Deadlocks in process management

Allocation of resources had always been a hazardous issue for programmers in multi-tasking and especially multi-threaded environments. In these environments, processes are being executed concurrently (with the help of context switches in the first case, and multiple processors in the second). In non-multitasking architectures, as in the case of many embedded systems software, only one main process is running, and execution is only diverted when interrupts are requested. Some principles of multitasking apply to non-multitasking architectures as well if interrupts are seen as processes that cannot be interrupted except by higher priority interrupts.

Issues of multi-tasking

When two or more processes request a unique resource various problems can arise. Starvation and race conditions are two of them. Starvation means that a process is requesting a resource that is being used by higher priority processes for an infinite amount of time. Race conditions occur when one resource is being used by different processes at the same time. This can lead to faults and unexpected results if the resource is not able to handle simultaneous requests.

Resource allocation is not solely a software issue. In hardware, when multiple circuits try to use a transmission line simultaneously, and they are all able to actively drive this line, then a short circuit could occur [1]. In software, under certain conditions, competing for one or more resources can block the function of every competing process. If this situation affects many processes then it could result in failure of the whole system.

The Coffman is the basic type of deadlocks appearing in process management which is different from the definition of a network deadlock. In the latter, nodes (instead of processes) are expecting messages that are not delivered due to a fault in the communication process. This report concentrates on deadlocks in process management and in non multitasking environments. In an embedded system, resources might be transmission lines, external peripherals, devices etc. while if there exists an OS they might also be files, databases etc.

Definition of a deadlock

In a deadlock situation, as described by Coffman, processes are requesting resources that are held by other blocked processes. There are certain conditions that must be true for this type of deadlocks to occur. Mutual exclusion (mutex) is one of them and dictates that a process using one resource, blocks all other processes from using the same resource. If the waiting processes withhold other free
resources the problem could turn into a circular wait which is the essence of a deadlock. The final condition that must be true is the absence of pre-emption which means that the resources allocated to some processes cannot be taken from them by others. A simplified scenario with two processes and two resources (A, B) is shown in Table 1.

<table>
<thead>
<tr>
<th></th>
<th>Process p1</th>
<th>Process p2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Request B</td>
<td>Request A</td>
</tr>
<tr>
<td></td>
<td>Mutex B</td>
<td>Mutex A</td>
</tr>
<tr>
<td></td>
<td>Request A</td>
<td>Request B</td>
</tr>
<tr>
<td></td>
<td>Sleep</td>
<td>Sleep</td>
</tr>
<tr>
<td></td>
<td>Request A</td>
<td>Request B</td>
</tr>
</tbody>
</table>

Table 1. Execution sequence excerpts that result in a deadlock

Process p1 finds resource B free and raises its mutex flag. After a context switch, p2 does the same for A. Now p1 needs A, but p2 cannot give it up unless it first uses B to complete its task. P1 will also not give up B unless it uses A. Problems like this can be modelled using “wait-for” graphs, where processes are represented by a circle and resources by a square. The acts of requesting and allocating are indicated by arrows. The equivalent graph of Table 1 is depicted in Figure 1. Graphs aid locating and thus preventing deadlocks. A deadlock can be detected in the graph when lines are forming a closed loop – they begin and end from the same point. Again, resources must be non pre-emptible in order for this loop to be indeed a deadlock.

![Graphical representation of Table 1 sequence](image)

Figure 1. Graphical representation of Table 1 sequence

Non multitasking environments are more straightforward in their development, but poor programming techniques can lead to undesirable situations. For example, consider a system with a serial line used for communication between two nodes. In this system, a global array is used to send one string over the serial interface, one character at a time. A receive interrupt (RxIR) is also enabled. Its service routine is programmed to wait until this array is free and then send a response over the serial interface after a specific message has been received. In the main routine a string is written to the buffer array initializing transmission. At the same time RxIR happens. Rx service
routine will wait for the buffer array to be empty, something that will never happen unless the RxSR returns. The whole system is deadlocked indefinitely. This, however, is a situation that can easily be detected and avoided while testing the system, whereas most deadlock situations in multi-tasking environments are difficult to detect and even more difficult to avoid.

**Countermeasures**

**Avoidance**

Deadlocks can be prevented, or even dealt with after occurring, depending on the application. Sometimes, the complexity and overhead of a countermeasure might not worth employing it. This usually applies to non-critical systems, where it is generally accepted that failure might be encountered at some point. When developing such systems special effort is applied to avoid deadlocks as much as possible within reasonable limits.

