EMBSYS 110 Module 6

Design and Optimization of Embedded and Real-time Systems

1 Introduction

Previously we have looked at the Demo active object as an example of statechart design and implementation using QP. Demo is based on an example in the PSiCC book. While it does nothing useful it nonetheless illustrates many of the transition possibilities.

Here we will start looking at some real-life statechart designs that are useful and more complex. We will demonstrate how statecharts can be applied to not only high-level business logic but also low-level hardware interface. We will first build a state-based UART driver and then build a state-based command console on top of it.

2 UART Driver

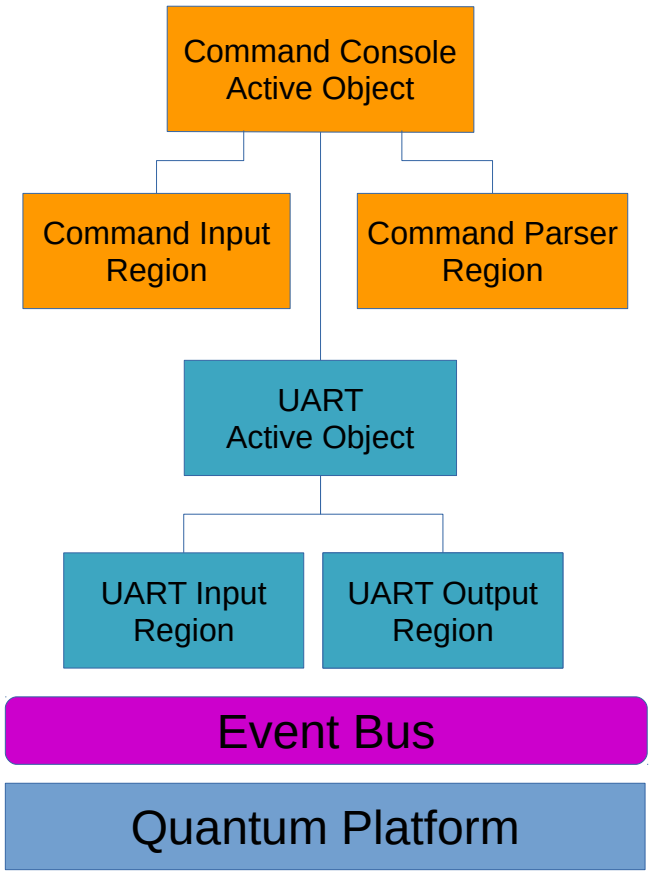
Though most laptops and PCs have long stripped out serial ports (or COM ports), the serial interface or in full the Universal Asynchronous Receiver-Transmitter (UART) interface remains ubiquitous to embedded systems development. Most development boards have built-in UART-to-USB converters to allow emulation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

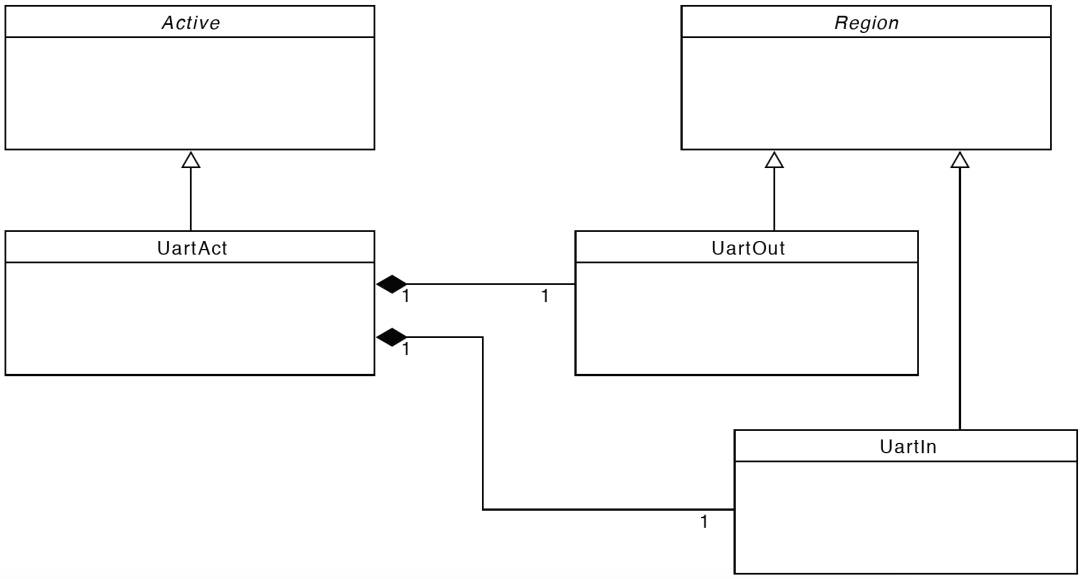
Asynchronous in UART means that there is no separate clock signal (line) in the interface. The clock used to sample the received data (on the Rx line) is derived from the received data by the receiver hardware. *Receiver-Transmitter* in UART means the interface supports simultaneous transmission and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

