CMPE2250 - Table of Contents

Overview

Reference Materials / Required Materials

9S12XCPU Reference Manual ("Little Pink")

9S12X Data Sheet ("Big Pink")

Libraries from CMPE1250 (Details below)

'Old' Course Pack (CP1503R13, Ross Taylor)

MC9S12XDP512 Micro Board

Analog Discovery 2

RS232 to USB Cable

CodeWarrior and Visual Studio Code

MCP4812 Dual SPI DAC

CMPE2250 requires full completion of CMPE1250. The assumption is that you have a complete set of libraries and that you can operate those libraries.

Code documentation and submission practices established in CMPE1250 will continue in CMPE2250, and your instructor may have additional requirements (and requirements will be more strict).

The following library functions will be used throughout the lecture component of the course, and you may be required to use these functions or equivalents:

Switch/LED (SWL_)

● `Init`	Initialize port(s) for use, configure default states
● `On`	Turn the specified indicator LED on
● `Off`	Turn the specified indicator LED off
● `Tog`	Toggle the state of the specified indicator LED
● `Pushed`	true if the specified switch is pressed
● `Any`	true if any switch is pressed
● `Transition`	true if the specified switch has transitioned to down since last checked

PLL (PLL_)

● `To20MHz`	Use the PLL facility to bring the bus rate to 20MHz

Timer (Timer_)

● `Init`	Enable the timer with the specified prescale. Arm `OC0` for OC function with specified initial interval and pin action. Save bus rate for the `Sleep` function.
● `Sleep`	Block for the specified interval in ms, using timer channel 6. Works with the existing prescale and requires timer to be enabled. Does not affect `OC0` operation.

7-Segment Display (Segs_)

- Includes private helpers for port manipulation (latch, mode low, mode high)

`Init`	Initialize port(s) for use, configure default states
`Normal`	Display the specified value at the specified address, normal decode (DP opt)
`Custom`	Display the custom segment control value at the specified address
`CodeB`	Display the specified value at the specified address, CodeB decode
`16H`	Display the specified 16-bit value in HEX on the specified row
`16D`	Display the specified 16-bit value in DEC on the specified row (Err > 9999)
`8H`	Display the specified 8-bit value in HEX starting at the specified address
`Clear`	Clear the display
`SayErr`	Display `Err` on the specified row
`SayHelp`	Display `HELP` on both rows of the display (Code B)

LCD (lcd_)

- Includes private helpers for port manipulation (RWU, RWD, EU, ED, RSU, RSD, DELAY)

`Busy`	Private: Wait for LCD to be not busy
`Init`	Initialize port(s) for use, configure default states
`Inst`	Send the specified byte as a command to the LCD
`Data`	Send the specified byte as data to the LCD
`Addr`	Set the DD RAM address (cursor position)
`AddrXY`	Decode X/Y coordinates and set `Addr`
`String`	Send a `NUL`-terminated string to the LCD
`StringXY`	Send a `NUL`-terminated string to the LCD at the specified position
`DispControl`	Set display and cursor options (on/off, blink/no blink)
`Clear`	Clear the LCD display
`Home`	Move the cursor to the home position
`CGAddr`	Set the CG Data address
`CGChar`	Send a character pattern array to CG RAM

You will continue to add and modify libraries in CMPE2250, although your instructor will leave more of the ‘what’ and ‘how’ to you this term.

In addition to the standard reference materials, you will be provided with all necessary lecture notes, data sheets, or other references that you will require. These items will be provided in Teams or on Moodle.

Attendance is important for success! It is strongly recommended that you attend all classes. In-class discussions will reveal useful information related to notes and work activities that are best not missed.

Interrupts (9S12X)

Reference Materials

2.2.5.5 – 9S12XCPU Reference Manual (I Mask Bit)

7.5.3 - 7.5.5 – 9S12XCPU Reference Manual (Interrupt Mechanics)

16.5.2 – Data Sheet (Interrupt Nesting - Optional)

16.5.3.1 – Data Sheet (Wake Up from Stop or Wait Mode)

Our micro supports interrupts!

Simply, the micro can be interrupted from what it is doing, can service the source of the interrupt, and can then resume what it was doing.

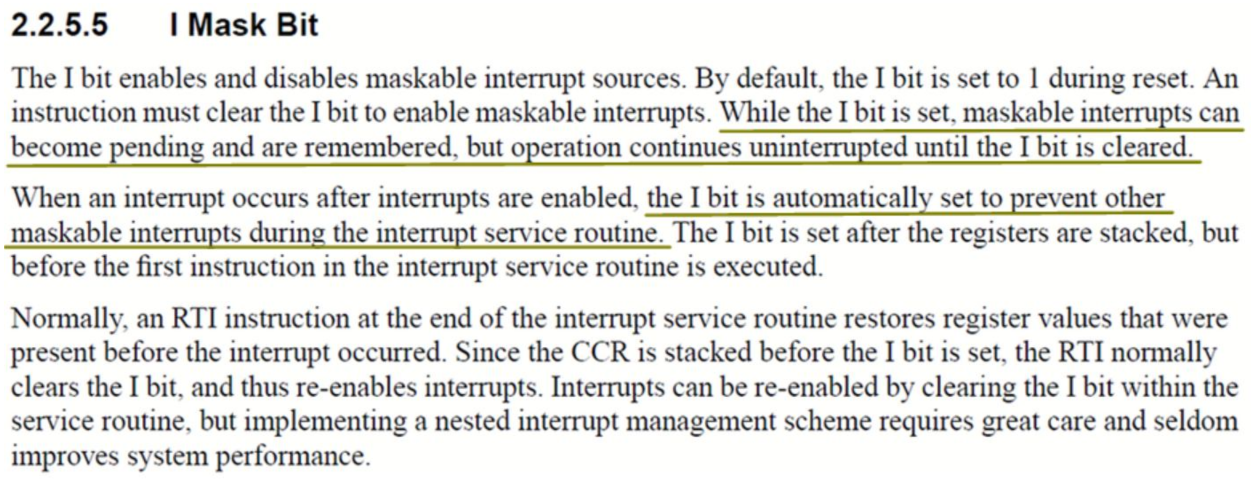
You do this when you are eating dinner and the doorbell rings.

The interrupt mechanism permits the micro to respond very quickly to interrupt sources and mitigates the need for state polling in many circumstances. The use of interrupts also permits the micro to nap and wait for interrupts; a feature that can significantly reduce power consumption.

The micro uses stack space to store the current context, and to restore it when done servicing the interrupt. See section A.2 of the S12XCPU Reference Manual!

Maskable Interrupts

Some interrupt sources may be masked, and some may not. When masked, an interrupt will not occur immediately, but it may become pending. This is controlled by the '`I`' bit in the CCR.



2.2.5.5 I Mask Bit

The I bit enables and disables maskable interrupt sources. By default, the I bit is set to 1 during reset. An instruction must clear the I bit to enable maskable interrupts. While the I bit is set, maskable interrupts can become pending and are remembered, but operation continues uninterrupted until the I bit is cleared.

When an interrupt occurs after interrupts are enabled, the I bit is automatically set to prevent other maskable interrupts during the interrupt service routine. The I bit is set after the registers are stacked, but before the first instruction in the interrupt service routine is executed.

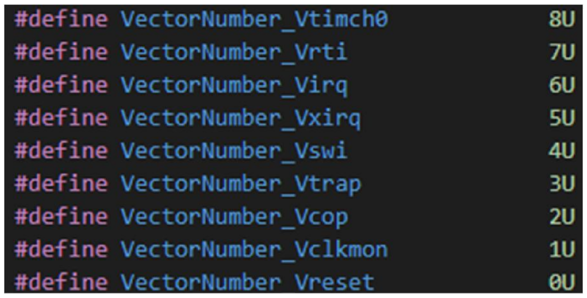
Normally, an RTI instruction at the end of the interrupt service routine restores register values that were present before the interrupt occurred. Since the CCR is stacked before the I bit is set, the RTI normally clears the I bit, and thus re-enables interrupts. Interrupts can be re-enabled by clearing the I bit within the service routine, but implementing a nested interrupt management scheme requires great care and seldom improves system performance.

We don't normally directly manipulate the `I` bit in the CCR in C, but there is an `EnableInterrupts` and a `DisableInterrupts` macro available in CodeWarrior to clear and set the I bit respectively. Our program template contains (it always has) an `EnableInterrupts` call.

NOTE: The call to the interrupt handling routine and the `RTI` instruction are handled by the compiler in C.

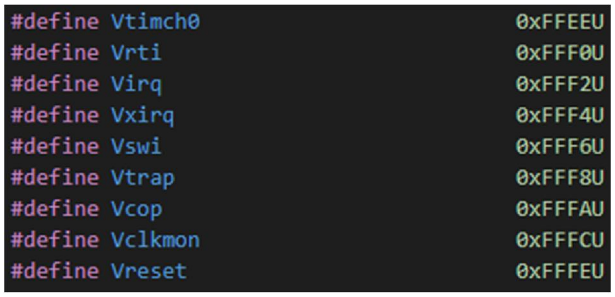
Sources of Interrupts

Interrupts on this device are pre-defined and have a priority. The derivative file contains definitions for each interrupt, as well as the vector location:



```
#define VectorNumber_Vtimch0     8U
#define VectorNumber_Vrti        7U
#define VectorNumber_Virq        6U
#define VectorNumber_Vxirq       5U
#define VectorNumber_Vswi        4U
#define VectorNumber_Vtrap       3U
#define VectorNumber_Vcop        2U
#define VectorNumber_Vclkmon     1U
#define VectorNumber_Vreset      0U
```

Interrupt Number (source)



```
#define Vtimch0                  0xFFEEU
#define Vrti                     0xFFF0U
#define Virq                     0xFFF2U
#define Vxirq                    0xFFF4U
#define Vswi                     0xFFF6U
#define Vtrap                    0xFFF8U
#define Vcop                     0xFFFAU
#define Vclkmon                  0xFFFCU
#define Vreset                   0xFFFEU
```

Interrupt Vector (address)

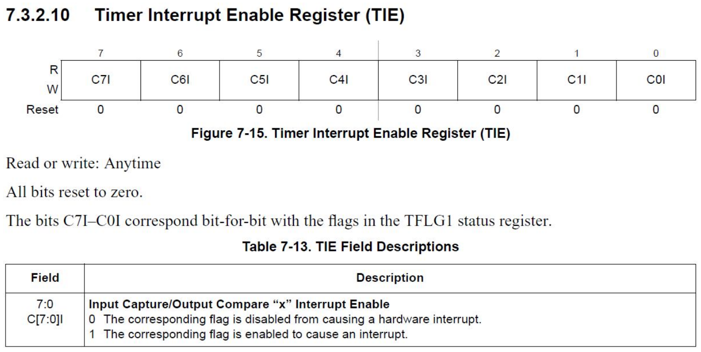
The interrupt number determines the priority of the interrupt where the lower the number the higher the priority. The vector address stores the address of the first instruction of interrupt handler (where the interrupt service routine (*ISR)* is). Note: vectors are stored in ROM and are programmed into the micro automatically by the tool chain when you run your program.

As a programmer you must determine what interrupts you are interested in. You then create ISRs and formally turn on the interrupt source. Some of these, like the reset vector for example, are done for you.

Integrated modules will contain zero or more interrupt sources, and a single interrupt vector may be used for multiple conditions. The SCI, for example, uses a single interrupt to handle multiple conditions.

If multiple conditions are associated with a single interrupt source, you must be prepared to determine which condition caused the interrupt, unless you only enabled one of the possible interrupt conditions.

Modules that support interrupt generation will typically have a configuration bit that enables the interrupt. The timer, for example, can generate interrupts on an output compare event. Section 7.3.2.10 of Big Pink describes the configuration register for timer interrupts:



7.3.2.10 Timer Interrupt Enable Register (TIE)

	7	6	5	4	3	2	1	0
R/W	C7I	C6I	C5I	C4I	C3I	C2I	C1I	C0I
Reset	0	0	0	0	0	0	0	0

Figure 7-15. Timer Interrupt Enable Register (TIE)

Read or write: Anytime

All bits reset to zero.

The bits C7I–C0I correspond bit-for-bit with the flags in the TFLG1 status register.

Table 7-13. TIE Field Descriptions

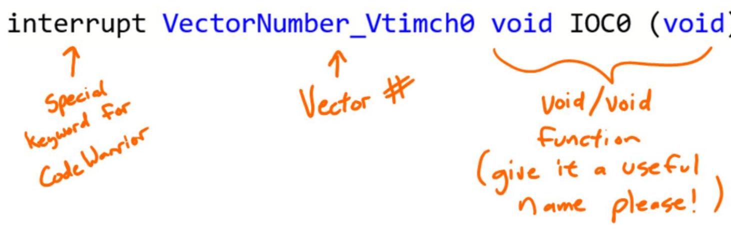
Field	Description
7:0 C[7:0]I	**Input Capture/Output Compare "x" Interrupt Enable** 0 The corresponding flag is disabled from causing a hardware interrupt. 1 The corresponding flag is enabled to cause an interrupt.

