CC2511 Week 7

Transition to one lecture per week

- From now onwards, we will have only a single lecture per week for this subject.
- The pace of content delivery is slowing down to give you more time to focus on the design project.
- Use the additional time to work on your project and ask questions of me.

Assignment 1

- **Check the gradebook on LearnJCU** to make sure that you have been marked off for both the drawing and the working implementation.
- You must receive a satisfactory grade on assignment 1 before the end of the semester to pass the subject.
- Don’t wait. Just get it over and done with.

Reminder

Due on Friday:

- Assignment 2 Schematic in Altium format.
- Bill of Materials (BOM) for all the components that you will need to buy in the format of the example (Assignment 1) on LearnJCU.

Today

- Interrupts
- Asynchronous I/O
- String buffers.

Interrupts

- Interrupts provide **event handling**.
- Your program can be **interrupted** in response to external events.
  - For example, a falling edge on a digital input, or a received char on a UART.
- The **interrupt handler** is a special function that gets run when the interrupt occurs.

Why do we need interrupts?

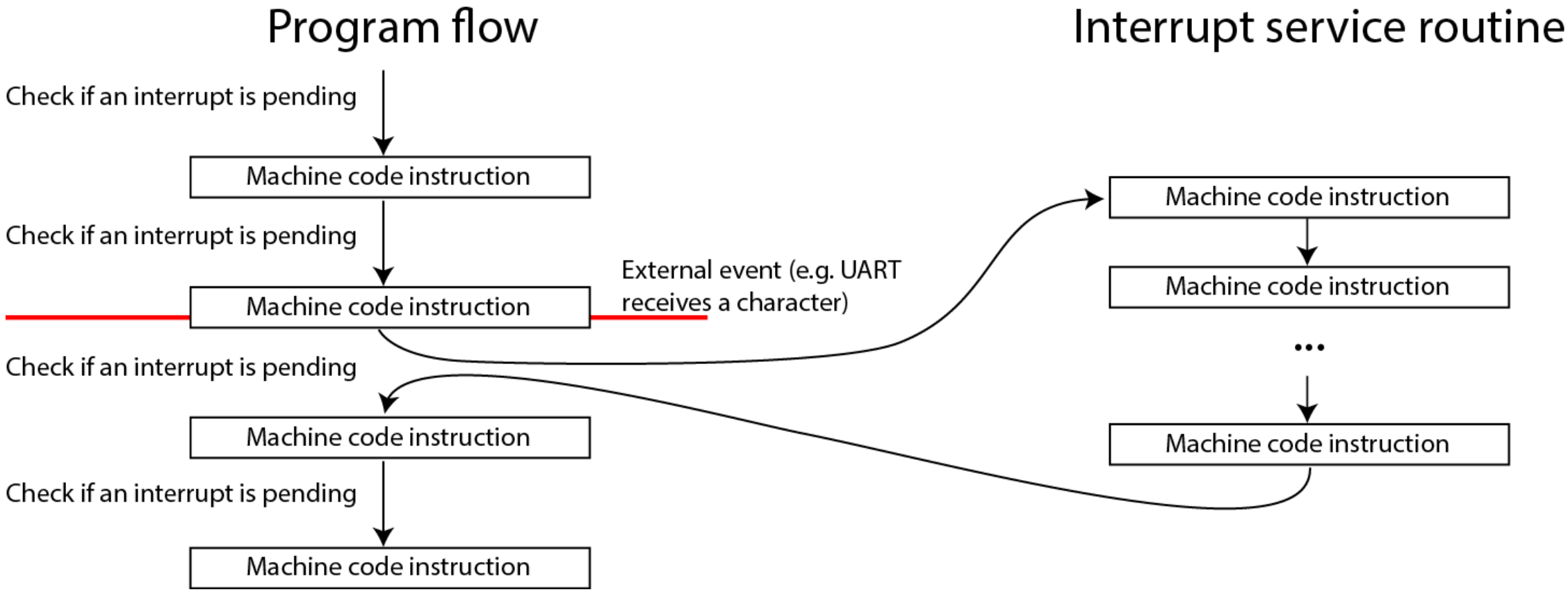
```
for (;;) {
     if (UART received char) { ... }
     if (switch pressed) { ... }
     /* ... many others ... */
}
```

- Continually polling wastes CPU time and consumes power.
- An event may go unnoticed for a long time if the CPU is busy.