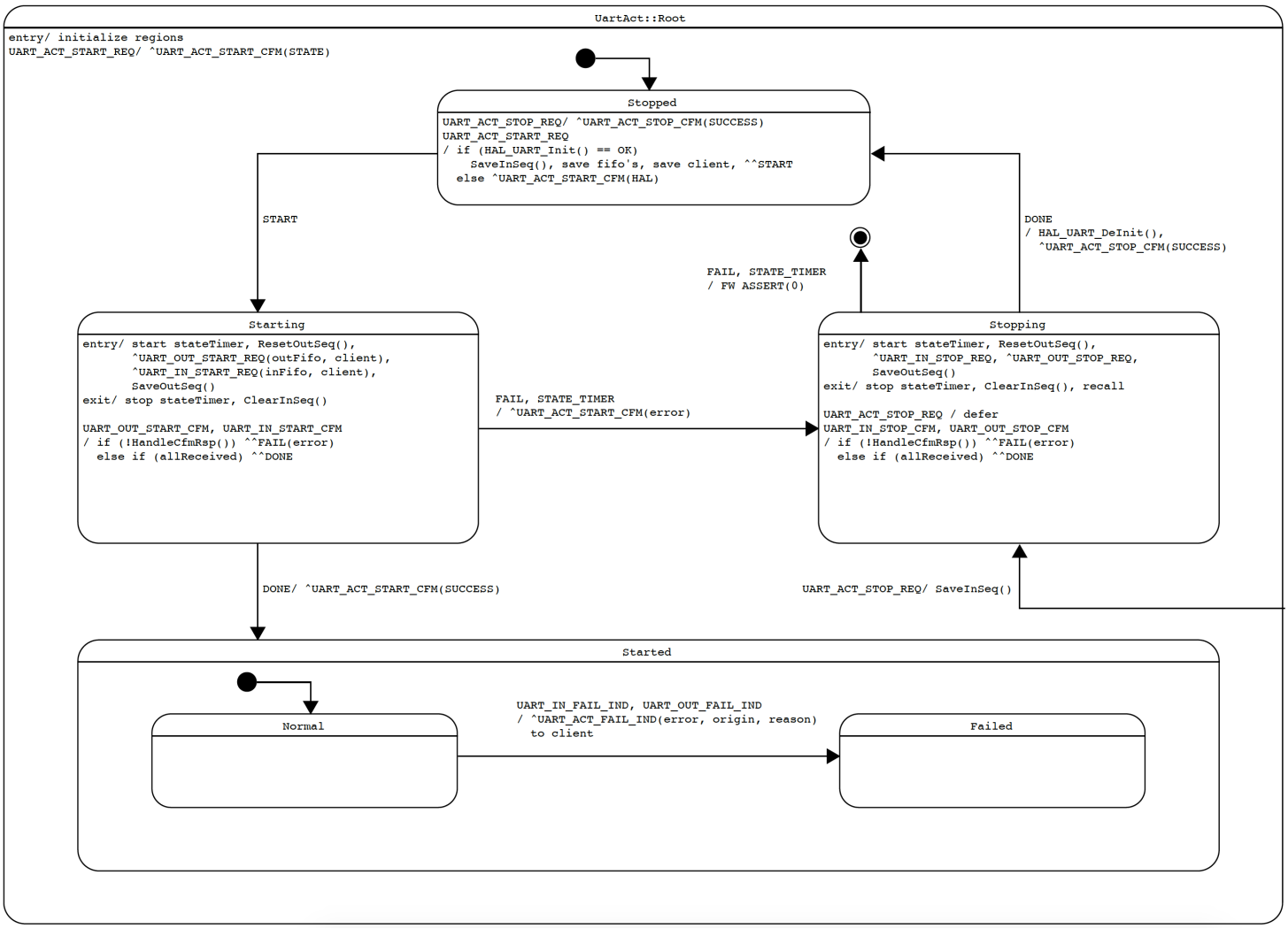
2.1 UartAct Active Object

At the top level we models our UART driver with a concrete active object class named *UartAct*. It supports multiple objects to be instantiated from the class, with each object configured to use a different UART port on the micro-controller. Later we will see how we use one for an interactive command console and one for communicating with a 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.







