University of Technology, Sydney

Faculty of Engineering and Information Technology

Subject:	**48434 Embedded Software**
Assessment Number:	**3**
Assessment Title:	**Lab 3 – Interrupts and Timers**
Tutorial Group:	

Students Name(s) and Number(s)

Student Number	Family Name	First Name

Declaration of Originality:

The work contained in this assignment, other than that specifically attributed to another source, is that of the author(s). It is recognised that, should this declaration be found to be false, disciplinary action could be taken and the assignments of all students involved will be given zero marks. In the statement below, I have indicated the extent to which I have collaborated with other students, whom I have named.

Statement of Collaboration:

Signature(s)

key

Marks

Opening comments / function descriptions	/0.5
Naming conventions / code structure	/0.5
Periodic timer	/1
Real-time clock	/2
SCI interrupts	/3
Protocol implementation	/1
TOTAL	/8

Office use only ☺

Assessment Submission Receipt

Assessment Title:	**Lab 3 – Interrupts and Timers**	**Mark**
Student's Name:		
Date Submitted:		
Tutor Signature:		Office use only ☺

Lab 3 – Interrupts and Timers

Interrupts. Periodic timers. Timer output channel compare. Real-time clock.

Introduction

Interrupts are an essential feature of a microcontroller. They enable the software to respond, in a timely fashion, to internal and external hardware events. For example, the reception and transmission of bytes via the SCI is more efficient (in terms of processor time) using interrupts, rather than using a polling method. Performance is improved because tasks can be given to hardware modules which "report back" when they are finished. The enhanced capture / timer (ECT) unit is also an essential feature of a microcontroller that needs to operate as part of a real-time system. For inputs, it enables the detection of external events and "time stamps" them. It can also accumulate (count) pulses occurring at the input pins. For outputs, it can be used to generate events at certain times.

Objectives

1. To use interrupts with the serial communication interface.
2. To set up a periodic timer.
3. To set up a real-time interrupt and implement a clock function.
4. To expand the implementation of the ModCon serial protocol.
5. To examine time relationships using a DSO / logic analyzer.

Equipment

- 1 ModCon microcontroller board – UTS
- 1 HCS12 USB programmer / debugger – PEMicro
- 2 USB cables
- 1 5V DC power supply, 1 12V DC power supply, PC connector
- Freescale CodeWarrior IDE for the 68HC12

Safety

This is a Category A laboratory experiment. Please adhere to the Category A safety guidelines (issued separately).

Cat. A lab

L3.2

Interrupts Overview

Declaring Interrupt Service Routines in C

In CodeWarrior, an interrupt service routine is declared with the non-ANSI C keyword `interrupt`, as well as a number corresponding to an entry in the vector table. For example, to declare an ISR for SCI0, you would use:

```
void interrupt 20 SCI0_ISR(void)
{
  /* code goes here */
}
```

This syntax tells the compiler that the function is an interrupt function (i.e. it should be terminated with `rti` rather than `rts`) and initializes the corresponding entry in the vector table. For the MC9S12, the reset vector is vector number 0 (at address 0xFFFE), vector number 1 is located just before the vector 0 (at address 0xFFFC), and so on.

Enabling and Disabling Interrupts

Interrupts can be enabled and disabled with the macros defined in the `hidef.h` file which you can include by placing `#include <hidef.h>` in your main file. The macros are:

```
#define EnableInterrupts   {__asm cli;}
#define DisableInterrupts  {__asm sei;}
```

Before the main loop of your program, but after setting up various modules, you will need to enable interrupts, as they are disabled by default.

Real-Time Interrupt

The Real-Time Interrupt (RTI) generator in the Clocks and Reset Generator (CRG) module can be used to initiate a periodic interrupt with a fairly coarse resolution. This is a handy function to implement a real-time operating system that switches between tasks where exact timing is not required. It can also be used to implement a real-time clock.

Further Information

For more information on interrupts, consult the Freescale *Core User Guide* document.

Serial Communication Interface using Interrupts

Consider the common case of an application that uses the serial communication interface (SCI). The SCI hardware receives characters at an asynchronous rate. In order to avoid loss of data in periods of high activity, the characters need to be stored in a FIFO buffer. The background task (main program) can process the characters at a rate which is independent of the rate at which the characters arrive. It must process the characters at an *average* rate which is faster than the *average* rate at which they can arrive, otherwise the FIFO buffer will become full and data will be lost. In other words, the buffer allows the input data to arrive in bursts, and the main program can access them when it is ready.