This register permits enabling interrupts for any combination of the timer channels. Turning on channel zero would permit your current library code to generate an interrupt when an output compare occurs, and you would not need to poll for the event, as you did in CMPE1250.

Interrupt Service Routine (ISR)

When the interrupt occurs, the address for the ISR is pulled from the matching vector and the ISR is called. CodeWarror facilitates this with a function that has a special form:

```
/********************************************************************/
//    Interrupt Service Routines
/********************************************************************/
interrupt VectorNumber_Vtimch0 void IOC0 (void)
{
  // ack interrupt by writing to that bit
  TFLG1 = TFLG1_C0F_MASK;

  // rearm for next event
  TC0 = TC0 + new_interval;
}
```



```
interrupt VectorNumber_Vtimch0 void IOC0 (void)
```

Special keyword for CodeWarrior
Vector #
Void/Void Function (give it a useful name please!)

This function will be called automatically when the interrupt occurs (in this case, when a successful input capture/output compare event occurs).

As with polling, the event (and now the interrupt) must be acknowledged by writing a one to the corresponding interrupt flag. You were doing this last term, but instead of responding to the interrupt in an ISR, you were polling for the event flag. The flag clearing and rearming operations are the same.

Note: during an ISR the `I` bit is turned on (interrupts inhibited). This prevents interrupts from nesting.

In general, you are required to acknowledge the interrupt, and some interrupts have slightly different acknowledgement (clearing) mechanisms. We will cover each as we encounter them. If an interrupt is not acknowledged, the ISR will be called repeatedly, as the interrupt will always be pending.

Timer Interrupt

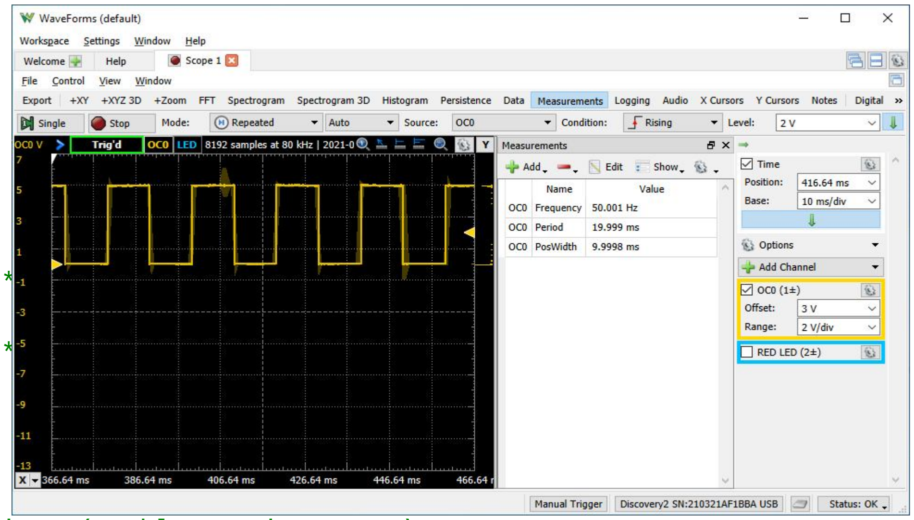
When you were developing your timer library, you were given a prototype for the `Timer_Init` function that optionally enabled the interrupt for `OC0`:

```
void Timer_Init (
  unsigned long ulBusClock,   // current bus rate (saved for Timer_Sleep use)
  Timer_Prescale prescale,    // desired prescale
  unsigned int uiOffset,      // offset from TCNT for first event
  int enableInt,              // enable interrupts?
  Timer_PinAction pinAction)  // event action on OC0 pin (9)
```

You may now complete the `Timer_Init` function by enabling/disabling the interrupt for `OC0`:

```
// enable interrupts on OC0 if parameterized as true (7.3.2.10)
if (enableInt)
  TIE_C0I = 1;
```

With the interrupt enabled, it will no longer be necessary for you to poll for the `OC0` event! This has quite a bit of impact on how we will write programs going forward. The executing code will be interrupted when the event occurs, and the code will automatically resume after the ISR has been called. What is left to do in main?


WaveForms (default)

Name		Value
OC0	Frequency	50.001 Hz
OC0	Period	19.999 ms
OC0	PosWidth	9.9998 ms

Mode: Repeated, Auto. Source: OC0. Condition: Rising. Level: 2 V.
Time: Position: 416.64 ms, Base: 10 ms/div.
OC0 (1±): Offset: 3 V, Range: 2 V/div.
8192 samples at 80 kHz
Discovery2 SN:210321AF1BBA USB
Status: OK

```
void main(void)
{
  // main entry point
  _DISABLE_COP();
  EnableInterrupts;

  /*********************
  // initializations
  /*********************
  PLL_To20MHz();
  Segs_Init();
  SWL_Init();

  // for interrupt technique (enable OC0 interrupt)
  Timer_Init (20E6, Timer_Prescale_32, 6250, 1, Timer_Pin_Toggle);

  for (;;)
  {
    // nothing to do!
  }
}

/*********************************************************************/
//    Interrupt Service Routines
/*********************************************************************/
interrupt VectorNumber_Vtimch0 void IOC0 (void)
{
  // ack interrupt by writing to that bit
  TFLG1 = TFLG1_C0F_MASK;

  // rearm for next event
  TC0 = TC0 + 6250; // +10ms
}
```

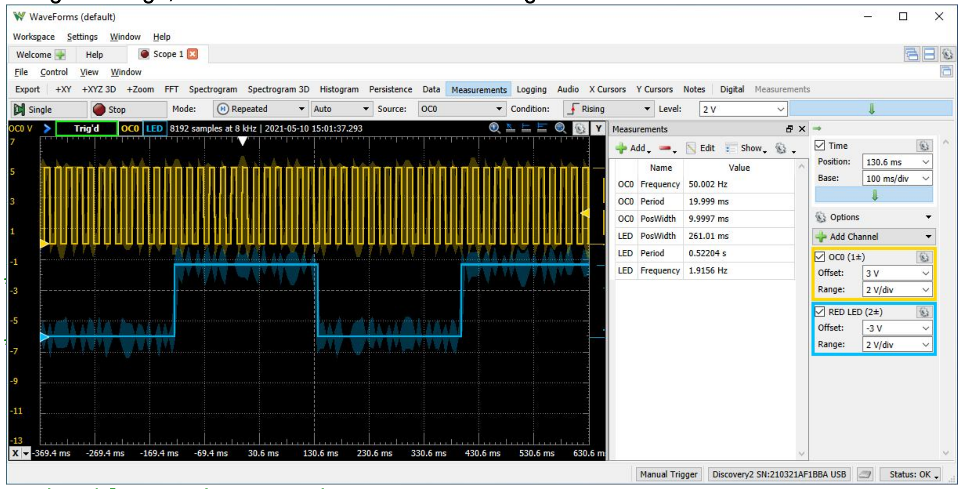
You are free to put any code in main, if you are aware that this code may be interrupted. Interrupting code is normally OK, but care must be taken not to interrupt code that:

- may be doing something that is time sensitive
- assumes a state that the ISR might change

We will explore this in more depth as we go, but here are some rules for ISRs:

- Should be as short as possible! Remember: interrupts are suppressed during an ISR, so a long ISR can cause additional interrupts to become pending.
- Should acknowledge the source of the interrupt, as a rule, not best practice!
- Should not contain code that causes unpredictable states within the main program.

The following code in main will run as fast as it can and will be interrupted every `OC0` event (10ms in this case). Because the ISR is not using the segs, there should be no conflicting state issue:


WaveForms (default)

8192 samples at 8 kHz | 2021-05-10 15:01:37.293

Mode: Repeated, Auto; Source: OC0; Condition: Rising; Level: 2 V

	Name	Value
OC0	Frequency	50.002 Hz
OC0	Period	19.999 ms
OC0	PosWidth	9.9997 ms
LED	PosWidth	261.01 ms
LED	Period	0.52204 s
LED	Frequency	1.9156 Hz

Time Position: 130.6 ms; Base: 100 ms/div

OC0 (1±) Offset: 3 V; Range: 2 V/div

RED LED (2±) Offset: -3 V; Range: 2 V/div

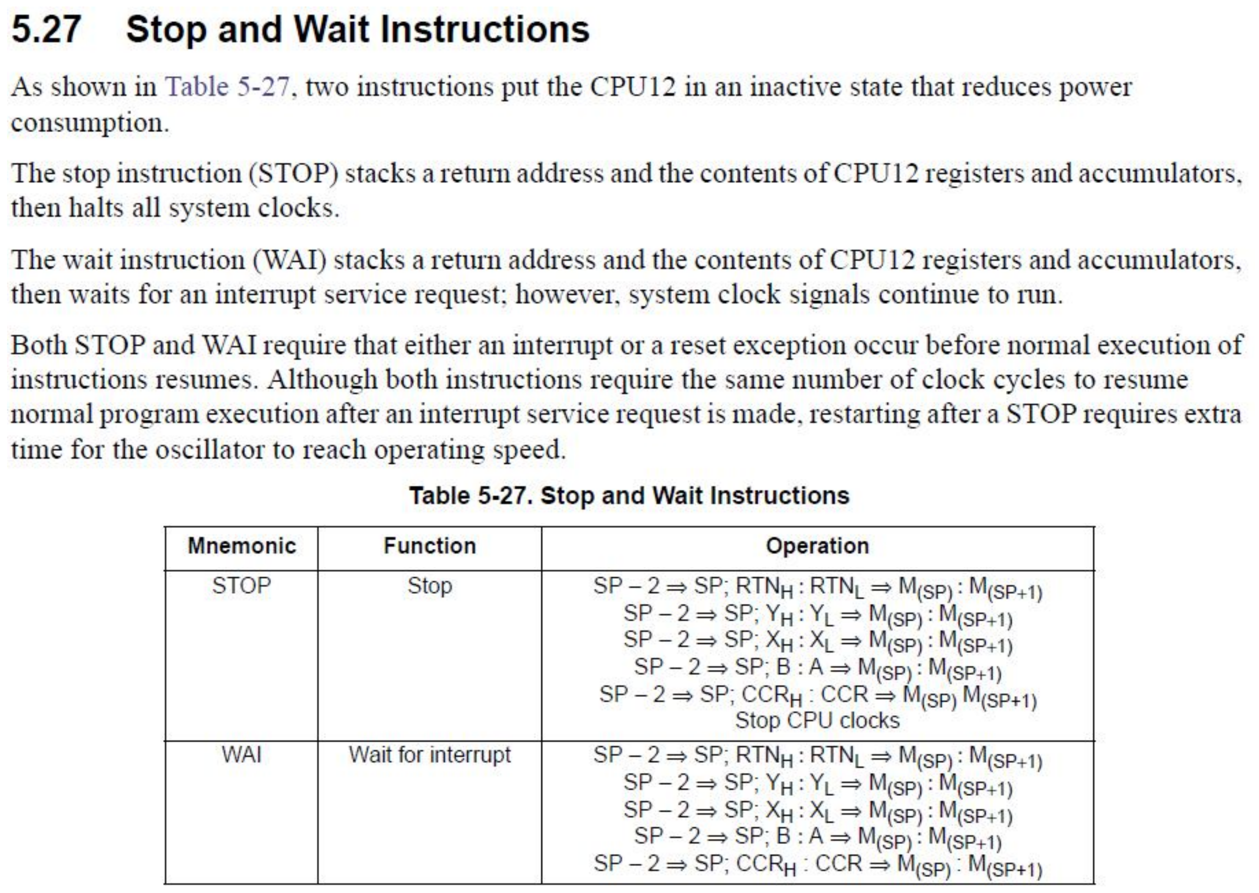
```
unsigned int _count = 0;

void main(void)
{
  // main entry point
  _DISABLE_COP();
  EnableInterrupts;

  /***********************
  // initializations
  /***********************
  PLL_To20MHz();
  Segs_Init();
  SWL_Init();

  // for interrupt technique (enable OC0 interrupt)
  Timer_Init (20E6, Timer_Prescale_32, 6250, 1, Timer_Pin_Toggle);

  for (;;)
  {
    // count as fast as you can!
    Segs_16D (_count = _count + 1 > 9999 ? 0 : _count + 1, Segs_LineTop);

    if (!_count)
      SWL_TOG (SWL_RED); // show how quickly flat out run wraps at 10k boundary
  }
}
```

There is a lot to glean from what this sample code is doing. The interrupt manages to maintain a stable 10ms toggle on the OC0 channel, as the interrupt, effectively, has the highest priority. When not servicing the ISR, the CPU is committed to running the main loop at full speed. No time is lost to polling, so this is a much more efficient way to write code that is responsive to events.

Directing 100% CPU utilization at a task is nice if all that effort is solving a problem. Directing 100% CPU at polling or blocking is not compatible with low power consumption design. If you are waiting for an event, and truly can't do anything until the event occurs, polling is a terrible option. It would be very power inefficient to loop at maximum speed polling for an event. It would be equally bad to spin around in a loop doing nothing waiting for an interrupt.