In the case that there are multiple, but not infinite, instances of a resource, the Banker’s algorithm can be used. Whenever a process is asking for a resource, this algorithm checks whether there are enough resources for the rest of the processes before doing the allocation. If there are, the system remains in ‘safe state’. If not, the allocation is averted.

**Prevention**

In embedded systems for critical applications (i.e. safety control systems for a vehicles, avionics) failure is unacceptable. In these cases deadlocks are usually prevented by designing the system in such way that they could not happen. The most straight-forward way to do this is to remove one of the necessary conditions of a deadlock.

If mutual exclusion is removed there will be no deadlocks. Reducing the amount of non-sharable resources reduces the possibility of a deadlock happening. The same result can be achieved when only one instruction is needed to complete the task after the acquisition of a resource. In the serial line communication example mentioned above, as soon as the character is written on the buffer (which takes one CPU cycle) control passes on to the UART which ensures that the character is send over the line and that eventually the buffer will be free. This is in contrast to the buffer array whose contents will be written sequentially on the Tx buffer by the CPU. This array needs mutual exclusion to work properly. Caution must be taken when using the latter.

If all resources needed for a specific task are allocated simultaneously, again, no deadlock can happen. This can be implemented by using one instruction for the acquisition of all the required
resources at once. Another alternative is to avoid withholding any resources when waiting for others to be released. This would be a more starvation prone system. The third alternative involves the releasing of resources when a process is requesting a resource that cannot be allocated to it. In other words: conditional pre-emptiveness. Implementation of this technique can be done by copying essential data or state information in buffers when handing in a resource and recover them when the resource is given back to the original process.

The final technique that can be used is ordering of the resources. For example, resources A and B have an order of 1 and 2 respectively. When a process needs both of these resources, it must always request first A and then B. This ensures that no circular waits will appear during the execution.

Recovery

If complexity and overhead of employing a deadlock prevention technique are too high related to how critical the function of the system is, a recovery method can be used to exit the processes gracefully (or not) after a deadlock has been detected. Decision of the order in which the processes are aborted could be based on various criteria like how many resources it acquires, how many instructions remain for it to complete, its priority etc.

Aborting the deadlock process, or abort one process at a time until the deadlock is eliminated, could prevent the whole system from crashing, but would not be considered a much graceful recovery. If sensitive data are at stake, execution can be rolled back to some safe state. It can then be resumed from the beginning of the previously deadlocked or another specific process in a more controllable manner.

Recovery algorithms are executed provided that a deadlock has been detected. Sometimes it might be hard to distinguish between a deadlock and a process that is taking too long to respond. Detection programs are usually executed on the background and make trade-offs between adding too much overhead and effectively recognising that a deadlock has happened. A detection algorithm can be an implementation of the “wait-for” graph. Circular waits are an indication of something being wrong in the normal execution flow.

Conclusion

Deadlocks in execution flow are not a trivial issue. They cannot be detected by a simple examination of the code, especially in multi-tasking environments where virtually any process can be loaded and executed. Sophisticated code and detection algorithms can be employed to deal with the situation. However, complexity and overhead of these countermeasures should always be analogous to how critical the function of the system is.
LABORATORY EXERCISES

Exercise 1.1. Parallel IO

A microcontroller is an integrated circuit that contains a processor, memory and I/O devices that exchange data through the use of buses. The Infineon C167CS has a core module which contains the processor and the Interrupt controller and various peripheral devices (ADC, PWM etc.) that are implemented as separate modules on the same chip, or on expansion boards.

Signals can be sent to the various peripherals with the use of I/O ports. In this exercise, for demonstration purposes, signals are sent to the external LED module which is located on the expansion board. Communication is achieved through port 2 of the microcontroller. This is our ‘gate’ for the LED module on the expansion board. Each pin of port 2 corresponds to one LED.

In programming terms, a port is a variable whose type depends on the number of pins on the port. Port 2 is 16-bit, bit-addressable, which means that it can be represented by an integer (16 bits) and that bitwise operations can be performed on it in order to change the status of individual pins. Pins are connected active low, therefore a value of 0 on a pin will activate this pin as well as the corresponding LED, while a value of 1 will deactivate it.

