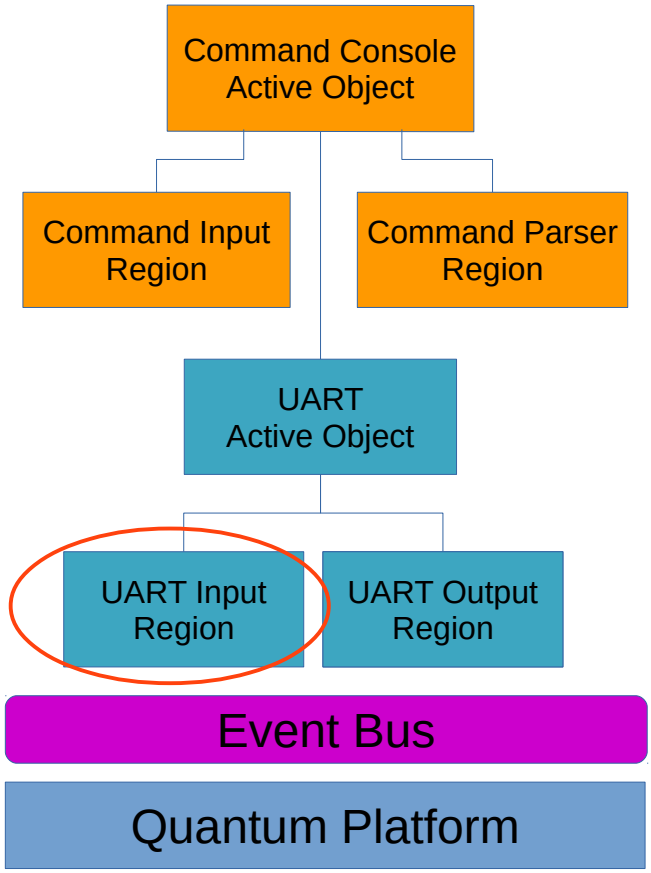
EMBSYS 110 Module 7

Design and Optimization of Embedded and Real-time Systems

1 Introduction

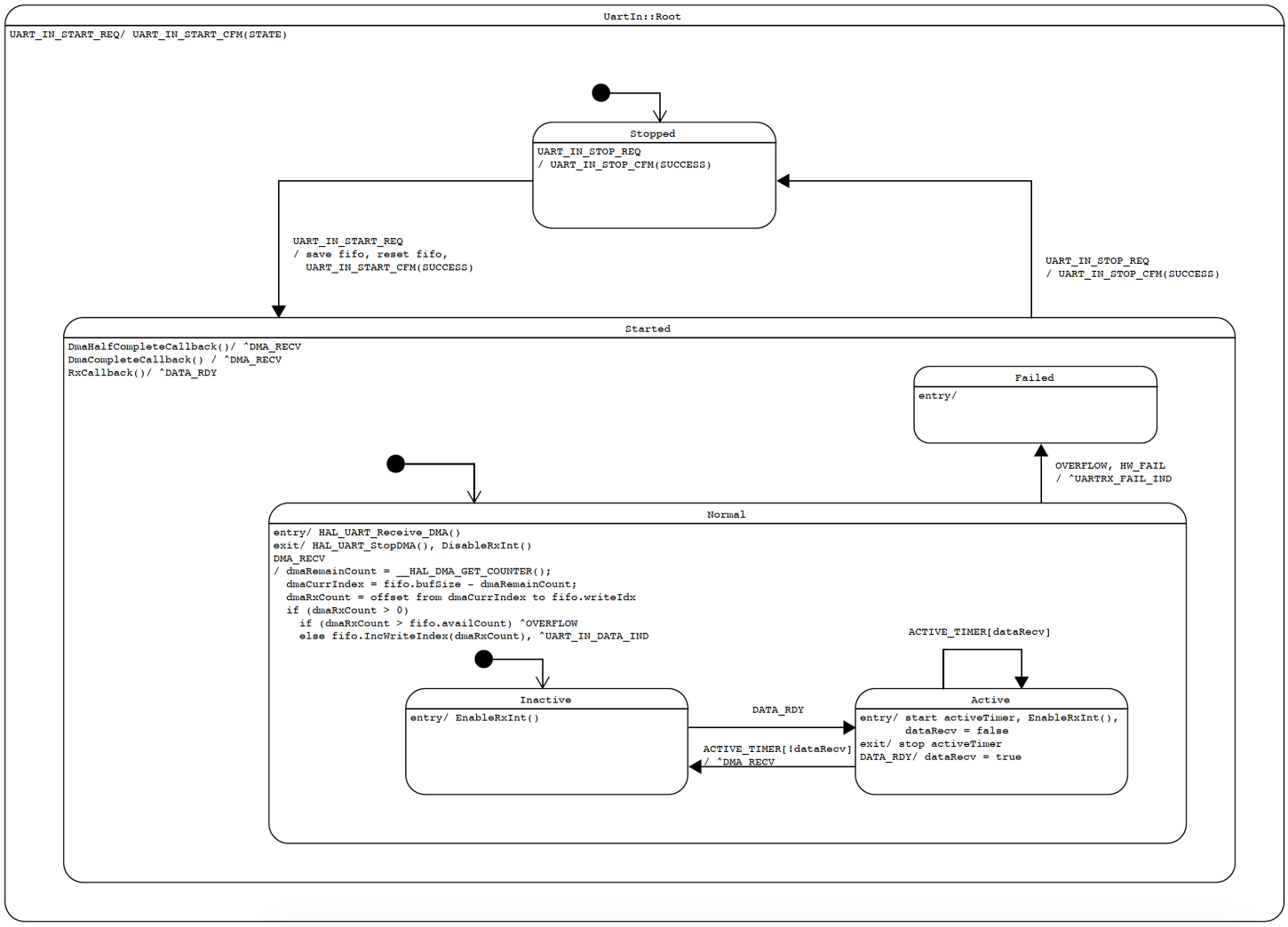
Previously we have looked at the UART active object and the UART Output region. This time we will continue to look at the input path, namely the UART Input region.