2.1.1 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:

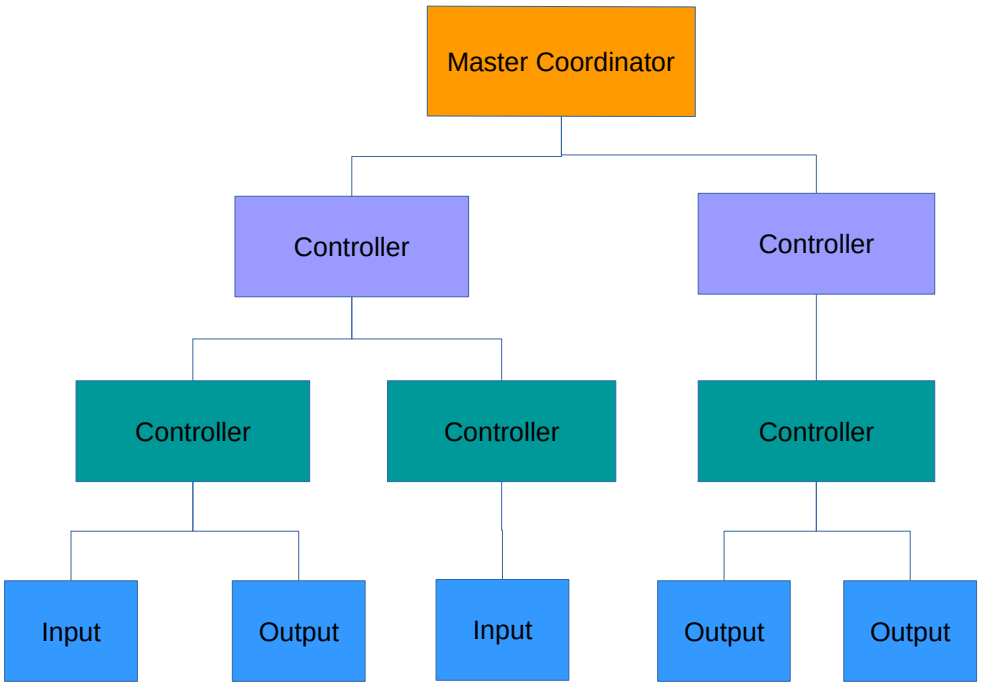
1. Command Input Region (CmdInput)
2. Command Parser Region (CmdParser)

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.

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!\"").

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 layer. As we move up the layers, we will find components of a higher level of intelligence, adding values 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 aways responsive to events addressed 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.

