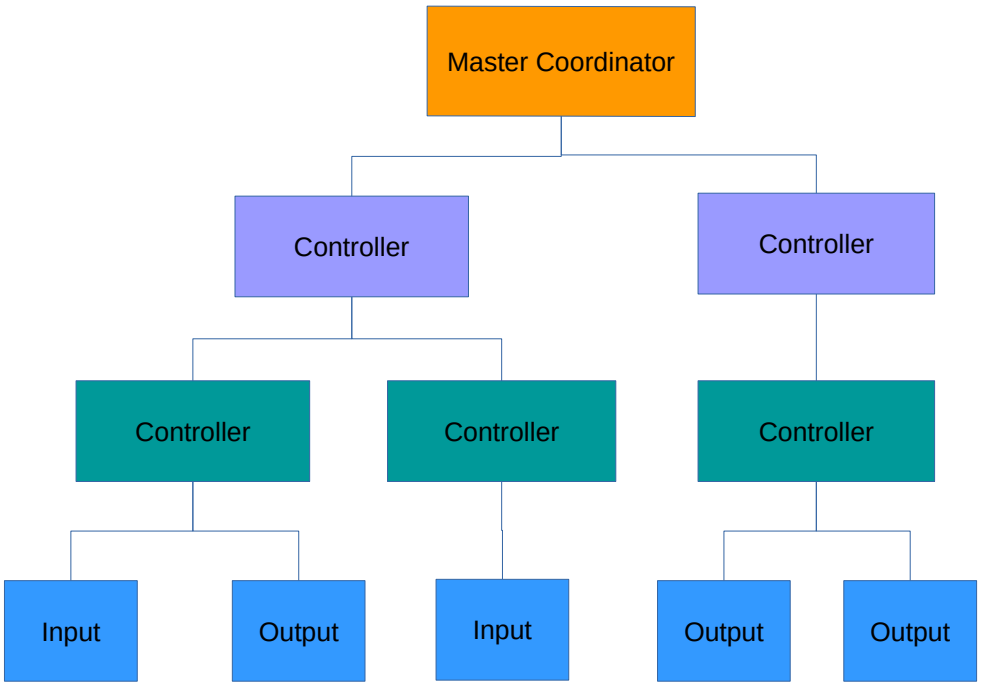
1. Command Input Region (CmdInput)
2. Command Parser Region (CmdParser)

3. UART Active Object (UartAct)

The Command Input Region (CmdInput) is responsible for detecting CR/LF-terminated command line input from a stream of received characters. It is also responsible for handling history buffers using UP/DOWN arrow keys. See **Src/app/Console/CmdInput** for details.

The Command Parser Region (CmdParser) is responsible for parsing a complete command line string into space-delimited tokens/arguments. It handles quoted arguments (e.g. "red apple") and escaped quotation characters (e.g. "John says \"Hello!\""). See **Src/app/Console/CmdParser** for details.

The UART Active Object (UartAct) is responsible for interfacing to the selected physical UART port for both input and output directions. It delegates the responsibilities of managing the input path to the UART Input Region (UartIn) and that of the output path to the UART Output Region (UartOut).



This hierarchical decomposition of a software system into layers, with a component in any layer controlling one or more components in the layer immediately below it, is particularly useful for embedded real-time systems. In the bottom layer, we can find sensor and actuator components. They perform basic I/O operations on the hardware devices on behalf of their controlling components in the upper layers. As we move up the layers, we will find components of a higher level of intelligence, achieving more complex goals by coordinating the activities of their subordinate components.

A key feature of the hierarchical control pattern is that each component in the hierarchy is a state-machine. Indeed each component is a hierarchical state-machine (HSM). Note there are two

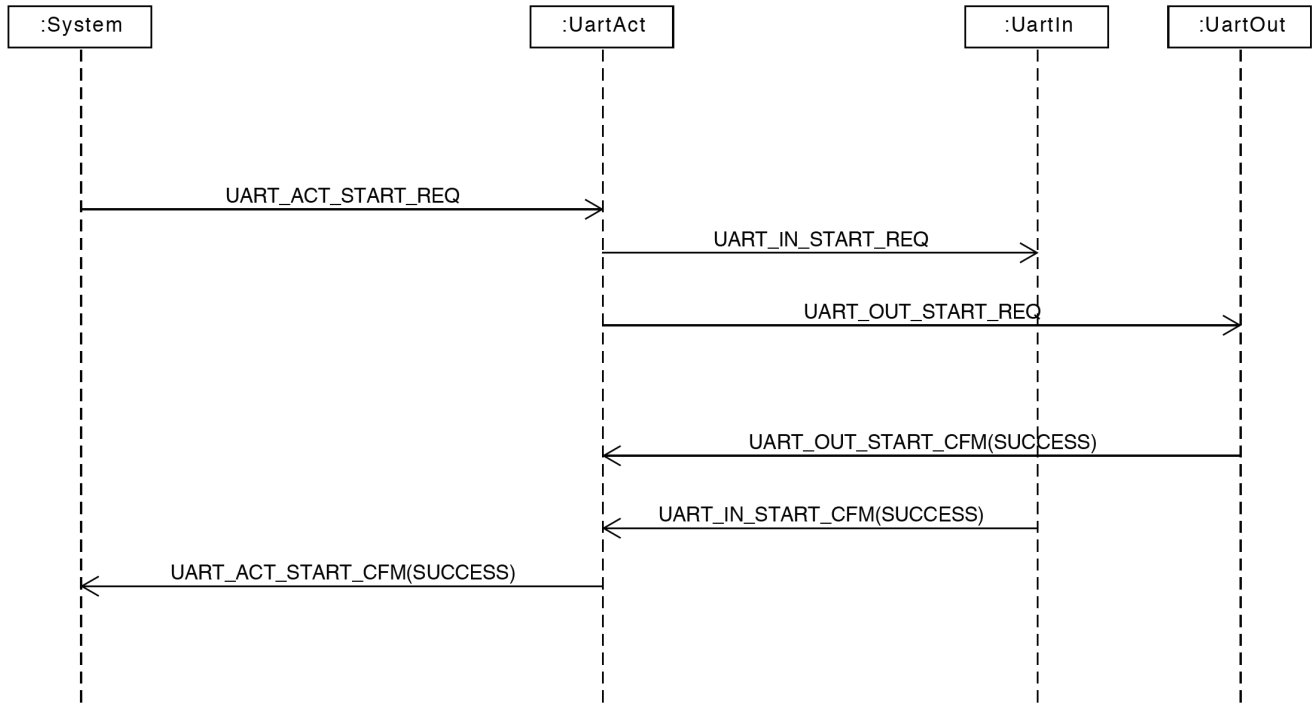
dimensions of hierarchy here – one at the component level and the other at the state level within a single component. The benefit of having each component being a state-machine is significant. Each component is always responsive to events posted to it, and working together they achieve a high degree of parallelism. With a well-defined top-down control hierarchy the overall system behaviors are both deterministic and responsive.