STOP and WAIT

Our micro contains special modes of operation that permit low power wait states. In these modes, the micro stops instruction execution and waits for any interrupt to bring it back to normal mode:



5.27 Stop and Wait Instructions

As shown in Table 5-27, two instructions put the CPU12 in an inactive state that reduces power consumption.

The stop instruction (STOP) stacks a return address and the contents of CPU12 registers and accumulators, then halts all system clocks.

The wait instruction (WAI) stacks a return address and the contents of CPU12 registers and accumulators, then waits for an interrupt service request; however, system clock signals continue to run.

Both STOP and WAI require that either an interrupt or a reset exception occur before normal execution of instructions resumes. Although both instructions require the same number of clock cycles to resume normal program execution after an interrupt service request is made, restarting after a STOP requires extra time for the oscillator to reach operating speed.

Table 5-27. Stop and Wait Instructions

Mnemonic	Function	Operation
STOP	Stop	$SP - 2 \Rightarrow SP$; $RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $Y_H : Y_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $X_H : X_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $B : A \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $CCR_H : CCR \Rightarrow M_{(SP)}\ M_{(SP+1)}$ Stop CPU clocks
WAI	Wait for interrupt	$SP - 2 \Rightarrow SP$; $RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $Y_H : Y_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $X_H : X_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $B : A \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $CCR_H : CCR \Rightarrow M_{(SP)} : M_{(SP+1)}$

The device is also configurable at the module level to permit additional power saving measures. We will explore these as we cover those modules.

STOP is heavy-handed and would normally be used when the micro needs to wait for some time before wake-up. Stopping the clocks will stop other activities we might be doing (and possibly some sources of interrupts necessary to wake the device), so we won't explore the STOP instruction. Not that it will be an issue for us but starting and stopping the crystal oscillator repeatedly may cause wear and tear on the crystal (although I can't find a specific reference to this).

https://www.nxp.com/docs/en/application-note/AN1706.pdf

https://www.nxp.com/docs/en/application-note/AN3208.pdf

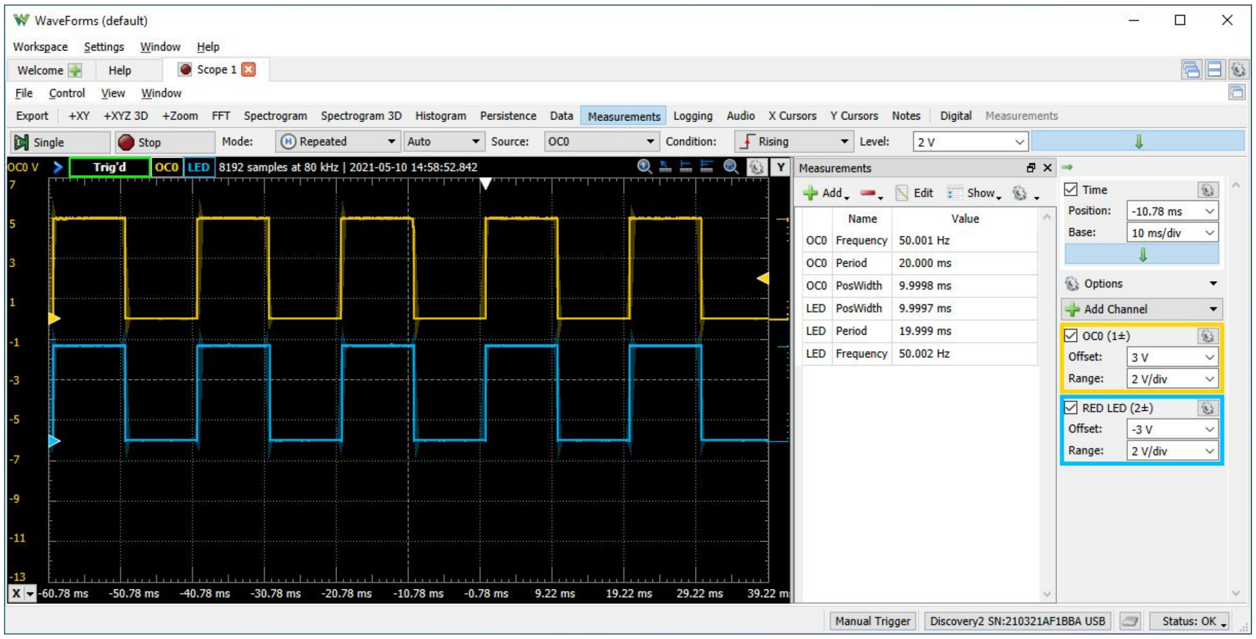
To enter the wait state, we must execute the machine specific `WAI` instruction. There is no C equivalent, or macro for this. To execute this instruction, you must place it in an `asm` (assembly language) command. Like the `interrupt` keyword, only CodeWarrior will recognize the `asm` command (VS Code will not view this as valid C), and the arguments with it.

The following code shows how the main loop may wait for an interrupt, without the downside of wasting CPU cycles to do so:

```
for (;;)
{
  asm wai;  // enter wait state ()

  // count as fast as interrupts occur (should be every 10ms in this case)
  Segs_16D (_count = _count + 1 > 9999 ? 0 : _count + 1, Segs_LineTop);

  //if (!_count) *** removed so we see LED toggle each interrupt
    SWL_TOG (SWL_RED);
}
```

Because the code is effectively blocking on the WAI instruction, the code in the main loop will only execute *after* an interrupt event. In this case, the timer is the only interrupt source, so this code will run precisely once every interrupt (10ms in this case). This assumes the main code takes less than 10ms to execute, or it will be interrupted by the timer again!

You will often use a similar pattern to manage the flow of your programs. Given the chance to execute code at intervals, you will be able to make measurements, update displays, check switches, and perform other tasks reliably. You may throttle the loop as necessary to balance responsiveness and power efficiency.


WaveForms (default)
Workspace Settings Window Help
Welcome Help Scope 1
File Control View Window
Export +XY +XYZ 3D +Zoom FFT Spectrogram Spectrogram 3D Histogram Persistence Data Measurements Logging Audio X Cursors Y Cursors Notes Digital Measurements
Single Stop
Trig'd OC0 LED
-60.78 ms -50.78 ms -40.78 ms -30.78 ms -20.78 ms -10.78 ms -0.78 ms 9.22 ms 19.22 ms 29.22 ms 39.22 ms
Add Edit Show
Options
Add Channel
Manual Trigger Discovery2 SN:210321AF1BBA USB Status: OK

Mode: Repeated Auto Source: OC0 Condition: Rising Level: 2 V

8192 samples at 80 kHz | 2021-05-10 14:58:52.842

Measurements

	Name	Value
OC0	Frequency	50.001 Hz
OC0	Period	20.000 ms
OC0	PosWidth	9.9998 ms
LED	PosWidth	9.9997 ms
LED	Period	19.999 ms
LED	Frequency	50.002 Hz

Time
Position: -10.78 ms
Base: 10 ms/div

OC0 (1±)
Offset: 3 V
Range: 2 V/div

RED LED (2±)
Offset: -3 V
Range: 2 V/div

Nesting of Interrupts

Don’t.

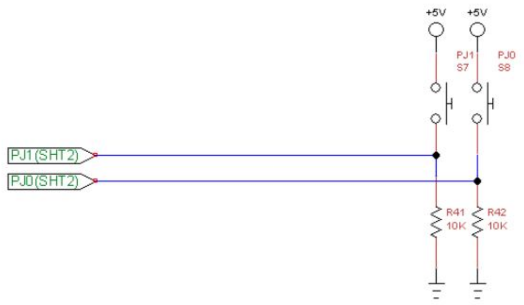
PORT J Interrupts (GPIO Interrupts)

Several modules on the micro may generate interrupts as a normal part of their function. Like the timer, we will explore the context-specific nature of these interrupts as we proceed through the course.

Some physical pins on the micro can generate interrupts when a logic change is detected on the pin. For example, if the pin transitions from a logic 0 (0V) to a logic 1 (5V) by an external actor, an interrupt can be generated. This is known as a rising-edge trigger. Falling edge is also possible. Together, the two are known more generally as *edge-detection*.

Your board contains two switches located to the bottom-left of the micro. These switches are connected to `PJ1` and `PJ0`. Port J supports interrupt generation, as the normal module behavior (UART) for these pins requires it. If the port is used for GPIO, that ability remains.

Electrically, these switches are configured to pull the pins high when pressed:



This would mean that the pins could be configured to trigger an interrupt on a rising edge (or less intuitively on a falling edge). Pressing a switch should trigger an interrupt. The ISR should acknowledge the interrupt and set in motion “switch pushed behavior”.

Because these pins are essentially being used for GPIO, the normal steps for operating the pins should be completed:

```
// setup port J for interrupts
PTJ  &= 0b11111100; // clear port J or will interrupt once for free (22.3.2.54)
DDRJ &= 0b11111100; // j0:1 inputs (22.3.2.56)
PPSJ |= 0b00000011; // j0:1 rising edge (22.3.2.59)
PIEJ |= 0b00000011; // j0:1 cause interrupts (22.3.2.60)
```

The bottom two lines simply select the triggering for the interrupts and formally enable the interrupts.

In the case of Port J, there are multiple interrupts sources (each pin) that are handled by a single interrupt handler. It is the responsibility of the ISR to sort out what the interrupt source is, and to individually acknowledge each source. It is unlikely that a person would push both buttons at *exactly* the same time, but that can be simulated by inhibiting interrupts for a long period of time:

```
void main(void)
{
  int iFreeCount = 0;

  _DISABLE_COP();
  EnableInterrupts;

  Segs_Init();
  SWL_Init();

  // setup port J for interrupts
  PTJ  &= 0b11111100; // clear port J or will interrupt once for free (22.3.2.54)
  DDRJ &= 0b11111100; // j0:1 inputs (22.3.2.56)
  PPSJ |= 0b00000011; // j0:1 rising edge (22.3.2.59)
  PIEJ |= 0b00000011; // j0:1 cause interrupts (22.3.2.60)

  // this test proves that if both switches are pressed at the same time
  //  there will only be one interrupt generated, meaning
  //  that the status flag would need to be read, and action taken
  //  for all bits that are set, flag is cleared on read
  for (;;)
  {
    unsigned long counter;

    /////////////////////////////////////////////////////
    asm sei; // suspend interrupts (set I in CCR)

    // do big delay to give a chance to push both buttons
    for (counter = 0; counter < 500000; counter++)
      ;

    asm cli; // permit interrupts (clear I in CCR)
    /////////////////////////////////////////////////////

    // counter will show when blocking delay is over
    Segs_16D (iFreeCount++, Segs_LineBottom);

    // fixup freecount on wrap
    if (iFreeCount > 9999)
      iFreeCount = 0;
  }
}
```

```
// ISRs ////////////////////////////////////////////////////////////////////////
//////////
interrupt VectorNumber_Vportj void IntJ (void)
{
  if (PIFJ_PIFJ0) // read of PIFJ (22.3.2.61)
  {
    // ack interrupt
    //PIFJ_PIFJ0 = 1; // can't do R/W clearing (preserves other bits, so clears other flags!)
    PIFJ = PIFJ_PIFJ0_MASK; // write only clear (just this ONE bit)

    // show something happened:
    SWL_TOG (SWL_RED);
  }

  if (PIFJ_PIFJ1)
  {
    // ack interrupt
    PIFJ = PIFJ_PIFJ1_MASK; // write only clear

    // show something happened:
    SWL_TOG (SWL_GREEN);
  }
}
```

The ISR in this case needs to resolve the source of the interrupt, as either or both switches could be pushed. Care must be taken in clearing the flags, as flag clearing requires that you write a `1` to the source (*using read/write code that preserves other bits will inadvertently clear the other flags*).

Remember that it is critical that the ISR is as short as possible. If possible, the ISR would just set flags that the main loop would use for subsequent processing (and that handling code could ultimately be interrupted).

Your instructor will demo this code with you, to ensure that all elements are fully understood. This isn't the only interrupt source that uses a consolidated handler. This will appear again when we cover the SCI.

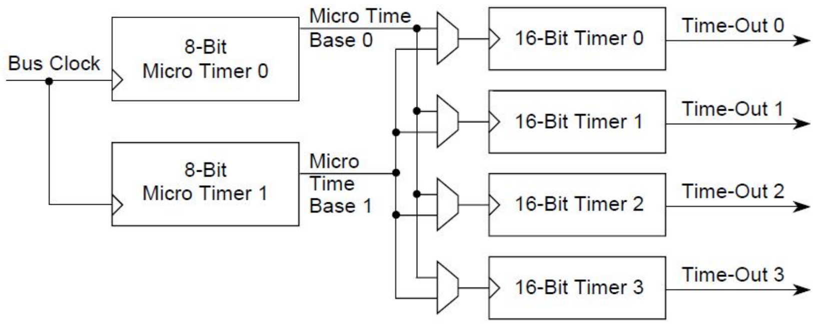
Periodic Interrupt Timer (PIT) – Basics

The timer isn't the only source of timing on your micro!

If you are looking to simply generate periodic interrupts, using the PIT is a good choice.

The PIT module does not have any external pins and is only used to generate interrupts (and trigger events in the XGATE module, but we won't be doing that).