The following figure shows the situation for character reception.

An interrupt-driven input routine

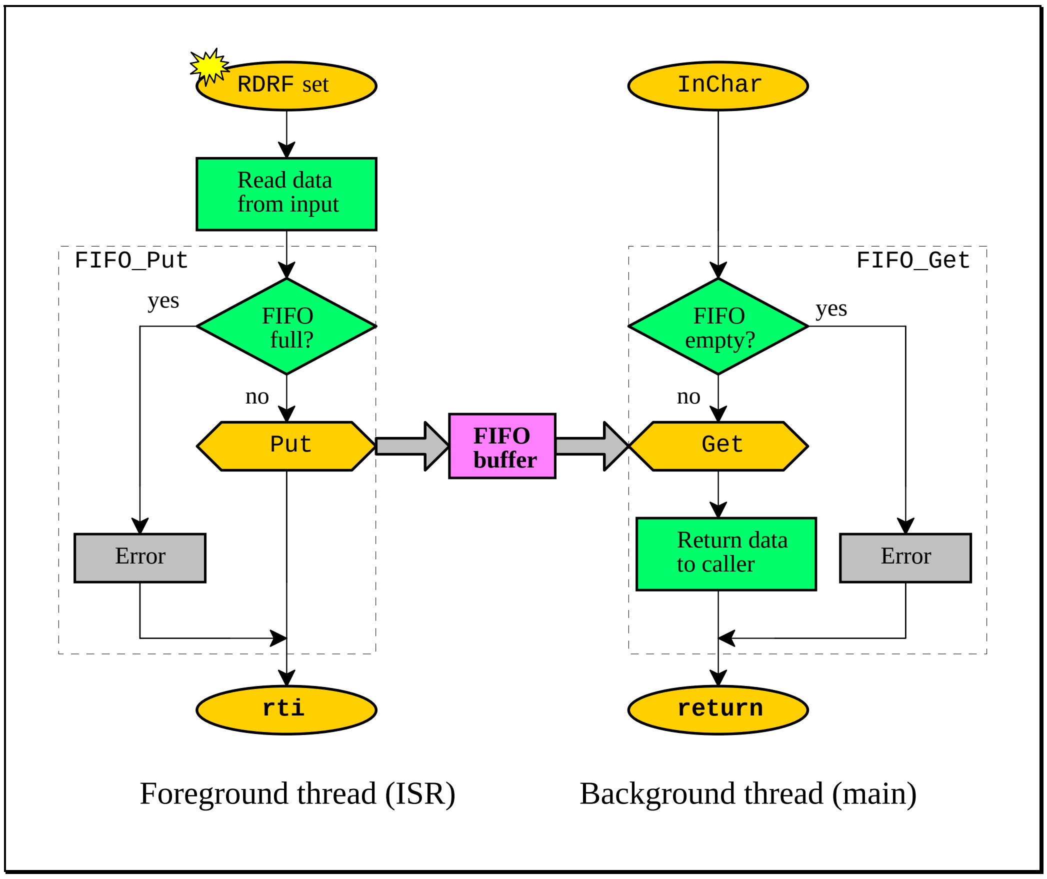


Figure L3.1

The structure for interrupt-driven character transmission is similar, but depends on whether the output device requests an interrupts when it is in the ready state, or when it *transitions* from busy to ready.

A problem with the SCI unit…that needs a "workaround"

Due to a mask set erratum, it is not possible to simultaneously use the transmit and receive interrupts on the MC9S12A512 (it only detects an odd number of interrupts, so that if both TDRE and RDRF are set, it will not generate an interrupt request). To get around this problem, we will have to implement the transmit interrupt using a timer output compare channel. See the next section for a brief description of the timer module.

Critical Sections in C

In C, macros should be defined to enter and exit critical sections of code:

C code to protect a critical section

```
#define EnterCritical() { asm pshc; asm sei; asm leas 1,sp; }
#define ExitCritical()  { asm leas -1,sp; asm pulc; }
```

You should understand the operation of these macros from a cursory knowledge of the MC9S12 assembly language.

The `EnterCritical()` macro firstly pushes the CCR register onto the stack, then sets the I bit to disable interrupts. The stack pointer has now been modified, unbeknownst to your compiled C code, which probably requires the stack pointer to be in its original position so as to address local variables. Therefore, the last assembly language statement in the macro increments SP to bring it back to its original position (before we pushed the CCR). We have now placed a "hidden" byte onto the stack.