To change the state of the port, the routines provided by DAvE are used. The following code will infinitely change an alternating pattern of on-off states of the LEDs from 0101010101010101 to 1010101010101010 where, as previously stated, zeros represent an on, and ones represent an off state of a LED. Function IO_vWritePort() is used to access all pins simultaneously.

```c
unsigned int count;
while(1)
{
    IO_vWritePort(P2, 0x5555);
    for (count=0; count<55000; count++) {}
    IO_vWritePort(P2, 0xAAAA);
    for (count=0; count<70000; count++) {}
}
```

Table 2. Accessing LEDs through parallel line

Because the CPU is able to perform more than $10^6$ instructions per second, if we constantly change states, the LEDs will not be able to adapt fast enough to the changes, therefore, a delay is added. This is done by using a `for` loop that keeps the CPU ‘busy’ between successive changes. As we see from the equivalent assembly code of a `for` loop (Table 3), the delay time can be roughly
determined by the time it takes for the CPU to execute instructions **CMPI1** and **JMPR** multiplied by the maximum value of the variable **count**. The time increases even more if we change the maximum value of **count** to a number that is greater than **0xFFFF**. Since we are working on a 16-bit microcontroller, more instruction cycles will be needed to compare with **count**.

```assembly
0000 E005  MOV  R5,#00H
0002  ?C0001:
0002 86F5BFDA  CMPI1  R5,#0DABFH
0006 8DFD  JMPR  cc_ULT,?C0001
0008  ?C0002:
0008 CB00  RET
```

Table 3. Assembly code of a for loop

In the next example an intermediate variable, **val**, is going to be used to write an incrementing value on the LEDs. Specifically, value **0x1** is going to be left-shifted so that each LED turns on and then off sequentially. After 16 shifts, **0x1** has to be re-assigned to **val**.

```c
void main(void)
{  
    // USER CODE BEGIN (Main,2)
    unsigned long count;
    unsigned int val=0;
    // USER CODE END

    MAIN_vInit();

    // USER CODE BEGIN (Main,4)
    while(1)
    {
        IO_vWritePort(P2, ~val); //write the value on the LEDs
        for (count=0; count<65000; count++) {} //delay
        val = val << 1; //left shifting val lights up the next LED
        if(!val)
            val=0x1;
    }
    // USER CODE END
}  //  End of function main
```

Table 4. Writing an incrementing value on the LEDs. Code from **main()** routine.

If, in each iteration, after left-shifting **val**, we also increment it by one then each LED that lights up will stay on. Again, a final condition has to check whether all the lights are turned on, so that the process starts from the start.

Instead of using variable **val**, function **IO_vReadPort()** can also be used to read the current status of the port and change it accordingly.

The expansion board of the microcontroller provides output pins for checking various signals. The pins belong to port 6 on the expansion board, which is connected to the microcontrollers through port 7. Writing to port 7 is done in exactly the same way as previously done for port 2. Function **IO_vSetPin()** can make the process easier since it provides the possibility of accessing an individual pin. Using an oscilloscope the voltage changes can be observed provided that non-open drain mode (push-pull) is selected for this pin in **DAvE**.
In push-pull operation both the upper and the lower transistors of the output driver of the pin are enabled. This way the driver can switch each transistor on or off and thus drive the line either to a higher or a lower level (logic ‘1’ or ‘0’). We can then observe this change with the oscilloscope. In Table 5, the voltage changes from 0V to 2V whenever \texttt{IO\_vSetPin()} is called. This process is in contrast to open drain mode where the upper transistor is disabled, so the driver can only drive the line to a lower state. When logic ‘1’ is needed, the lower transistor switches off and the line enters in high impedance state (high-z), a ‘floating’ state. The oscilloscope will record no changes in voltage if \texttt{IO\_vSetPin()} is called. If we want to change the voltage to a desirable level at this state, we must connect an external pullup device to the pin, like a pullup resistor.

<table>
<thead>
<tr>
<th>V</th>
<th>void main(void) {</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>unsigned long count;</td>
</tr>
<tr>
<td>3</td>
<td>MAIN_vInit();</td>
</tr>
<tr>
<td>1</td>
<td>while(1){</td>
</tr>
<tr>
<td>-1</td>
<td>IO_vSetPin(P7_P7_0); //active high</td>
</tr>
<tr>
<td>-3</td>
<td>for(count=0; count&lt;65000; count++);</td>
</tr>
<tr>
<td>-5</td>
<td>IO_vResetPin(P7_P7_0); //active low</td>
</tr>
<tr>
<td></td>
<td>for(count=0; count&lt;65000; count++);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

Table 5. Oscilloscope output of first pin of port 7 in push/pull operation (left) equivalent code from \texttt{main()} routine (right)

**Exercise 1.2 Timer**

In the previous exercise, a \texttt{for} loop was used to force a delay between pin state changes. More control over this interval can be obtained by using timers. Timers are structures that can be used to time an event, or trigger another after some specific time has passed. They do this by incrementing or decrementing a register with a predefined frequency.