The PIT is relatively easy to setup. It uses two count-down timers driven by the bus clock:



The values in the count-down stages are 8-bits and 16-bits respectively, and automatically have 1 added to the register value. This means that the PIT intervals generated by the two registers jointly are between 1 and 2^{24} bus cycles.

There are four PIT channels. Each channel has its own interrupt. Each channel permits selection between two 8-bit micro timers, but these notes will only cover using micro timer 0 (the default).

Setting up the PIT requires, at minimum, the following instructions (assumes 20MHz bus clock):

```
// enable interrupt on chan 0
PITINTE = 0b00000001;  // 13.3.0.5

// enable chan 0
PITCE = 0b00000001;  // 13.3.0.3

PITMTLD0 = 199; // (200)
PITLD0 = 49999; // (50000)
// should yield 500ms interval (200 * 50000 * 50ns = 0.5s per interrupt)

// finally, enable periodic interrupt, normal in wait, PIT stalled in freeze
PITCFLMT = 0b10100000;  //13.3.0.1
```

You could create a PIT library that permits operation of each PIT channel, with a specified interval, likely in µs. This would provide simpler use compared to the timer, where periodic interrupts are required. Your functions could work out the required register values to provide the desired interval, making the functions trivial to operate for the caller. The caller being you, during a lab exam.

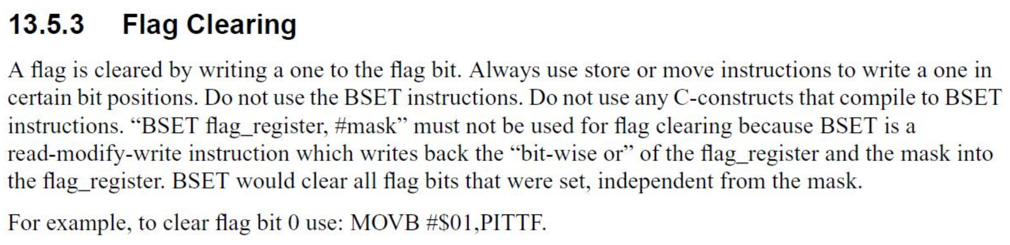
Note that the code above assumes a single operator and trashes the PIT configuration registers. If you put this code in a library, you would write the register assignments with more care.

You will need to include an ISR for the channel(s) you have activated. Note that no rearming is required – the PIT automatically reloads the countdown values and repeats the cycle:

```
interrupt VectorNumber_Vpit0 void PIT0Int (void)
{
  // clear flag
  PITTF = PITTF_PTF0_MASK; // can't R/W - clears other flags, write only

  // take action!
}
```

PIT Flag Clearing

As with other flag clearing operations, it is important to not clear the interrupt flag with an instruction that uses a read/write to set the desired bit. These instructions will attempt to preserve other set bits, inadvertently clearing other pending flags!



13.5.3 Flag Clearing

A flag is cleared by writing a one to the flag bit. Always use store or move instructions to write a one in certain bit positions. Do not use the BSET instructions. Do not use any C-constructs that compile to BSET instructions. “BSET flag_register, #mask” must not be used for flag clearing because BSET is a read-modify-write instruction which writes back the “bit-wise or” of the flag_register and the mask into the flag_register. BSET would clear all flag bits that were set, independent from the mask.

For example, to clear flag bit 0 use: MOVB #$01,PITTF.

As per normal procedure, you should attempt to keep the execution length of the code in the ISR as short as possible. Remember that ISRs automatically inhibit interrupts, so a long ISR may cause pending interrupts to stack.

Calculation of PIT Clock Values

When you are attempting to determine what values you need to use to generate the interval you desire, there are a few rules of thumb to follow:

- First determine the total number of bus cycles in the interval you want.
- Find the largest practical 16-bit factor and plug it into PITLD, then plug the remaining factor for the total product into PITMTLD.

For example, if you want a periodic interrupt interval of 50ms, figure out the total number of bus cycles:

50ms / 50ns = 1M bus cycles.

A suitable large 16-bit factor is 50000, so 50000 goes into `PITLD` (as 49999, as 1 is automatically added).

1M / 50000 = 20, so 20 goes into `PITMTLD` (as 19, as 1 is automatically added) : (20 x 50000 = 1M).

NOTE: Working in the reverse order also works and is arguably a better technique for working with irregular intervals. With a 20MHz bus rate, the longest interval is ~839ms. The shortest interval, in theory, is 50ns, but the ISR would not be able to service such a short interval.

Using the PIT instead of the timer is a nice alternative, as it does not interfere with activities you might want to use the timer for, and automatically rearms. The PIT additionally has dedicated interrupts associated with each channel, so you can have good isolation for each PIT task.

Extra Notes on Interrupts

As you have seen so far in the course, it is possible to request interrupts from a variety of sources on the 9S12X microcontroller. Ideally you will have observed interrupt generation on the RDRF condition for the SCI, and for a rising edge of PJ0 or PJ1.

This document will discuss aspects of interrupts that we have not yet formally covered:

- Maskable interrupts may be inhibited
- Interrupts have priority
- Interrupts may be pending, when inhibited (or not)

Interrupt Inhibit

If you write a section of code that could be adversely affected by an interrupt, you may create a critical section of code by inhibiting interrupts temporarily. This is done by setting the interrupt inhibit bit (the 'I' bit) in the CCR, through the SEI assembly instruction. The interrupt inhibit bit is cleared with the CLI assembly instruction. During the period that the interrupt inhibit flag is set, maskable interrupts will be inhibited.

Sections 2.2.5.5, and 7.5.3 of big pink indicate that an ISR will automatically enable the I bit, so interrupts may not be interrupted. Inhibiting a maskable interrupt does not prevent it from becoming pending; it simply prevents it from causing the ISR from occurring.

Interrupt Priority

Each maskable interrupt source has a priority. The higher the vector in memory, the higher the priority. In C the higher the interrupt number in the derivative file, the lower the priority, so interrupt 0x00 (reset) has the highest priority. Priority is determined by criticality or importance of the interrupt source. The SCI0 has priority 0x20, and Port J has priority 0x24, for example.

Pending Interrupts

If a requested interrupt occurs while maskable interrupts are inhibited, or during an ISR, the interrupt will become pending. This interrupt will be serviced once the interrupt inhibit flag is cleared, and when an active ISR terminated. The next ISR executed is determined by priority.

Section 7.5.5 of big pink indicates that if an interrupt occurs during the handling of an interrupt, the behavior of the RTI instruction (normally executed at the end of an ISR) is different. If no interrupt is pending, code resumes from the point of interruption. If an interrupt is pending, the code resumes in the next ISR.

If an interrupt is pending and the conditions are met to trigger that same interrupt, the instance of that second interrupt is lost. In other words, a particular pending interrupt cannot stack – the flag may only be set once. This is one argument for keeping the ISRs as short as possible: ISRs inherently mask interrupts.

Practical Examination

The following example explores what occurs when an SCI and PortJ interrupt both occur while maskable interrupts are inhibited. In the example, maskable interrupts are inhibited during a long delay – this provides the opportunity to trigger the SCI RDRF condition, and Port J switch press condition:

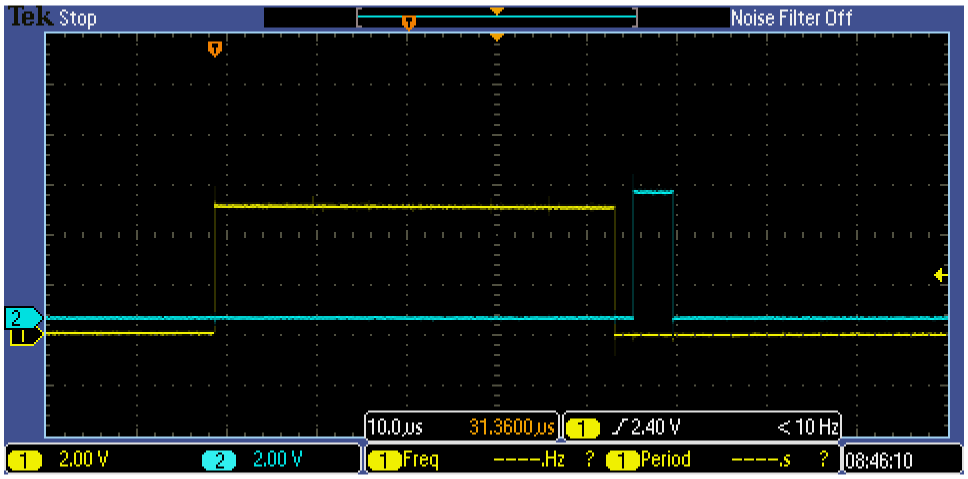
```
asm sei;
for (delayloop = 0; delayloop < 500000; delayloop++)
  ;
asm cli;
```

Internally, each ISR raises a GPIO pin at the start of the ISR and lowers it at the end of the ISR. The SCI ISR output is shown on channel 1 in the capture below, and the PortJ ISR is on channel 2.

SCI ISR	PortJ ISR

```
interrupt VectorNumber_Vsci0 void ISR_SCI0 (void)
{ // total execution time measured at ~44.2us
  // demo, PA6 high
  PORTA |= 0b01000000;

  MuxLEDOut8(6, sci0Bread());

  PORTA &= ~0b01000000;
}
```
	```
	interrupt VectorNumber_Vportj void IntJ (void)
	{ // total execution time measured at ~4.4us
	  // demo, PA7 high
	  PORTA |= 0b10000000;
	  // ack interrupt
	  PIFJ_PIFJ1 = 1;
	  // toggle led (much faster than SCI ISR)
	  SWL_LED_TOG (SWL_RED);
	  PORTA &= ~0b10000000;
	}
	```

The SCI interrupt will take longer to complete, as there is lengthier code in this ISR.



Capture of GPIO port manipulations from ISRs for SCI and PORTJ
Channel 1: PA6, Channel 2: PA7

Regardless of the order of the triggering of the interrupts during the interrupt inhibit, they will be processed in the same order when the inhibit is lifted. SCI has a higher priority than Port J, so the SCI ISR occurs first. Note the very short time between the two ISR executions - just 2.2us, as measured, which includes GPIO operations.

From this we can learn that the interrupt requests are logically queued by priority.

Concurrent Interrupt Requests

What if two interrupt requests occur at exactly the same time?

This may be tested by creating two interrupt sources that will fire at exactly the same clock cycle. The demo below tests this with the periodic interrupt timer.

In this scenario, two independent periodic timer channels are set to fire each at 0.1s intervals, with the same start time. Since both channels will trigger at the same time, the handling order is dictated by the channel priority, and both ISRs will occur back-to-back (appearing as one handling event from the perspective of the code being interrupted, similar to the SCI/Port J example above).

```
interrupt VectorNumber_Vpit0 void PIT0Int (void)
{
  // clear flag
  // note: 13.5.3 - must write to bit to clear, no read allowed! (no bset)
  PITTF = PITTF_PTF0_MASK;

  SWL_LED_TOG (SWL_GREEN);
}

interrupt VectorNumber_Vpit1 void PIT1Int (void)
{
  // clear flag
  // note: 13.5.3 - must write to bit to clear, no read allowed! (no bset)
  PITTF = PITTF_PTF1_MASK;

  SWL_LED_TOG (SWL_YELLOW);
}
```

`PIT0 (0x66)` has higher priority than `PIT1 (0x67)`, so the `PIT0` ISR would fire first, with an RTI into `PIT1` immediately after.

Pulse Width Modulator

Your micro contains a PWM module, as described in chapter 8 of Big Pink.

The purpose of the PWM module is to create continuous waveforms with programmable duty rates. There are eight 8-bit channels available for use (may be configured as four 16-bit channels). There are many clocking options which allow a wide range of frequencies.

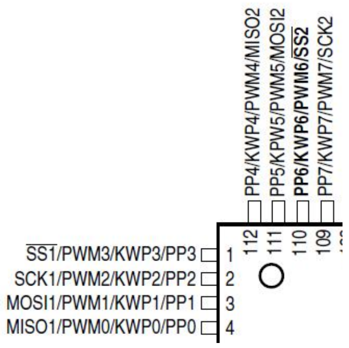
You will use PWM to create fixed frequencies suitable for audio tone generation, LCD backlight control, or as a simple square-wave generator. You may also use *actual modulation* with PWM to control servos, operate a Class D amplifier, or generate RGB colors with an RGB LED. This topic may be covered in other courses in terms three and four as well.

The module consists of eight channels, but effectively, they are internally organized as two groups of four channels, each group based on a dedicated clock.

Channels 0, 1, 4, and 5 are part of group A

Channels 2, 3, 6, and 7 are part of group B

Each group may also select between a scaled and unscaled clock source. This flexible clocking scheme allows for great control of the frequencies generated but makes operation a little more complex than other clock-based modules we have seen thus far. The fact that each group shares a clock source also limits how each group may be used.



Each PWM channel has an associated pin on the chip, and the PWM appears on Port P.

Note: pin assignments are a little weird for the PWM module.