2 UartIn Region

The statechart of UartIn is shown below.

Copyright (C) 2019 Lawrence Lo. All rights reserved.



The key design ideas are listed below.

1. It uses DMA to transfer received UART data from the receive buffer to the software 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 transmission and reception paths in UartAct::InitUart(). The code fragment that configures DMA reception is listed below:

```
// Configure the DMA handler forGPIO_InitStruct reception process
m_rxDmaHandle.Instance                 = m_config->rxDmaStream;
m_rxDmaHandle.Init.Channel             = m_config->rxDmaCh;
m_rxDmaHandle.Init.Direction           = DMA_PERIPH_TO_MEMORY;
m_rxDmaHandle.Init.PeriphInc           = DMA_PINC_DISABLE;
m_rxDmaHandle.Init.MemInc              = DMA_MINC_ENABLE;
```

Copyright (C) 2019 Lawrence Lo. All rights reserved.

```
    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);
```

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

2. While UartAct configures DMA reception, it does not enable any DMA transactions. It is the job of UartIn to enable DMA transactions 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 it does is to generate the internal event *DMA_RECV* to itself, which indicates that *DMA data have been received.*

```
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));
}
```

HAL_UART_RxCpltCallback() is a weak callback function provided by the STM32 HAL DMA driver. We override the callback to *map* the HAL object passed in to the HSMN of the corresponding UartIn region.

4. The core of UartIn is done in the handling of DMA_RECV event in the Started-Normal state.

```
case DMA_RECV: {
    EVENT(e);
    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) {
```

Copyright (C) 2019 Lawrence Lo. All rights reserved.

```
            if (dmaRxCount > me->m_fifo->GetAvailCount()) {
                Evt  *evt = new Evt(OVERFLOW, GET_HSMN());
                me->PostSync(evt);
            } else {
                me->m_fifo->IncWriteIndex(dmaRxCount);
                Evt *evt = new UartInDataInd(me->m_client, GET_HSMN(),
                                             GEN_SEQ());
                Fw::Post(evt);
            }
        }
        return Q_HANDLED();
```

Note it is okay if no data have been received when handling DMA_RECV. Note also that the only CPU processing on the FIFO is to increment its write index.

5. Using *DMA Half-completion* or *Completion* interrupts alone would work if UART data keep coming in, as the FIFO would get filled in continuously. However if the UART port interfaces with a command console, relying on those 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 no DMA interrupts would 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 activities on the UART receiving path. Once an activity is detected it enters the Active state. Note that the UART RXNE interrupt is automatically disabled by the ISR and has to be re-enabled as needed by UartIn. This is a common pattern to defer interrupt handling to an HSM (via events), which avoids an interrupt from repeatedly triggering the ISR before it is processed or acknowledged.

   In the Active state, if no UART activities have been detected within the timeout period (say 10ms) it generates the same DMA_RECV event as if an DMA completion interrupt has occurred. Note how all uses of the flag *dataRecv* (cleared, set and tested) are clearly shown in the Active state.