Two groups of timers \texttt{GPT1} and \texttt{GPT2} are provided in \textit{C167CS}. Timer 2 of group \texttt{GPT1} is going to be used. When set to \textit{Timer mode}, \texttt{T2} uses the CPU clock for synchronization. A prescaler is also used to decrease the frequency in which the timer is counting. The prescaler does this by dividing the CPU frequency by an integer that is determined by the \texttt{T2I} bit field and the following formula,

\[ f_{T2} = \frac{f_{CPU}}{8.2^{<T2I>}} \]

\[ r_{T2}[\mu s] = \frac{8.2^{<T2I>}}{f_{CPU}[MHz]} \]
Where $f_{T2}$ is the frequency of T2 timer and $r_{T2}$ is the period between two successive changes of the timer data register. Table 6 lists timer related values for the CPU frequency of 20MHz.

### Table 6. Frequencies and time intervals for various values of T2l field when the CPU is operating in 20 MHz

<table>
<thead>
<tr>
<th>Prescaler factor</th>
<th>Timer Input Selection T2l/T3l/T4l</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000B</td>
</tr>
<tr>
<td>Input Frequency</td>
<td>2.5 MHz</td>
</tr>
<tr>
<td>Resolution</td>
<td>400 ns</td>
</tr>
<tr>
<td>Period</td>
<td>26.2 ms</td>
</tr>
</tbody>
</table>

By choosing a T2l value of 6 (110 in binary), $f_{CPU}$ will be divided by the factor 512. If “count down” is selected the timer will decrement its value by one every 25.6 μs, starting from the value in the timer register (in this case 0x9896). In the next decrement after zero is reached, the timer will underflow and T2 interrupt will happen. In our case, T2 will happen after 1 sec (0x9896 * 25.6 μs).

Execution now passes to GPT1_vITmr2() function, which, as asked from exercise 1.2, inverses the value of the LEDs.

Function GPT1_vLoadTmr() loads the timer register with the desirable value. This function has higher priority over any increment or decrement of the timer register. Each time it is called, a ‘reset’ of the timer to the given value is performed and counting starts again from that point. After an overflow or an underflow happens, the timer is disabled, so we have to reload it using the same function.

By adding the variable count in GPT1.c and incrementing it every time T2 interrupt is called, we can count the number of seconds that have passed. This works because T2 interrupt is called once every second. If we change the value of T2l field, we will have to adjust the timer register so that count effectively holds the value of the elapsed seconds.
void main(void)
{
    // USER CODE BEGIN (Main,2)
    // USER CODE END
    MAIN_vInit();
    // USER CODE BEGIN (Main,4)
    IO_vWritePort(P2, 0xFFFF);
    while(1);
    // USER CODE END
} // End of function main

void GPT1_viTmr2(void) interrupt T2INT
{
    // USER CODE BEGIN (Tmr2,2)
    count++ ; //one more second passed
    GPT1_vLoadTmr(GPT1_TIMER_2, 0x9896); //reload the timer for
    one second count down
    IO_vWritePort(P2, ~count); //write the seconds passed on the LEDs
    // USER CODE END
} // End of function GPT1_viTmr2

Table 7. Counting the seconds elapsed: (left) main.c, (right) GPT1.c

The while statement at the end of the main routine is very essential for this program because not only keeps the operation of the microcontroller to a continuous state, but also provides a ‘safe’ place for the execution to return after the service routine of T2 returns.

Exercise 2. Producing a PWM waveform

A PWM waveform will be created using the T2 timer to control the ‘high’ and ‘low’ intervals of the first pin of port 7. For a PWM frequency of 50Hz (period T= 0.02s = 20ms) and 5% duty cycle, the pin must stay high for 1 ms and low for 19 ms. To achieve this we load the timer register with the value 0x0026 which corresponds to 1 ms with a T2I value of 6. When the T2 interrupt is called, the pin is toggled to the low state and timer is reloaded with value 0x02E5, which corresponds to an underflow interval of 19 ms. This process is repeated with the help of the boolean variable low that is true whenever the state of the pin at the time the interrupt is called is low.

When the program starts the timer will be already initialized to the low interval value. This is done in DAvE by checking the ‘Enabling T2 after initialization’ checkbox and putting the value, 0x02E5 in ‘Timer T2’ field. We now have to set the pin to ‘1’ (low) and initialize variable low to 1 (Table 8).
If we want to change the duty cycle, we simply load the timer register with different values. For example, for a 10% duty cycle, the register will be loaded with 0x004D high time value, and 0x02BE low time value. The code is identical to that presented in Table 8.