The following physical connections have already been made for your board:

Channel	Group	Assignment
0	A	Blue LED in RGB LED
1	A	Green LED in RGB LED
2	B	[Not Assigned]
3	B	LCD Backlight
4	A	Red LED in RGB LED
5	A	[Not Assigned]
6	B	Speaker
7	B	[Not Assigned]

Clock choices for each group affect the entire group. For this reason, outputs with similar clocking needs have been placed within the same group.

Clock Generation

The PWM configures an A and B clock separately, and the A and B clocks may be scaled again to create an SA and SB clock. Each channel may independently choose between the unscaled or scaled clock, but will use the clock source for the group that it is in. This means that an A group channel may only choose the A or SA clock, and a B group channel may only choose the B or SB clock.

A and B Clocks

The `PWMPRCLK` registers allows for a 2^N prescale for each group independently, and clocks down from the bus rate. This register establishes the A and B clocks.

SA and SB Clocks

The `PWMSCLA` and `PWMSCLB` registers are used for subsequent clock division and generate the SA and SB clocks. These registers accept full 8-bit values. The rates for the SA and SB clocks are found as:

SA = A / (2 x `PWMSCLA`)

SB = B / (2 x `PWMSCLB`)

Note: If the `PWMSCL` value is zero, this is considered full scale, and will result in divide by 512.

Each PWM channel may independently select the clock source out of its group with the `PWMCLK` register (PWM Clock Select Register). Again, the A/SA, B/SB clock choice is determined by the channel group.

The waveform generated for each channel is based on values written to the `PWMPERx` (period) and `PWMDTYx` (duty) registers. These registers determine the period and duty of the waveform in counts of the selected clock.

Each channel additionally allows polarity selection via the `PWMPOL` register.

- If the polarity bit is a 1 for a channel, the output will start high, and then go low when the duty count is reached (duty counts high time).
- If the polarity bit is a 0 for a channel, the output will start low, and then go high when the duty count is reached (duty counts low time).

To have a high amount of control over the generated waveform, you will typically want to have the largest, even value possible for the period. This will permit a high degree of control for the duty, as the duty must be less than or equal to the period. For example, a 50% duty waveform needs to have a duty that is ½ the period. There are many values that would allow this. If you wanted a 2% duty waveform, however, the duty would need to be 1/50th of the period. To support this level of flexibility, you should always shoot for high period values. This will influence the clock rate, as the clock rate will need to increase proportionally.

`Case A`
Create a 1KHz 50% duty square wave on channel 2.

Following our best practices, let's assume a high value for period (200), and to generate a 50% duty cycle, we need ½ of that value for the duty (100).

This means that 200 PWM clock counts need to generate a 1KHz waveform. The period of a 1KHz waveform is 1ms, so 200 clock counts must equal 1ms, so 1 clock count needs to be 5µs.

The bus period is 50ns (20MHz), so the total amount of scaling that needs to be achieved is 5µs / 50ns, or 100. Channel 2 is in the B group, so we need to come up with clock B and clock SB values that total division by 100.

Remember that clock B is a power of two divisor, and SB is any 8-bit value that is multiplied by 2. This leaves more than one combination that will equal 100:

Clock B (2^N)	Clock SB (2 x N)	Total Division
2^0 = 1	N = 50, (2 x 50) = 100	1 x 100 = 100
2^1 = 2	N = 25, (2 x 25) = 50	2 x 50 = 100

You may use any combination of values to create a suitable SB clock for channel 2.

Once you know these values, you need to:

- Program the clocks for the appropriate group
- Select the appropriate clock source for the channel
- Set the desired polarity for the desired channel
- Program the period and duty for the channel
- Enable the channel

We could use the following code to achieve the above (see waveform and measurements in appendix):

```
PLL_To20MHz();

// set Clock B to divide by 2 (8.3.2.4)
PWMPRCLK &= 0b10001111;
PWMPRCLK |= 0b00010000; // 2^1 = 2 (upper nibble for B)

// set Clock SB to divide by 50 (8.3.2.10)
PWMSCLB = 25; // remember: it does x 2

// select Clock SB for channel 2 (8.3.2.3)
PWMCLK_PCLK2 = 1; // 1 means use clock SB

// set polarity as start high, go low (does not matter here) (8.3.2.2)
PWMPOL_PPOL2 = 1;

// program period (8.3.2.13)
PWMPER2 = 200;

// program duty (8.3.2.14)
PWMDTY2 = 100; // half period, so 50% duty

// enable channel (8.3.2.1)
PWME_PWME2 = 1;
```

`Case B (assumes you understand Case A)`
Create a waveform with a 20µs period, and 19µs 'On' time, on pin 111.

This is a 95% positive duty waveform, with pol = 0, at 1/20µs = 50KHz, on channel 5.

Being this fast, it may require us to use a period of 20, and a duty of 1. The micro is not always fast enough at the default bus rate to generate higher frequencies at periods of 200 counts.

20 PWM clock periods must equal 20µs, so 1 clock period is 1µs. To clock the bus rate down to 1µs, we need 1µs / 50ns, or divide by 20. NOTE: for 200 clock periods to equal 20us, we would need a clock of 100ns. This is difficult to achieve, but a period of 100 clocks would be possible.

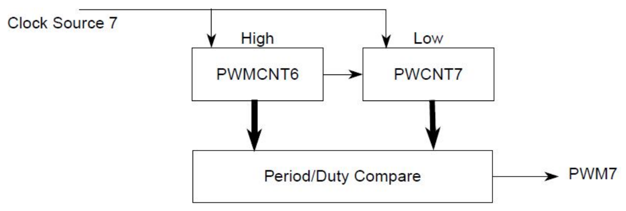
Once again, there are several ways to get a total division factor of 20, but div by 1 then div by 20 is obvious.

We could use the following code to achieve the above (see waveform and measurements in appendix):

```
PLL_To20MHz();

// set Clock A to divide by 1 (8.3.2.4)
PWMPRCLK &= 0b11111000; // 2^0 = 1 (lower nibble for A)

// set Clock SA to divide by 20 (8.3.2.10)
PWMSCLA = 10; // remember: it does x 2

// select Clock SA for channel 5 (8.3.2.3)
PWMCLK_PCLK5 = 1; // 1 means use clock SA

// set polarity as start low, go high (8.3.2.2)
PWMPOL_PPOL5 = 0;

// program period (8.3.2.13)
PWMPER5 = 20;

// program duty (8.3.2.14)
PWMDTY5 = 1; // 1/20th period, 5% duty

// enable channel (8.3.2.1)
PWME_PWME5 = 1;
```

16-bit PWM (8.4.2.7)

The PWM module permits concatenation of two 8-bit channels into a 16-bit channel. Each adjacent pair of 8-bit channels can do this. The numerically higher of the pair is the channel that selects the channel attributes and represents the physical pin that the signal appears on. In Big Pink, the higher channel becomes the least significant byte for 16-bit registers, so this is a little confusing, but consistent with the architecture. Both channels are used for period and duty values, where the high byte is stored in the lower channel.



Concatenate Channels 6 and 7

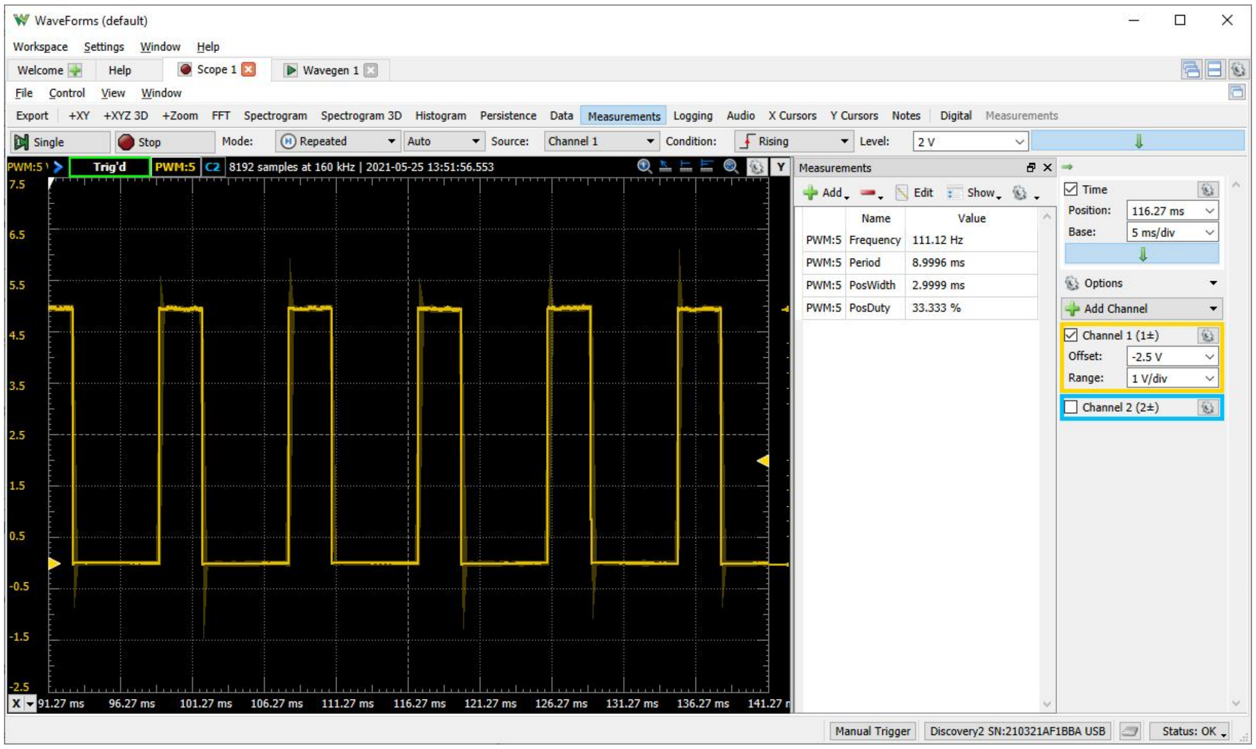
0 Channels 6 and 7 are separate 8-bit PWMs.

1 Channels 6 and 7 are concatenated to create one 16-bit PWM channel. Channel 6 becomes the high order byte and channel 7 becomes the low order byte. Channel 7 output pin is used as the output for this 16-bit PWM (bit 7 of port PWMP). Channel 7 clock select control-bit determines the clock source, channel 7 polarity bit determines the polarity, channel 7 enable bit enables the output and channel 7 center aligned enable bit determines the output mode.

For example, you could configure channel 7 as a 16-bit channel (sacrificing channel 6):

```
PLL_To20MHz();

// channel 7 is a B channel!
// set Clock B to divide by 2 (8.3.2.4)
PWMPRCLK &= 0b10001111;
PWMPRCLK |= 0b00010000; // 2^1 = 2 (upper nibble for B)

// set Clock SB to divide by 2 (8.3.2.10)
PWMSCLB = 1; // remember: it does x 2
// must use fast clocks if PER is high (lots of counts)

// select Clock SB for channel 2 (8.3.2.3)
PWMCLK_PCLK7 = 1; // 1 means use clock SB

// set polarity as start high, go low (does not matter here) (8.3.2.2)
PWMPOL_PPOL7 = 1;

// program period (8.3.2.13) 45000 = 0xAFC8, msb goes first
//PWMPER6 = 0xAF;
//PWMPER7 = 0xC8;
//or
PWMPER67 = 45000; // nice, support file has 16-bit registers defined

// program duty (8.3.2.14) 15000 = 0x3A98, msb first
//PWMDTY6 = 0x3A; // 1/3 period, so 33.3% duty
//PWMDTY7 = 0x98;
//or
PWMDTY67 = 45000 / 3; // 16-bit values provide *much* higher control over period and duty ratio than 8-bit

// set as a 16-bit channel (the actual 16-bit concatenation)
PWMCTL_CON67 = 1;

// enable channel (8.3.2.1)
PWME_PWME7 = 1;
```

Using the PWM in 16-bit mode affords much higher control over the period and duty times. The example above shows how a 1/3 duty could be implemented accurately, even with an arbitrary period.


WaveForms (default)
Workspace Settings Window Help
Welcome Help Scope 1 Wavegen 1
File Control View Window
Export +XY +XYZ 3D +Zoom FFT Spectrogram Spectrogram 3D Histogram Persistence Data Measurements Logging Audio X Cursors Y Cursors Notes Digital Measurements
Single Stop
Trig'd PWM:5 C2
7.5 6.5 5.5 4.5 3.5 2.5 1.5 0.5 -0.5 -1.5 -2.5
91.27 ms 96.27 ms 101.27 ms 106.27 ms 111.27 ms 116.27 ms 121.27 ms 126.27 ms 131.27 ms 136.27 ms 141.27
Add Edit Show
Options
Add Channel
Manual Trigger

Mode: Repeated, Auto | Source: Channel 1 | Condition: Rising | Level: 2 V

8192 samples at 160 kHz | 2021-05-25 13:51:56.553

Measurements

	Name	Value
PWM:5	Frequency	111.12 Hz
PWM:5	Period	8.9996 ms
PWM:5	PosWidth	2.9999 ms
PWM:5	PosDuty	33.333 %

Time
Position: 116.27 ms
Base: 5 ms/div

Channel 1 (1±)
Offset: -2.5 V
Range: 1 V/div

Channel 2 (2±)

Discovery2 SN:210321AF1BBA USB | Status: OK

The benefits of using 16-bit mode will become readily apparent if operating a servo motor; 8-bits (1/255) may not be enough to get the resolution you need for full range and control.