The advantage of interrupts

- The event handler runs immediately when the event occurs.
- The CPU can be in a low-power state until an interrupt “wakes” it.
- Interrupt priorities can be defined, so high priority events are handled first.

Interrupts can run between any machine code instructions



How interrupts work

When an interrupt is triggered:

1. The current instruction pointer is saved (so the program can resume after the interrupt has been handled).
2. The memory address of the interrupt handler is located by examining a pre-configured list called the **interrupt vector table**.
3. The relevant entry in the interrupt vector table is loaded into the instruction pointer, thereby transferring control to that function.
4. When the interrupt handler is complete, it restores the previous instruction pointer.

Terminology

- The **interrupt handler** or **interrupt service routine** (**ISR**) is the function that runs in response to an interrupt event.
- The **interrupt vector** is the address of the ISR.
- A **pending** interrupt is one that has been triggered but the ISR has not yet run.
- A **maskable** interrupt can be ignored or disabled in software, e.g. events occurring on peripherals such as GPIO.
- A **non-maskable interrupt (NMI)** cannot be ignored or disabled, e.g. reset or a memory access errors.

Configuring an interrupt handler

- On ARM CPUs, the interrupt vector table starts at address 0 in flash memory.
- Configure the compiler/linker to fill in this vector table with the specified functions.
- Finally, enable (“unmask”) the relevant interrupts by setting bits in the control registers as per the reference manual.

The interrupt vector table in Kinetis

- At the top of the interrupt vector table is the **initial stack pointer** and **initial program counter**.
- These are used when the board is reset, including a “power-on reset” that occurs when power is applied.

```
__attribute__ ((section (".vectortable"))) const tVectorTable __vect_table = { /* Interrupt vector table */

/* ISR name                          No. Address      Pri Name                          Description */
&__SP_INIT,                       /* 0x00  0x00000000   -   ivINT_Initial_Stack_Pointer   used by PE */
{
(tIsrFunc)&__thumb_startup,       /* 0x01  0x00000004   -   ivINT_Initial_Program_Counter used by PE */
(tIsrFunc)&Cpu_INT_NMIInterrupt,  /* 0x02  0x00000008   -2   ivINT_NMI                     used by PE */
```