The `ExitCritical()` macro firstly decrements the SP so that it points to the saved CCR, and then pulls it back into the CCR register. The SP is now in its original position (before we pushed the CCR) and code can continue as normal.

We should restrict the usage of these macros to small portions of code where we understand the stack limitations

The problem with this approach is that the "hidden" saved CCR register that is sitting on the stack can be overwritten if the subsequent C code utilises the stack. We are therefore prohibited from making a function call (which utilises the stack for the return address and parameter passing) within our critical section of code. Worse than that though, the compiler may place its own function calls to library routines, such as 32-bit arithmetic operations, without us being explicitly aware. In these instances, what appears to be correct code will "crash" the software when the CCR register gets loaded with a corrupted value.

Timers

The Enhanced Capture Timer (ECT) module has the capability of capturing events and time-stamping them, and of generating events at certain times. It also has a pulse accumulator that can be used to count external pulses without the need for software intervention. A modulus down-counter can be used to generate periodic interrupts.

Timer Module

A simplified block diagram of the timer module is shown below:

A simplified ECT block diagram

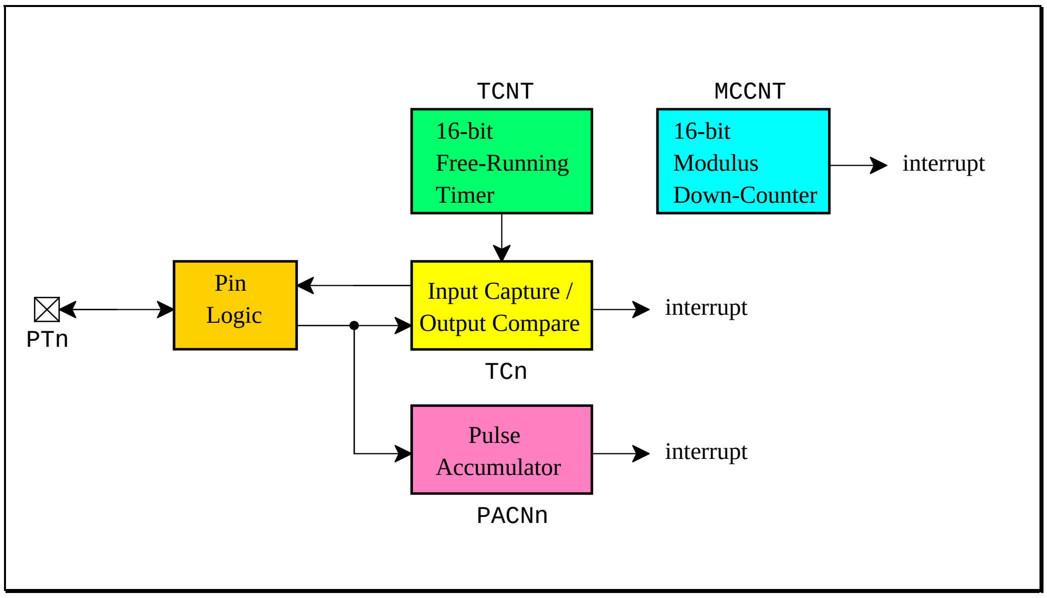


Figure L3.2

