7. Менеджер Прерываний

Управление аппаратными средствами связанного с асинхронными событиями. Большинство из них приходят от аппаратной периферии, к примеру: таймер достигает некоторого заданного значения периода, или UART который сообщает о прибытии данных. Другие порождаются «внешним окружением» нашей платы. К примеру, нажатие пользователем идиотского переключателя, может привести к «зависанию» платы, и вы тратите целый день, пытаясь понять, где допустили ошибку

Все микроконтроллеры имеют аппаратную поддержку прерываний. Прерывание - это асинхронное событие, при срабатывании которого происходит остановка выполняемого кода. Срабатывание происходит в приоритетном порядке(чем более важное прерывание, тем выше его приоритет, это означает, что низкоприоритетные прерывания будет приостановлены прерываниями с более высоким приоритетом). Специальная процедура, вызываемая в ответ на прерывание, для его обработки называется Interrupt Service Routine (ISR). (Процедура Обработки Прерываний )

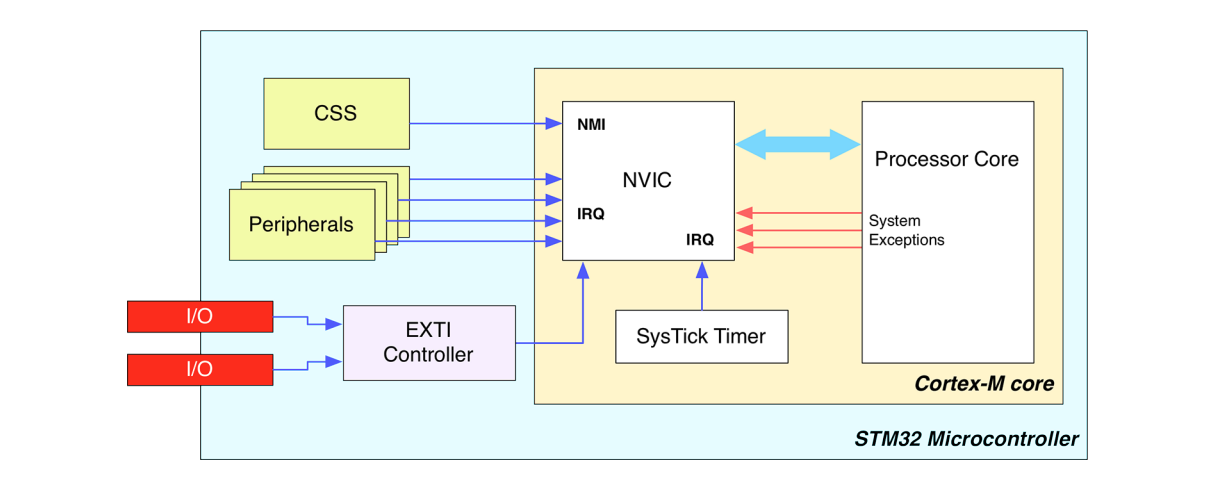
Прерывания обеспечивают многозадачность: аппаратное обеспечение обеспечивает сохранение текущего контекста исполнения (то есть стэковый кадр, счетчик команд, и несколько других вещей) перед вызовом ISR. Они используются RTOS для представления идеи задач. Без поддержки со стороны апаратного обеспечения, невозможно иметь настоящую вытесняющую многозадачность, которая бы позволяла переключаться между несколькими контекстами исполнения без непоправимой потери текущего исполняемого потока.

Прерывания могут вызываться как аппаратным обеспечением, так и программным. Архитектура ARM различает два типа: прерывания вызываемые аппаратным обеспечением и исключения вызываемые программным обеспечением (к примеру, доступ к недействительному участку памяти). В терминологии ARM, прерывания - тип исключений.

Процессоры Cortex-M предоставляю специальный юнит предназначенный для управления исключениями. Она называется Nested Vectored Interrupt Controller (NVIC) и эта глава посвящена программированию этого действительно фундаментального аппаратного компонента. Однако, здесь мы работаем исключительно с обработкой прерываний. Обработка исключений будет рассмотренная в следующей главе, посвященной продвинутой отладке.

7.1 NVIC контроллер

NVIC это обособленный(существующий как отдельная независимая сущность) аппаратный юнит микроконтроллеров на базе ядра Cortex-M ответственный за обработку исключений. На Рис 1 показаны взаимодействия между юнитом NVIC, Процесcорным Ядром и периферией. Здесь мы можем выделить два типа периферии: внешнюю по отношению к ядру Cortex-M, но включенную в STM32 MCU(такая как таймеры, UART и т.д.) и периферийные устройства, полностью внешние по отношению к MCU. Источником прерывания последнего типа периферийных устройств являются I/O порты MCU, которые могут быть сконфигурированы как порты I/O общего назначения (к примеру, кнопка может быть подключена к порту сконфигурированному как вход) или для управления внешней периферией (к примеру, I/Os настроен для изменение данных с помощью интернет phy(прим: <https://ru.wikipedia.org/wiki/PHY>) через RMII интерфейс). Обособленный программируемый контроллер называется External Interrupt/Event Controller (EXTI), он отвечает за взаимодействие между внешними сигналы I/O и контроллером NVIC, как мы можем видеть на изображении ниже.

Рис 1: взаимодействия между контроллером NVIC, ядром Cortex-M и периферией STM32

Как было сказанно до этого, ARM различает системные исключения, которые возникают внутри CPU, и апаратные исключения приходяшие от внешней переферии, так же называемые Interrupt Requests (IRQ). Программисты управляют исключениями ипользуя специальные ISR, которые написанны на языке высокого уровня(чаще всего используя язык С). The processor knows where to locate these routines thanks to an indirect table containing the addresses in memory of Interrupt Service Routines. This table is commonly called vector table, and every STM32 microcontrollers defines its own. Let us analyze this in depth.

7.1.1 Таблица векторов в STM32

Для все процессоров Cortex-M определен фиксированый набор исключений( 15 для ядер Cortex-M3/4/7 и тринадцать для ядер Cortex-M0/0+)

All Cortex-M processors define a fixed set of exceptions (fifteen for the Cortex-M3/4/7 cores and thirteen for Cortex-M0/0+ cores) common to all Cortex-M families and hence common to all STM32- series. Мы уже встречались с ними в Главе 1. Here, you can find the same table (Table 1) for your convenience. It is a good idea to take a quick look to these exceptions (we will study fault exceptions better in a following chapter dedicated to advanced debugging). • Reset: this exception is raised just after the CPU resets. Its handler is the real entry point of the running firmware. In an STM32 application all starts from this exception. The handler contains some assembly-coded functions designed to initialize the execution environment, such as the main stack, the .bss area, etc. A following chapter dedicated to the booting process will explain this deeply. • NMI: this is a special exception, which has the highest priority after the Reset one. Like the Reset exception, it cannot be masked (that is disabled), and it can be associated to critical and non-deferrable activities. In STM32 microcontrollers it is linked to the Clock Security System (CSS). CSS is a self-diagnostic peripheral that detects the failure of the HSE. If this happens, HSE is automatically disabled (this means that the internal HSI is automatically enabled) and a NMI interrupt is raised to inform the software that something is wrong with the HSE. More about this feature in Chapter 10.