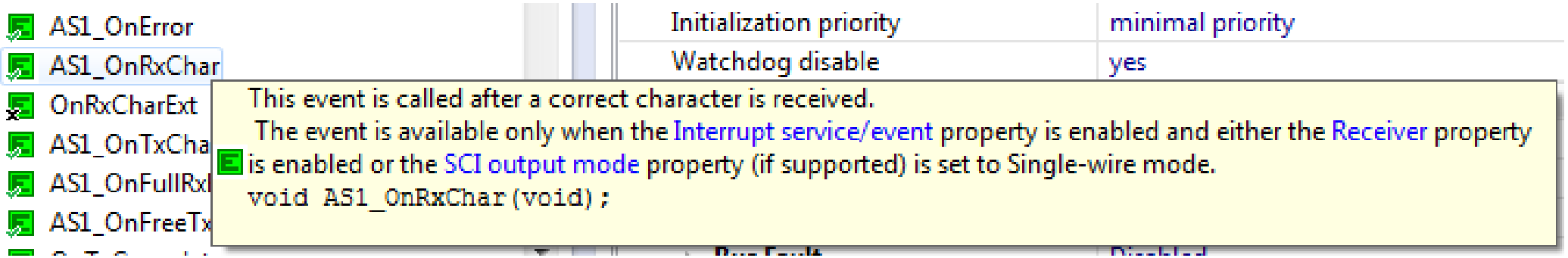
The interrupt vector table in Kinetis

- Processor Expert components insert their own interrupt handlers.
- These pre-written interrupt handlers service the hardware as required to clear the interrupt.
  - Example: often one must read from a data register or else the interrupt will keep triggering.

```
(tIsrFunc)&Cpu_Interrupt,           /* 0x1B  0x0000006C   -   ivINT_I2C0                unused by PE */
(tIsrFunc)&Cpu_Interrupt,           /* 0x1C  0x00000070   -   ivINT_SPI0                unused by PE */
(tIsrFunc)&Cpu_Interrupt,           /* 0x1D  0x00000074   -   ivINT_I2S0_Tx             unused by PE */
(tIsrFunc)&Cpu_Interrupt,           /* 0x1E  0x00000078   -   ivINT_I2S0_Rx             unused by PE */
(tIsrFunc)&Cpu_Interrupt,           /* 0x1F  0x0000007C   -   ivINT_UART0_LON           unused by PE */
(tIsrFunc)&ASerialLdd1_Interrupt,   /* 0x20  0x00000080   8   ivINT_UART0_RX_TX         used by PE */
(tIsrFunc)&ASerialLdd1_Interrupt,   /* 0x21  0x00000084   8   ivINT_UART0_ERR           used by PE */
(tIsrFunc)&Cpu_Interrupt,           /* 0x22  0x00000088   -   ivINT_UART1_RX_TX         unused by PE */
```

Processor Expert Events

- You can add your own code into each ISR using Events in Processor Expert.




```
void AS1_OnRxChar(void)
{
  /* Write your code here ... */
}
```

Event handling

- Components such as AsynchroSerial define events such as “OnRxChar” (received character) and “OnTxChar” (transmitted character).
- These are called from interrupt handlers.
- **They can be triggered at any time, so the code needs to be designed accordingly.**

Interrupt handlers

- When an interrupt service routine (ISR) is running, other interrupts (of the same or lower priority) cannot be serviced.
- Therefore an ISR should be short and simple.
- Usually save the result and defer processing to the main application loop.
- **Inside an ISR, never wait on another event!**

Interacting with hardware in an ISR

- It’s bad form to wait on hardware in an ISR.
- Examples:

1. Set a GPIO pin: OK to do in an ISR because this is fast.
2. Transmit a message over UART: Avoid because this requires a busy-wait loop.

- If your ISR takes a long time, other interrupts will be delayed.

Interacting with software in an ISR

- From the perspective of your main program, **variables that are shared with ISRs could change at any time**.
- This contradicts the usual assumptions of how variables work:

```
bool switch_pressed = false;
while (!switch_pressed) {
    /* wait for ISR to change “switch_pressed” */
}
```

The need for volatile variables

```
bool switch_pressed = false;
while (!switch_pressed) {
     /* wait for ISR to change “switch_pressed” */
}
```

- An optimising compiler could conclude that switch_pressed will never change.
- **All code below the while loop could be deleted** because it can never be reached!

Volatile variables

- The keyword **volatile** informs the compiler that the variable in question might unexpectedly change.
  - For example, by an interrupt service routine.

```
volatile bool switch_pressed = false;
while (!switch_pressed) {
    /* wait for ISR to change “switch_pressed” */
}
```

Consequence of using “volatile”

When you declare a variable as volatile:

- **Reads cannot be eliminated** by optimisation, because the compiler does not assume it has exclusive use of the variable.
- **Apparently redundant writes cannot be eliminated** by optimisation, because the writes may have side-effects elsewhere.

```
int i;
i = 1; // Redundant: can be removed during optimisation
i = 2;
```

Volatile pointers

- To declare a pointer that points into volatile memory:
  - This is probably the one that you want.

```
volatile int *p;
```

- To declare a pointer whose address is volatile, but points into regular memory:

```
int* volatile p;
```

- To declare a pointer whose address is volatile, and points into volatile memory:

```
volatile int* volatile p;
```

Sharing global variables between C files

- Processor Expert places event handlers in a separate C file.
- Communication between your main program loop and event handlers is usually through global variables.
- To share global variables between files, there's another qualifier: **extern**.

Sharing global variables between C files

```
// file1.c
// Outside any function:
int global;
```

- Declare the variable as usual in the first file.

```
// file2.c
// Outside any function:
extern int global;
```

- Use the **extern** qualifier in the second and subsequent C files.