Finally in this section, we note that UartAct represents the *whole* of the UART driver. It encapsulates UartIn and UartOut, and makes sure both are initialized and shut down properly in the right order. At the same time UartAct represents a *part* of the Console active object, responsible for only a part of the duties assumed by Console (i.e. interfacing to the UART port). The dual roles of an HSM, being a *part* and a *whole* at the time, is called *part-whole statecharts*. It is discussed in:

- Pazzi, Luca. Systems of Systems Modeled by a Hierarchical Part-Whole State-Based Formalism. March 2013. (https://www.researchgate.net/publication/236156084_Systems_of_Systems_Modeled_by_a_Hierarchical_Part-Whole_State-Based_Formalism)

2.2 Initialization Sequence

One of the main jobs of UartAct is to coordinate the startup and shutdown sequences of its subordinate components, namely UartIn and UartOut regions. Its logic is illustrated in the statechart above. It looks rather complicated because its logic *is indeed* complicated in order to handle various exception cases right in the design. The sequence diagram below shows a normal scenario:



Highlights of the design include:

1. Upon UART_ACT_START_REQ in the Stopped state, UartAct initializes the UART device via a call to the HAL layer, namely HAL_UART_Init(). It is shown in the statechart as:

   ```
   UART_ACT_START_REQ
   / InitUart(),
     if (HAL_UART_Init() == OK)
       m_inEvt = req, m_client = req.GetFrom(),
       save FIFOs to m_outFifo and m_inFifo,
       Raise(START)
     else SendCfm(UART_ACT_START_CFM(HAL))
   ```

   Note that in statecharts, unless we rely on automatic code generation, we don't need to write down the complete source code. In most cases, a concise description is the most useful. In this example the main action is to call the HAL initialization function. If it fails, UartAct responds with a failure confirmation. If it succeeds, UartAct saves the event parameters passed in (e.g. pointers to the input and output FIFOs) and raises a reminder event *START* to itself, which will be processed immediately next.

2. Communications between UartAct and UartIn/UartOut is asynchronous (non-blocking). UartAct sends a start request event to each region and waits for response events from them. While UartAct is waiting, it is available to process any other events that may arrive in the meantime, such as UART_ACT_STOP_REQ or another UART_ACT_START_REQ.

   Note that responses from the two regions may arrive in any order, not necessarily following the order of start requests sent to them.

3. It is a good practice to use a timer to protect against being stuck in a wait state indefinitely, which may otherwise occur if not all responses are received.

4. Sequence number is used to match a confirmation (CFM) to the corresponding request (REQ), or a response (RSP) to the corresponding indication (IND). Recall the four types of interface events, namely REQ/CFM and IND/RSP. Sequence number is important to avoid race conditions.

5. An event of a given type may arrive in any state, and therefore a robust design needs to handle *all* types of events in *all* states. It sounds challenging, however it is just what a traditional state-transition table does, with each row assigned to a state and each column to an event type. With state nesting in statecharts, this is made simpler since we can implement default handling of an event in a super state and only override it in substates that handle it differently.

   For example UartAct rejects a UART_ACT_START_REQ in the Root state with the error code ERROR_STATE meaning *invalid state*. It is overriden only in the Stopped state for the normal path. In essence UartAct only accepts a start request when it is currently stopped. In another other states, including Starting, Stopping and Started, it rejects a start request. It is up to the user

object (which can be an upper layer controller object or an end-user) to decide what to do, such as:

a) retrying automatically,

b) propagating the error up the hierarchy,

c) reporting the error to the end-user, or

d) handling the error in an application specific manner.

6. Note the asymmetry between UART_ACT_START_REQ and UART_ACT_STOP_REQ. While a start request is only accepted in the Stopped state, a stop request is accepted in any states. First a stop request is handled in the Root state by transiting into the Stopping state. It covers all states except the Stopping state and the Stopped state.

   In the Stopping state, since shutdown is already in progress it defers any further stop requests by saving them in the deferred event queue owned by each HSM. Any deferred requests will be recalled upon exit from the Stopping state. The state-machine will then transition back to the Stopped state which will simply accept any stop requests with *success* since it is already stopped.

7. Component startup or shutdown is tricky. Our non-blocking design is *efficient* since it allows multiple components to be started or shut down simultaneously. If there are dependencies among components, we can add more starting and stopping states to perform the actions in stages (see System component).

   Our design is *robust* since it handles requests and confirmations arriving at any time in any order *right in the design*. This is quite a contrary to the common approach of coding up the happy paths first and fixing any issues found in test cases. When dealing with hardware interrupts or network packets typical in embedded systems, we simply cannot assume events arrive in a particular order. Statecharts enable us to visualize and analyze all these exception cases even before we start writing code. It is much more effective than trying to capture them empirically with a large set of test cases.

   See this paper on common statechart patterns employed by the control software of the Very Large Telescope (VLT) in Chile: Behavioural Models for Device Control (cern.ch).