* Hard Fault: is the generic fault exception, and hence related to software interrupts. When the other fault exceptions are disabled, it acts as a collector for all types of exceptions (e.g., a memory access to an invalid location raised the Hard Fault exceptions if the Bus Fault one is not enabled). • Memory Management Fault¹: it occurs when executing code attempts to access an illegal location or violates a rule of the Memory Protection Unit (MPU). More about this in a following chapter. • Bus Fault¹: it occurs when AHB interface receives an error response from a bus slave (also called prefetch abort if it is an instruction fetch, or data abort if it is a data access). Can also be caused by other illegal accesses (e.g. an access to a non existent SRAM memory location). • Usage Fault¹: it occurs when there is a program error such as an illegal instruction, alignment problem, or attempt to access a non-existent co-processor. • SVCCall: this is not a fault condition, and it is raised when the Supervisor Call (SVC) instructions is called. This is used by Real Time Operating Systems to execute instructions in privileged state (a task needing to execute privileged operations executes the SVC instruction, and the OS performs the requested operations - this is the same behavior of a system call in other OS). • Debug Monitor¹: this exception is raised when a software debug event occurs while the processor core is in Monitor Debug-Mode. It is also used as exception for debug events like breakpoints and watchpoints when software based debug solution is used. • PendSV: this is another exception related to RTOS. Unlike the SVCall exception, which is executed immediately after a SVC instruction is executed, the PendSV can be delayed. This allows the RTOS to complete tasks with higher priorities. • SysTick: this exception is also usually related to RTOS activities. Every RTOS needs a timer to periodically interrupt the execution of current code and to switch to another task. All STM32 microcontrollers provide a SysTick timer, internal to the Cortex-M core. Even if every other timer may be used to schedule system activities, the presence of a dedicated timer ensures portability among all STM32 families (due to optimization reasons related to the internal die of the MCU, not all timers could be available as external peripheral). Moreover, even if we aren’t using an RTOS in our firmware, it is important to keep in mind that the ST CubeHAL uses the SysTick timer to perform internal time-related activities (and it also assumes that the SysTick timer is configured to generate an interrupt every 1ms). The remaining exceptions that can be defined for a given MCU are related to IRQ handling. Cortex-M0/0+ cores allows up to 32 external interrupts, while Cortex-M3/4/7 cores allows silicon manufacturers to define up to 240 interrupts. Where can we find the list of usable interrupts for a given STM32 microcontrollers? The datasheet of that MCU is certainly the main source about available interrupts. However, we can simply refer to the vector table provided by ST in its HAL. This table is defined inside the startup file for our MCU, the assembly file ending with .S we have learned to import in our Eclipse project in Chapter ¹This exception is not available in Cortex-M0/0+ based microcontrollers.

It is important to clarify some things about the vector table. 1. The name of the exception handlers is just a convention, and you are totally free to rename them if you like a different one. They are just symbols (as are variables and functions inside a program). However, keep in mind that the CubeMX software is designed to generate ISR with those names, which are an ST convention. So, you have to rename the ISR name too. 2. As said before, the vector table must be placed at the beginning of the flash memory, where the processor expects to find it. This is a Link Editor job that places the vector table at the beginning of the flash data during the generation of the absolute file, which is the binary file we upload to the flash. In a following chapter we will study the content of ldscripts/sections.ld file, which contains the directives to instruct GNU LD about this.

**7.2 Активация Прерываний**

Когда STM32 MCU загружается, только исключения Reset, NMI и Card Fault включены по умолчанию. Остальные исключения и прерывания периферии отключены и они должны быть включены по запросу. Для включения IRQ, CubeHAL предоставляет следующую функцию:

void HAL\_NVIC\_EnableIRQ(IRQn\_Type IRQn);

где IRQn\_Type перечисление содержащее все исключения и прерывания определенных для конкретного MCU. IRQn\_Type часть ST Device HAL, and it is defined inside a header file specific for the given STM32 MCU in the Eclipse folder system/include/cmsis/. These files are named stm32fxxxx.h. For example, for an STM32F030R8 MCU the right filename is stm32f030x8.h (the pattern name of these files is the same of start-up files).

Соответствующей функцией для отключения IRQ является:

void HAL\_NVIC\_DisableIRQ(IRQn\_Type IRQn);

It is important to remark that the previous two function enable/disable an interrupt at the NVIC controller level. Looking a Figure 1, you can see that an interrupt line is asserted by the peripheral connected to that line. For example, the USART2 peripheral asserts the interrupt line that corresponds to the USART2\_IRQn interrupt line inside the NVIC controller. This means that the single peripheral must be properly configured to work in interrupt mode. As we will see in the remain of this book, the majority of STM32 peripherals are designed to work, among the others, in interrupt mode. By using specific HAL routines we can enable the interrupt at peripheral level. For example, using the HAL\_USART\_Transmit\_IT() we implicitly configure the USART peripheral in interrupt mode. Clearly, it is also required to enable the corresponding interrupt at NVIC level by calling the HAL\_NVIC\_EnableIRQ().

Теперь, самое время поиграть с прерываниями.

7.2.1 External Lines and NVIC

As we have seen in Figure 1, STM32 microcontrollers provide a variable number of external interrupt sources connected to the NVIC through the EXTI controller, which in turn is capable to manage several EXTI lines. The number of interrupt sources and lines depends on the specific STM32 family. GPIO are connected to the EXTI lines, and it is possible to enable interrupts for every MCU GPIO, even if the most of them share the same interrupt line. For example, for an STM32F4 MCU, up to 114 GPIOs are connected to 16 EXTI lines. However, only 7 of these lines have an independent interrupt associated with them.

Figure 3 shows EXTI lines 0, 10 and 15 in an STM32F4 MCU. All Px0 pins are connected to EXTI0, all Px10 pins are connected to EXTI10 and all Px15 pins are connected to EXTI15. However, EXTI lines 10 and 15 share the same IRQ inside the NVIC (and hence are serviced by the same ISR)³. This means that:

* Only one PxY pin can be a source of interrupt. For example, we cannot define both PA0 and PB0 as input interrupt pins.
* For EXTI lines sharing the same IRQ inside the NVIC controller, we have to code the corresponding ISR so that we must be able to discriminate which lines generated the interrupt.

Figure 3: The relation between GPIO, EXTI lines and corresponding ISR in an STM32F4 MCU

The following example⁴ shows how to use interrupts to toggle the LD2 LED every time we press the user-programmable button, which is connected to the PC13 pin. First, we configure in the GPIO PC13 to fire an interrupt every time it goes from the low level to the high one (lines 49:52). This is accomplished setting GPIO .Mode to be equal to GPIO\_MODE\_IT\_RISING (for the complete list of available interrupt related modes, refer to Table 2 in Chapter 6). Next, we enable the interrupt of the EXTI line associated with the Px13 pins, that is EXTI15\_10\_IRQn

7.2.2 Enabling Interrupts With CubeMX

CubeMX can be used to easily enable IRQs and to automatically generate the ISR code. The first step is to enable the corresponding EXTI line using the Chip view, as shown in Figure 5.