Building a Library

You should create a new library for your PWM functions.

NOTE: the files `pwm.h` and `pwm.c` already exist in the scope of the complier, so make sure you don't use conflicting names!

It is OK if your initialization function(s) destroy(s) previous clock values for channels in the same group.

You may have separate groups of functions to support 8-bit and 16-bit operations.

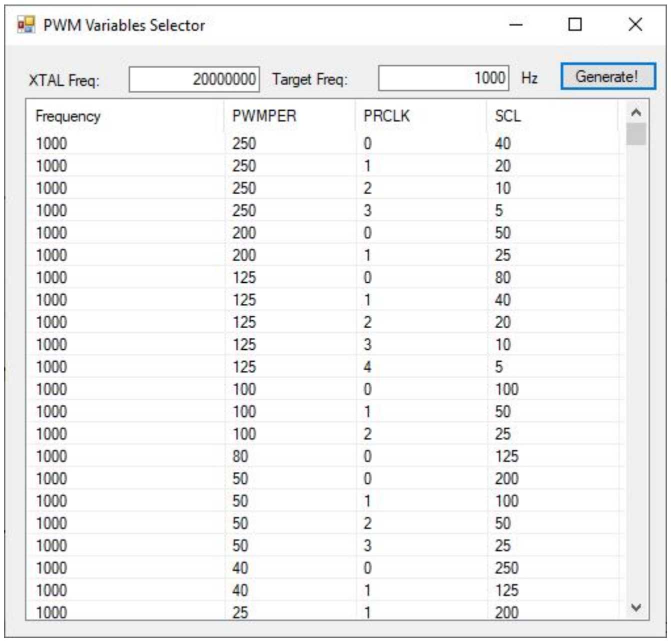
You will not be directed in library specifics. It is up to you to decide what functions will be of use, keeping in mind that you will be expected to use the PWM module for a full range of solutions.

Quality implementation now will save you trouble and coding during the practical examinations!

You should run your library plan by your instructor to get some feedback before you commit.

Helper Programs

You are permitted to write helper programs to assist in rapid calculation of PWM configuration values. Creating a program to calculate best-fit values for a particular frequency, for example, would be allowed during practical exams, if, and only if, you are demonstrably the author of the program.

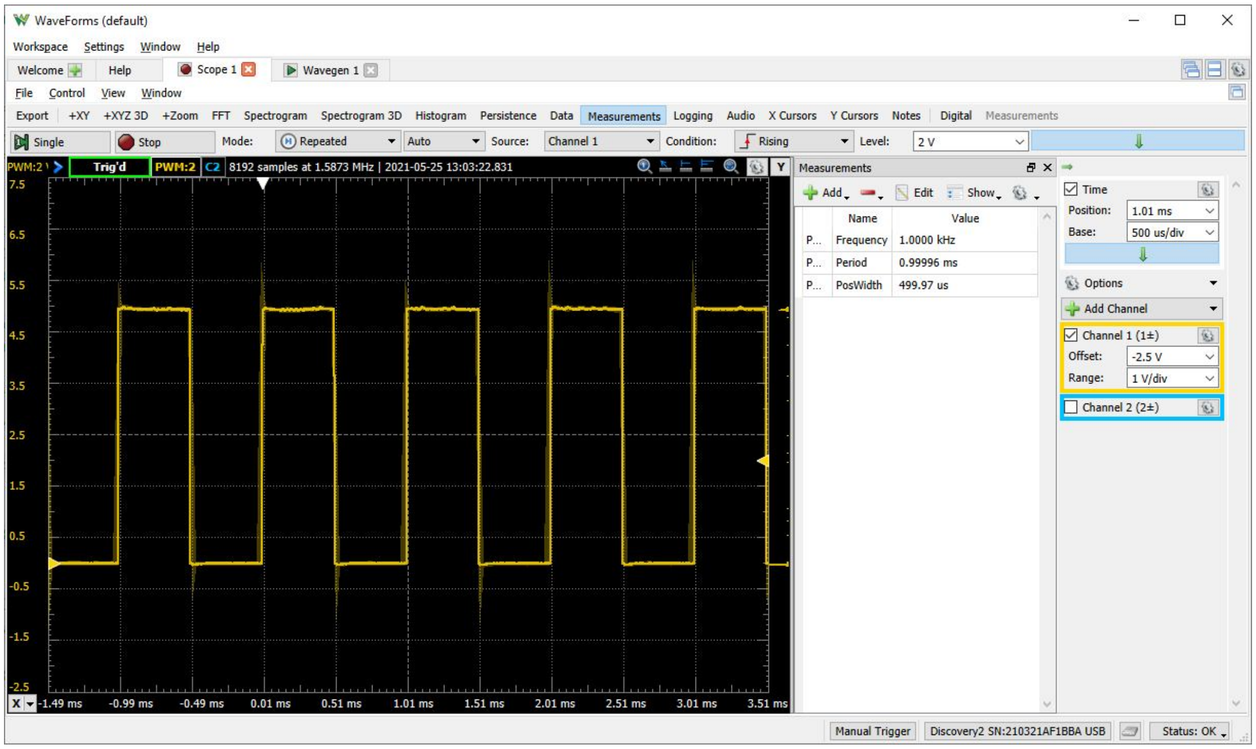


XTAL Freq: 20000000 Target Freq: 1000 Hz

Frequency	PWMPER	PRCLK	SCL
1000	250	0	40
1000	250	1	20
1000	250	2	10
1000	250	3	5
1000	200	0	50
1000	200	1	25
1000	125	0	80
1000	125	1	40
1000	125	2	20
1000	125	3	10
1000	125	4	5
1000	100	0	100
1000	100	1	50
1000	100	2	25
1000	80	0	125
1000	50	0	200
1000	50	1	100
1000	50	2	50
1000	50	3	25
1000	40	0	250
1000	40	1	125
1000	25	1	200

Appendix

'Case A' Waveform and measurements:


WaveForms (default)
Workspace Settings Window Help
Welcome Help Scope 1 Wavegen 1
File Control View Window
Export +XY +XYZ 3D +Zoom FFT Spectrogram Spectrogram 3D Histogram Persistence Data Measurements Logging Audio X Cursors Y Cursors Notes Digital Measurements
Single Stop
Trig'd PWM:2 C2
7.5 6.5 5.5 4.5 3.5 2.5 1.5 0.5 -0.5 -1.5 -2.5
-1.49 ms -0.99 ms -0.49 ms 0.01 ms 0.51 ms 1.01 ms 1.51 ms 2.01 ms 2.51 ms 3.01 ms 3.51 ms
Add Edit Show
Options
Add Channel
Manual Trigger
Discovery2 SN:210321AF1BBA USB
Status: OK

8192 samples at 1.5873 MHz | 2021-05-25 13:03:22.831

Mode: Repeated | Auto | Source: Channel 1 | Condition: Rising | Level: 2 V

Measurements

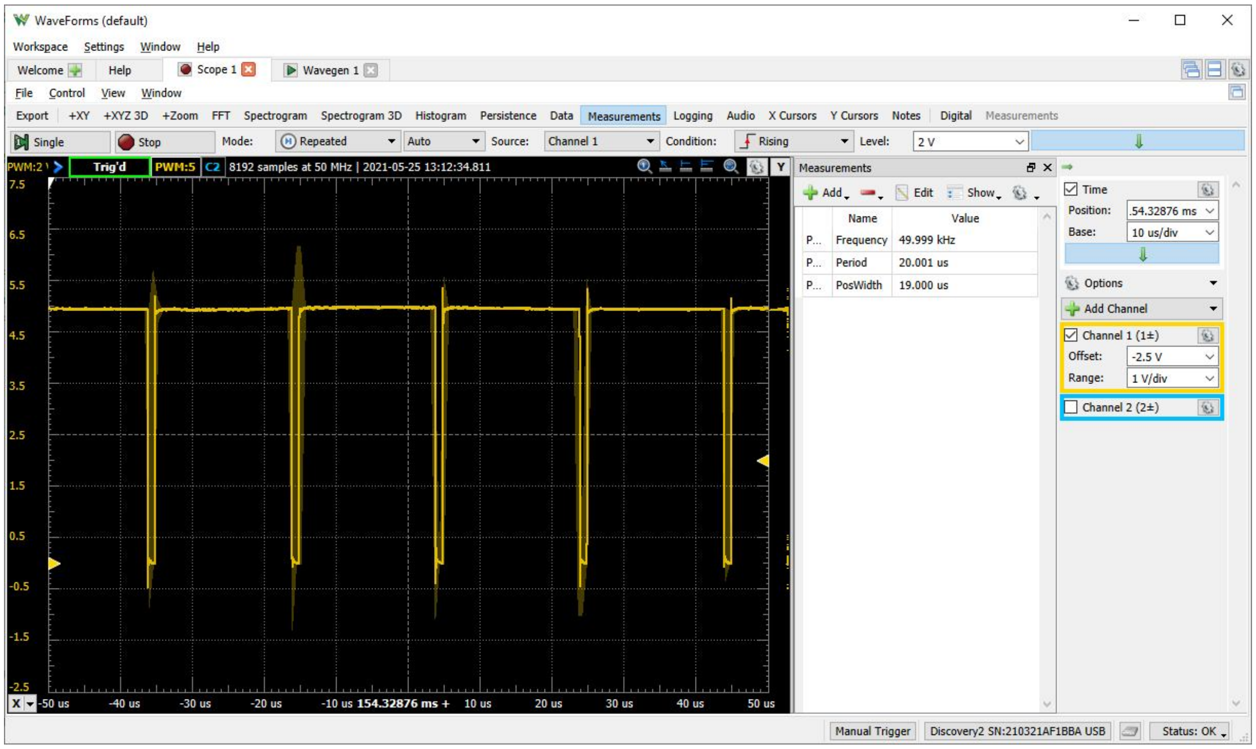
	Name	Value
P...	Frequency	1.0000 kHz
P...	Period	0.99996 ms
P...	PosWidth	499.97 us

Time — Position: 1.01 ms; Base: 500 us/div

Channel 1 (1±) — Offset: -2.5 V; Range: 1 V/div

Channel 2 (2±)

'Case B' Waveform and measurements:


WaveForms (default)
Workspace Settings Window Help
Welcome Help Scope 1 Wavegen 1
File Control View Window
Export +XY +XYZ 3D +Zoom FFT Spectrogram Spectrogram 3D Histogram Persistence Data Measurements Logging Audio X Cursors Y Cursors Notes Digital Measurements
Single Stop
Trig'd PWM:5 C2
7.5 6.5 5.5 4.5 3.5 2.5 1.5 0.5 -0.5 -1.5 -2.5
-50 us -40 us -30 us -20 us -10 us 154.32876 ms + 10 us 20 us 30 us 40 us 50 us
Add Edit Show
Options
Add Channel
Manual Trigger
Discovery2 SN:210321AF1BBA USB
Status: OK

8192 samples at 50 MHz | 2021-05-25 13:12:34.811

Mode: Repeated | Auto | Source: Channel 1 | Condition: Rising | Level: 2 V

Measurements

	Name	Value
P...	Frequency	49.999 kHz
P...	Period	20.001 us
P...	PosWidth	19.000 us

Time — Position: 154.32876 ms; Base: 10 us/div

Channel 1 (1±) — Offset: -2.5 V; Range: 1 V/div

Channel 2 (2±)

PWM Practice Problems

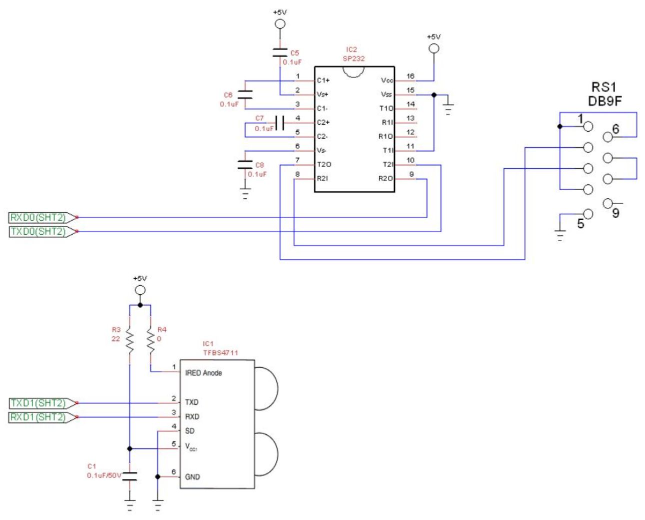
1. What is the longest period you may achieve with the PWM when using a bus rate of 20MHz? Show your work.
2. Show how you would configure channel 7 for a 20ms, 25% negative duty output.
3. Configure the speaker to produce a 1kHz tone.
4. Configure the LCD backlight for perceived ‘half brightness’. Ensure you use a frequency that is fast enough to not be noticeable by humans. Does the LCD appear to be at half brightness when the duty is 50%, or does this happen at a different duty?
5. Configure the RGB LED to produce a random color each time the center switch transitions.