3 Console

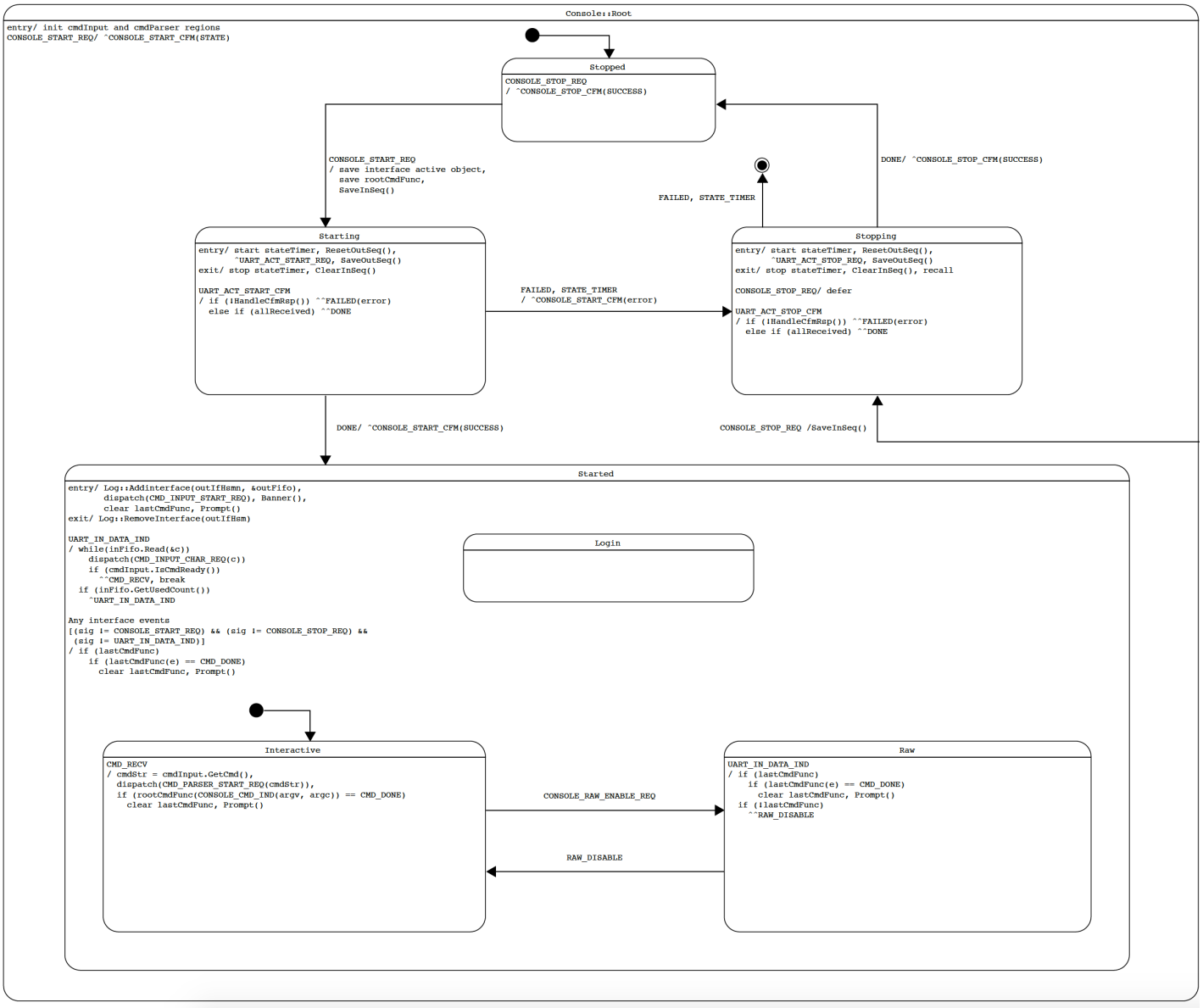
In a desktop or network programming environment you'd take a debug console for granted, which allows you to print out messages for debugging purposes and sometimes send commands to control the program.

In embedded systems a command console like this can be a luxury. You often have to build one from scratch by yourself. Even if a vendor provides one in its SDK it could be limited in features or efficiency.

Copyright (C) 2019 Lawrence Lo. All rights reserved.

Here we introduce a versatile command console based on the UART active object we have previously created. The console by itself an active object named *Console*, and it contains two helper regions named *CmdInput* and *CmdParser*. CmdInput is responsible for processing input characters to detect command line strings and handle command history. CmdParser is responsible for parsing a command line string into arguments. The Console allows a user to define his/her own command handlers. Each command handler acts as an event handler, and therefore not only can it call functions or send events to other active objects, it can also receive events from them. The support of two way event communication is ideal for debugging, system control and integration testing on event-driven systems.

The statechart of Console is shown below:



Copyright (C) 2019 Lawrence Lo. All rights reserved.

A few key design considerations are explained below:

1. In the block diagram shown on the first page we saw that the *Console* active object is dependent on *UartAct*. Typically this is done through a standard technique called dependency injection in which a reference/pointer to the object being depended on is *passed* or *injected* to the depending object via its constructor. For example the constructor of Console would look like this:

   ```
   Console::Console(UartAct *uartAct, ...)
   ```

   In our event framework, active objects do not maintain references to other active objects since they do not call each other's public member functions (API) directly. Due to this higher degree of decoupling, *dependency injection* is done in a different way by passing HSMNs (HSM numbers) rather than pointers to the objects.

   The HSMN of the UartAct instance to be used by the Console is passed via the CONSOLE_START_REQ event. The parameter *ifHsmn* in the event constructor below passes in the HSMN of the UartAct instance serving as the UART interface for the Console object being started:

   ```
   ConsoleStartReq(Hsmn to, Hsmn from, Sequence seq, CmdFunc cmdFunc,
                   Hsmn ifHsmn, bool isDefault = true)
   ```

   This allows multiple instances of *Console* to be created, each depending on any one of the UartAct instances available. The listing below shows the code in main() used to initialize two Console instances (CONSOLE_UART2 and CONSOLE_UART1), with one using the UART2 port and the other using the UART1 port.

   ```
   evt = new ConsoleStartReq(CONSOLE_UART2, HSM_UNDEF, 0, ConsoleCmd,
                             UART2_ACT, true);
   Fw::Post(evt);
   evt = new ConsoleStartReq(CONSOLE_UART1, HSM_UNDEF, 0, ConsoleCmd,
                             UART1_ACT, false);
   Fw::Post(evt);
   ```

   Note – CONSOLE_UART2 is initialized as a "default" interface (last param = true) to which log messages are output by default.

2. Console uses the familiar pattern we have seen before to *start* and *stop* the HSM(s) it depends on. This pattern handles the normal path as well as the exception paths in a robust manner.