Once we have enabled an IRQ, we need to instruct CubeMX to generate the corresponding ISR. This configuration is done through the Configuration view, clicking on the NVIC button. A list of ISRs that can be enabled appears, as shown in Figure 6.

**8. Универсальный асинхронный приёмопередатчик (UART)**

На сегодняшний день, в электронной индустрии, существует действительно большое количество последовательных протоколов обмена и аппаратных интерфейсов. Большинство из них фокусируются на высокой пропускной способности, к примеру такие современные стандарты как USB 2.0/3.0, Firewire (IEEE 1394) и так далее. Some of these standards come from the past, but are still widespread especially as communication interface between modules on the same board. Один из них Universal Synchronous/Asynchronous Receiver/Transmitter interface, так же известный как USART.

Почти каждый микрокотроллер предоставляет как минимум один переферийный интерфейс UART. Почти каждый STM32 MCU предоставляет как минимум два интерфейса UART/USART, но большенство из них предоставляют более двух интерфейсов( у некоторых количества доходит до восьми интерфейсовs) according the number of I/O supported by the MCU package.

In this Chapter we will see how to program this really useful peripheral using the CubeHAL. Moreover, we will study how to develop applications using the UART both in polling and interrupt modes, leaving the third operative mode, the DMA, to the next chapter.

8.1 Введение в UART и USART

Before we start diving into the analysis of the functions provided by the HAL to manipulate universal serial devices, it is best to take a brief look to the UART/USART interface and its communication protocol. When we want two exchange data between two (or even more) devices, we have two alternatives: we can transmit it in parallel, that is using a given number of communication lines equal to the size of the each data word (e.g., eight independent lines for a word made of eight bits), or we can transmit each bit constituting our word one by one. A UART/USART is a device that translates a parallel sequence of bits (usually grouped in a byte) in a continuous stream of signals flowing on a single wire. When the information flows between two devices inside a common channel, both devices (here, for simplicity, we will refer to them as the sender and the receiver) have to agree on the timing, that this how long it takes to transmit each individual bit of the information. In a synchronous transmission, the sender and the receiver share a common clock generated by one of the two devices (usually the device that acts as the master of this interconnection system).

Figure 1: A serial communication between two devices using a shared clock source

In Figure 1 we have a typical timing diagram¹ showing the Device A sending one byte (0b01101001) serially to the Device B using a common reference clock. The common clock is also used to agree on when to start sampling the sequence of bits: when the master device starts clocking the dedicated line, it means that it is going to send a sequence of bits. In a synchronous transmission the transmission speed and duration are defined by the clock: its frequency determines how fast we can transmit a single byte on the communication channel². But if both devices involved in data transmission agree on how long it takes to transmit a single bit and when to start and finish to sample transmitted bits, than we can avoid to use a dedicated clock line. In this case we have an asynchronous transmission.

Figure 2 shows the timing diagram of an asynchronous transmission. The idle state (that is, no transmission occurring) is represented by the high signal. Transmission begins with a START bit, which is represented by the low level. The negative edge is detected by the receiver and 1.5 bit periods after this (indicated in Figure 1s T1.5bit), the sampling of bits begins. Eight data bits are sampled. The least significant bit (LSB) is typically transmitted first. An optional parity bit is then transmitted (for error checking of the data bits). Often this bit is omitted if the transmission channel is assumed to be noise free or if there are error checking higher up in the protocol layers. The transmission is ended by a STOP bit, which last 1.5 bits.

8.2 Инициализация UART

8.2.1 Конфигурация UART с помощью CubeMX

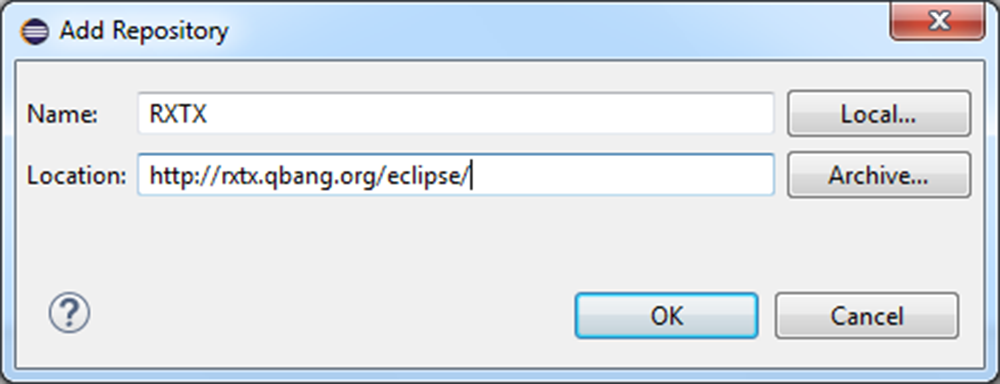
8.3 UART Communication in Polling Mode

8.3.1 Установка терминала на последовательном порте в Windows

Под Windows существует простое и надежное решение. Оно основано на двух плагинах. Первый плагин - это обертка над RXTX¹¹ Java library. Для его установки, перейдите Help->Install software… меню, затем нажмите кнопку Add… и заполните поля следующим образом (см. Изображение 6):