3 UartOut Region

3.1 Separation of Data and Control

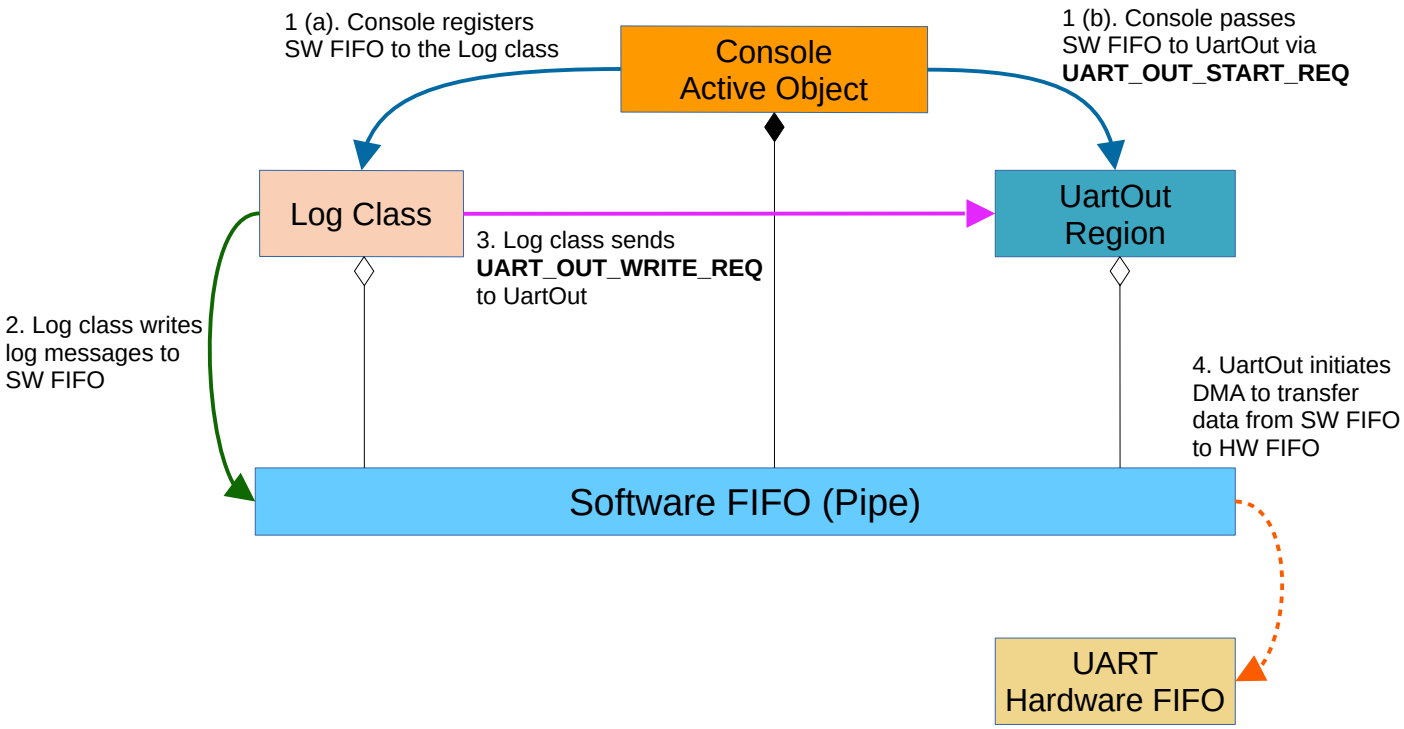
UartOut is an orthogonal region derived from Region and eventually from QHsm. It is responsible for managing the UART output path, i.e. writing from the microcontroller to a connected serial device such

as a PC (via a USB-to-serial adapter) or an external WIFI module. For efficiency we aim at reducing the number of memory copies. The ultimate goal is *zero-copy* which is particularly desirable for embedded systems since they have limited computing power and they often need to move a lot of data from one port to another.

We have previously shown the advantages of an event-driven system. Naturally in such a system we use events as a communication medium among components, such as UART_OUT_START_REQ and UART_OUT_START_CFM.

For a client object to send data to a UART port, an obvious way is to use events like UART_OUT_SEND_DATA_REQ to carry the output data as event parameters. However this approach involves first copying the payload data into an event buffer, followed by a second copy out of the event buffer into the hardware FIFO. Apart from having extra copying, this approach relies on the CPU to do the heavy lifting of moving data from SRAM to peripheral registers.

To optimize for data throughput, we adopt a different approach. We by-pass events when sending payload data from a client object to UartOut. We use a software FIFO (see type *Fifo* in fw_pipe.h) as a direct pipe linking the source and destination of a data transfer. In this design a FIFO is unidirectional, and therefore we will need an output FIFO for UartOut and an input FIFO for UartIn. The operation is illustrated and outlined below.



1. First a control event UART_OUT_START_REQ carries information about the output FIFO to UartOut. It establishes the direct data path between the client (Console/Log) and UartOut.

2. The client writes payload data directly into the output FIFO via the method Fifo::Write(). After writing to the FIFO, the client still needs to inform UartOut such that it will flush the FIFO data to the hardware. This is done via a control event named UART_OUT_WRITE_REQ. Note that this event does not carry any payload data with it. It is only a request to UartIn to flush out any data that may have been stored in the FIFO. An important optimization trick is that the client only sends UART_OUT_WRITE_REQ to UartOut if the FIFO was originally empty when Fifo::Write() is called. This minimizes the overhead of sending control events to UartOut.

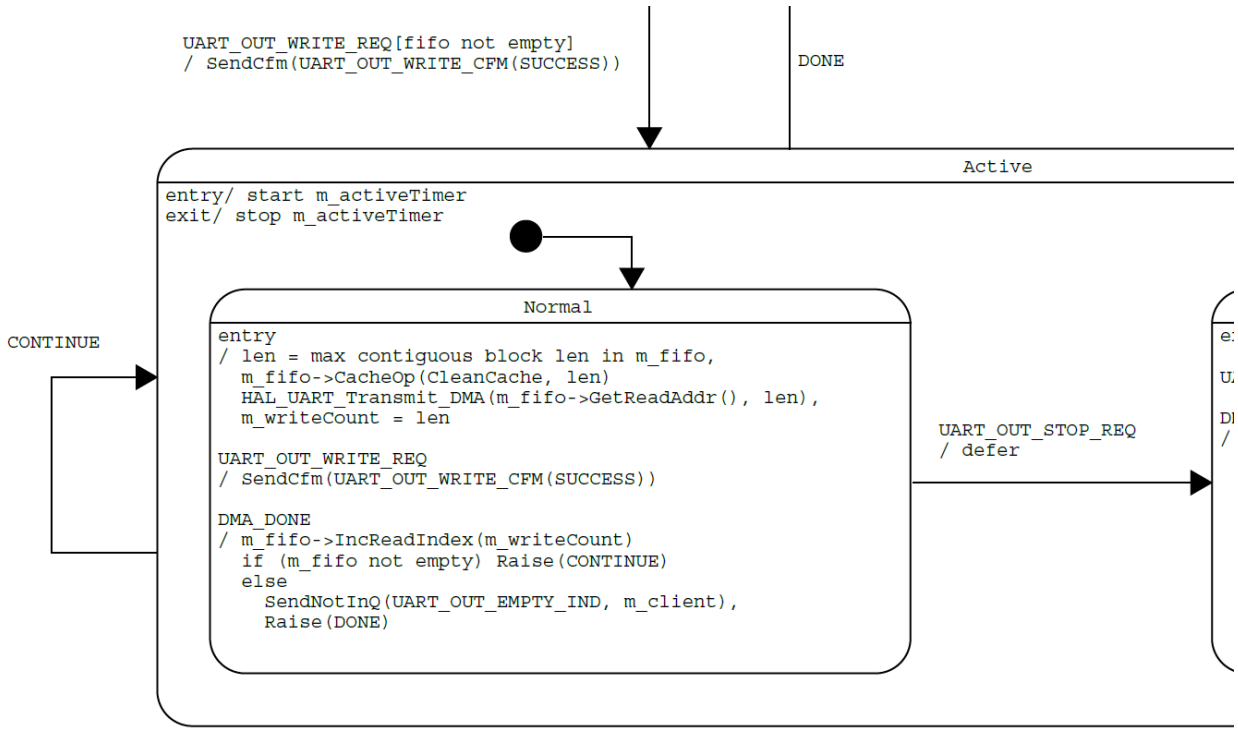
   The concept is illustrated by the pseudocode below:

```
bool status = false;
result = fifo->Write(buffer, len, &status);
if (status) {
    Evt *evt = new Evt(UART_OUT_WRITE_REQ, UART_OUT);
    Fw::Post(evt);
}
```