3. One of the main responsibilities of Console is to process characters received from the UART port. It detects CR/LF characters which mark the end of command line strings. It implements command history buffers which can be navigated with the UP and DOWN arrow keys. The detailed processing logic is separated into an orthogonal region named *CmdInput*.

   Character reception from the UART port is indicated by the event *UART_IN_DATA_IND*, and its processing is shown in the statechart fragment below.

Copyright (C) 2019 Lawrence Lo. All rights reserved.

```
UART_IN_DATA_IND
/ while(inFifo.Read(&c))
    dispatch(CMD_INPUT_CHAR_REQ(c))
    if (cmdInput.IsCmdReady())
      ^^CMD_RECV, break
  if (inFifo.GetUsedCount())
    ^UART_IN_DATA_IND
```

The corresponding source code is listed below with a few observations:

a) Each received character (*c*) is dispatched to the CmdInput region (*m_cmdInput*). It saves received characters and handles special characters such as CR/LF, UP and DOWN.

b) CmdInput provides a public method *IsCmdReady()* to query if a complete command line string has been detected. If so Console post the internal event CMD_RECV to itself to trigger immediate handling of the received command line.

c) If there are characters remaining in the FIFO after a command line string has been detected, Console *must* send a *reminder* to itself since the UartIn region will not send further UART_IN_DATA_IND unless new data are received.

```
case UART_IN_DATA_IND: {
    char c;
    while(me->m_inFifo.Read(reinterpret_cast<uint8_t *>(&c), 1)) {
        CmdInputCharReq req(me->GetCmdInputHsmn(GET_HSMN()),
                            GET_HSMN(), c);
        me->m_cmdInput.dispatch(&req);
        if (me->m_cmdInput.IsCmdReady()) {
            Evt *evt = new Evt(CMD_RECV, GET_HSMN(), GET_HSMN());
            me->PostSync(evt);
            break;
        }
    }
    if (me->m_inFifo.GetUsedCount()) {
        Evt *evt = new UartInDataInd(GET_HSMN(), GET_HSMN(), 0);
        Fw::Post(evt);
    }
    return Q_HANDLED();
}
```

4. Since CMD_RECV is posted with PostSync(), it is posted to the front of the event queue and hence processed immediately (or synchronously). Upon CMD_RECV, Console gets the command line string from the CmdInput region and dispatches it to the CmdParser region via the event CMD_PARSER_START_REQ.

```
CMD_RECV
/ cmdStr = cmdInput.GetCmd(),
  dispatch(CMD_PARSER_START_REQ(cmdStr)),
  if (rootCmdFunc(CONSOLE_CMD_IND(argv,| argc)) == CMD_DONE)
    clear lastCmdFunc, Prompt()
```

Copyright (C) 2019 Lawrence Lo. All rights reserved.

The corresponding source code is listed below:

```
        case CMD_RECV: {
            EVENT(e);
            STRING_COPY(me->m_cmdStr, me->m_cmdInput.GetCmd().GetRawConst(),
                        sizeof(me->m_cmdStr));
            CmdParserStartReq req(me->GetCmdParserHsmn(GET_HSMN()),
                                  GET_HSMN(), me->m_cmdStr, me->m_argv,
                                  &me->m_argc, ARRAY_COUNT(me->m_argv));
            me->m_cmdParser.dispatch(&req);
            ConsoleCmd ind(GET_HSMN(), me->m_argv, me->m_argc);
            me->RootCmdFunc(&ind);
            return Q_HANDLED();
        }
```

CmdParser gets the command line string from the start request and parsers it into arguments (supporting quoted strings, escaped quote characters, etc). The results are written back to Console::m_argv and Console::m_argc passed in via the start request.

4 Command Handlers

4.1 Command Table

The Console active object supports plugins for command handlers, which allow developers to define their own debug and test commands. Commands are grouped into multiple levels, with each level organized by a command table. The root level of commands are defined in ConsoleCmd.cpp under src/Console.

The root command table looks like this:

```
static CmdHandler const cmdHandler[] = {
    { "test",       Test,       "Test function", 0 },
    { "timer",      Timer,      "Timer test function", 0 },
    { "fib",        Fibonacci,  "Fibonacci generator", 0 },
    { "sys",        SystemCmd,  "System", 0 },
    { "wifist",     WifiStCmd,  "Wifi(stm32) control", 0 },
    { "demo",       DemoCmd,    "Demo from Psicc", 0 },
    { "?",          List,       "List commands", 0 },
};
```

where *CmdHandler* is a type defined as:

```
typedef CmdStatus (*CmdFunc)(Console& console, Evt const *e);
struct CmdHandler {
    char const *key;            // Command to match.
    CmdFunc func;               // Handler function.
```

Copyright (C) 2019 Lawrence Lo. All rights reserved.

```
    char const *text;           // Text description.
    bool isSuper;               // True if only superuser can run it.
};
```
Note *CmdFunc* is a type defined for function pointers commonly used to store a callback functions.