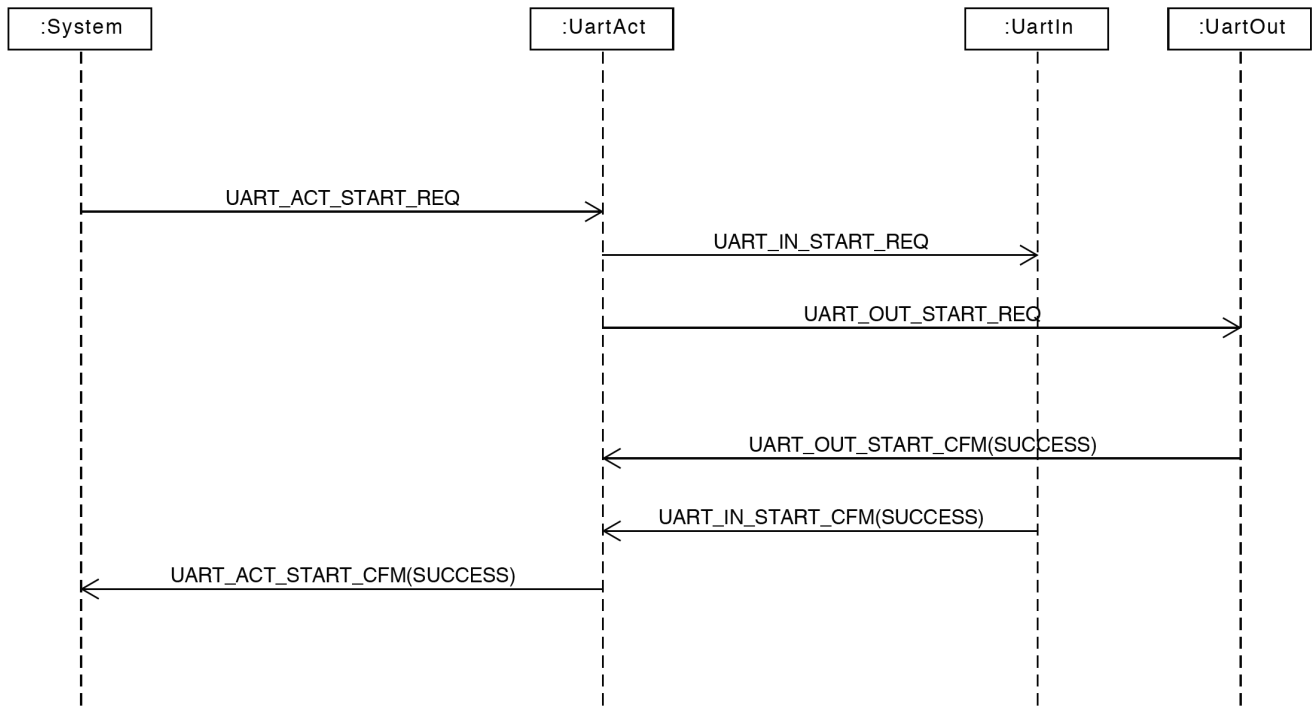
Finally in this section, we note that UartAct represents the *whole* of the UART driver. It encapsulates UartIn and UartOut, and makes sure both are initialized and shutdown properly in the right order. At the same time UartAct represents a *part* of the Console active object, responsible for only a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

2.1.2 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 various exception cases.

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
   / if (HAL_UART_Init() == OK)
       SaveInSeq(), save fifo's, save client, ^^START
     else ^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. incoming request sequence number, pointers to the FIFOs and client ID) and posts the internal event *START* to the front of its event queue (via Active::PostSync()).

   In UML the standard symbol '^' means posting an event. '^^' is my own symbol to denote posting an event synchronously to the front of the event queue.

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 another UART_ACT_START_REQ or UART_ACT_STOP_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 its corresponding request (REQ), or a response (RSP) to its 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 put default handlings of an event in a super state and override it only in the substate that handles 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 overrides it 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 a special deferred event queue. Any deferred requests will be recalled upon exit from the Stopping state. The state-machine will then transit 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 shutdown simultaneously. If there are dependencies among components, we can add more starting and stopping states to perform the actions in stages.

   Our design is robust since it handles start/stop requests/confirmations arriving at any time in any order *at the design level*. This is quite a contrary to the common approach of coding it up as you go and fixing it when its fails a unit test case. When dealing with hardware signals or network packets, which are typical in embedded systems, we simply cannot assume events will arrive in certain 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 unit test cases. (This is not a statement against unit testing, but suggests that unit testing cannot replace proper design.)

2.2 UartOut Region

2.2.1 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 a 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 an extra *copy*, this approach relies on the CPU to do the heavy lifting in copying, that is moving data into and out of CPU registers (load/store architecture).

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 outlined below:

1. First a control event UART_OUT_START_REQ carries information about the output FIFO to UartOut. It establishes the direct link between the client 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 is 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:

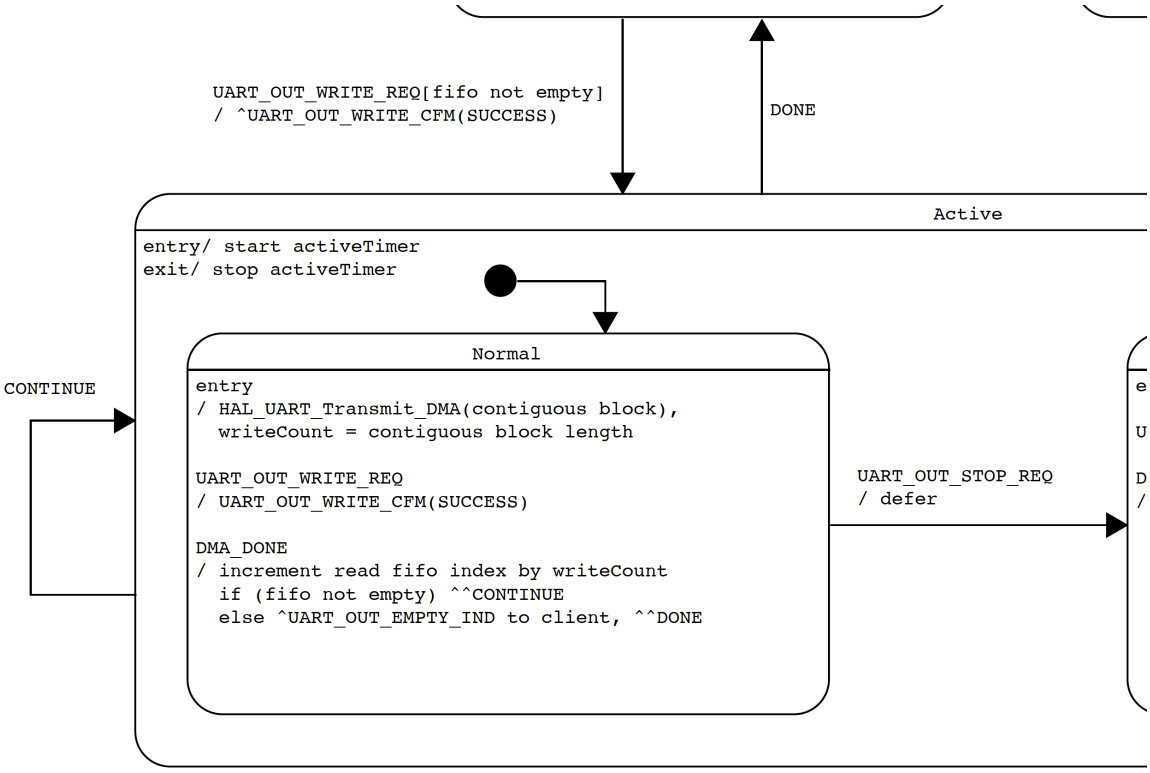
   ```
   bool status = false;
   result = fifo->Write(buffer, len, &status);
   if (status) {
       Evt *evt = new Evt(UART_OUT_WRITE_REQ, destination);
       Fw::Post(evt);
   }
   ```

   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 peripheral FIFO without CPU involvement (after the initial setup). Upon entry to the Active-Normal state, UartOut initiates a DMA transfer by calling HAL_UART_Transmit_DMA() and