Name: RXTX

Location: <http://rxtx.qbang.org/eclipse/>

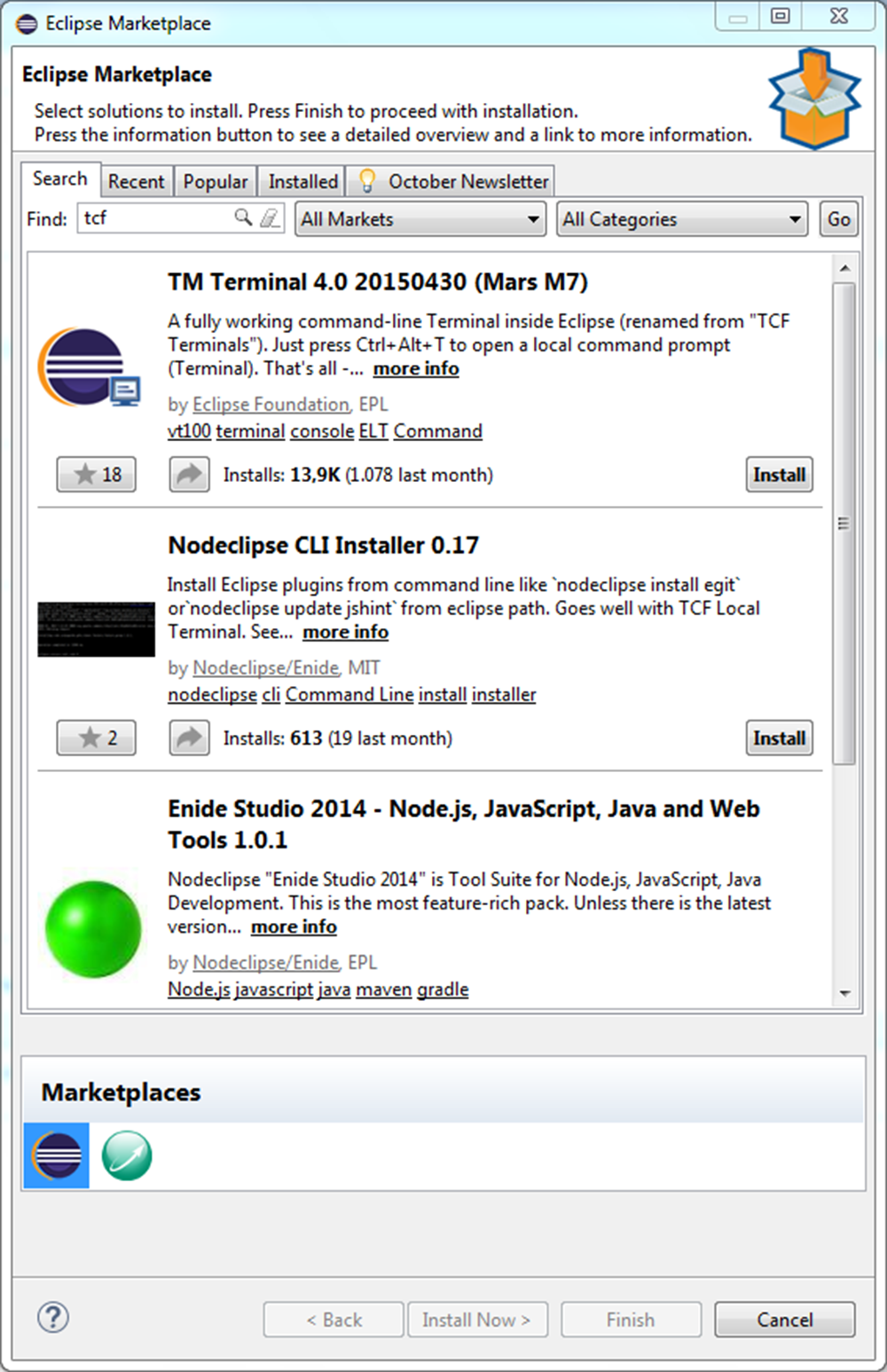


Изображение 6: Диалог добавления нового репозитория плагинов

Нажмите OK и установите релиз RXTX 2.1-7r4 следуя инструкциям. Как только установка будет завершена, перейдите Help->Eclipse Marketplace…. В поле Find введите “tcf”. Через некоторое время должен появиться плагин TM Terminal, как показано на Изображении 7. Нажмите Install и следуйте инструкциям. По запросу, перезапустите Eclipse.

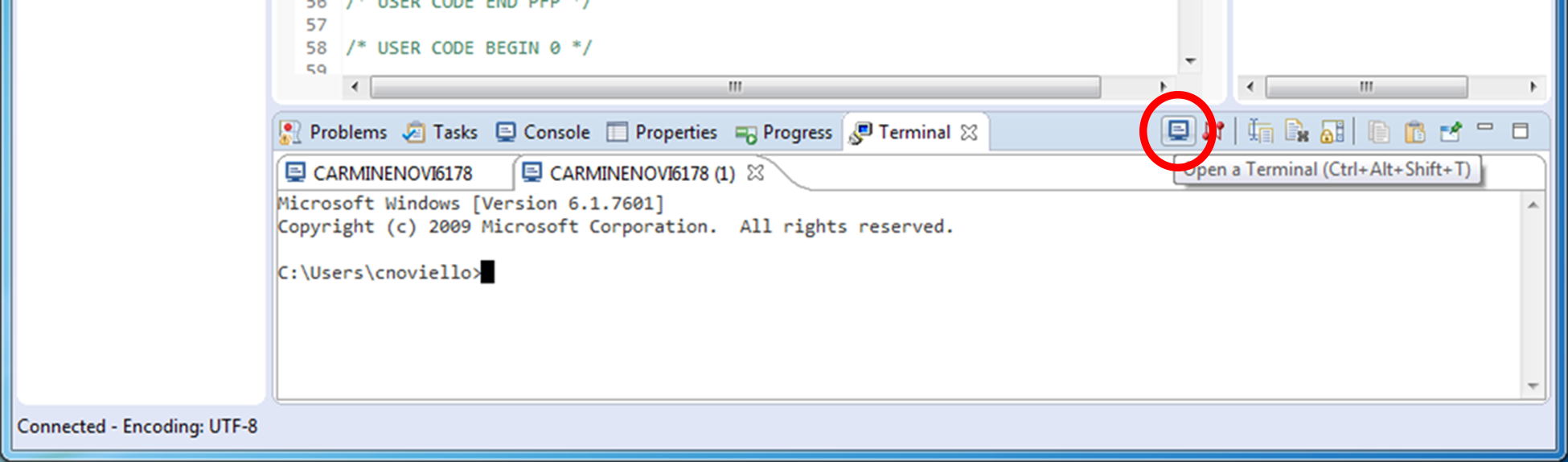
\_\_\_

⁹http://bit.ly/1jsQjnt ¹⁰http://www.columbia.edu/kermit/ ¹¹<http://rxtx.qbang.org/>