When Console is started, a *root handler function* is registered to it:

```
Evt *evt;
evt = new ConsoleStartReq(CONSOLE_UART2, HSM_UNDEF, 0, ConsoleCmd,
                          UART2_ACT, true);
Fw::Post(evt);
```

Console calls the root handler function (e.g. *ConsoleCmd*) whenever a command line is parsed. It *does not* directly know any command handlers other than this root handler. It is this root handler that passes the root command table to Console:

```
CmdStatus ConsoleCmd(Console &console, Evt const *e) {
    return console.HandleCmd(e, cmdHandler, ARRAY_COUNT(cmdHandler), true);
}
```

4.2 Handler Functions

All handler functions, including the root handler, have the same prototype of:

```
typedef CmdStatus (*CmdFunc)(Console& console, Evt const *e);
```

It receives two parameters:

1. Console &console – A back reference to the Console active object. It allows a handler function to use common utilities provided by Console since as output formatting functions, string to number conversion functions, etc
2. Evt const *e – Every handler function is by itself an event handler, and naturally an event object is passed in to it. This is the central concept in an event driven system – everything is triggered by an event, even for debug and test code. As we will see in examples shortly, the first event to arrive for a detected command is *Console::CONSOLE_CMD*. The corresponding event type *Console::ConsoleCmd* carries the number of arguments and the array of arguments parsed. If a console timer is started, the timeout event is Console::CONSOLE_TIMER.

   Other events can arrive as a result of the interactions between a handler function and other HSMs. For example when a handler function calls a UART output function (e.g. Console::Print()), a *UART_OUT_EMPTY_IND* will arrive as soon as the output FIFO has become empty. This is useful for flow control when a handler needs to send a large amount of data continuously. See Fibonacci() for an example use case.

   Console provides an array of generic variables accessible via the Var() method, along with a generic timer accessible via GetTimer(). The variables are useful to remember *state* information

Copyright (C) 2019 Lawrence Lo. All rights reserved.

between event arrivals for a handler function. A handler function returns *CMD_CONTINUE* to indicate it has not finished processing the command yet, i.e. it expects further events to arrive. Console will keep this handler function as *current*, and dispatch received events to it. When a handler function is done processing the command it returns *CMD_DONE* to inform Console to clear it as the *current* handler. Console will no longer dispatch received events to any handler functions until a new command is detected.

Note that a handler function runs in the context or thread of the Console active object.

4.3 Example Handlers

The following are some examples of command handler functions:

1. Command line parsing (Console/ConsoleCmd.cpp):

```
static CmdStatus Test(Console &console, Evt const *e) {
    switch (e->sig) {
        case Console::CONSOLE_CMD: {
            Console::ConsoleCmd const &ind =
                static_cast<Console::ConsoleCmd const &>(*e);
            console.PutStr("ConsoleCmd Test\n\r");
            console.Print("Command = %s\n\r", ind.Argv()[0]);
            if (ind.Argc() > 1) {
                console.Print("%d arguments:\n\r", ind.Argc() - 1);
                for (uint32_t i = 1; i < ind.Argc(); i++) {
                    console.Print("[%d] %s\n\r", i, ind.Argv()[i]);
                }
            }
            break;
        }
    }
    return CMD_DONE;
}
```

2. Timer testing (Console/ConsoleCmd.cpp):

```
static CmdStatus Timer(Console &console, Evt const *e) {
    switch (e->sig) {
        case Console::CONSOLE_CMD: {
            Console::ConsoleCmd const &ind =
                static_cast<Console::ConsoleCmd const &>(*e);
            if (ind.Argc() < 2) {
                console.PutStr("timer period_ms - timer test\n\r");
                return CMD_DONE;
            }
            uint32_t period = STRING_TO_NUM(ind.Argv()[1], 0);
            console.Print("period = %d\n\r", period);
```

Copyright (C) 2019 Lawrence Lo. All rights reserved.

```
            console.GetTimer().Start(period, Timer::PERIODIC);
            console.Var(0) = 0;
            break;
        }
        case Console::CONSOLE_TIMER: {
            console.Print("timeout %d\n\r", console.Var(0)++);
            break;
        }
    }
    return CMD_CONTINUE;
}
```

3. Continuously UART output (Console/ConsoleCmd.cpp):

```
CmdStatus Fibonacci(Console &console, Evt const *e) {
    uint32_t &prev2 = console.Var(0);
    uint32_t &prev1 = console.Var(1);
    uint32_t &count = console.Var(2);
    switch (e->sig) {
        case Console::CONSOLE_CMD: {
            prev2 = 1;
            prev1 = 1;
            count = 0;
            console.Print("[%d] %d\n\r", count++, prev2);
            console.Print("[%d] %d\n\r", count++, prev1);
            break;
        }
        case UART_OUT_EMPTY_IND: {
            while (1) {
                uint32_t curr = prev2 + prev1;
                prev2 = prev1;
                prev1 = curr;
                uint32_t result = console.Print("[%d] %lu\n\r", count, curr);
                if (result == 0) {
                    break;
                }
                if (++count > 45) {
                    prev2 = 1;
                    prev1 = 1;
                    count = 2;
                }
            }
            break;
        }
    }
    return CMD_CONTINUE;
}
```

Copyright (C) 2019 Lawrence Lo. All rights reserved.

4. Second level handlers (WifiSt/WifiStCommands.cpp):

```
static CmdHandler const cmdHandler[] = {
    { "test",       Test,       "Test function", 0 },
    { "interact",   Interact,   "Interactive mode", 0 },
    { "?",          List,       "List commands", 0 },
};

CmdStatus WifiStCmd(Console &console, Evt const *e) {
    return console.HandleCmd(e, cmdHandler, ARRAY_COUNT(cmdHandler));
}
```

   Refer to the root command table to see how *WifiStCmd* is linked to it.