The timer has a free-running counter and a modulus down-counter

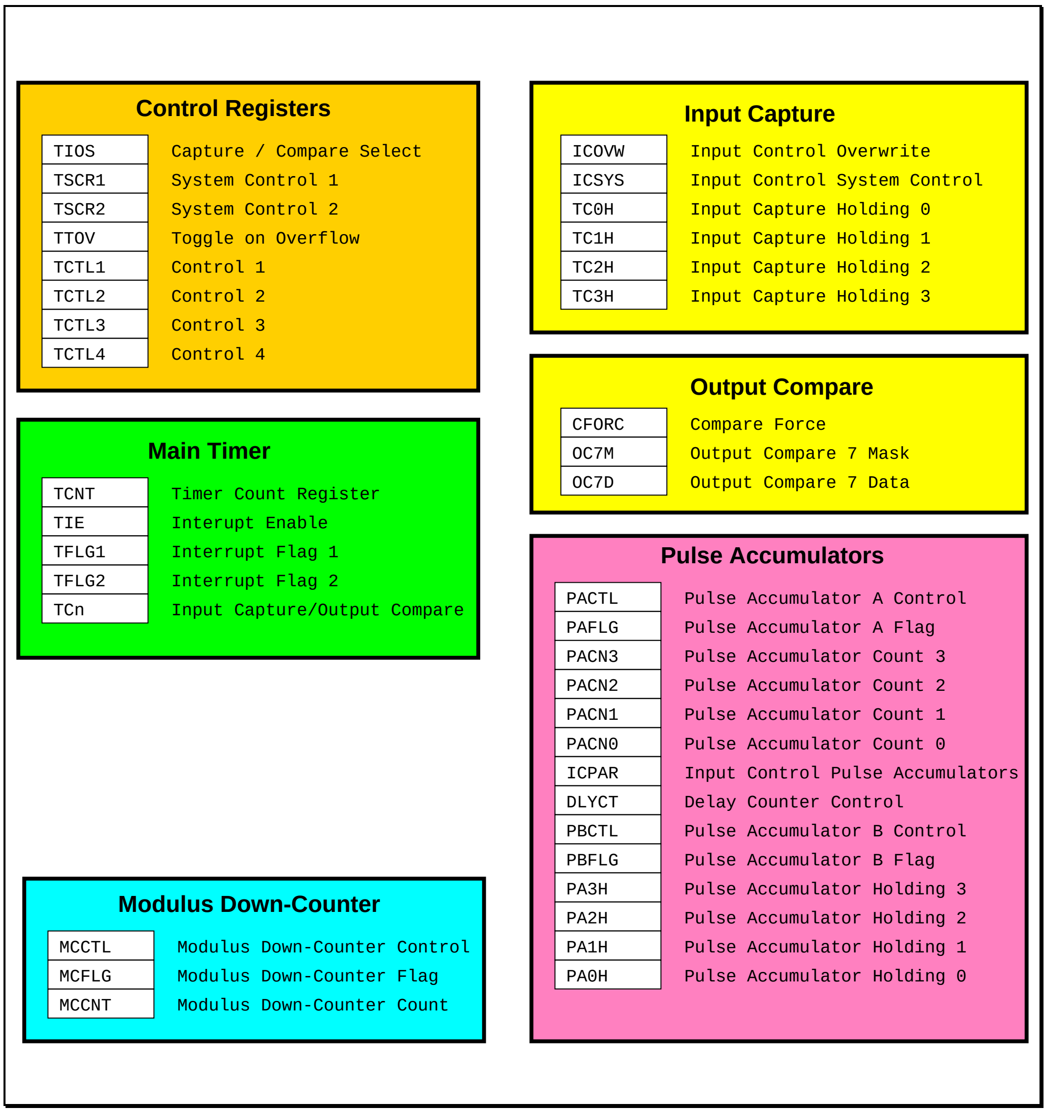
There is a 16-bit free-running timer called `TCNT`. This is used to time-stamp an input event (an *input capture*) or to trigger an output event (an *output compare*). There is also a 16-bit modulus down-counter that can be used to generate periodic interrupts with much greater precision than the RTI module of the CRG block.

The "input capture / output compare" block is just a register called `TCn`, where `n` is the channel number, that gets loaded with the current value of `TCNT` for an input capture event, and which holds a desired value of `TCNT` to trigger an output compare event.

The types of events to capture, or to initiate on a successful compare, are setup through various control registers. For inputs, it is possible to capture rising and falling edges. Outputs can be made to toggle, clear or be set.

A programmer's model of the timer module is shown below:

Programmer's model of the ECT



Control Registers

TIOS	Capture / Compare Select
TSCR1	System Control 1
TSCR2	System Control 2
TTOV	Toggle on Overflow
TCTL1	Control 1
TCTL2	Control 2
TCTL3	Control 3
TCTL4	Control 4

Input Capture

ICOVW	Input Control Overwrite
ICSYS	Input Control System Control
TC0H	Input Capture Holding 0
TC1H	Input Capture Holding 1
TC2H	Input Capture Holding 2
TC3H	Input Capture Holding 3

Main Timer

TCNT	Timer Count Register
TIE	Interupt Enable
TFLG1	Interrupt Flag 1
TFLG2	Interrupt Flag 2
TCn	Input Capture/Output Compare