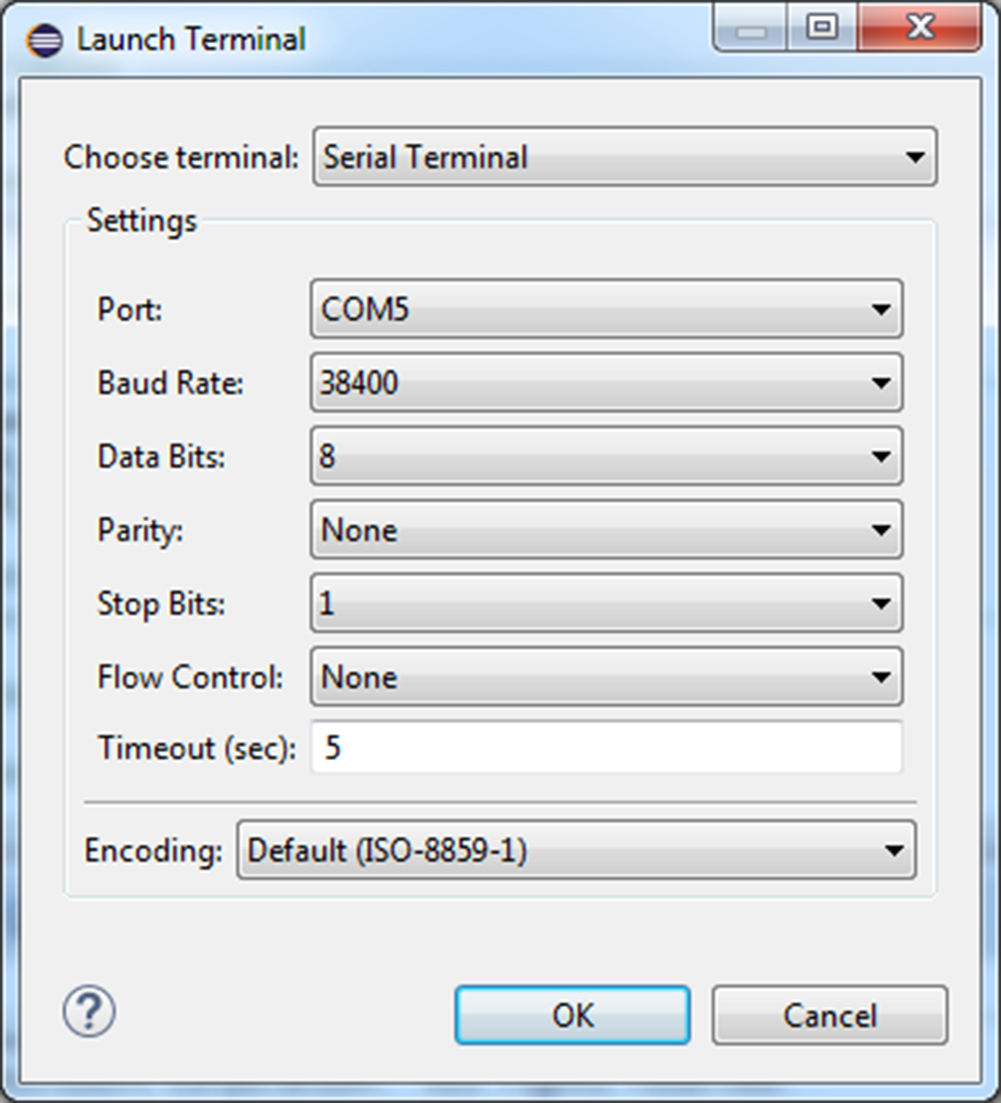
Изображение 7: Eclipse Marketplace

Для открытия панели терминала вы можете просто нажать Ctrl+Alt+T, или вы можете перейти Window->Show View->Other… меню и найти область Terminal.



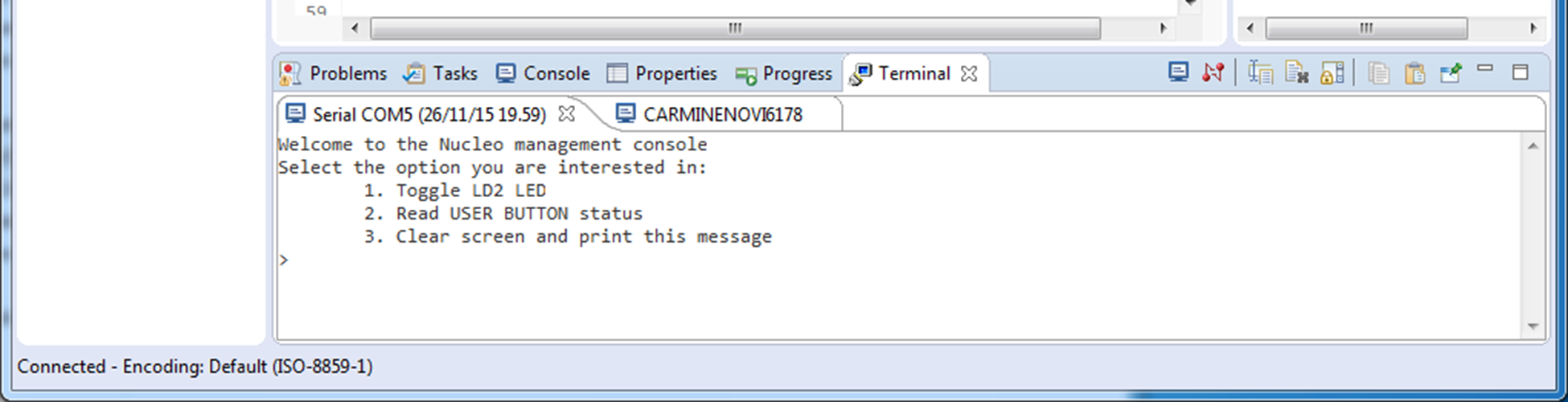
Изображение 8: Как запустить новый терминал

Поумолчанию, панель терминала открывает новое приглашение командной строки. Нажмите на иконку Open a Terminal(обведено красным кругом на Изображении 8). В диалоговом окне Launch Terminal (смотрите Изображение 9) выберете Serial Terminal в качестве типа терминала, затем выберете COM-порт соотвествующий Nucleo VCP, и Baud Rate(скорость передачи) установите 38400 Бит/сек. Нажмите Ок.



Изображение 9: Диалог конфигурирования терминала

Теперь вы можете сбросить Nucleo. Консоль управления, которую мы запрограммировали используя библиотеку HAL\_UART, должна появится в окне терминала на последовательном порте, как показано на Изображение 10.



Изображение 10: Консоль управления Nucleo отображается в терминале

8.3.2 Установка терминала на последовательном порте в Linux и MacOS X

8.4 UART Communication in Interrupt Mode

Let us consider again the first example of this chapter. What’s wrong with it? Since our firmware is all committed to this simple task, there is nothing wrong by using the UART in polling mode. The MCU is essentially blocked waiting for the user input (the HAL\_MAX\_DELAY timeout value blocks the HAL\_UART\_Receive() until one char is sent over the UART). But what if our firmware has to accomplish other cpu-intensive activities in real-time? Suppose to rearrange the main() from the first example in the following way:

while (1) {

opt = readUserInput();

processUserInput(opt);

if(opt == 3) 42

goto printMessage;

performCriticalTasks();

}

In this case we cannot block the execution of function processUserInput() waiting for the user choice, but we have to specify a much more short timeout value to the HAL\_UART\_Receive() function, otherwise performCriticalTasks() is never executed. However, this could cause the loss of important data coming from the UART peripheral (remember that the UART interface has a one byte wide buffer).

To address this issue the HAL offers another way to exchange data over a UART peripheral: the interrupt mode. To use this mode, we have to accomplish the following tasks:

• To enable the USARTx\_IRQn interrupt and to implement the corresponding USARTx\_IRQHandler() ISR.

• To call HAL\_UART\_IRQHandler() inside the USARTx\_IRQHandler(): this will perform all activities related to management of interrupts generated by the UART peripheral¹².

• To use the functions HAL\_UART\_Transmit\_IT() and HAL\_UART\_Receive\_IT() to exchange data over the UART. These functions also enables the interrupt mode of the UART peripheral: in this way the peripheral will assert the corresponding line in the NVIC controller so that the ISR is raised when an event occurs.

• To rearrange our application code to deal with asynchronous events.

Before we rearrange the code from the first example, it is best to take a look to the available UART interrupts and to the way HAL routines are designed.

\_\_\_\_\_

¹²If we use CubeMX to enable the USARTx\_IRQn from the NVIC configuration section (as shown in Chapter 7), it will automatically place the call to the HAL\_UART\_IRQHandler() from the ISR.