5 WiFi Module

5.1 Introduction

The WiFi module we use is X-NUCLEO-IDW04A1 from STMicro. First we should check out its user manual to learn about its pin-out and schematics. The document is available from STMicro's website:

UM2183 User manual. Getting started with the X-NUCLEO-IDW04A1 Wi-Fi expansion board based on SPWF04SA module for STM32 Nucleo. DocID030409 Rev 2. July 2017.
(ST UM2183 X-NUCLEO-IDW04A1 User Manual.pdf)

Another good reference is the *STSW-IDW004 WiFi Hands On Training* presentation. It contains step-by-step guide to most of the features supported by the module.

The WiFi module supports UART or SPI interface to a host processor. We will demonstrate how to use the UART interface to connect to our STM32F401 Nucleo board.

5.2 UART Communications

The first thing to do with the module is to establish communications with it. Plug in the module onto the Nucleo board, with care about orientation and pin alignment. After power up the Nucleo board, you should see a blue LED flashing.

In our project we have an active object named *WifiSt* serving as a controller for the ST WiFi module. It depends on the active object *UartAct* to interface with the module over UART. It is started via the event WIFI_ST_START_REQ posted in main():

evt = new `WifiStStartReq`(WIFI_ST, HSM_UNDEF, 0, **UART1_ACT**);
Fw::Post(evt);

The last parameter for the start request event is the HSMN of the UartAct instance to be used for the

Copyright (C) 2019 Lawrence Lo. All rights reserved.

UART interface. As shown above it uses *UART1_ACT* which is configured in UartAct.cpp as:

```
// Define UART configurations.
UartAct::Config const UartAct::CONFIG[] = {
    { UART2_ACT, USART2, USART2_IRQn, USART2_IRQ_PRIO,
      GPIOA, GPIO_PIN_2, GPIO_AF7_USART2, DMA1_Stream6, DMA_CHANNEL_4,
      DMA1_Stream6_IRQn, DMA1_STREAM6_PRIO,
      GPIOA, GPIO_PIN_3, GPIO_AF7_USART2, DMA1_Stream5, DMA_CHANNEL_4,
      DMA1_Stream5_IRQn, DMA1_STREAM5_PRIO },
    { UART1_ACT, USART1, USART1_IRQn, USART1_IRQ_PRIO,
      GPIOA, GPIO_PIN_9, GPIO_AF7_USART1, DMA2_Stream7, DMA_CHANNEL_4,
      DMA2_Stream7_IRQn, DMA2_STREAM7_PRIO,
      GPIOA, GPIO_PIN_10, GPIO_AF7_USART1, DMA2_Stream5, DMA_CHANNEL_4,
      DMA2_Stream5_IRQn, DMA2_STREAM5_PRIO },
};
```

We can see that UART1 uses the GPIO pin **PA9** for Tx and pin **PA10** for Rx. This matches the pin-out of the WiFi module as shown in the User Guide:

Table 9: SPWF04SA module UART interface with STM32 Nucleo board

SPWF04SA Pin/Signal	STM32 pin	Placement
6/TXD	PA9	CN10 – pin 21 CN5 – Pin 1 To use this connection: mount R31 and remove R39 (Default)
	PA3(1)	CN10 – pin 37 CN9 – Pin 1 To use this connection: mount R39 and remove R31
8/RXD	PA10	CN10 – pin 33 CN9 – Pin 3 To use this connection: mount R37 and remove R38 (Default)
	PA2(1)	CN10 – pin 35 CN9 – Pin 2 To use this connection: mount R38 and remove R37
		CN8 – pin 1

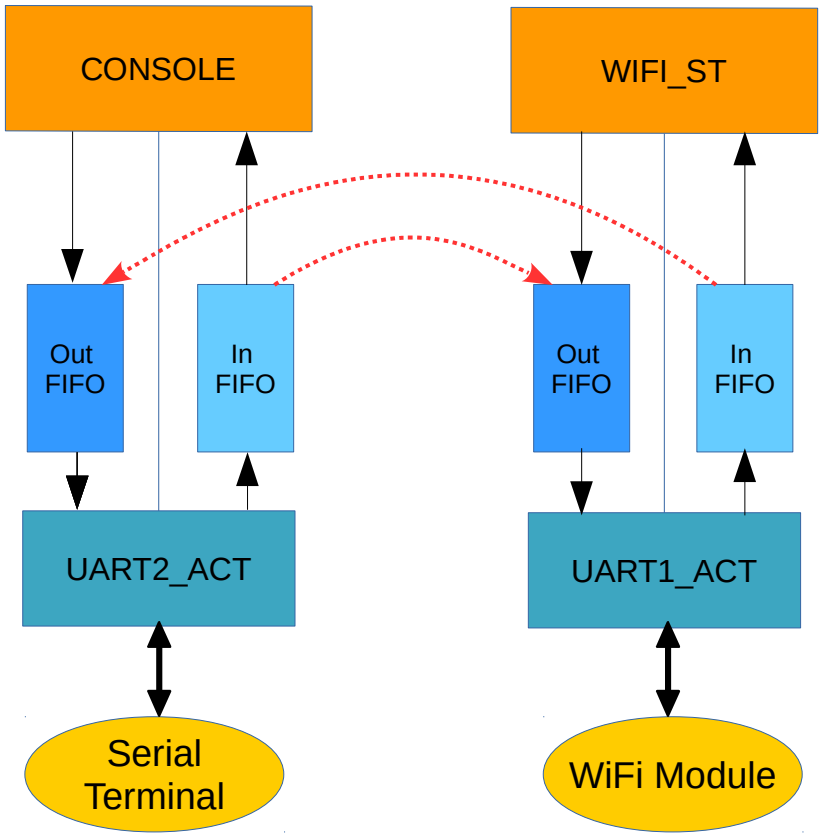
(source: *UM2183 User manual. Getting started with the X-NUCLEO-IDW04A1 Wi-Fi expansion board based on SPWF04SA module for STM32 Nucleo. DocID030409 Rev 2. July 2017. Page 12.)*