Output Compare

CFORC	Compare Force
OC7M	Output Compare 7 Mask
OC7D	Output Compare 7 Data

Pulse Accumulators

PACTL	Pulse Accumulator A Control
PAFLG	Pulse Accumulator A Flag
PACN3	Pulse Accumulator Count 3
PACN2	Pulse Accumulator Count 2
PACN1	Pulse Accumulator Count 1
PACN0	Pulse Accumulator Count 0
ICPAR	Input Control Pulse Accumulators
DLYCT	Delay Counter Control
PBCTL	Pulse Accumulator B Control
PBFLG	Pulse Accumulator B Flag
PA3H	Pulse Accumulator Holding 3
PA2H	Pulse Accumulator Holding 2
PA1H	Pulse Accumulator Holding 1
PA0H	Pulse Accumulator Holding 0

Modulus Down-Counter

MCCTL	Modulus Down-Counter Control
MCFLG	Modulus Down-Counter Flag
MCCNT	Modulus Down-Counter Count

Figure L3.3

There are numerous control registers used to set up the ECT module. Only a few are needed to interact with the ECT once it has been set up for a particular application.

Modulus Down-Counter

A block diagram of the modulus down-counter is shown below:

A simplified modulus down-counter block diagram

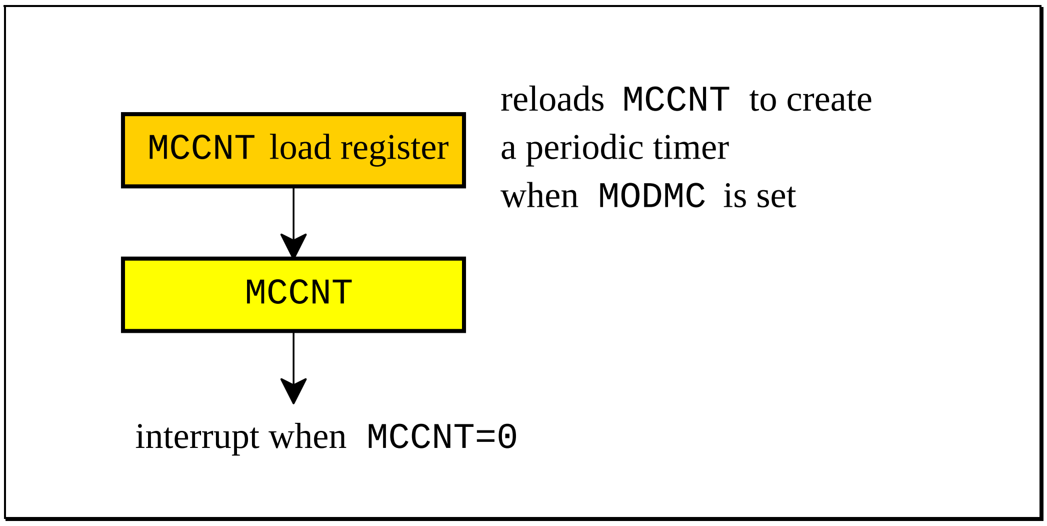


Figure L3.4

The modulus down-counter can be used as a time-base to generate a periodic interrupt. It can also be used to latch the values of the input capture registers and the pulse accumulators to their holding registers. The action of latching can be programmed to be periodic or only once.

Output Compare

A channel set up as an *output compare* channel will trigger an output action when the output compare register is equal to the free-running timer. A block diagram of the output compare action is shown below:

A simplified output compare block diagram

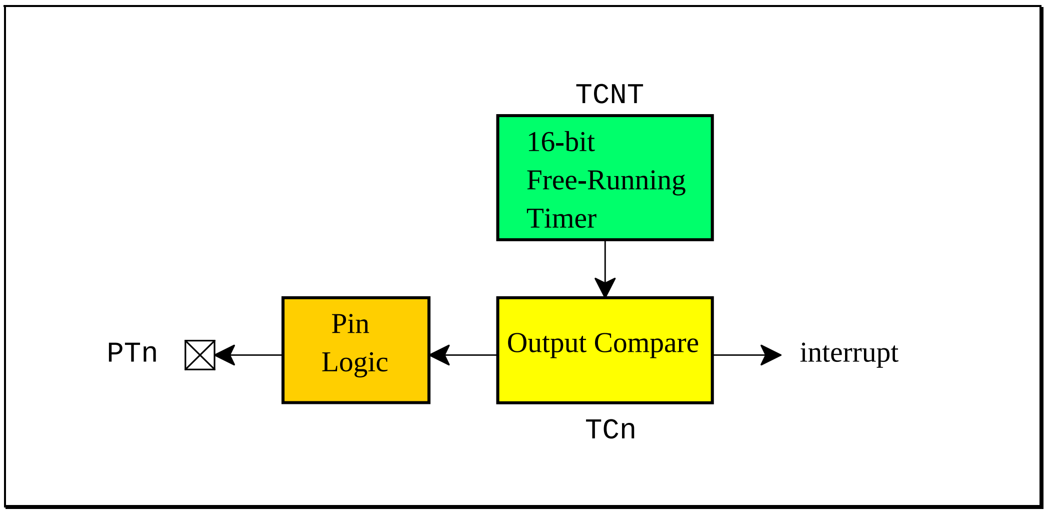


Figure L3.5

A compare result output action can be set up using the `TCTL1` and `TCTL2` registers. The options are:

Output compare actions

Action
Timer disconnected from output pin logic
Toggle `OCn` output line
Clear `OCn` output line to zero
Set `OCn` output line to one

One simple application of output compare is to create a fixed time delay. Let `delay` be the number of cycles you wish to wait. The steps to create the delay are:

1. Read the current 16-bit `TCNT`.
2. Set the 16-bit output compare register to `TCNT + delay;`
3. Clear the output compare flag.
4. Wait for the output compare flag to be set.

This method will only work for values of `delay` that fall between a minimum value (the time it takes to implement steps 1 to 3) and 65536. It will function properly even if `TCNT` rolls over from `0xffff` to `0`, since the 16-bit addition is really a modulo `0x10000` addition.

Coupled Output Compare

The output compare on channel 7 can be coupled to the other channels

The channel 7 output compare module can be configured such that an output compare event on it will cause changes on some or all of the other output compare pins. This coupled behaviour can be used to create synchronous signals. For example, we can create pulses that start together or end together.

Input Capture

A channel can be set up as an *input capture* channel. We can use input capture to measure the period or pulse width of TTL-level signals. The input capture system can also be used to trigger interrupts on rising or falling transitions of external signals. A simplified block diagram of a channel set up for input capture is shown below:

A simplified input capture block diagram

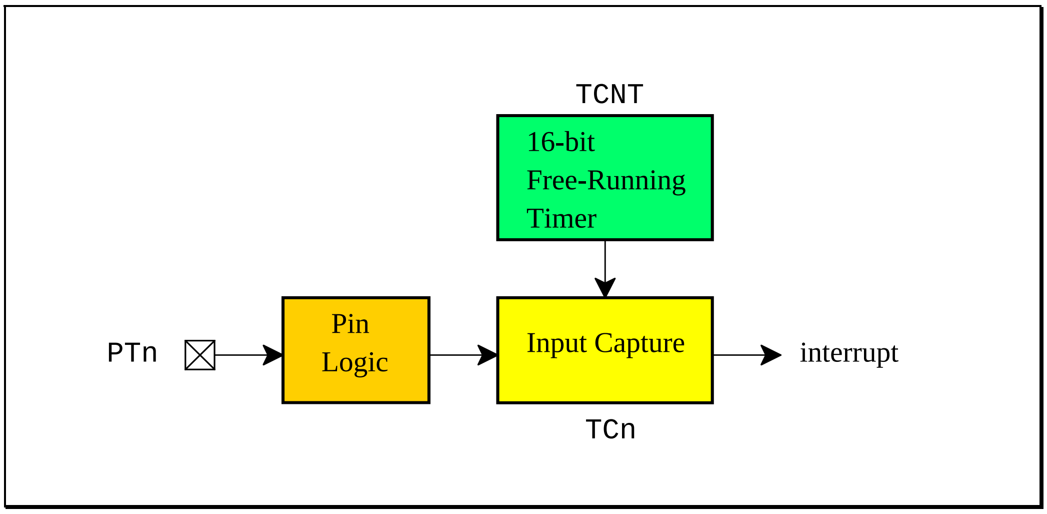


Figure L3.6

The input capture edge detection circuits can be set up using the `TCTL3` and `TCTL4` registers. The options are:

Input compare configurations

Configuration
Capture disabled
Capture on rising edges only
Capture on falling edges only
Capture on any edge (rising or falling)

Two or three actions result from a capture event:

1. The current 16-bit `TCNT` value is copied into the input capture register, `TCn`.
2. The input capture flag is set in `TFLG1`.
3. An interrupt is requested when the mask bit is `1` in `TIE`.