remember the number of bytes to be transferred in a member variable named *writeCount*. After that, the thread of UartOut will block on the event queue and will not consume any CPU resources. In the meantime the value of *writeCount*, being an extended state variable, will persist. 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();
        }
```

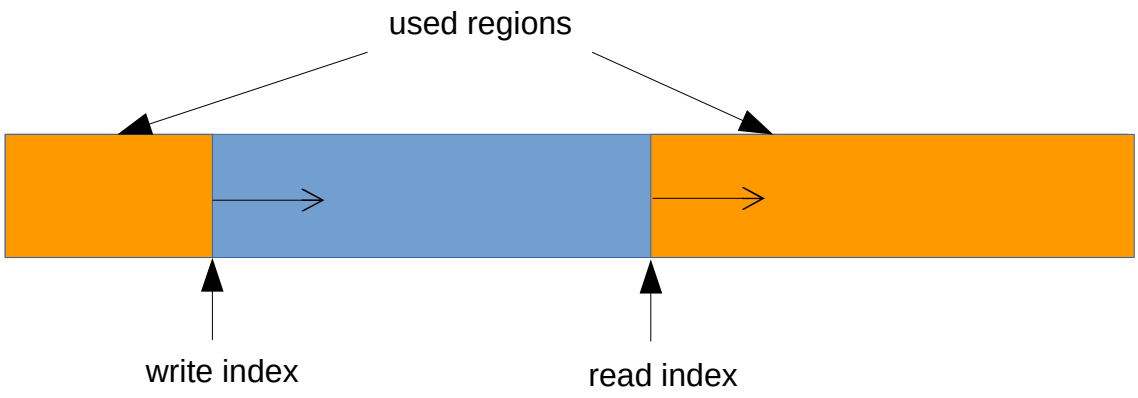


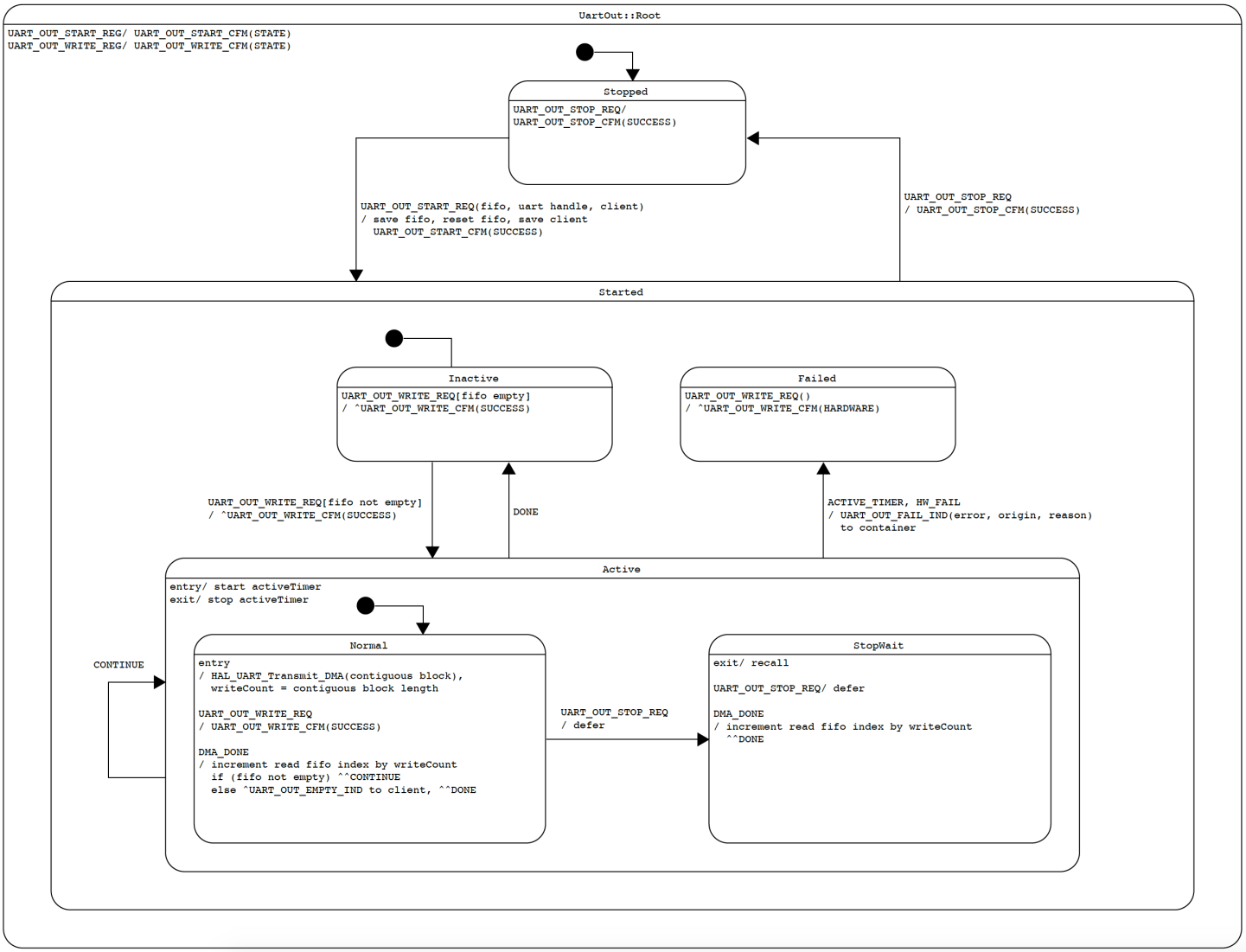
When DMA completes, a DMA_DONE event will be generated and received by UartOut. It

will then remove the data that have just been transferred from the FIFO by incrementing the *read index*. You can image 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 (activeTimer) is automatically restarted as Active is re-entered. If there are no more data it sends DONE to itself causing it to transit from the Active state back to the Inactive state.

The design also 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 transit 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.





2.2.2 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 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 and GPIO data are sent

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 onto the bus, and new data can now be written to the transmit buffer (a.k.a output data register, etc.).

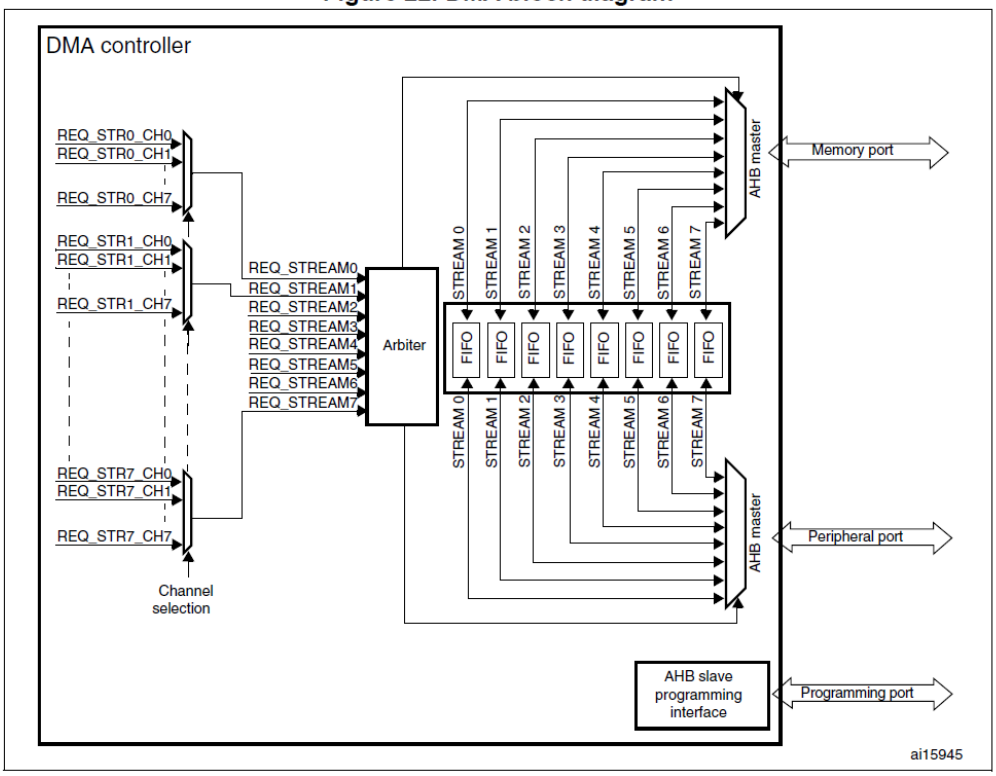
   If we were not using DMA, the interrupt service routine (ISR) will be invoked to write directly or indirectly new data to the transmit buffer. Using interrupts is more efficient than *polling* because the CPU is no longer required to constantly check when the transmit buffer has become empty and writes new data to it. 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. a single word vs a FIFO) and only a small amount of data is transferred per interrupt. This is indeed the case for STM32F4, and where DMA brings a key advantage. A DMA transfer is automatically triggered upon a *transmit buffer empty* interrupt to move data from a programmable source memory location (with auto increment) to the destination transmit buffer, all without CPU involvement.

3. *Receive buffer not empty* interrupts for UART reception. This will be discussed when we introduce the UartIn Region.