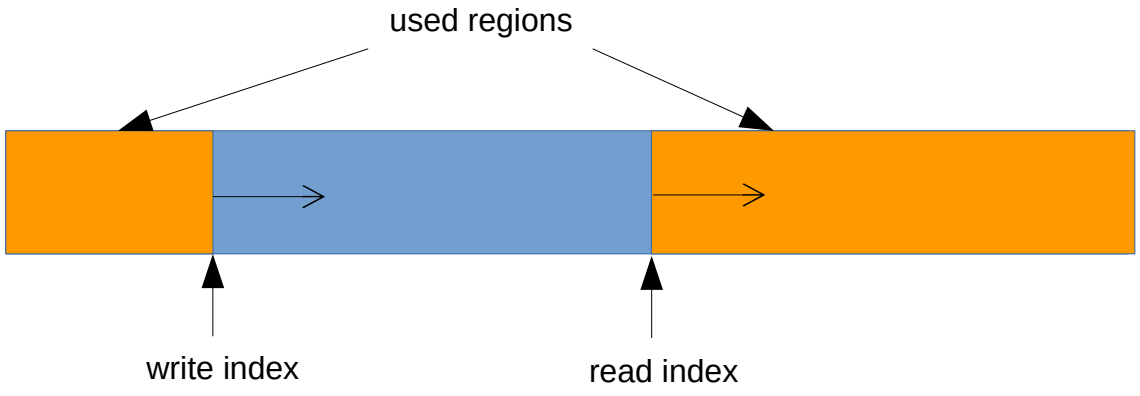
   Due to this optimization, once UartOut receives UART_OUT_WRITE_REQ it must ensure the FIFO has become empty before it goes back to the Inactive state to avoid residual data remaining in the FIFO. Check the UartOut statechart to visualize this behavior.

3. DMA, Direct Memory Access, is used to transfer data directly from a software FIFO to a hardware peripheral FIFO without CPU intervention (apart from initial setup). Upon entry to the Active-Normal state, UartOut initiates a DMA transfer by calling HAL_UART_Transmit_DMA() and remembers the number of bytes to be transferred in a member variable named *m_writeCount*. After that, UartOut waits for the DMA completion event and does not consume any CPU cycles. The code fragment and the corresponding statechart are shown below:

```
QState UartOut::Normal(UartOut * const me, QEvt const * const e) {
    switch (e->sig) {
        case Q_ENTRY_SIG: {
            Fifo &fifo = *(me->m_fifo);
            uint32_t addr = fifo.GetReadAddr();
            uint32_t len = fifo.GetUsedCount();
            if ((addr + len) > fifo.GetEndAddr()) {
                len = fifo.GetEndAddr() - addr;
            }
            HAL_UART_Transmit_DMA(&me->m_hal, (uint8_t*)addr, len);
            me->m_writeCount = len;
            return Q_HANDLED();
        }
```



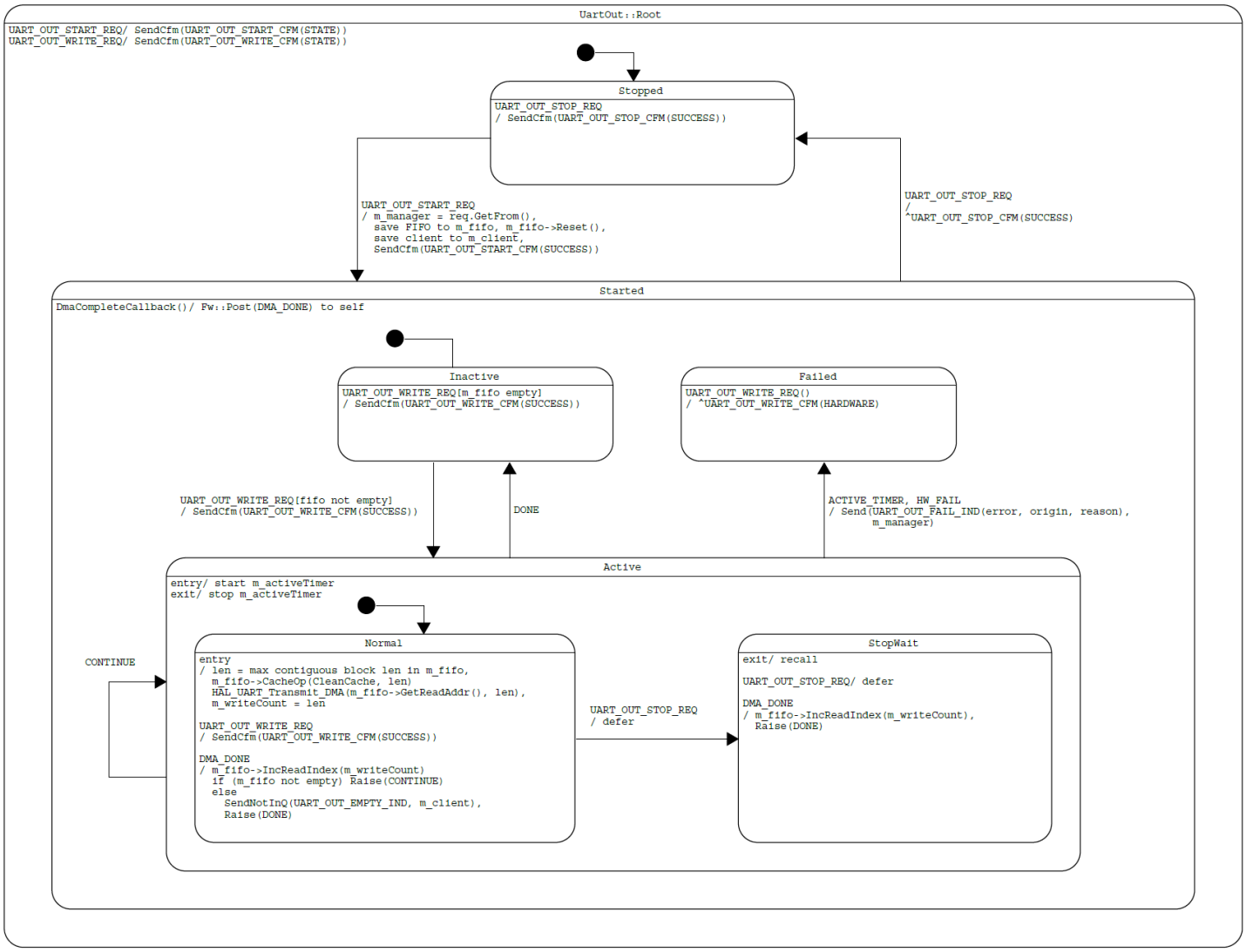
When DMA completes, a DMA_DONE event will be posted to UartOut. It then removes the transmitted data from the software FIFO by incrementing the *read index*. You can imagine the read index chasing the write index in the figure below. When data are written to the FIFO the write index moves to the right; when data are read or removed from the FIFO the read index moves to the right (with wrap-around).