To make a sweep from 5% to 10% duty cycle in 2 seconds we must first make some calculations in order to derive the values of the timer register for each full cycle. We first calculate the values for the first half of the sweep. This will occur at the first second. Since the full cycle period of 20ms...

![Table 8. Producing a PWM waveform](image)

![Figure 2. Output of pin 7.1 showing PWM waveform of 5% duty cycle](image)
remains the same throughout the sweep, in the interval of 1 s, there will be 50 cycles. During these 50 cycles the high time should go from 1 ms to 2 ms, which gives us a difference of 1 ms. So, the change in high time between two successive cycles should be \(1\text{ms} \div 50 = 0.02 \text{ ms} = 20 \mu\text{s}\). That is, if the first high time interval is 1 ms, the second will be 1.02 ms, the third 1.04 ms etc. The low time is just the remaining of the full cycle and will be 19 ms the first time, 18.98 ms the second etc.

Implementing this on the microcontroller gives us a 0x0026 high time timer register value for the first cycle, 0x0027 for the second etc. Equivalently, for low time the register will be 0x02e5, 0x02e4, 0x02e3 etc. We start with T2 value 0x0026 initialized in DAvE, pin 1 set to 0 (high) with function IO_vResetPin(), and variable low = 0.

```
//file main.c
void main(void)
{
    // USER CODE BEGIN (Main,2)
    MAIN_vInit();
    // USER CODE END

    // USER CODE BEGIN (Main,4)
    IO_vResetPin(P7_P7_0);
    while(1);
    // USER CODE END
}

//file GPT1.c

// USER CODE BEGIN (GPT1_General,6)
unsigned int t_h = 0x0027;
unsigned int t_l = 0x02e5;
unsigned char low = 0;
int sgn = 1;
// USER CODE END

Table 9. PWM sweep from 5% to 10% duty cycle.

The timer variables \(t_h, t_l\) in GPT1.c file are used as follows: \(t_h\) is the interval in which the pin will stay in high state and \(t_l\) in low state. Variable \(low\), again, defines the state of the pin. In each interrupt, low variable is checked. If the state of the pin was high the timer is loaded with the low state interval value \(t_l\) which is then decremented by 1. If the state was low, timer is loaded with value \(t_h\) which is afterwards incremented by one. When \(t_l\) reaches value 0x02be one second will have passed. Thus the process must be reversed. This is done with the introduction of the multiplier
```
sgn. This variable reverses the incrementing to decrementing and vice versa. From this point on, in each interrupt, \( t_l \) increments by one or \( t_h \) decrements by one. The process will again be reversed when \( t_l \) reaches \( 0x02e5 \).

Exercise 3. Serial Communication

In Asynchronous serial communication, two devices can exchange data sent on non-predefined timing intervals. To achieve this, a start bit is sent before each word transmission to prepare the line while a stop bit indicates the end of the transmission of this single word. A word of 8 bits of data, no parity bit (used for error detection), and one stop bit have been chosen.

In \( C167CS \), communication is taking place with the help of the \textit{transmit} and \textit{receive} buffers \texttt{S0TBUF} and \texttt{S0RBUF} respectively. For transmitting the first byte on the line \texttt{S0TBUF} is checked using \texttt{ASC0\textunderscore ubTxBufFree()} function. This function checks if \texttt{S0TBIR} bit of the \texttt{S0TBIC} control register is set. If it is set then \texttt{S0TBUF} is empty, so a transmission can begin. \texttt{S0TBIC\textunderscore S0TBIR} is set to 1 upon initialization of the ASC interface. \texttt{S0TBIR} is the flag for the buffer interrupt request, but since this interrupt is disabled, it is used as an indication of when the buffer is empty.

First a routine is created for handling the transmission of a string through the serial interface. This function is placed in \texttt{main.c} and contains the code shown in Table 10.

\begin{verbatim}
//****************************************************************************
// USER CODE BEGIN (MAIN_General,9)
void ASC_PrintString(const char* String) {
    unsigned char *slide; //next character to be sent
    slide = String;
    while(*slide != '\0') { //stop at the end of the string
        while(!ASC0\textunderscore ubTxBufFree()); //wait for S0TBUF to be free
        ASC0\textunderscore vSendData(*slide++); //send the character
    }
}
//****************************************************************************

table 10. Function for sending one string of character through the serial interface (main.c)

Variable \texttt{slide} points to the next character to be transmitted through ASC and is increased after the completion of each transmission.