What does “extern” actually mean?

- A variable declared “extern” is not allocated any memory.
- Extern means that another C file is responsible for the allocation, i.e. it is “externally allocated”.
- The actual memory location becomes known at the linking stage.

Initial values and extern

```
// main.c
// Outside any function:
int index = 0;
```

```
// events.c
// Outside any function:
extern int index;
```

An initial value (if present) must be specified in the file where the variable is declared.

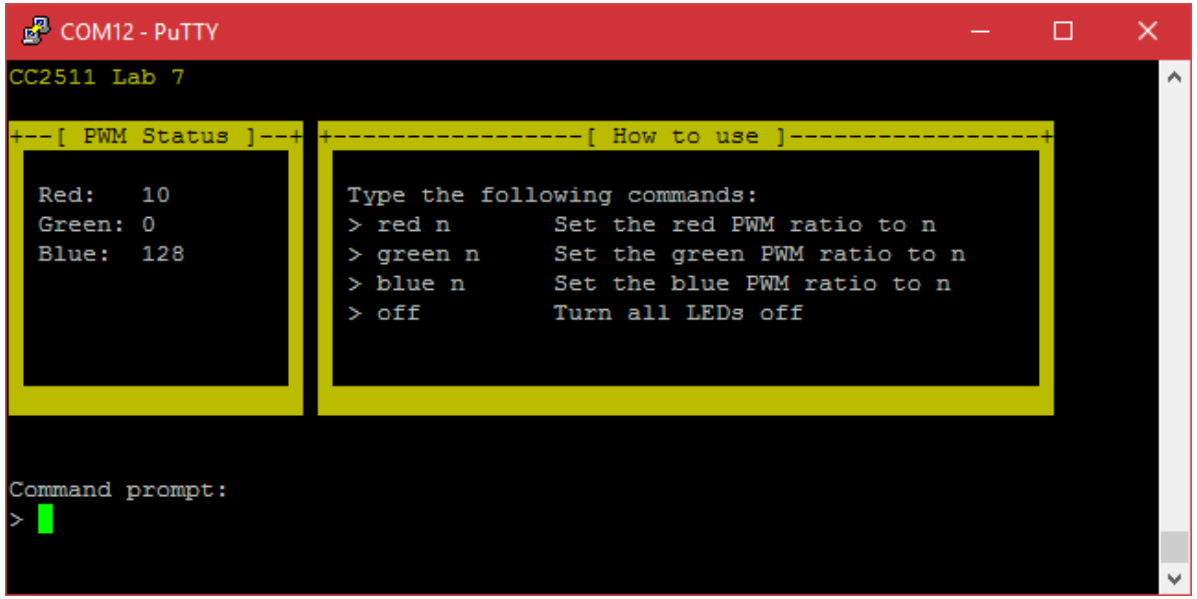
Combining volatile and extern

```
// main.c
// Outside any function:
volatile int global = 0;
```

```
// events.c
// Outside any function:
extern volatile int global;
```

Using interrupts in practice: build a command-line user interface

- A **command-line user interface** is a convenient way to interact with an embedded system.
- The user types commands that are executed when the Enter key is pressed.

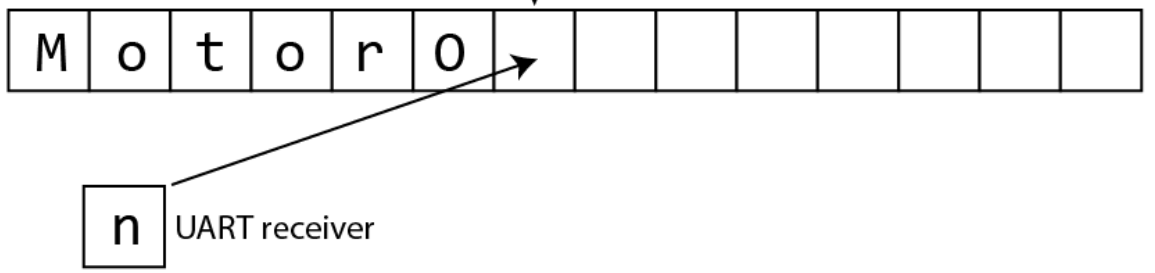


Key concept: buffer

- A **buffer** is an array that is used to hold data being transmitted, received, generated or processed.
- Typical use: receive characters one at a time and place them into the buffer until a complete message has arrived.

```
char buf [100];
int buf_index;
```

Recommended convention:
Index variable refers to first empty space.




```
buf[buf_index] = /* new char from UART */;
buf_index++;
```

How to build a command-line interface?