Upon DMA completion UartOut must ensure the FIFO is empty before leaving the Active state. If there are more data it sends the internal event *CONTINUE* to itself causing it to re-enter the

Active state and initiate the next DMA transfer. Note how the safeguard timer (m_activeTimer) is automatically restarted as Active is re-entered. If there are no more data it sends DONE to itself causing it to transition from the Active state back to the Inactive state.

The design considers the exception case that UartOut is requested to stop while a DMA transfer is in progress. If UART_OUT_STOP_REQ is received in the Active-Normal state it will transition to the Active-StopWait state to wait for the DMA transfer to complete before handling the stop request. The complete statechart for UartOut is shown below.



3.2 DMA and Interrupts

DMA is very efficient in transferring a large amount of data between memory and peripherals (via data registers). It can work with I2C, SPI, UART or GPIO. It can work in both directions, i.e. to and from peripherals. Since there is an overhead in setting up each transaction it may not be efficient to send a small amount of data in each transaction.

In STM32 terms, a DMA transaction consists of a sequence of data transfers. The number of transfers in each transaction and the transfer data width (8-bit, 16-bit or 32-bit) are programmable. Once a DMA transaction has been setup, DMA transfers are triggered by various event sources, such as:

1. Timer interrupts used to achieve precise timing when writing data to GPIO. For example, we can use DMA to trigger data output to a GPIO port on every rising/falling edge of a hardware timer signal. In this way the timer signal serves as a clock signal to send data synchronously to the peripheral.

2. *Transmit buffer empty* interrupts for UART transmission. This type of interrupts informs the CPU that data previously written to the transmit buffer have just been sent out to the bus, and new data can now be written to the transmit buffer (a.k.a output data register, etc.).

   If we were not using DMA, the interrupt service routine (ISR) will be invoked to write new data to the transmit buffer. Using interrupts is more efficient than *polling* because the CPU is not required to constantly check when the transmit buffer has become empty. Interrupts inform the CPU asynchronously as soon as the transmit buffer has become empty, and as a result the CPU is free to do other tasks or enter sleep mode while waiting for the next interrupt to occur.

   Still there is an overhead associated with each interrupt to save and restore the CPU context (registers). This overhead is particularly significant if the transmit buffer is shallow (e.g. just a single word or a tiny FIFO) and only a small amount of data is transferred per interrupt. This is indeed the case for STM32L4, and where DMA brings a key advantage. A DMA transfer is automatically triggered upon a *transmit buffer empty* interrupt to move data from a source memory location (with auto increment) to the destination transmit buffer, all without CPU intervention.

3. *Receive buffer not empty* interrupts for UART reception. This will be discussed in the section on UartIn Region.

To understand the hardware support of DMA on the STM32L475 microcontroller, see its Reference Manual RM0351 Rev 8 (Feb 2021).

https://www.st.com/resource/en/reference_manual/dm00083560-stm32l47xxx-stm32l48xxx-stm32l49xxx-and-stm32l4axxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

1. See **Figure 29 and 30 on page 338 and 339** for a block diagram of the DMA1 and DMA2 controllers.

   Each DMA controller supports 8 simultaneous *channels*, where each channel can be mapped to only one *request type* (peripheral) at a time.

2. See **Table 44 and 45 on page 340** for all the possible request types that can be mapped to each channel on the DMA1 and DMA2 controllers.

3. Note the DMA channel and request mapping architecture varies among different STM32 processor families. Make sure you check out the corresponding reference manuals when using other STM32 families such as STM32F4, STM32H7 or STM32L4+.

4. The UART and DMA configuration for both the Tx and Rx paths are done in UartAct::InitUart(), which is extracted below for reference.

```
void UartAct::InitUart() {
    ...
    // Configure the DMA handler for Transmission process
    m_txDmaHandle.Instance                 = m_config->txDmaCh;
    m_txDmaHandle.Init.Request             = m_config->txDmaReq;
    m_txDmaHandle.Init.Direction           = DMA_MEMORY_TO_PERIPH;
    m_txDmaHandle.Init.PeriphInc           = DMA_PINC_DISABLE;
    m_txDmaHandle.Init.MemInc              = DMA_MINC_ENABLE;
    m_txDmaHandle.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
    m_txDmaHandle.Init.MemDataAlignment    = DMA_MDATAALIGN_BYTE;
    m_txDmaHandle.Init.Mode                = DMA_NORMAL;
    m_txDmaHandle.Init.Priority            = DMA_PRIORITY_LOW;
    HAL_DMA_Init(&m_txDmaHandle);
    // Associate the initialized DMA handle to the UART handle
    __HAL_LINKDMA(&m_hal, hdmatx, m_txDmaHandle);
    // Configure the DMA handler forGPIO_InitStruct reception process
    m_rxDmaHandle.Instance                 = m_config->rxDmaCh;
    m_rxDmaHandle.Init.Request             = m_config->rxDmaReq;
    m_rxDmaHandle.Init.Direction           = DMA_PERIPH_TO_MEMORY;
    m_rxDmaHandle.Init.PeriphInc           = DMA_PINC_DISABLE;
    m_rxDmaHandle.Init.MemInc              = DMA_MINC_ENABLE;
    m_rxDmaHandle.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
    m_rxDmaHandle.Init.MemDataAlignment    = DMA_MDATAALIGN_BYTE;
    m_rxDmaHandle.Init.Mode                = DMA_CIRCULAR;
    m_rxDmaHandle.Init.Priority            = DMA_PRIORITY_HIGH;
    HAL_DMA_Init(&m_rxDmaHandle);
    // Associate the initialized DMA handle to the the UART handle
    __HAL_LINKDMA(&m_hal, hdmarx, m_rxDmaHandle);

    // Configure the NVIC for DMA
    // NVIC for DMA TX
    NVIC_SetPriority(m_config->txDmaIrq, m_config->txDmaPrio);
    NVIC_EnableIRQ(m_config->txDmaIrq);
    // NVIC for DMA RX
    NVIC_SetPriority(m_config->rxDmaIrq, m_config->rxDmaPrio);
    NVIC_EnableIRQ(m_config->rxDmaIrq);
    ...
}
```