8.4.1 UART Related Interrupts

Every STM32 USART peripheral provides the interrupts listed in Table 6. These interrupts include both IRQs related to data transmission and to communication errors. They can be divided in two groups:

\*IRQs generated during transmission: Transmission Complete, Clear to Send or Transmit Data Register empty interrupt.

\*IRQs generated while receiving: Idle Line detection, Overrun error, Receive Data register not empty, Parity error, LIN break detection, Noise Flag (only in multi buffer communication) and Framing Error (only in multi buffer communication).

Table 6: The list of USART related interrupts

| Interrupt Event | Event Flag | Enable Control Bit |
| --- | --- | --- |
| Transmit Data Register Empty | TXE | TXEIE |
| Clear To Send (CTS) flag | CTS | CTSIE |
| Transmission Complete | TC | TCIE |
| Received Data Ready to be Read | RXNE | RXNEIE |
| Overrun Error Detected | ORE | RXNEIE |
| Idle Line Detected | IDLE | IDLEIE |
| Parity Error | PE | PEIE |
| Break Flag | LBD | LBDIE |
| Noise Flag, Overrun error and Framing Error in multi buffer communication | NF or ORE or FE | EIE |

These events generate an interrupt if the corresponding Enable Control Bit is set (third column of Table 6). However, STM32 MCUs are designed so that all these IRQs are bound to just one ISR for every USART peripheral (see Figure 11¹³). For example, the USART2 defines only the USART2\_- IRQn as IRQ for all interrupts generated by this peripheral. It is up to the user code to analyze the corresponding Event Flag to infer which interrupt has generated the request.

\_\_\_\_\_\_\_

13The Figure 9s taken from the STM32F030 Reference Manual (RM0390).

Figure 11: How the USART interrupt events are connected to the same interrupt vector

The CubeHAL is designed to automatically do this job for us. The user is warned about the interrupt generation thanks to a series of callback functions invoked by the HAL\_UART\_IRQHandler(), which must be called inside the ISR.

From a technical point of view, there is not so much difference between UART transmission in polling and in interrupt mode. Both the methods transfer an array of bytes using the UART Data Register (DR) with the following algorithm:

\*From a technical point of view, there is not so much difference between UART transmission in polling and in interrupt mode. Both the methods transfer an array of bytes using the UART Data Register (DR) with the following algorithm:

\*For data reception, wait until the Received Data Ready to be Read(RXNE) is not asserted true, and then store the content of the USART->DR register inside the application memory.

The difference between the two methods consists in how they wait for the completion of data transmission. In polling mode, the HAL\_UART\_Receive()/HAL\_UART\_Transmit() functions are designed so that it waits for the corresponding event flag to be set, for every byte we want to transmit. In interrupt mode, the function HAL\_UART\_Receive\_IT()/HAL\_UART\_Transmit\_IT() are designed so that they do not wait for data transmission completion, but the dirty job to place a new byte inside the DR register, or to load its content inside the application memory, is accomplished by the ISR routine when the RXNEIE/TXEIE interrupt is generated¹⁴. To transmit a sequence of bytes in interrupt mode, the HAL defines the function:

\_\_\_\_\_

¹⁴This is the reason why transferring a sequence of bytes in interrupt mode is not a smart thing when the communication speed is too high, or when we have to transfer a great amount of data very often. Since the transmission of each byte happens quickly, the CPU will be congested by the interrupts generated by the UART for every byte transmitted. For continuous transmission of great sequences of bytes at high speed is best to use the DMA mode, as we will see in the next chapter

HAL\_StatusTypeDef HAL\_UART\_Transmit\_IT(UART\_HandleTypeDef \*huart, uint8\_t \*pData, uint16\_t Size);

where:

\*huart: it is the pointer to an instance of the struct UART\_HandleTypeDef seen before, which identifies and configures the UART peripheral;

\*pData: it is the pointer to an array, with a length equal to the Size parameter, containing the sequence of bytes we are going to transmit; the function will not block waiting for the data transmission, and it will pass the control to the main flow as soon as it completes to configure the UART.

Conversely, to receive a sequence of bytes over the USART in interrupt mode the HAL provides the function:

HAL\_StatusTypeDef HAL\_UART\_Receive\_IT(UART\_HandleTypeDef \*huart, uint8\_t \*pData, uint16\_t Size);

where: • huart: it is the pointer to an instance of the struct UART\_HandleTypeDef seen before, which identifies and configures the UART peripheral; • pData: it is the pointer to an array, with a length at lest equal to the Size parameter, containing the sequence of bytes we are going to receive. The function will not block waiting for the data reception, and it will pass the control to the main flow as soon as it completes to configure the UART. Now we can proceed rearranging the first example.

As you can see in the above code, the first step is to enable the USART2\_IRQn and to assign it a priority¹⁵. Next, we define the corresponding ISR inside the stm32xxxx\_it.c file (not shown here) and we add the call to the HAL\_UART\_IRQHandler() function inside it. The remaining part of the example file is all about restructuring the readUserInput() and processUserInput() functions to deal with asynchronous events. The function readUserInput() now checks for the value of the global variable UartReady. If it is equal to SET, it means that the user has sent a char to the management console. This character is contained inside the global array readBuf. The function then calls the HAL\_UART\_Receive\_IT() to receive another character in interrupt mode. When readUserInput() returns a value greater than 0, the function processUserInput() is called. Finally, the function HAL\_UART\_RxCpltCallback(), which is automatically called by the HAL when one byte is received, is defined: it simply sets the global UartReady variable, which in turn is used by the readUserInput() as seen before. It is important to clarify that the function HAL\_UART\_RxCpltCallback() is called only when all the bytes specified with the Size parameter, passed to the HAL\_UART\_Receive\_IT() function, are received. What about the HAL\_UART\_Transmit\_IT() function? It works in a way similar to the HAL\_UART\_- Receive\_IT(): it transfers the next byte in the array every time the Transmit Data Register Empty(TXE) interrupt is generated. However, special care must be taken when calling it multiple times. Since the function returns the control to the caller as soon as it finishes to setup the UART, a subsequent call of the same function will fail and it will return the HAL\_BUSY value. Suppose to rearrange the function printWelcomeMessage() from the previous example in the following way:

void printWelcomeMessage(void) { HAL\_UART\_Transmit\_IT(&huart2, (uint8\_t\*)"\033[0;0H", strlen("\033[0;0H")); HAL\_UART\_Transmit\_IT(&huart2, (uint8\_t\*)"\033[2J", strlen("\033[2J")); HAL\_UART\_Transmit\_IT(&huart2, (uint8\_t\*)WELCOME\_MSG, strlen(WELCOME\_MSG)); HAL\_UART\_Transmit\_IT(&huart2, (uint8\_t\*)MAIN\_MENU, strlen(MAIN\_MENU)); HAL\_UART\_Transmit\_IT(&huart2, (uint8\_t\*)PROMPT, strlen(PROMPT)); }

The above code will never work correctly, since each call to the function HAL\_UART\_Transmit\_IT() is much faster than the UART transmission, and the subsequent calls to the HAL\_UART\_Transmit\_IT() will fail.