The Serial Communications Interface (SCI)

Your 9S12 micro contains SCI modules for asynchronous serial communications. You will use one of the SCI modules to communicate with the PC, although the SCI module may communicate with any other compatible UART (Universal asynchronous receiver-transmitter). If you are directly interfacing to another UART, you will likely not use RS-232, but TTL levels instead.

In order to connect to a PC the TTL-level signals (0-5V) from the SCI must be converted to RS-232 levels (~±10V). Your board contains a chip dedicated to this task, and the signals are brought to a standard RS-232 DB9 connector on the top edge of your board. RS-232 permits much longer distances between devices and has *some* resistance to signal interference. NOTE: The use of RS-232 does not change the timing of the UART signal, it just uses a different signaling scheme.

Your board has an IrDA transceiver as well on `SCI1`. We may not get to use this port in this course, but IrDA is good for ~1m and uses infrared light as the physical layer. Using infrared light provides *extreme* electrical isolation between the two communicating devices.



While RS-232 supports additional signaling options, we will be using only three wires: ground, transmit data, and receive data.

In asynchronous communications, the transmitter may begin a data send operation to the receiver at any time. Once started, a complete block of data (known as a data character) must be completely transmitted. The delay between data characters may be any length. Transmission of the individual bits in the data character is driven by a local clock. The transmitter and receiver must operate independent clocks that are approximately equal in rate to correctly exchange data. The term 'asynchronous' refers to the fact that the clocks on the two devices are independent (there is no synchronizing clock signal), and communication can be initiated at any time.

Because we are reading and writing bytes in serial communications, the SCI module acts as a parallel-to-serial and serial-to-parallel converter.

The RS-232 protocol allows for a wide range of signaling characteristics. Here are a few:

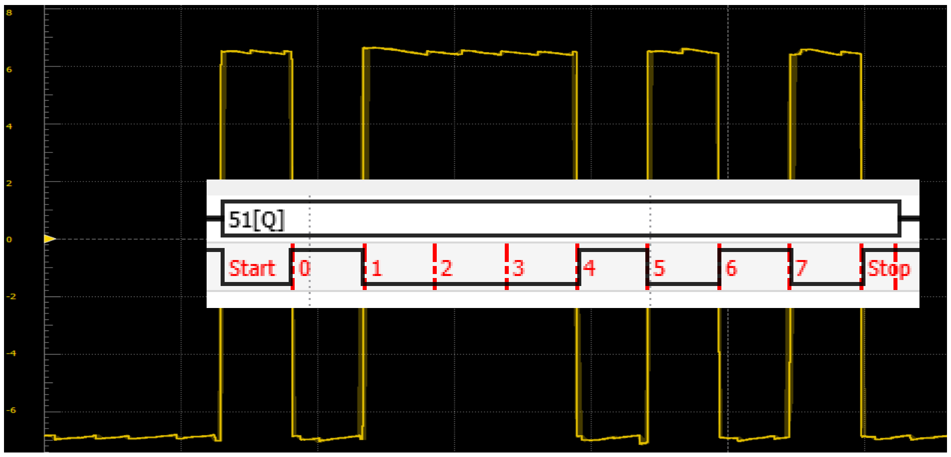
- 'Standard' transmission rates from 75-115200 BAUD* (these are specified speeds only, like 9600, not just any value in that range).
- Data may be sent as 7-bit standard ASCII characters, 8-bit extended ASCII characters, or binary data (note: this is interpretation, data is data).
- Simple error checking via parity is possible.
- The minimum time between characters (stop bits) may be adjusted.
- Handshaking and flow control are possible.

The SCI on our board also supports a 9-bit mode, but we won't use this, as it is non-standard, and is more complex to operate.

We will use the very common `8N1` configuration. This means one start bit, eight bits of data, no parity, and one stop bit. The start bit signals the start of communications. The eight bits that follow are the data payload, and the stop bit is used to set a minimum time between characters. The stop bit was once used to ensure that the receiver had time to process the received character, but this is not normally a consideration for modern equipment.

*BAUD is a pseudo acronym that effectively means "Bits per Second". You may find references that describe it as "Bits of Actual Usable Data", but this is confusing, as the start bit, stop bit(s), and optional parity bit are not part of the data payload. The actual signal is 8 of 10 bits transmitted in the 8N1 format.

Note: when converted to RS-232, the UART TTL signal is inverted, bipolar, and non-return-to-zero!



The yellow trace in the background is the RS-232 signal viewed on a scope (±7V).

The image imposed on top is from the protocol analyzer in the AD2 (0-5V).

Data is sent least significant bit first, so the data appears 'backwards' from how we normally view it.

When the signal is idle, the RS-232 level is negative with respect to ground. This is known as 'MARK'.

When the signal is active, the RS-232 level is positive with respect to ground. This is known as 'SPACE'.

By not using common (ground) for any valid signal (non-return-to-zero), RS-232 is easier to troubleshoot: if a signal is at ground, it is not connected correctly!

Setting the BAUD rate

The SCI module uses a clock 16 times the BAUD rate for sampling. To reserve clock for this, the bus rate is inherently divided by 16, then by the 13-bit `SCIBD` register. For example, if you wanted a BAUD rate of 9600, you would divide `20E6 / 16 / 9600 = 130.20833`. You can't put a fraction into the integral register, but 130 is 99.84% of ideal.

For some rates, the denominator can get pretty big, which is really bad in integer division. For example, with the fastest 'standard' rate of `115200`, we run into a problem:

`20E6 / 16 / 115200 = 10` in the integer realm, but that value was truncated from `10.85`, so a value of `11` would provide a more accurate BAUD rate. You can still use integer division to calculate the value for the SCIBD register, you just need to provide rounding that will push the value up the next whole number if truncation would lose a fraction over `0.5`. This is achieved by multiplying the operands by `10`, adding `5` to the result, then dividing the overall result by `10`. Doing this adds a 'half' to what would be the digit to the right of the decimal point. When truncation occurs, if the original value was less than `5` then there is no change; if the value was `5` or more, the added `5` 'rounds' it to the next digit prior to truncation.

Consider the `115200` BAUD example from above (all calculated with integer types):

`20E6 / 16 / 115200 = 10` (original integer value)

`20E6 * 10 / 16 = 12500000`

`12500000 / 115200 = 108`

`108 + 5 = 113`

`113 / 10 = 11` (with rounding implemented)

Using this method will provide more accurate BAUD rate values.

As it turns out, that oversampling permits some tolerance in the BAUD rate, and if the actual rate is within ~2% of ideal, the communications should function.

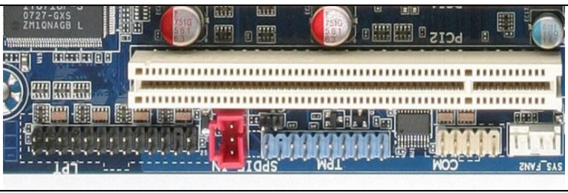
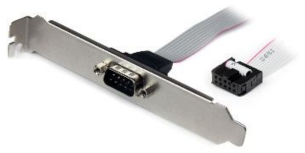
Remember that whatever value you put in the BAUD register must be limited to 13 bits, so the max divisor is `8191`. Using a divisor of zero will disable the BAUD generator.

Most communications programs will expect one of several fixed BAUD rates. The following is a list of 'standard' BAUD rates, where ones in red are particularly common:

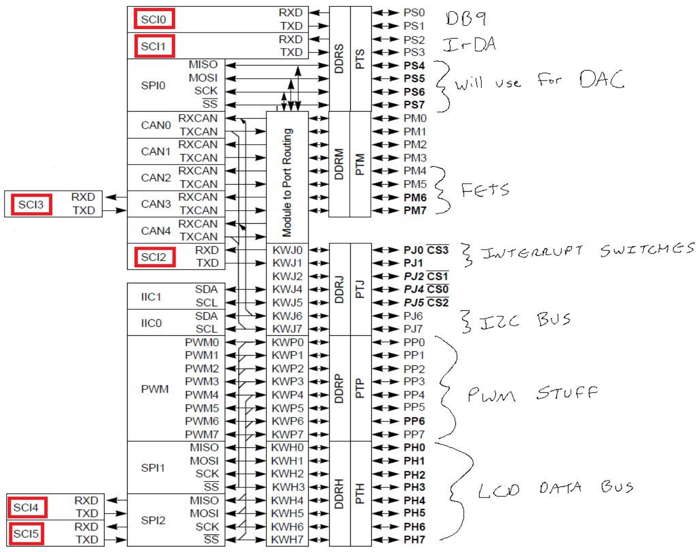
75, 110, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200.

If your computer is not equipped with an actual serial port, you may need to use a USB to RS-232 adaptor cable. Some of these cables will not work with all rates (generally the very low ones).

If you are using a PC, there is generally a serial port available, but it might only be pinned out on the motherboard, and you would need to procure a cable to bring it to a connector on one of the expansion slots. Laptops, generally, do not have serial ports available – you must use a converter cable from USB.

There are several SCI modules on our chip, and the port you are using needs to be distinguished when using the registers:



The naming convention is to use the module number in the register name:

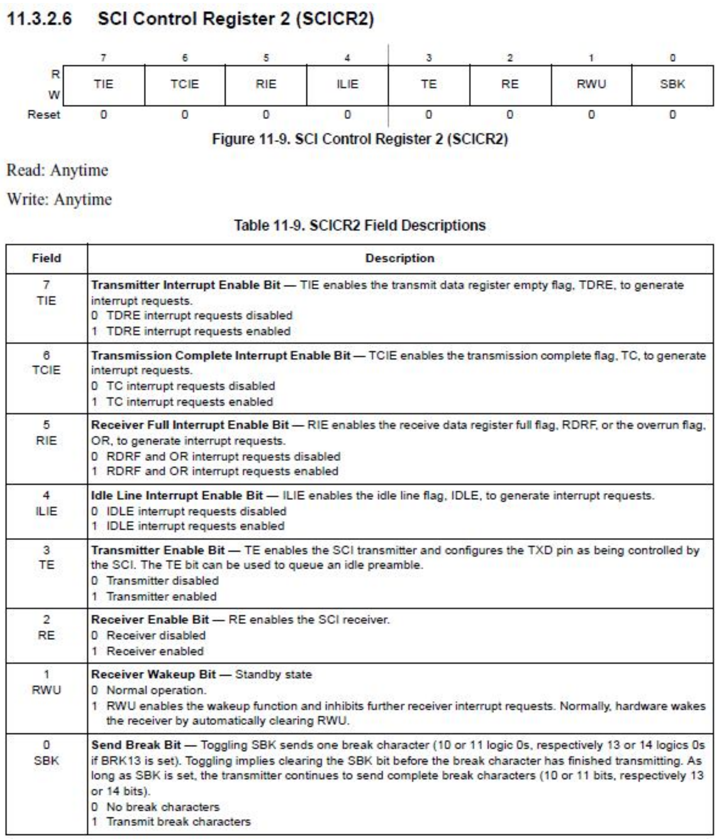
```
SCI0BD = 130;
```

`SCI0` is the SCI module that is connected to the DB9 on your board. `SCI1` is the SCI module that is connected to the IrDA module (infrared data). The remaining SCI ports should not be used, as these pins are allocated for other devices or modules (by the design of the board).

Once the BAUD rate is set (both devices must be set to the same BAUD rate), you must enable the SCI. Like other modules, this unit is powered off out of reset.