With the UART interface setup between the Nucleo board (host processor) and the WiFi module, we should be able to write applications on the Nucleo board to control the WiFi module such as connecting to an access point, send data to or receive data from a remote server. Like many popular WiFi modules (e.g. ESP8266), the ST IDW04A1 module uses an AT command set over UART, which is defined in:

UM2114 User manual TCP/IP protocol stack for SPWF04Sx Wi-Fi modules. DocID029798 Rev 2. November 2017.
(ST UM2114 SPWF04Sx AT Command Reference.pdf)

Copyright (C) 2019 Lawrence Lo. All rights reserved.

For initial development, however, it would be much more convenient to talk to the WiFi module interactively, given that the AT command set is text-based and human-readable. To support that our application simply needs to bridge the *Console* and *WifiSt* active objects together via the same pairs of software FIFOs connected to *UartAct*. This is illustrated in the block diagram below:



5.3 Interactive Mode

At the serial terminal, type "**wifist interactive**" to enter interactive mode to communicate with the WiFi module:

```
348845 CONSOLE_UART2> wifist interact
Enter ST WiFi Interactive mode. Hit CTRL-C to exit
355988 WIFI_ST(14): Normal WIFI_ST_INTERACTIVE_ON_REQ from CONSOLE_UART2(2) seq=2
355988 WIFI_ST(14): Normal EXIT
355988 WIFI_ST(14): Interactive ENTRY
+WIND:21:WiFi Scanning
+WIND:35:WiFi Scan Complete:00
```

Type "**at+s.sts**" to see the status of the module, such as firmware version.

Copyright (C) 2019 Lawrence Lo. All rights reserved.

```
at+s.sts
AT-S.List
AT-S.Var:build=170216-fd39c59-SPWF04S
AT-S.Var:fw_version=1.0.0
AT-S.Var:boot_version=1.0
AT-S.Var:var_version=1
...
AT-S.OK
```

5.4 Firmware Upgrade

Firmware upgrade is risky. Perform this at your own risk. If your WiFi module is corrupted you may not be able to use it. I may not be able to help you restore it.

This section is for reference only and is NOT required.

As seen in the status report, the original firmware version of the WiFi module is 1.0.0. A new firmware version is available on STMicro's website:

http://www.st.com/content/st_com/en/products/embedded-software/wireless-connectivity-software/stsw-wifi004.html

The hex file of the WiFi module firmware (version 1.1.0) is named:

SPWF04S-171117-0328fe3-Full.hex

Refer to *Lab 3.1 : Upgrade through UART* of *STSW-IDW004 WiFi Hands On Training* for upgrade instructions.

This is a summary of the steps:

1. Insert a jumper to JP2 of the WiFi module. Plug the module onto the Nucleo board. Reset both boards (SW1 on module and B2 on Nucleo board).
2. Program a special firmware named *FW_Update_UART_Nucleo-F401RE.bin* on the Nucleo board. It is available in the package *STM32CubeExpansion_WIFI1_V3.0.2* (or later versions) downloaded from STMicro website (under *Utilities/PC_Software/FW_Update_UART* folder).

   One way to program a binary file to the Nucleo board is to use IAR-EARM. See this document for instructions:

   https://www.iar.com/contentassets/554f673197f74bfa9059dd4094d61bee/instructions-how-to-flash-one-bin-file-with-ewarm.pdf

   Note – On the Linker -> Input page, select the binary file listed above under *Raw binary image File*. Set Symbol to "__ICFEDIT_intvec_start__" and *Section* to ".intvec". There is no need to enter MYSYM to *Keep symbols*.

   Use the following linker script file (name it *stm32f401xE.icf* and place it to project folder).

Copyright (C) 2019 Lawrence Lo. All rights reserved.

Select it in *Linker -> Config -> Linker configuration file* setting.

```
define symbol __ICFEDIT_intvec_start__ = 0x08000000;
define symbol __ICFEDIT_region_ROM_start__ = 0x08000000;
define symbol __ICFEDIT_region_ROM_end__   = 0x0807FFFF;
define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x20017FFF;
define symbol __ICFEDIT_size_cstack__ = 0x2000;
define symbol __ICFEDIT_size_heap__   = 0x2000;

define memory mem with size = 4G;
define region ROM_region   = mem:[from __ICFEDIT_region_ROM_start__   to
__ICFEDIT_region_ROM_end__];
define region RAM_region   = mem:[from __ICFEDIT_region_RAM_start__   to
__ICFEDIT_region_RAM_end__];

define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };

initialize by copy { readwrite };
do not initialize  { section .noinit };

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

place in ROM_region   { readonly };
place in RAM_region   { readwrite,
                        block CSTACK, block HEAP };
```

Note – You may see an error message when launching the debug session. If it happens after downloading has completed you can ignore it.