- In a receive ISR, place characters into a string buffer.
- Once the enter key is pressed, interpret (“parse”) the string.

- Allocate the buffer (as a global variable, outside any function):

```
volatile char buffer [100];
volatile unsigned int index = 0;
```

- The variables must be volatile because they are shared between the main loop and an ISR.

Character buffers

```
volatile char buffer [100];
volatile unsigned int index = 0;
```

- There are various conventions for how to manage a buffer.
  - A pointer to the first empty item, a pointer to the last filled item, the index of the last filled item, …
- **We will define `buffer[index]` as the first unused item.**
- In other words, the index variable counts the number of characters in the buffer.

Filling up a buffer: the event handler

```
// Assuming these variables are declared without “extern” in another file
extern volatile char buffer [100];
extern volatile unsigned int index; // no initial value on extern declarations
void AS1_OnRxChar() {
    char c;
    if (ERR_OK == AS1_RecvChar(&c)) {
        buffer[index] = c; // save the character
        index++; // increment the index
        // TODO echo the char back to the user
        // TODO handle the enter key
    }
}
```

Detecting the enter key

- A terminal emulator (e.g. PuTTY) might send ‘\r’ (carriage return) and/or ‘\n’ (linefeed) when the Enter key is pressed.
- Detect these characters to identify the end of the command.
- Don’t forget to add a trailing NULL to the buffer!

```
buffer[index] = 0; // add a trailing NULL
```

Buffer overflow

- Must not accept new characters when the buffer is full:
  - Compare the index variable with the size of the buffer.
  - Discard characters that would overflow the buffer.
  - Don’t forget to save space for the trailing NULL.

Managing string buffers

To write into string buffers:

- Place characters one-by-one (e.g. as they arrive); or
- Use **snprintf** for string formatting and number-to-string conversion.

To read from string buffers:

- Use **strcmp** to compare a buffer against a known string; or
- Use **sscanf** to read “formatted input” including string-to-number conversion.

snprintf

- **snprintf** writes formatted text into a buffer.

```
#include <stdio.h>

#define LENGTH 20
char buf [LENGTH];
snprintf(buf, LENGTH, "PWM ratio: %i\n", pwm_ratio);
```

String compare

- To compare strings, use **strcmp** or **strcmpi**
  - strcmp is case sensitive.
  - strcmpi is case insensitive.
- Returns zero when the strings are equal.
  - Does not return a boolean!

```
if (0 == strcmp(buf, "stop")) {
    // buf contains stop
}
```

Using sscanf

- Example: interpret the command “PWM 100”

```
char buf [100];
int pwm;

// ... read text into buf

sscanf(buf, "PWM %i", &pwm);
```

sscanf

```
char buf [100];
int pwm;

// read text into buf

sscanf(buf, "PWM %i", &pwm);
```

- This will read the literal string “PWM”, then a space, then an integer.
- sscanf will do a string-to-integer conversion.
- The next argument is a pointer for where to write the converted integer.

sscanf: return value

- Did the string match the format?
- **The return value of sscanf is the number of conversions performed.**

```
if (sscanf(buf, "PWM %i", &pwm) == 1) {
     // the string matched the format, i.e.
     // the command “PWM” was entered correctly.
}
```

Format specifiers

```
unsigned int i;
sscanf(buf, "PWM %i", &pwm);
```

- Problem! “%i” is for signed integers but the output is unsigned.
- Must be exactly precise about the data types when using sscanf.