Software Requirements

1. The software is to incorporate all the features of Lab 2.
2. A real-time clock must be implemented using the CRG's RTI module that keeps track of the minutes and seconds of elapsed time since start up. The RTI ISR should be called as little as possible, since its only job is to keep track of seconds. The updating of the real-time clock *seconds* and *minutes* should be performed in the background (in `main.c`), i.e. the ISR should only update the elapsed milliseconds and microseconds. In addition, Port E, pin 7 (the red LED) should be set up as an output and toggled *approximately* every half a second.
3. A periodic interrupt must be implemented using the modulus down-counter with a period of 2 ms.
4. An output capture compare interrupt must be implemented on channel 7 that simulates the "Transmit Complete" function, i.e. that generates an interrupt after a time that approximately equals the time it takes to send a character out the SCI. Remember that the format of data sent is 8N1 which means 10 bits are sent for each character.

   Note: Due to a mask set erratum, it is not possible to use the "Fast Flag Clear All" function with the modulus down-counter and output capture compare. Therefore, individual flags of the ECT must be cleared separately.
5. The SCI software must use a fully interrupt-driven approach to sending and receiving characters, using FIFO buffers.

6. Extra commands of the ModCon serial protocol to be implemented are:

ModCon to PC	PC to ModCon
	0x09 Special – Toggle debug mode
0x0C Time	

7. The ModCon response to a “0x09 Special – Toggle debug mode” packet should be to toggle the internal state of a *non-volatile* `Debug` variable (it will need to be stored in EEPROM).

8. When in “debug” mode, the ModCon should send a “0x0C Time” packet each time the real-time clock changes (every second).

9. When in “debug” mode, the following bits of Port T should be toggled in the respective ISR for debug purposes:

ISR	Port T bit number
Modulus down-counter	4
RTI	5
ECT Channel 7	6

   Note: Set the direction of the port pins to output.

   You should verify the timing of the interrupts using a DSO or logic analyser, whilst in the “debug” mode.

10. TortoiseSVN must be used for version control.

Debugging Tip

Whilst debugging interrupts, it is important to understand the operation of some of the hardware registers – in this case the most important is the `SCI0SR1` status register. Careful reading of the SCI Block User Guide tells us that if `RDRF` is set, then a read of the `SCI0DRL` register will clear the bit. Consequently, if you set a break point inside `SCI0_ISR`, the debugger will happily read `SCI0SR1`, then `SCI0DRL`, and display the results in the "Data:1" pane as though they are global variables. Unfortunately, this satisfies the hardware requirements for a clear of the `RDRF` bit, and subsequently your code may not work correctly (since `RDRF` is now 0). The "workaround" to this problem is to set breakpoints in the ISR **after** your code has examined the `SCI0SR1` status register:

```
void interrupt 20 SCI0_ISR(void)
{
  // Receive a character
  if (SCI0CR2_RIE)
  {
    // Clear RDRF flag by reading the status register
    if (SCI0SR1_RDRF)
      (void)FIFO_Put(&RxFIFO, SCI0DRL);
   }
```

This may be a problem for other hardware modules which use a similar mechanism for clearing status flags.

Marking

The software should be ready for marking on the date specified in the Timetable in the Learning Guide.

Software marking will be carried out in the laboratory, in the format of an oral exam.

Marking criteria are on the front page. Also refer to the document "Software Style Guide".

Software Specification

The following non-volatile debug variable should be declared.

EEPROM.h

```
...
// Debug flag
#define    Debug       _EW(0x420)
...
```

The following header files are suggested, but not mandatory.

clock.h

```
// ----------------------------------------
// Filename: clock.h
// Description: Routines for maintaining
//   a real-time clock
// Author: (your name)
// Date: (the date)

#ifndef CLOCK_H
#define CLOCK_H

// new types
#include "types.h"

extern UINT8 Clock_Seconds, Clock_Minutes;

// ----------------------------------------
// Clock_Setup
//
// Sets up the clock
// Input:
//   prescaleRate is the desired prescale rate
//   modulusCount is the deired modulus count
//   These are set by the user according to Table 3.2
//   in the CRG Block User Guide, to  ensure ticks
//   of the clock occur every 65.536 ms
// Output:
//   TRUE if the clock was setup successfully
// Conditions:
//   none
//
void Clock_Setup(const UINT8 prescaleRate, const UINT8 modulusCount);

// ----------------------------------------
// Clock_Update
//
// Updates the clock by converting milliseconds
//   and microseconds into seconds and minutes
// Updates the clock by converting milliseconds
//   and microseconds into seconds and minutes
// Input:
//   none
// Output:
//   TRUE if clock seconds have changed
// Conditions:
//   Assumes that the clock has been set up
//
BOOL Clock_Update(void);

#endif
```