An important observation in this code segment is the ‘dangerous’ execution flow between checking whether the \texttt{Tx} buffer is free and sending the next character. As it has already been mentioned in the first part of this report, a race condition could take place if different parts of the program where accessing the buffer simultaneously. This situation could happen if an interrupt service routine that used the \texttt{Tx} buffer was called just after function \texttt{ASC0\textunderscore ubTxBufFree()} returned
1 (buffer empty) and before **ASC0_vSendData()** was called. If this interrupt sent one character through ASC and then immediately returned, it would depend on the time it takes for the CPU to restore the registers before continuing execution in the main routine in relation to the baud rate set for ASC whether there would be any errors in the transmission.

For the program that we are working now there is no possibility of error (just one process and no interrupts enabled) thus no measures were taken because they would only add unnecessary complexity. The code for sending the string ‘Test’ through the serial interface can be found in Appendix A.

The timer will be used next for sending the seconds elapsed to the PC terminal. In order to send the number of the time counter (Table 7) through the serial interface, we have to convert it to ASCII or the receiver will not be able to translate it properly. Function **itos()** will handle this conversion. The number to be converted, as well as a pointer to an existing string in memory (variable slide), are passed as arguments. The range of values of an unsigned integer is 0 - 65535, thus the array that slide is pointing to should have a minimum value of 6 (five digits and a null character). The function scans the number from last towards the first decimal digit and converts each character to ASCII.

```
//file main.c (continued from table 10)
void itos(const char *String, unsigned int Number){
    unsigned char *slide;

    slide = String + 5; //move to the last element of the ASCII array
    *slide = '0';        //fill it with null character
    slide--;            //move to the previous

    while ( Number >= 10 ){       //skip if only one decimal digit
        *slide = (unsigned char)(0x30 + (Number % 10)); //convert last decimal digit to ASCII
        Number /= 10;          //get rid of last digit
        slide--;                //move to the previous
    }
    *slide = (unsigned char)(0x30 + Number); //this is the first digit of Number
                                            //and the last one to be written to the array
    }
// USER CODE END (MAIN_General,9)
```

Table 11. Converting an integer to ASCII (file main.c)

Finally, we modify **GPT1_vITmr2()** function as shown in Table 12. Two variables are initialized: **reset_ivl** holds the seconds passed since the last reset of the microcontroller in decimal form and **ivl_str** contains the same value in ASCII form. Each time a timer interrupt occurs **reset_ivl** is increased by one, its value is converted to ASCII and then sent through the serial interface. If **reset_ivl** reaches the last value possible for a 16-bit integer, it will wrap around and start counting from the start. The array of characters still contains the last value which now has more characters than the new one, so we have to reset it before continuing.
//file GPT1.c
//******************************************************************************
// @Global Variables
//******************************************************************************

// USER CODE BEGIN (GPT1_General,7)
unsigned int reset_ivl = 0;
unsigned char ivl_str[6] = {0x20,0x20,0x20,0x20,0x20,0x20};
// USER CODE END

void GPT1_viTmr2(void) interrupt T2INT { // USER CODE BEGIN (Tmr2,2)
    int i;
    GPT1_vLoadTmr(GPT1_TIMER_2, 0x9896);
    reset_ivl++;
    itos(ivl_str,reset_ivl); //calling integer to strin conversion
    ASC_PrintString(ivl_str); //printing in terminal
    ASC_PrintString(","); //printing a delimiter
    if(reset_ivl == 65535) { //reset the array
        for(i = 0; i<5; i++)
            ivl_str[i] = 'x20';
    }
    // USER CODE END}
} //  End of function GPT1_viTmr2

Table 12. Timer interrupt service routine sends the seconds elapsed through the serial interface (file GPT1.c)

Figure 3. Hyper Terminal output of seconds elapsed since last reset.
For the next exercise a character is going to be received through the serial interface. Receive interrupt is enabled (Figure 4). Its service routine sends an immediate response back to the line depending on the character that was received (Appendix B). Priority was set to level 10 (group 2). Priority of this interrupt should always be relatively high. If the other end of the line is sending characters sequentially, some characters might be lost if they are not read on time from the buffer (i.e. if another interrupt with higher priority has occurred). This would raise an overrun error status flag.