Format specifiers

Format specifier	C type	Meaning
`%f`	`float`	Single precision float
`%lf`	`double`	Double precision float
`%d`	`int`	Signed integer (base 10)
`%i`	`int`	Signed integer (leading 0x means hex and leading 0 means octal)
`%u`	`unsigned int`	Unsigned integer (decimal format)
`%hi`	`short int`	Short signed integer (16 bits)
`%hu`	`short unsigned int`	Short unsigned integer (16 bits)
`%hhi`	`signed char`	Signed char (8 bits). *Only on compilers supporting the C90 standard.*
`%hhu`	`unsigned char`	Unsigned char (8 bits). *Only on compilers supporting the C90 standard.*

Parsing multiple commands

- Repeatedly call sscanf for each command.

```
int16 val;
if (sscanf(buf, "pwm %hu", &val) > 0) {
    // command was PWM
} else if (sscanf(buf, "led %hu", &val) > 0) {
    // command was led
}
```

- sscanf returns the number of conversions performed.
  - The return value will be zero if the format string doesn’t match the buffer.

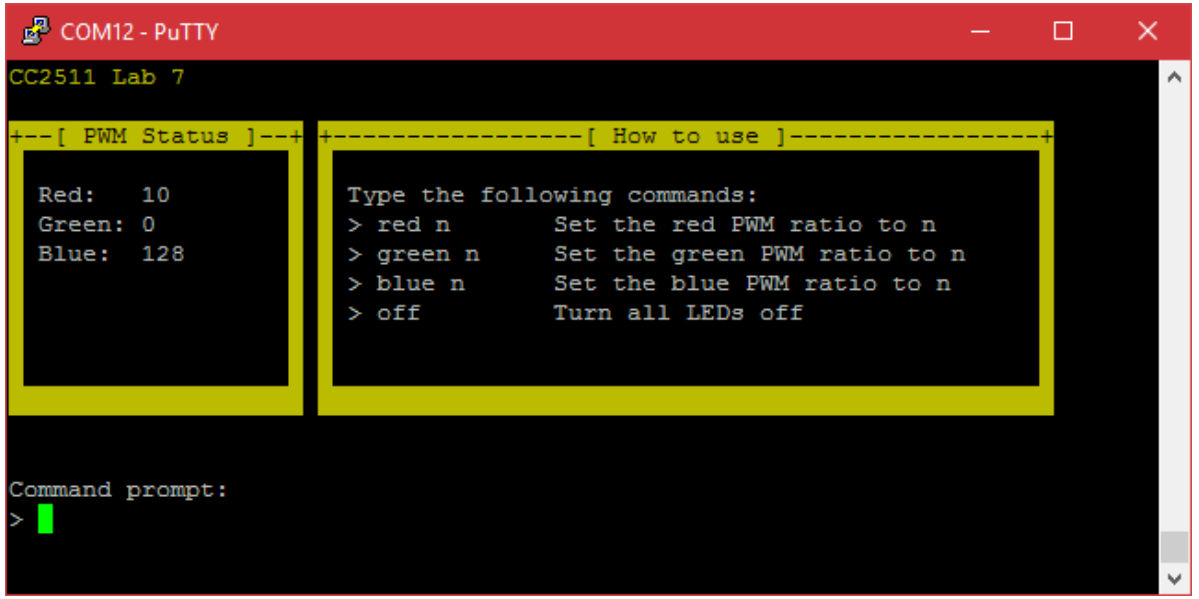
String buffers: troublesome points

Common mistakes:

- **Always make sure strings are null terminated** before calling C library functions like strcmp and sscanf.
- **Be aware of the size of the buffer.** The C language provides no protection against writing off the end of the array.
- **Choose the correct format specifier in sscanf** otherwise sscanf might write too many bytes and clobber other variables.

This week’s lab

- Control the PWM ratio sent to each of the 3 LEDs.
- Receive characters under interrupt and parse the string.
- This task is more demanding that previous labs. Start early!



Summary

- Interrupts add “event handling” capability.
- Interrupt service routines (ISRs) should be short. They should never busy-wait on peripherals.
- To implement a command-line user interface, receive characters into a buffer and then process this buffer.