3. Install the PC flasher tool from STMicro:

   http://www.st.com/en/development-tools/flasher-stm32.html

   It is named *stmicroelectronics -> Demonstrator GUI* in the start menu in Windows.

   Follow the instructions in the Hands On Training manual. Note – You may see an error message about erase failure. If this happens you may try to reset the boards, relaunch the PC flasher tool and continue to program the new firmware to the WiFi module.

4. After successfully upgrading firmware on the module, remove the jumper on JP2 and reset the boards. You should see the blue LED flashing again and the following status in the console:

```
at+s.sts
AT-S.List
AT-S.Var:build=171117-0328fe3-SPWF04S
AT-S.Var:fw_version=1.1.0
AT-S.Var:boot_version=1.0
AT-S.Var:var_version=2
...
```

Copyright (C) 2019 Lawrence Lo. All rights reserved.

5.5 Connecting to Access Point

First scan the network for access points (APs):

```
AT+S.FSP=ScanResult,,
1:       BSS C0:7C:D1:9A:F5:F8 CHAN: 01 RSSI: -86 SSID: 'Comcastic' CAPS: 0431 WPA2 WPS
2:       BSS 6E:E2:0C:DF:9F:70 CHAN: 01 RSSI: -76 SSID: 'John's Guest Network' CAPS: 1511 WPA WPA2
3:       BSS 6C:70:9F:DF:0C:E2 CHAN: 01 RSSI: -76 SSID: 'John's Wi-Fi Network' CAPS: 1511 WPA2
4:       BSS B0:7F:B9:A0:B9:65 CHAN: 08 RSSI: -84 SSID: 'NETGEAR_EXT' CAPS: 0C01 WPS
...
```

Connect to the desired AP by providing the SSID and password as follows:

```
AT+S.WIFI=0
AT+S.SSIDTXT=<ssid>
AT+S.SCFG=wifi_wpa_psk_text,<password>
AT+S.SCFG=wifi_priv_mode,2
AT+S.SCFG=wifi_mode,1
AT+S.WIFI=1
```

The connection will be immediately established and every time when the module is reset:

```
+WIND:1:Poweron:171117-0328fe3-SPWF04S
+WIND:13:Copyright (c) 2012-2017 STMicroelectronics, Inc.  All rights Reserved:SPWF04SA
+WIND:0:Console active
+WIND:3:Watchdog Running:20
+WIND:32:WiFi Hardware Started
+WIND:19:WiFi Join:10:C3:7B:A3:AF:60
+WIND:25:WiFi Association successful:<ssid>
+WIND:51:WPA Handshake Complete
+WIND:24:WiFi Up:0:192.168.1.129
+WIND:24:WiFi Up::fe80:0:0:0:280:e1ff:febd:d23c
+WIND:84:NTP Server delivery:2018.3.16:5:1.52.45
```

5.6 WebSocket Example

Here we set up a websocket connection from the WiFi module to a demo server at *echo.websocket.org* at port 80:

```
AT+S.WSOCKON=echo.Websocket.org,80,,,,,,,
AT-S.On:0
AT-S.OK
```

Next we write 5 bytes of data to the socket:

```
AT+S.WSOCKW=0,1,1,0,5
Hello
AT-S.OK
+WIND:88:WebSocket Data:0:1:1:5:5
```

You can query the status of the socket (ID = 0) with the WSOCKQ command. It will show the number of received bytes pending to be read:

Copyright (C) 2019 Lawrence Lo. All rights reserved.

```
AT+S.WSOCKQ=0
AT-S.Query:5
AT-S.OK
```

To get a list of open websockets, type the command:

```
AT+S.WSOCKL
AT-S.List:0:1:5:80
AT-S.OK
```

To read pending received data, type:

```
AT+S.WSOCKR=0,
AT-S.Reading:5:5
HelloAT-S.OK
```

6 IMU Module

We will learn how to integrate the STMicro IKS01A1 sensor module to our platform. First we download the user manual to learn about its components, pin-out and schematics:

UM1820 User manual Getting started with motion MEMS and environmental sensor expansion board for STM32 Nucleo. DocID026959 Rev 4. STMicroelectronics. May 2015.
(*ST UM1820 IKS01A1 User manual.pdf*)

This module contains 4 components (chips) with 6 sensors altogether:

1. LSM6DS0 – Accelerometer and gyroscope
2. LIS3MDL – Magnetometer
3. LPS25HB – Pressure sensor
4. HTS221 – Humidity and temperature sensors.

Note that the module only supports I2C interface to each of the chips. As a result even though some of the chips support the fast SPI interface we won't be able to use it with the IKS01A1 module.

STMicro provides a driver library for the module. It can be downloaded at:

http://www.st.com/content/st_com/en/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32cube-expansion-packages/x-cube-mems1.html

We will be integrating part of the drivers into our statechart framework. We will see the challenges of integrating a typical *blocking* driver into a *non-blocking* event-driven system. In the end we will appreciate how nicely they together on top of our QP-based statechart framework. The details will be explained in the class.

Copyright (C) 2019 Lawrence Lo. All rights reserved.