5. We have extracted the hardware configurations to a constant table named *UartAct::CONFIG[]*, which makes it much easier to adapt to different boards than having them hard-coded. The table

supports multiple instances of UartAct (currently only one). The entry matching each instance is pointed to by the member variable *m_config*.

```
// Define UART configurations.
UartAct::Config const UartAct::CONFIG[] = {
    { UART1_ACT, USART1, USART1_IRQn, USART1_IRQ_PRIO,
      // Tx parameters.
      GPIOB, GPIO_PIN_6, GPIO_AF7_USART1, DMA2_Channel6, DMA_REQUEST_2,
      DMA2_Channel6_IRQn, DMA2_CHANNEL6_PRIO,
      // Rx parameters.
      GPIOB, GPIO_PIN_7, GPIO_AF7_USART1, DMA2_Channel7, DMA_REQUEST_2,
      DMA2_Channel7_IRQn, DMA2_CHANNEL7_PRIO },
    // Add more configurations here...
};
```

where the *UartAct::Config* structure is defined as:

```
typedef struct {
    // Key
    Hsmn hsmn;
    // Common parameters.
    USART_TypeDef *uart;
    IRQn_Type uartIrq;
    uint32_t uartPrio;

    // Tx parameters.
    GPIO_TypeDef *txPort;
    uint32_t txPin;
    uint32_t txAf;
    DMA_Channel_TypeDef *txDmaCh;
    uint32_t txDmaReq;
    IRQn_Type txDmaIrq;
    uint32_t txDmaPrio;

    // Rx parameters.
    GPIO_TypeDef *rxPort;
    uint32_t rxPin;
    uint32_t rxAf;
    DMA_Channel_TypeDef *rxDmaCh;
    uint32_t rxDmaReq;
    IRQn_Type rxDmaIrq;
    uint32_t rxDmaPrio;
} Config;
```

6. The pin allocation, DMA channel and request mappings for all peripherals in use are listed in **Src/periph.cpp** for ease of reference.

7. Here we have a division of labor. While the statechart captures high-level behaviors, the nuts and bolts of hardware configurations are encapsulated by helper functions such as *UartAct::InitUart()* and the underlying STM32Cube library functions such as HAL_UART_Init(), HAL_DMA_Init(), etc.

3.3 From Interrupts to Events

Interrupt service routines (ISR) are defined in **Src/stm32l4xx_it.cpp**.

As shown in the configuration table *UartAct::CONFIG[]* above, the serial console uses the peripheral

named USART1 with the Tx DMA mapped to *DMA2 Channel 6* and *Request 2*. It means that when a DMA transaction has completed, an interrupt of the source *DMA2_Channel6* will be generated and the ISR named *DMA2_Channel6_IRQHandler()* will be called.

How does an ISR get called by the CPU automatically when an interrupt occurs?

Recall that each type of interrupts has an associated entry in the interrupt/exception vector table located at address 0x00000000 (in Flash) for a Cortex-M processor. See **Table 58 on page 396** of the STM32L475 Reference Manual RM0351 Rev 8 (Feb 2021).

https://www.st.com/resource/en/reference_manual/dm00083560-stm32l47xxx-stm32l48xxx-stm32l49xxx-and-stm32l4axxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

Each entry in the vector table stores the (starting) address of the ISR to jump to when the corresponding interrupt occurs. The entry for *DMA2_Channel6* is located at address 0x00000150 and it stores the address of (or *points to*) the function *DMA2_Channel6_IRQHandler()* in **Src/stm32l4xx_it.cpp**.

The vector table itself is defined in **Startup/startup_stm32l475vgtx.s** as listed below:

```
    .section  .isr_vector,"a",%progbits
  .type  g_pfnVectors, %object
  .size  g_pfnVectors, .-g_pfnVectors

g_pfnVectors:
  .word  _estack
  .word  Reset_Handler
  .word  NMI_Handler
  .word  HardFault_Handler
   ...
  .word  PendSV_Handler
  .word  SysTick_Handler

  /* External Interrupts */
   ...
  .word     OTG_FS_IRQHandler
  .word     DMA2_Channel6_IRQHandler
  .word     DMA2_Channel7_IRQHandler
   ...
```

Note that all these ISRs are declared with weak linkage in **startup_stm32l475vgtx.s** as below:

```
    .weak      DMA2_Channel6_IRQHandler
    .thumb_set DMA2_Channel6_IRQHandler,Default_Handler
```

If an ISR is not overridden it is defaulted to *Default_Handler* which is nothing more than an infinite loop:

```
Default_Handler:
Infinite_Loop:
  b  Infinite_Loop
```

In our example it is overridden by our own implementation in **Src/stm32l4xx_it.cpp**:

```
// UART1 TX DMA
// Must be declared as extern "C" in header.
extern "C" void DMA2_Channel6_IRQHandler(void) {
    QXK_ISR_ENTRY();
    UART_HandleTypeDef *hal = UartAct::GetHal(UART1_ACT);
    HAL_DMA_IRQHandler(hal->hdmatx);
    QXK_ISR_EXIT();
}
```

In *DMA2_Channel6_IRQHandler()* we try to use the STM32 HAL as much as possible. First we need to get back the HAL object of type *UART_HandleTypeDef* by calling the *UartAct::GetHal()* static method. We provided the HSMN (UART1_ACT) as the key for lookup.

Next we provide the HAL DMA interrupt handler with the DMA handle in the retrieved HAL object (returned by *UartAct::GetHal()*). It will figure out which specific type of DMA interrupt has occurred and call the corresponding callback function. Like an ISR, a HAL callback function is declared with weak linkage which allows an application to override it with its own implementation.