![Asynchronous/Synchronous Serial Interface (ASC)](image)

**Figure 4.** DAvE configuration for enabling the receive interrupt

The receive interrupt uses the **Tx** buffer. This might be dangerous if something was on the process of being sent through ASC interface. However, the main program is not sending anything except from the string “Give me a letter” at the start of the execution. If at the exact same time the receive interrupt routine was called, some characters might appear wrong on the terminal and the response of the microcontroller would appear somewhere between the initial string.
Exercise 4. Using the Controller Area Network

The CAN standard includes a CSMA/CD protocol that permits many nodes to communicate without the help of an intermediate device that controls the communication and without data loss in the event of a collision. Message identifier bits are used in an arbitration process that defines the node that finally acquires the line. Whenever a node detects a frame with a message id of higher priority than its own, it retreats leaving the line to the winning node without any loss of information.

In the next program the old standard 11-bit identifiers are going to be used for the communication between two nodes which will be two C167CS microcontrollers. One of the nodes will initialize the transmission executing the code shown in Table 13. This node will have a Tx message object (MO) id 0x1 and Rx MO id 0x2. The receiving node will have Tx MO id 0x2 and Rx MO 0x1. Both microcontrollers will have Tx message objects of 2 data bytes length.

*Note: loading a value at Tx MOs through DAvE does not work. Have to do it by hand (Table 13).
Table 13. This code is only loaded in the microcontroller that initializes the communication

Timer **T2** is enabled with an interrupt priority of level 10 (group 2). It is not started at initialization of the program but will be started after a response has been received from the CAN interface. CAN1 interrupt is enabled with priority of level 11 (group 2).

All accesses to the CAN interface are handled with one interrupt and its equivalent service routine **CAN1_viCAN1()**. The specific event that triggers the interrupt is determined by the **INTID** field of the Port Control/Interrupt Register (PCIR). A value of 4 (2 + N where N is the number of the message object) indicates that the interrupt was caused by an event on MO 2. If **NEWDAT** (data received) of MO 2 is set and no overwrite errors occurred we can read the contents of object 2 using the following code:

```c
//file CAN1.c
//**************************************
// @Imported Global Variables
//**************************************

// USER CODE BEGIN (CAN1_General,6)
unsigned int Rmsg; //holds received integer
TCAN1_Obj recv_obj; //software MO for received message

// USER CODE END

void CAN1_vlInit(void) interrupt XP0INT
{
    uword uwIntID;
    while (uwIntID = C1PCIR & 0x00ff) {
        switch (uwIntID & 0x00ff) {
            case 4: // Message Object 2 Interrupt
                CAN1_OBJ[1].MCR = 0xfffd; // reset INTPND
                if ((CAN1_OBJ[1].MCR & 0x0300) == 0x0200) { // if NEWDAT set
                    if ((CAN1_OBJ[1].MCR & 0x0c00) == 0x0800) { // if MSGLST set
                        // ...omitted
                    }
                }
        }
    }
}
```
CAN1OBJ[1].MCR = 0xf7ff; // reset MSGLST

} else {
    // The CAN1 controller has stored a new message into this object.
    // USER CODE BEGIN (CAN1,21)
    CAN1_vReleaseObj(2);
    CAN1_vGetMsgObj(2, &recv_obj);
    Rmsg = ((unsigned int)recv_obj.ubData[0] << 8) + recv_obj.ubData[1]; // retrieve the integer sent by peer
    IO_vWritePort(P2, Rmsg); // print it in the LEDs
    Rmsg++;
    GPT1_vLoadTmr(GPT1_TIMER_2, 0x9896); // load the timer to count one second
    GPT1_vStartTmr(GPT1_TIMER_2); // set the run bit for the timer (start it)
    // USER CODE END
}