\_\_\_

¹⁵The example is designed for an STM32F4. Please, refer to the book examples for your specific Nucleo

If speed is not a strict requirement for your application, and the use of the HAL\_UART\_Transmit\_IT() is limited to few parts of your application, the above code could be rearranged in the following way:

void printWelcomeMessage(void) { char \*strings[] = {"\033[0;0H", "\033[2J", WELCOME\_MSG, MAIN\_MENU, PROMPT}; for (uint8\_t i = 0; i < 5; i++) { HAL\_UART\_Transmit\_IT(&huart2, (uint8\_t\*)strings[i], strlen(strings[i])); while (HAL\_UART\_GetState(&huart2) == HAL\_UART\_STATE\_BUSY\_TX || HAL\_UART\_GetState(&huart2) == HAL\_UART\_STATE\_BUSY\_TX\_RX); } }

Here we transfer each string using the HAL\_UART\_Transmit\_IT() but, before we transfer the next string, we wait to the transmission completion. However, this is just a variant of the HAL\_UART\_- Transmit(), since we have a busy wait for every UART transfer.

A more elegant and performing solution is to use a temporary memory area where to store the byte sequences and to let the ISR to execute the transfer. A queue is the best options to handle FIFO events. There are several ways to implement a queue, both using static and dynamic data structure. If we decide to implement a queue with a predefined area of memory, a circular buffer is the data structure suitable for this kind of applications.

Figure 12: A circular buffer implemented using an array and two pointers

A circular buffer is nothing more than an array with a fixed size where two pointers are used to keep track of the head and the tail of data that still needs to be processed. In a circular buffer, the first and the last position of the array are seen “contiguous” (see Figure 12). This is the reason why this data structure is called circular. Circular buffers have an important feature too: unless our application has up to two concurrent execution streams (in our case, the main flow that places chars inside the buffer and the ISR routine that sends these chars over the UART), they are intrinsically thread safe, since the “consumer” thread (the ISR in our case) will update only the tail pointer and the producer (the main flow) will update only the head one.

Circular buffers can be implemented in several ways. Some of them are faster, others are more safe (that is, they add an extra overhead ensuring that we handle the buffer content correctly). You will find a simple and quite fast implementation in the book examples. Explaining how it is coded is outside the scope of this book. Using a circular buffer, we can define a new UART transmit function in the following way:

uint8\_t UART\_Transmit(UART\_HandleTypeDef \*huart, uint8\_t \*pData, uint16\_t len) { if(HAL\_UART\_Transmit\_IT(huart, pData, len) != HAL\_OK) { if(RingBuffer\_Write(&txBuf, pData, len) != RING\_BUFFER\_OK) return 0; } return 1; }

The function does just two things: it tries to send the buffer over the UART in interrupt mode; if the HAL\_UART\_Transmit\_IT() function fails (which means that the UART is already transmitting another message), then the byte sequence is placed inside a circular buffer. It is up to the HAL\_UART\_TxCpltCallback() to check for pending bytes inside the circular buffer:

void HAL\_UART\_TxCpltCallback(UART\_HandleTypeDef \*huart) { if(RingBuffer\_GetDataLength(&txBuf) > 0) { RingBuffer\_Read(&txBuf, &txData, 1); HAL\_UART\_Transmit\_IT(huart, &txData, 1); } }

The RingBuffer\_Read() it is not really fast as it could be with a more performant implementation. For some real world situations, the whole overhead of the HAL\_UART\_- TxCpltCallback() routine (that is called from the ISR routine) could be too high. If this is your case, you can consider to create a function like the following one:

void processPendingTXTransfers(UART\_HandleTypeDef \*huart) { if(RingBuffer\_GetDataLength(&txBuf) > 0) { RingBuffer\_Read(&txBuf, &txData, 1); HAL\_UART\_Transmit\_IT(huart, &txData, 1); } }

Then, you could call this function from the main application code or in a lower privileged task if you are using an RTOS.

8.5 Обработка ошибок

When dealing with external communications, the error management is an aspect that we must

strongly take in consideration. An STM32 UART peripheral offers some error flags related to

communication errors. Moreover, it is possible to enable a corresponding interrupt to be noticed

when the error occurs.

The CubeHAL is designed to automatically detect error conditions, and to warn us about them. We only need to implement the HAL\_UART\_ErrorCallback() function inside our application code. The HAL\_UART\_IRQHandler() will automatically invoke it in case an error occurs. To understand which error has been occurred, we can check the value of the UART\_HandleTypeDef->ErrorCode field. The list of error codes is reported in Table 7.

Table 7: List of UART\_HandleTypeDef->ErrorCode possible values

|  |  |
| --- | --- |
| UART Error Code | Description |
| HAL\_UART\_ERROR\_NONE | No error occurred |
| HAL\_UART\_ERROR\_PE | Parity check error |
| HAL\_UART\_ERROR\_NE | Noise error |
| HAL\_UART\_ERROR\_FE | Framing error |
| HAL\_UART\_ERROR\_ORE | Overrun error |
| HAL\_UART\_ERROR\_DMA | DMA Transfer error |

The HAL\_UART\_IRQHandler() is designed so that we should not care with the implementation details of UART error management. The HAL code will automatically perform all needed steps to handle the error (like clearing event flags, pending bit and so on), leaving to us the responsibility to handle the error at application level (for example, we may ask to the other peer to resend a corrupted frame).

**Read Carefully**

At the time of writing this chapter, December 2nd 2015, a subtle bug prevents the right management of the Overrun error. You can read more about it on the official ST forum¹⁶. You can reproduce this bug even with the second example of this chapter. Run the example on your Nucleo, and hit the key ‘3’ on your keyboard leaving it pressed. After a while, the firmware will hang. This happens because, after the Overrun error occurs, the HAL does not restart the receiving process again. You can address this bug implementing the HAL\_UART\_ErrorCallback() function in the following way:

void HAL\_UART\_ErrorCallback(UART\_HandleTypeDef \*huart) {

if(huart->ErrorCode == HAL\_UART\_ERROR\_ORE) HAL\_UART\_Receive\_IT(huart, readBuf, 1);

}

}

8.6 I/O Retargeting

В главе 5 мы изучали как использовать semihosting для вывода отладочных сообщений в консоль OpenOCD, используя функцию стандартной библиотеки языка C - printf(). Если вы уже использовали данную возможность, то вы знаете о двух серьезных ограничениях:

* semihosting действительно замедляет выполнение прошивки
* it also prevents your firmware from working if it is executed without a debug session (из-за того что semihosting реализованно с использование програмных брейкпойнтов).