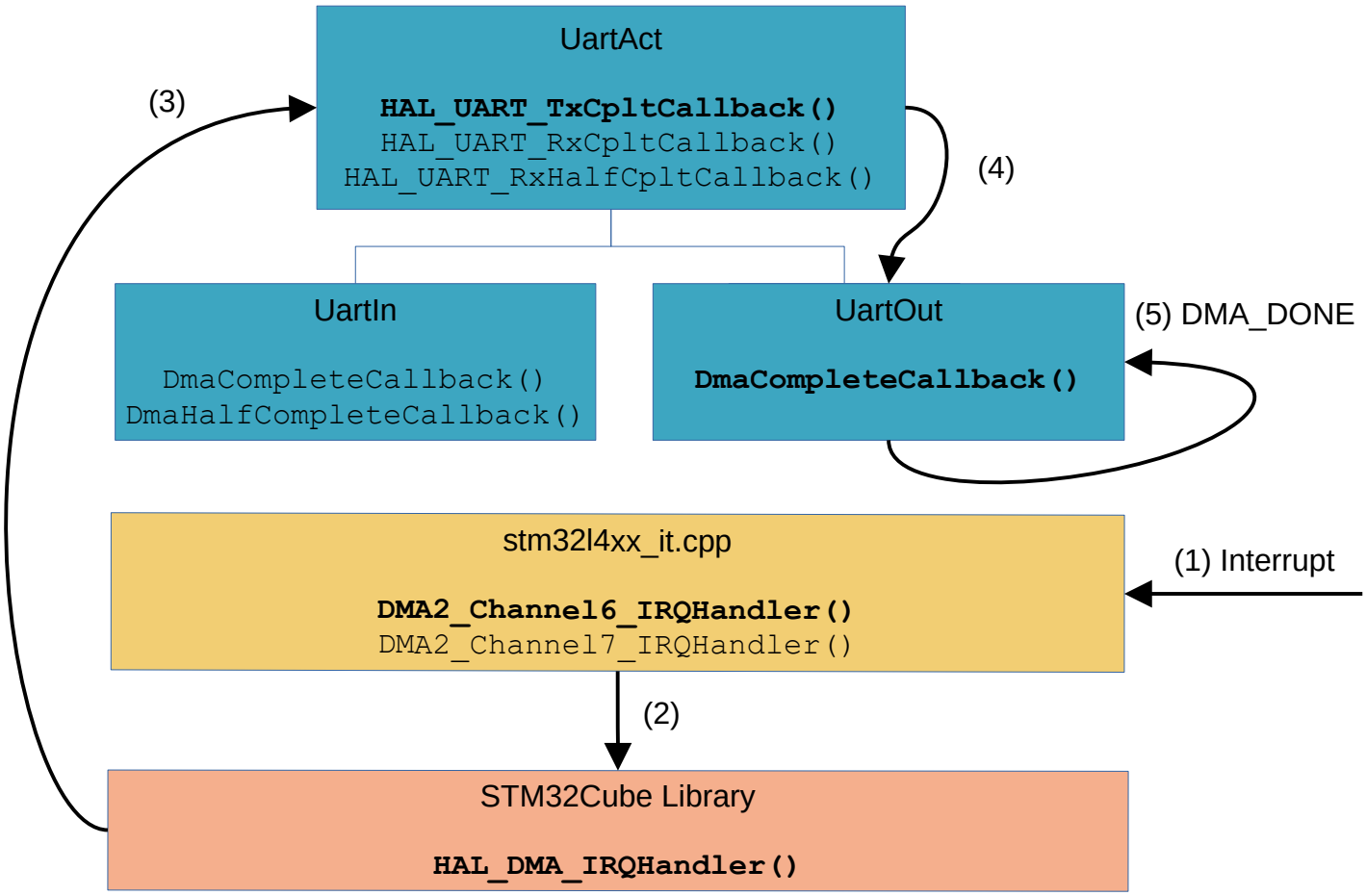
The listing below shows our implementation in UartAct.cpp to handle the callback for *UART transmission DMA completion:*

```
extern "C" void HAL_UART_TxCpltCallback(UART_HandleTypeDef *hal) {
    Hsmn hsmn = UartAct::GetHsmn(hal);
    UartOut::DmaCompleteCallback(UART_OUT + UartAct::GetInst(hsmn));
}

extern "C" void HAL_UART_RxCpltCallback(UART_HandleTypeDef *hal) {
    Hsmn hsmn = UartAct::GetHsmn(hal);
    UartIn::DmaCompleteCallback(UART_IN + UartAct::GetInst(hsmn));
}

extern "C" void HAL_UART_RxHalfCpltCallback(UART_HandleTypeDef *hal) {
    Hsmn hsmn = UartAct::GetHsmn(hal);
    UartIn::DmaHalfCompleteCallback(UART_IN + UartAct::GetInst(hsmn));
}
```

This high-level callback in turn delegates to the corresponding UartOut region by calling *UartOut::DmaCompleteCallback* defined in UartOut.cpp as:

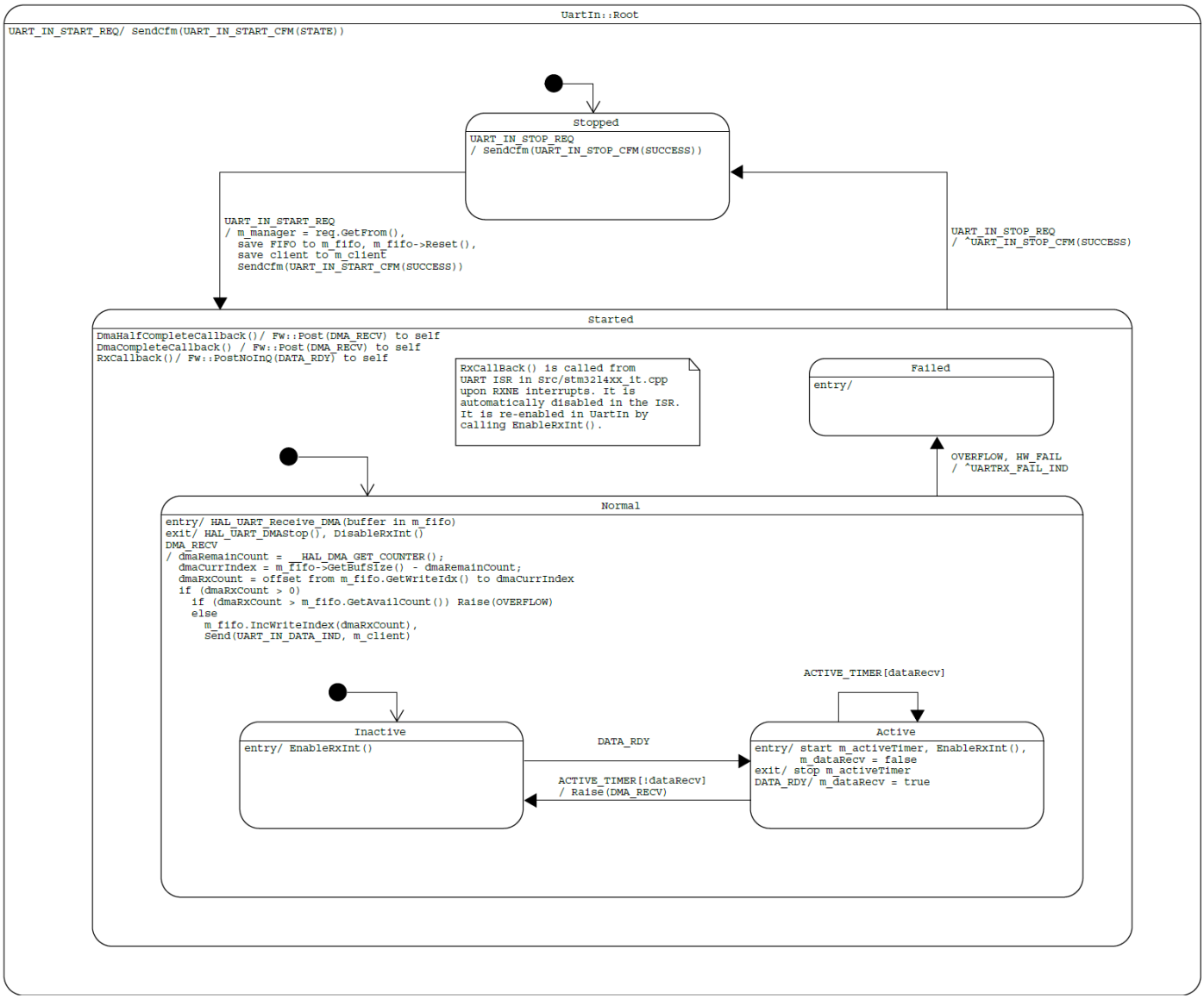
```
void UartOut::DmaCompleteCallback(Hsmn hsmn) {
    static Sequence counter = 10000;
    Evt *evt = new Evt(UartOut::DMA_DONE, hsmn, HSM_UNDEF, counter++);
    Fw::Post(evt);
}
```

Finally it generates the event *DMA_DONE* and posts it to the UartOut HSM to drive the desired behaviors specified by the statechart.