L3.14

timer.h

```c
// ----------------------------------------
// Filename: timer.h
// Description:
//   Routines to implement general purpose timers
//   Routines to support the modulus down-counter
//     as a periodic timer
// Author: (your name)
// Date: (the date)

#ifndef TIMER_H
#define TIMER_H

// new types
#include "types.h"

typedef enum
{
  TIMER_OUTPUT_DISCONNECT,
  TIMER_OUTPUT_TOGGLE,
  TIMER_OUTPUT_LOW,
  TIMER_OUTPUT_HIGH
} TTimerOutputAction;

typedef enum
{
  TIMER_INPUT_OFF,
  TIMER_INPUT_RISING,
  TIMER_INPUT_FALLING,
  TIMER_INPUT_ANY
} TTimerInputDetection;

typedef enum
{
  TIMER_Ch0,
  TIMER_Ch1,
  TIMER_Ch2,
  TIMER_Ch3,
  TIMER_Ch4,
  TIMER_Ch5,
  TIMER_Ch6,
  TIMER_Ch7
} TTimerChannel;

typedef struct
{
  BOOL outputCompare;
  TTimerOutputAction outputAction;
  TTimerInputDetection inputDetection;
  BOOL toggleOnOverflow;
  BOOL interruptEnable;
  BOOL pulseAccumulator;
} TTimerSetup;

// ----------------------------------------
// Timer_SetupPeriodicTimer
//
// Sets the period of the periodic timer
// Input:
//   microSeconds is the number of microseconds for one period
//   busClk is the actual bus clock rate in Hz,
// Output:
//   none
// Conditions:
//   none

void Timer_SetupPeriodicTimer(const UINT16 microSeconds,
  const UINT32 busClk);
```

```
// ----------------------------------------
// Timer_PeriodicTimerEnable
//
// Enables or disables the periodic timer
// Input:
//   enable is a Boolean value indicating whether to enable the timer
// Output:
//   none
// Conditions:
//   Assumes the timer has been set up

void Timer_PeriodicTimerEnable(const BOOL enable);

// ----------------------------------------
// Timer_Setup
//
// Sets up the Enhanced Capture Timer unit for
//   operations with the timers
// Input:
//   none
// Output:
//   none
// Conditions:
//   none

void Timer_Setup(void);

// ----------------------------------------
// Timer_Init
//
// Initializes a timer channel
// Input:
//   channelNb is the timer channel number
//   aTimerSetup is a structure containing the parameters to
//     be used in setting up the timer channel:
//       outputCompare is TRUE when the timer should be configured as an
//         output, otherwise it is configured as an input
//       outputAction is the action to take on a successful output compare
//       inputDetection is the type of input capture detection
//       toggleOnOverflow is TRUE when the timer should toggle on overflow
//       interruptEnable is TRUE to enable interrupts
// Output:
//   none
// Conditions:
//   Assumes that Timer_Setup has been called

void Timer_Init(const TTimerChannel channelNb,
  const TTimerSetup * const aTimerSetup);

// ----------------------------------------
// Timer_Enable
//
// Enables or disables a timer channel interrupt
// Input:
//   channelNb is the timer channel number
//   enableInt is a Boolean value indicating whether to enable the
//     interrupt on the timer channel
// Output:
//   none
// Conditions:
//   none

void Timer_Enable(const TTimerChannel channelNb, const BOOL enableInt);
```

```
// ----------------------------------------
// Timer_Enabled
//
// Returns the status of a timer
// Input:
//   channelNb is the timer channel number
// Output:
//   a Boolean value indicating whether the channel is enabled
// Conditions:
//   none

BOOL Timer_Enabled(const TTimerChannel channelNb);

// ----------------------------------------
// Timer_Set
//
// Sets a timer channel to generate an
//   interrupt after a certain number of busClk cycles
// Input:
//   channelNb is the timer channel number from 0-7
//   busClkCyclesDelay is the number of busClk cycles
//     to wait until an interrupt is triggered
// Output:
//   none
// Conditions:
//   none

void Timer_Set(const TTimerChannel channelNb,
  const UINT16 busClkCyclesDelay);

#endif
```

The following macros should be declared.

types.h

```
...
// Macros for critical sections
#define EnterCritical() { asm pshc; asm sei; asm leas 1,sp; }
#define ExitCritical()  { asm leas -1,sp; asm pulc; }
...
```

Start a `main.h` file and place various program constants, `#defines`, and header files in it.