Теперь, когда мы знакомы с управлением UART’ом, мы можем переопределить нужные системные вызовы (\_write(), \_read() и так далее) для перенаправления стандартных потоков STDIN, STDOUT и STDERR в Nucleo USART2. Это можно легко сделать следующим образом:

#if !defined(OS\_USE\_SEMIHOSTING)

#define STDIN\_FILENO 0

#define STDOUT\_FILENO 1

#define STDERR\_FILENO 2

UART\_HandleTypeDef \*gHuart;

void RetargetInit(UART\_HandleTypeDef \*huart) {

gHuart = huart;

/\* Disable I/O buffering for STDOUT stream, so that

\* chars are sent out as soon as they are printed. \*/

setvbuf(stdout, NULL, \_IONBF, 0);

}

int \_isatty(int fd) {

if (fd >= STDIN\_FILENO && fd <= STDERR\_FILENO)

return 1;

errno = EBADF;

return 0;

}

int \_write(int fd, char\* ptr, int len) {

HAL\_StatusTypeDef hstatus;

if (fd == STDOUT\_FILENO || fd == STDERR\_FILENO) {

hstatus = HAL\_UART\_Transmit(gHuart, (uint8\_t \*) ptr, len, HAL\_MAX\_DELAY);

if (hstatus == HAL\_OK)

return len;

else

return EIO;

}

errno = EBADF;

return -1;

}

int \_close(int fd) {

if (fd >= STDIN\_FILENO && fd <= STDERR\_FILENO)

return 0;

errno = EBADF;

return -1;

}

int \_lseek(int fd, int ptr, int dir) {

(void) fd;

(void) ptr;

(void) dir;

errno = EBADF;

return -1;

}

int \_read(int fd, char\* ptr, int len) {

HAL\_StatusTypeDef hstatus;

if (fd == STDIN\_FILENO) {

hstatus = HAL\_UART\_Receive(gHuart, (uint8\_t \*) ptr, 1, HAL\_MAX\_DELAY);

if (hstatus == HAL\_OK)

return 1;

else

return EIO;

}

errno = EBADF;

return -1; }

int \_fstat(int fd, struct stat\* st) {

if (fd >= STDIN\_FILENO && fd <= STDERR\_FILENO) {

st->st\_mode = S\_IFCHR;

return 0;

}

errno = EBADF;

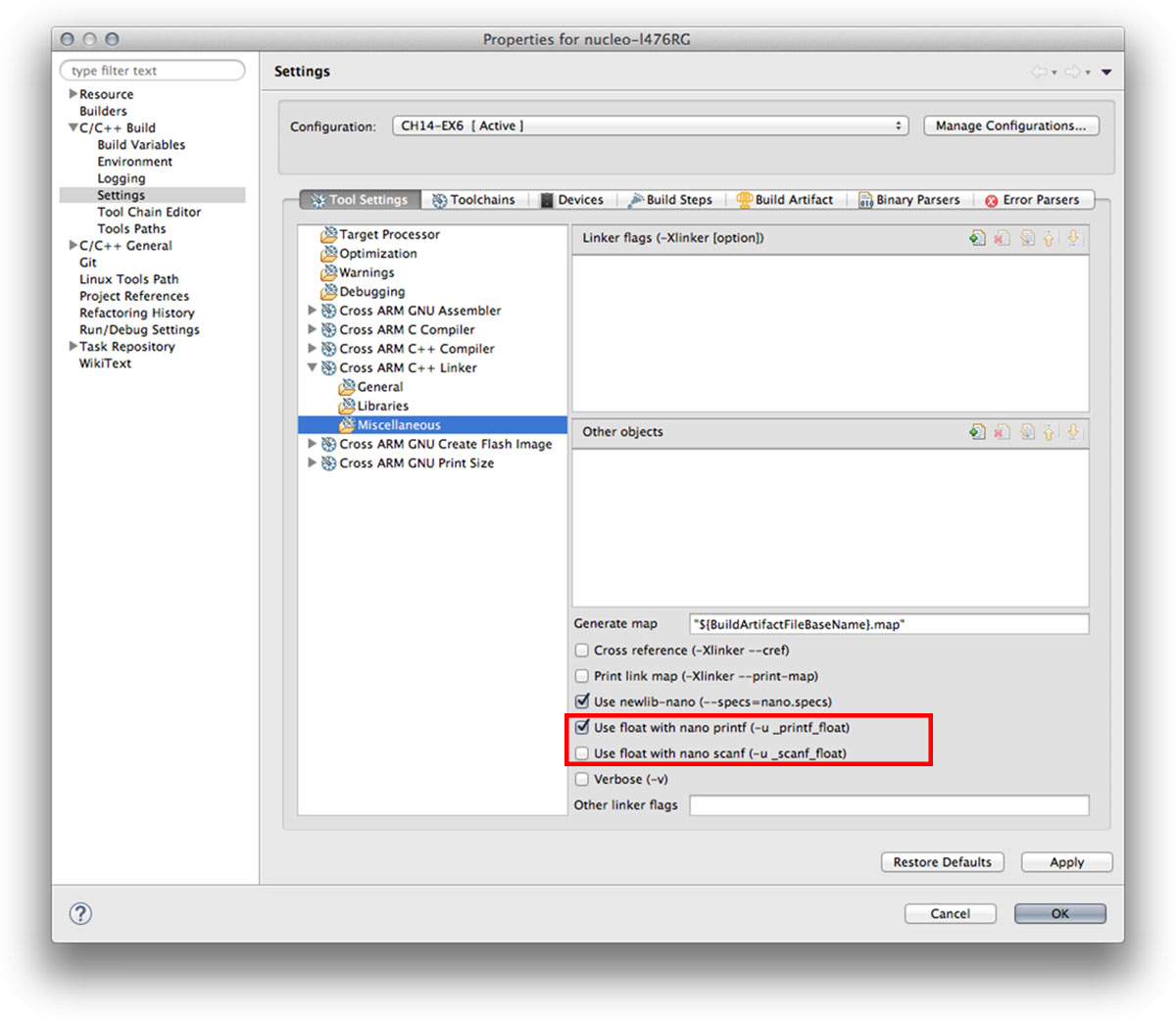
return 0;

}

#endif //#if !defined(OS\_USE\_SEMIHOSTING)

Для перенаправления стандартных потоков в вашей прошивке, вам нужно удалить макрос OS\_USE\_SEMIHOSTING на уровне проекта, и инициализируйте библиотеку вызвав RetargetInit(), передав указатель на UART\_HandleTypeDef(экземпляр UART2). Для примера, следующий код демонстрирует, как использовать функции printf()/scanf() в вашей прошивке:

Если вы намеренным использовать функции printf()/scanf() для вывода/чтения данных типа float в терминал на последовательном порту (так же, если вы собираетесь использовать sprintf() и другие похожие функции), вы должны явно включить поддержку float в newlib-nano, которая является более компактной вариацией C runtime library для встроенных систем. Проделайте следующие шаги, перейдите Project->Properties... меню, далее перейдите C/C++ Build->Settings->Cross ARM C++ Linker->Miscellaneous и установите флаги **Use float with nano printf/scanf** напротив требующихся вам возможностей, это показано на Изображение 13. Данное действие увеличит размер прошивки.



Изображение 13: Как включить поддержку float для printf() и scanf()