4 UartIn Region

The statechart of UartIn is shown below.



The key design ideas are:

1. It uses DMA to transfer received data from the hardware Rx FIFO to the software Rx FIFO *continuously*.

   DMA reception is first configured by UartAct when it initializes the UART port. This is done in *UartAct* rather than in *UartIn* because UART port configuration is done together for both the Tx and Rx paths in *UartAct::InitUart()*. The code fragment that configures DMA reception is listed below:

```
void UartAct::InitUart() {
    ...
    // Configure the DMA handler forGPIO_InitStruct reception process
    m_rxDmaHandle.Instance                 = m_config->rxDmaCh;
    m_rxDmaHandle.Init.Request             = m_config->rxDmaReq;
    m_rxDmaHandle.Init.Direction           = DMA_PERIPH_TO_MEMORY;
    m_rxDmaHandle.Init.PeriphInc           = DMA_PINC_DISABLE;
    m_rxDmaHandle.Init.MemInc              = DMA_MINC_ENABLE;
    m_rxDmaHandle.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
    m_rxDmaHandle.Init.MemDataAlignment    = DMA_MDATAALIGN_BYTE;
    m_rxDmaHandle.Init.Mode                = DMA_CIRCULAR;
    m_rxDmaHandle.Init.Priority            = DMA_PRIORITY_HIGH;
    HAL_DMA_Init(&m_rxDmaHandle);
    // Associate the initialized DMA handle to the the UART handle
    __HAL_LINKDMA(&m_hal, hdmarx, m_rxDmaHandle);
    ...
}
```

The DMA direction is set to *peripheral-to-memory.* Circular mode is enabled to ensure data reception is not interrupted when the end of the software Rx FIFO is reached. It will automatically wrap-around to the beginning of the software FIFO to continue data reception. To avoid existing FIFO data from being overridden, *HAL_DMA_Init()* enables both the DMA *Half-Completion* and *Completion* interrupts. This allows software to process (e.g. copy out) half of the software FIFO while the other half is continuously being filled by DMA.

2. While *UartAct::InitUart()* configures DMA reception, it does not enable any DMA transactions yet. It is done in UartIn upon entry to the Started-Normal state by calling *HAL_UART_Receive_DMA()*.

3. When a *DMA Half-completion* or *Completion* interrupt occurs, the function *UartIn::DmaCompleteCallback()* or *UartIn::DmaHalfCompleteCallback()* is called. They are shown in the statechart in the Started state. All they do is to raise the internal event *DMA_RECV* to itself, signaling DMA data have been received.

```
void UartIn::DmaCompleteCallback(Hsmn hsmn) {
    static Sequence counter = 10000;
    Evt *evt = new Evt(UartIn::DMA_RECV, hsmn, HSM_UNDEF, counter++);
    Fw::Post(evt);
}

void UartIn::DmaHalfCompleteCallback(Hsmn hsmn) {
    static Sequence counter = 10000;
    Evt *evt = new Evt(UartIn::DMA_RECV, hsmn, HSM_UNDEF, counter++);
    Fw::Post(evt);
}
```

4. The *DMA_RECV* event is handled in the Started-Normal state.

```
case DMA_RECV: {
    EVENT(e);
    // Sample DMA remaining count. It may keep decrementing as data are received.
```

```
    // Those arriving after this point will be processed upon the next DMA_RECV.
    uint32_t dmaRemainCount = __HAL_DMA_GET_COUNTER(me->m_hal.hdmarx);
    uint32_t dmaCurrIndex = me->m_fifo->GetBufSize() - dmaRemainCount;
    uint32_t dmaRxCount = me->m_fifo->GetDiff(dmaCurrIndex,
                                              me->m_fifo->GetWriteIndex());
    if (dmaRxCount > 0) {
        if (dmaRxCount > me->m_fifo->GetAvailCount()) {
            me->Raise(new Evt(FIFO_OVERFLOW));
        } else {
            me->m_fifo->CacheOp(UartIn::CleanInvalidateCache, dmaRxCount);
            me->m_fifo->IncWriteIndex(dmaRxCount);
            me->Send(new UartInDataInd(), me->m_client);
        }
    }
    return Q_HANDLED();
}
```

It is okay if no data have been received when handling *DMA_RECV*. The only CPU processing with the software FIFO is to increment its write index by calling *IncWriteIndex(dmaRxCount)*.

The call *CacheOp(UartIn::CleanInvalidateCache, dmaRxCount)* does nothing on STM32L475 since it does not have any cache memory. However on STM32F7 or STM32H7, it invalidates the software FIFO before reading from it.

5. Using *DMA Half-completion* or *Completion* interrupts alone would work if UART data keep streaming in, as the software FIFO would get filled up continuously. However if the UART port is used for an interactive command console, relying on DMA interrupts alone is not sufficient. If a user types a few characters, the FIFO is unlikely to be filled completely or halfway. As a result DMA interrupts would not be generated and those few characters would be stuck in the FIFO for a very long time.

   To solve this problem we refine our design by adding the substates *Inactive* and *Active* within *Started-Normal*. It uses the UART Receive Buffer Not Empty (RXNE) interrupt to detect UART Rx activities. Once Rx activity is detected it enters the Active state. The UART RXNE interrupt is automatically disabled in the ISR and is re-enabled as needed by UartIn. This is a common pattern of *deferring interrupt handling* to an upper application layer (HSM in our case) in order to avoid an interrupt from repeatedly triggering an ISR before it is completely processed or acknowledged.

   In the *Active* state, if no UART Rx activities have been detected within the past timeout period (say 10ms), it raises the same DMA_RECV event as if an DMA interrupt has occurred. Note all uses of the flag *m_dataRecv* (set, cleared and tested) are clearly shown in the Active state. This is how a character you type at the console gets processed by UartIn and echoed back by UartOut.

   An important feature of this design is that when the UART Rx line is idle, UartIn stays in the *Inactive* state with the m_activeTimer stopped, and hence it is not consuming any CPU cycles at all.