The following block diagram and table are extracted from *RM0368 Reference manual. STMicroelectronics. DocID025350 Rev 4. May 2015. Pages 167-170.*

Figure 22. DMA block diagram



(Source: RM0368 Reference manual. STMicroelectronics. DocID025350 Rev 4. May 2015. Page 167.)

Table 28. DMA1 request mapping (STM32F401xB/C and STM32F401xD/E)

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX		SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX		SPI3_TX
Channel 1	I2C1_RX	I2C3_RX				I2C1_RX	I2C1_TX	I2C1_TX
Channel 2	TIM4_CH1		I2S3_EXT_RX	TIM4_CH2	I2S2_EXT_TX	I2S3_EXT_TX	TIM4_UP	TIM4_CH3
Channel 3	I2S3_EXT_RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
Channel 4						USART2_RX	USART2_TX	
Channel 5			TIM3_CH4 TIM3_UP		TIM3_CH1 TIM3_TRIG	TIM3_CH2		TIM3_CH3
Channel 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2	I2C3_TX	TIM5_UP	
Channel 7			I2C2_RX	I2C2_RX				I2C2_TX

(Source: RM0368 Reference manual. STMicroelectronics. DocID025350 Rev 4. May 2015. Pages 169.)

Table 29. DMA2 request mapping (STM32F401xB/C and STM32F401xD/E)

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	ADC1				ADC1		TIM1_CH1 TIM1_CH2 TIM1_CH3	
Channel 1								
Channel 2								
Channel 3	SPI1_RX		SPI1_RX	SPI1_TX		SPI1_TX		
Channel 4	SPI4_RX	SPI4_TX	USART1_RX	SDIO		USART1_RX	SDIO	USART1_TX
Channel 5		USART6_RX	USART6_RX	SPI4_RX	SPI4_TX		USART6_TX	USART6_TX
Channel 6	TIM1_TRIG	TIM1_CH1	TIM1_CH2	TIM1_CH1	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
Channel 7								

(Sources: *RM0368 Reference manual. STMicroelectronics. DocID025350 Rev 4. May 2015. Page 170.)*

2.2.3 From Interrupts to Events

Using a similar project layout as in STM32 sample projects we define our interrupt service routines in stm32f4xx_it.cpp. Let's take UART2 transmission as an example. According to Table 28, DMA requests for *USART2_TX* can be mapped to DMA Controller 1 Stream 6 Channel 4. It means that when a DMA transaction has completed an interrupt of the source *DMA1_Stream6* will be generated, which causes the ISR named DMA1_Stream6_IRQHandler() to 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 vector address in the memory map. For STM32F4 the vector table starts at memory location 0x00000004 (0x00000000 is reserved to hold the initial value of the stack pointer). See Table 38 in *RM0368 Reference manual. STMicroelectronics. DocID025350 Rev 4. May 2015. Pages 201.* This is an extract for the DMA1_Stream6 interrupt:

15	22	settable	DMA1_Stream4	DMA1 Stream4 global interrupt	0x0000 007C
16	23	settable	DMA1_Stream5	DMA1 Stream5 global interrupt	0x0000 0080
17	24	settable	DMA1_Stream6	DMA1 Stream6 global interrupt	0x0000 0084
18	25	settable	ADC	ADC1 global interrupts	0x0000 0088
23	30	settable	EXTI9_5	EXTI Line[9:5] interrupts	0x0000 009C

In short, the vector table holds the starting address of the ISR to jump to when the corresponding interrupt occurs. In this example the memory location at 0x00000084 should store the starting address of the function *DMA1_Stream6_IRQHandler()* defined in stm32f4xx_it.cpp.

The vector table itself is defined in *system/src/cmsis/startup_stm32f401xe.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     DMA1_Stream5_IRQHandler
  .word     DMA1_Stream6_IRQHandler
  .word     ADC_IRQHandler
```

Note that all these ISRs are declared with weak linkage in *startup_stm32f401xe.S* as below:

```
   .weak      DMA1_Stream6_IRQHandler                 .thumb_set
DMA1_Stream6_IRQHandler,Default_Handler
```

This means if an ISR is not overridden by a real implementation 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 defined in stm32f4xx_it.cpp:

```
// UART2 TX DMA
// Must be declared as extern "C" in header.
extern "C" void DMA1_Stream6_IRQHandler(void) {
    QXK_ISR_ENTRY();
    UART_HandleTypeDef *hal = UartAct::GetHal(UART2_ACT);
    HAL_DMA_IRQHandler(hal->hdmatx);
    QXK_ISR_ENTRY();
}
```

In `DMA1_Stream6_IRQHandler()` we are trying 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 (UART2_ACT) as the key for lookup.

Next we provide the built-in DMA interrupt handler that comes with the STM32 HAL with 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));
}
```

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);
}
```

Now we get the event *DMA_DONE* generated for an occurrence of the *DMA1_Stream6* interrupt. This event is then fed into the UartOut HSM to drive the desired behaviors as specified by the statechart.