Table 14. Modifications on the CAN interrupt, receive section

Variable Rmsg stores the number that is received from the peer microcontroller. The CPU will only store messages with the id specified in DAvE for this object. To retrieve the data from within it we use CAN1_vGetMsgObj(). This function takes a pointer of type TCAN1_Obj as an argument, which is a struct built by DAvE. The address of recv_obj is passed to the function. The ubData portion of this struct is the number that was received. We also have to use CAN1_vReleaseObj() for the receive object to reset its NEWDAT flag and make it ready for the CPU to use. Next we have to read the number that was received. We can’t do this just by using the address of the first data byte because there might be alignment bits in-between the two received bytes inside the struct. Thus, we take the first byte (byte 0), cast it to unsigned int, shift it 8 times and then add the second byte. This should give us the number to write on the LEDs. We increase it by one, and then load the timer register. Note that since the timer was not started at initialization, we have to do it using function GPT1_vStartTmr(). The timer interrupt is modified next (Table 15). Tables Table 14 and Table 15 both show code segments that will be loaded to both the microcontrollers.

```
//file GPT1.c
void GPT1_viTmr2(void) interrupt T2INT
{  // USER CODE BEGIN (Tmr2,2)
    extern unsigned int Rmsg;
    extern TCAN1_Obj recv_obj;
    GPT1_vStopTmr_GPT1_TIMER_2(); // stop the timer
    recv_obj.ubData[1] = (unsigned char)Rmsg; // load the second byte
    recv_obj.ubData[0] = (unsigned char)(Rmsg >> 8); // load the first byte
    CAN1_ubRequestMsgObj(1); // wait for Tx MO to be available
    CAN1_vLoadData(1, recv_obj.ubData); // load contents of software MO to hardware MO
    CAN1_vTransmit(1); // do the transmission
    // USER CODE END
} // End of function GPT1_viTmr2
```

Table 15. Timer interrupt modified to send reply through CAN
After one second passes and the timer interrupt is called and Rmsg is loaded to the Tx object. Bytes 0 and 1 of software Tx object are filled with the first and second bytes of Rmsg equivalently. Function CAN1_ubRequestMsgObj() assures that MO 1 is empty when we try to send the new message. Finally, we have to stop the timer to ensure no further transmissions will be made until another message is received from the peer microcontroller.
Appendix A

//****************************************************************************
// @Filename MAIN.C
// @Project 3.1.dav
// Send string “Test” to Hyper Terminal
//****************************************************************************

//****************************************************************************
// @Prototypes Of Local Functions
//****************************************************************************

// USER CODE BEGIN (MAIN_General,9)

void ASC_PrintString(const char* String) {
    unsigned char *slide;
    slide = String;

    while(*slide != '\0'){
        while(ASC0_ubTxBufFree() == 0);
        ASC0_vSendData(*slide);
        slide++;
    }
}

void main(void)
{
    // USER CODE BEGIN (Main,2)

    // USER CODE END

    MAIN_vInit();

    // USER CODE BEGIN (Main,4)

    ASC_PrintString("Test");

    // USER CODE END

} // End of function main


Appendix B

//**************************************************************************
// @Filename      MAIN.C
// @Project 3.3.dav
// Responding to a character sent from Hyper Terminal with a word starting with that character
//**************************************************************************

// prototypes of local functions
//**************************************************************************

void ASC_PrintString(const char* String)  {
    unsigned char *slide = String;
    while(*slide != '0'){
        while(ASC0_ubTxBufFree() == 0);
        ASC0_vSendData(*slide++);
    }
}

void main(void)
{
    // USER CODE BEGIN (Main,2)
    // USER CODE END
    MAIN_vInit();
    // USER CODE BEGIN (Main,4)
    ASC_PrintString("Give me a letter\n");
    while(1);
    // USER CODE END
}

//****************************************************************************
// @Module        Asynchronous/Synchronous Serial Interface (ASC0)
// @Filename      ASC0.C
// @Project 3.3.dav
//**************************************************************************

void ASC0_viRx(void) interrupt S0RINT
{
    // USER CODE BEGIN (Rx,2)
    uword letter; //holds the received character
    unsigned char *alphabet[26] =
    { "Allocation",
        "Bandwidth",
        "Buffer",
        "Capacity",
        "Character",
        "Computer",
        "Device",
        "Electronic",
        "Engineering",
        "Frequency",
        "Generator",
        "Hardware",
        "Interface",
        "Infrared",
        "Junction",
        "Keyword",
        "Laser",
        "Module",
        "Network",
        "Observer",
        "Protocol",
        "Receiver",
        "Software",
        "Termination",
        "Transfer",
        "Understanding",
        "Vehicle",
        "Window",
    }
    // USER CODE END
}
letter = ASC0_uwGetData();                //get the character that was received (contents of SORBUF)
ASC_PrintString(" for ");              // print the “ for “ on terminal
if(letter>=0x41 && letter <=0x5A)       //letter is capital
    ASC_PrintString(alphabet[letter-0x41]);
else if(letter>=0x61 && letter <=0x7A)  //letter not capital
    ASC_PrintString(alphabet[letter-0x61]);
else                                  //not a letter
    ASC_PrintString("Only the alphabet pls!");
ASC_PrintString("\n");                  //line feed

// USER CODE END

} // End of function ASC0_viRx
Acknowledgements

The code on this report was developed in collaboration with my lab partner, Miguel Hervás Lázaro.

Bibliography