The `SCICR2` register is used to configure parts of the SCI. You are looking to turn on the transmitter and receiver, so the `TE` and `RE` bits of `SCICR2` need to be set to 1:

```
SCI0CR2 = 0b00001100;
```



11.3.2.6 SCI Control Register 2 (SCICR2)

	7	6	5	4	3	2	1	0
R/W	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
Reset	0	0	0	0	0	0	0	0

Figure 11-9. SCI Control Register 2 (SCICR2)

Read: Anytime

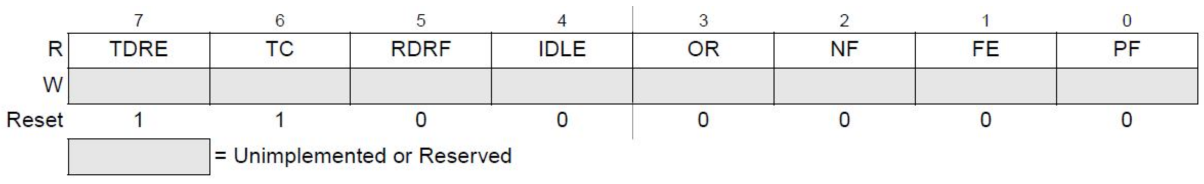
Write: Anytime

Table 11-9. SCICR2 Field Descriptions

Field	Description
7 TIE	**Transmitter Interrupt Enable Bit** — TIE enables the transmit data register empty flag, TDRE, to generate interrupt requests. 0 TDRE interrupt requests disabled 1 TDRE interrupt requests enabled
6 TCIE	**Transmission Complete Interrupt Enable Bit** — TCIE enables the transmission complete flag, TC, to generate interrupt requests. 0 TC interrupt requests disabled 1 TC interrupt requests enabled
5 RIE	**Receiver Full Interrupt Enable Bit** — RIE enables the receive data register full flag, RDRF, or the overrun flag, OR, to generate interrupt requests. 0 RDRF and OR interrupt requests disabled 1 RDRF and OR interrupt requests enabled
4 ILIE	**Idle Line Interrupt Enable Bit** — ILIE enables the idle line flag, IDLE, to generate interrupt requests. 0 IDLE interrupt requests disabled 1 IDLE interrupt requests enabled
3 TE	**Transmitter Enable Bit** — TE enables the SCI transmitter and configures the TXD pin as being controlled by the SCI. The TE bit can be used to queue an idle preamble. 0 Transmitter disabled 1 Transmitter enabled
2 RE	**Receiver Enable Bit** — RE enables the SCI receiver. 0 Receiver disabled 1 Receiver enabled
1 RWU	**Receiver Wakeup Bit** — Standby state 0 Normal operation. 1 RWU enables the wakeup function and inhibits further receiver interrupt requests. Normally, hardware wakes the receiver by automatically clearing RWU.
0 SBK	**Send Break Bit** — Toggling SBK sends one break character (10 or 11 logic 0s, respectively 13 or 14 logics 0s if BRK13 is set). Toggling implies clearing the SBK bit before the break character has finished transmitting. As long as SBK is set, the transmitter continues to send complete break characters (10 or 11 bits, respectively 13 or 14 bits). 0 No break characters 1 Transmit break characters

NOTE: The SCI can generate interrupts for four different conditions. Eventually, we will use interrupts to manage processing of received data. Initially, you will poll for received data, to get a basic idea about how data is sent and received.

Much of the status of the SCI module is revealed through the `SCISR1` register:



	7	6	5	4	3	2	1	0
R	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF
W	▨	▨	▨	▨	▨	▨	▨	▨
Reset	1	1	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 11-10. SCI Status Register 1 (SCISR1)

The two flags that are of principal interest are `TDRE` and `RDRF`. If `TDRE` is a 1 when the module is ready to accept a byte for transmission. You will check to ensure this flag is set before you attempt to send a byte.

The `RDRF` flag indicates that a new byte has been received. Initially you will use polling to check this bit to see if a byte has arrived for processing. If the flag is set, reading the flag then reading the byte from `SCIDRL` will clear it, making the module ready for another byte reception.

The following code demonstrates how to configure `SCI0` for 9600 BAUD communication. It will attempt to send out bytes as fast as possible, and display ASCII codes on the segs if a byte is received:

```
void main(void)
{
  // main entry point
  _DISABLE_COP();
  EnableInterrupts;

  /********************************************************************/
  // initializations
  /********************************************************************/
  PLL_To20MHz();
  SWL_Init();
  Segs_Init();

  // do SCI startups
  SCI0BD = 130; // 20E6 / (9600 * 16) // 11.3.2.1

  SCI0CR2 = 0b00001100; // turn on TX/RX // 11.3.2.6

  /********************************************************************/
  // main program loop
  /********************************************************************/
  for (;;)
  {
    // if the transmitter buffer is empty, load a new byte to send (TX)
    if (SCI0SR1_TDRE /*&& SWL_Transition (SWL_CTR)*/)
      SCI0DRL = rand() % 26 + 'A';

    // if a byte has been received, pull it!
    if (SCI0SR1_RDRF)
      Segs_8H (2, SCI0DRL);
  }
}
```

The main loop will run very quickly, as the micro is very fast relative to 9600 BAUD. In fact, a fun activity to try would be to figure out how many iterations the main loop runs for each character transmission (maybe add that code and put it on the second segs line).

Seeing the characters sent and received may be done at the UART level with the AD2, and at RS-232 levels with a terminal program.

Both the AD2 and terminal programs will allow you to send and receive characters, but you may also write programs in C# that use serial communications.

You will need to build up a library for the SCI that is capable of initializing the SCI, blocking send, non-blocking send, blocking receive, non-blocking receive, and possibly interrupt management:

```
// sci 0 - normal mode ********************************
// set baud, returns actual baud
unsigned long sci0_Init (unsigned long ulBusClock, unsigned long ulBaudRate);

// read a byte, non-blocking
// returns 1 if byte read, 0 if not
int sci0_read (unsigned char * pData);

// blocking byte read
// waits for a byte to arrive and returns it
unsigned char sci0_bread (void);

// send a byte over SCI (blocking)
void sci0_txByte (unsigned char data);

// send a null-terminated string over SCI
void sci0_txStr (char const * straddr);

// receive a string from the SCI
// up to buffer size-1 (string always NULL terminated)
// number of characters is BufferSize minus one for null
// once user enters the max characters, null terminate and return
// if user enters 'enter ('\r')' before-hand, return with current entry (null terminated)
// echo valid characters (non-enter) back to the terminal
// return -1 on any error, otherwise string length
int sci0_rxStr (char * const pTarget, int BufferSize);

// set/clear interrupt flags for SCI0
void sci0_SetIntFlag (unsigned char flags);
void sci0_ClrIntFlag (unsigned char flags);
// sci 0 - normal mode ********************************
```

These topics will be covered in class with demonstrations.

Using SCI Interrupts

The most useful interrupt for the SCI is arguably the *Receiver Full Interrupt*. This interrupt will occur when a byte has been received and is ready to be pulled from the receive buffer. Because we typically don't know when a byte will arrive, and one could *never* arrive, this interrupt can save a lot of polling. Additionally, if we want to use the micro for CPU intensive work while performing communications, particularly higher-speed communications, the interrupt model can make the code more responsive, less prone to missing data, and more efficient.

The SCI module routes all interrupt events to a single ISR, so if you request more than one interrupt source, you must determine the source of the interrupt in the ISR by flag checking. If you have only one interrupt cause enabled, you may skip this step.

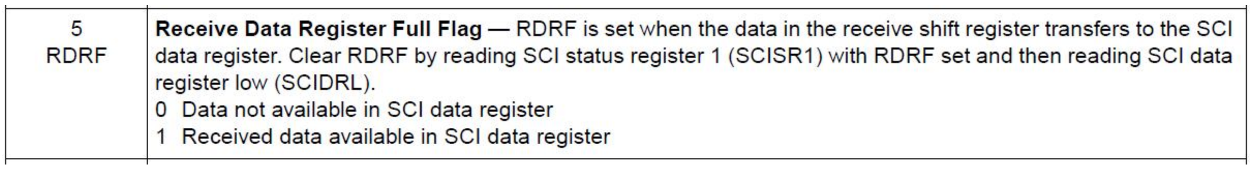
Interrupts for the SCI are managed through the `SCIxCR2` register, as shown above. To enable an interrupt when data is fully received, you would set `RIE` to `1`.

```
// setup interrupt for RDRF
SCI0CR2_RIE = 1;
```

In doing so, you have committed yourself to dealing with this interrupt. You will need a suitable ISR:

```
interrupt VectorNumber_Vsci0 void ISR_SCI0 (void)
{
    // single read to capture flags
    unsigned char status = SCI0SR1;

    // if you've done more than one interrupt on this device
    //  you need to identify the interrupts, otherwise, clear the
    //  one and only one you asked for...

    // TDRE: cleared by reading SCI0SR1 w/B7 set, then write to SCI0DRL
    // RDRF: cleared by reading SCI0SR1 w/B5 set, then reading from SCI0DRL

    // check SCI0SR1 for RDRF, this does the int clearing operation (use lib function)
    if (status & SCI0SR1_RDRF_MASK)
    {
      // retrieve byte by reading from SCI0DRL (use library method)
    }

    // other flags may still be set (if requested), so continue checking other int sources
    if (status & SCI0SR1_TC_MASK)
    {
      // send next byte by writing to SCI0DRL (use library method)
    }
}
```

Normally you will only have `RIE` enabled, so the ISR is simpler than shown above. Section 11.3.2.7 in Big Pink discusses flag clearing for each interrupt flag in the `SCIxSR1`:



5 RDRF	**Receive Data Register Full Flag** — RDRF is set when the data in the receive shift register transfers to the SCI data register. Clear RDRF by reading SCI status register 1 (SCISR1) with RDRF set and then reading SCI data register low (SCIDRL). 0 Data not available in SCI data register 1 Received data available in SCI data register

For `RIE` and any other interrupts you end up using, you must adhere strictly to the flag clearing mechanism. Your library functions should be written to automatically clear the flags through normal behavior.

The following program, for example, will display received characters on the segs using interrupts:

```
void main(void)
{
  // main entry point
  _DISABLE_COP();
  EnableInterrupts;

  PLL_To20MHz();
  Segs_Init();

  // start SCI at 38400
  (void)sci0_Init(20E6, 38400);

  // setup interrupt for RDRF
  SCI0CR2_RIE = 1;

  for (;;)
  {
    asm wai;
  }
}

interrupt VectorNumber_Vsci0 void ISR_SCI0 (void)
{
  // only one source of interrupt! RIE (RDRF), no need to check flags!
  // your blocking read will test RDRF and read data
  // flag is cleared via bread function!
  unsigned char data = sci0_bread();
  Segs_8H (0, data);
}
```

In the case of a single interrupt source, the ISR is much simpler to write! Most of the time this is what you will be doing, but you should be prepared for more complex operation of the SCI.

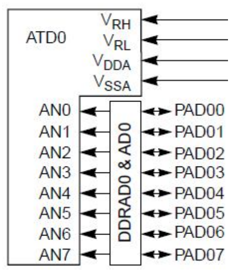
How you handle the data in the ISR is worthy of note. Remember that you don't want the ISR to be long in execution, as it will suspend subsequent interrupts. At high data rates, this could mean the loss of data. Ideally the ISR will simply put the received character into a buffer (queue or similar) for processing, and the main code would deal with it, but this is beyond the scope of this course.

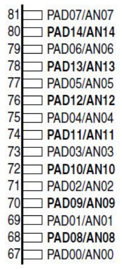
You will instead store the received data in a location visible to the main program code and will use a flag to indicate availability. As a result, the data processing work you will do will be relatively simple.

Using the Analog to Digital Module

There are two A/D blocks on your micro! We will be focusing on the one described in Chapter 5 of Big Pink (S12ATD10B8CV2). There are only minor differences between these modules (the one we use is 8-channel, the other is 16).

The A/D we are using is module `ATD0` and occupies port `PAD0`.





The ATD block contains independent power supply pins and a voltage reference pins. Your board uses a precision voltage reference to provide 5.12V for the reference voltage.

Since this is a 10-Bit A/D, there are 2^10, or 1024 steps it may report. Using a 5.12V reference means 5.12V/1024 at max reading = `5mV`/step.

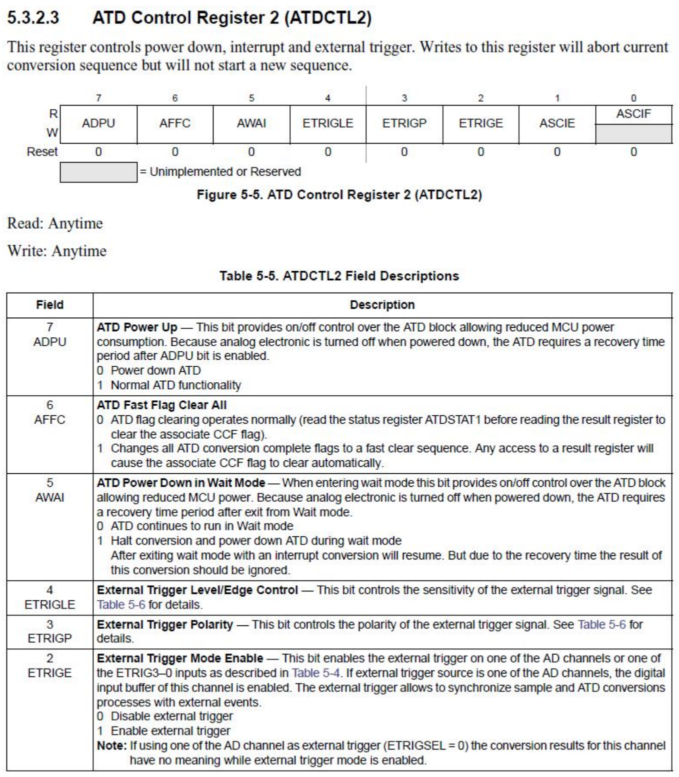
You will configure the A/D to report single quadrant (unsigned) 10-bit values. You may then multiply the A/D value by `5mV` to obtain the voltage present on the associated pin at the time of the sample.

The A/D module is very versatile, so we will make a few assumptions and choose an initialization path for the module that will generally serve our needs. If necessary, you may build functionality into your library that will use other modes of operation, where suitable.

The A/D module can be configured to cause interrupts when a conversion is complete, so we can make use of this feature to avoid polling once again, although polling is certainly possible. Given the very short conversion time for a sample, and the slow clock rate for the micro, polling actually has advantages over interrupts in many cases.

Initialization

The first register to be set in the initialization routine will be the `ATD0CTL2` register:



5.3.2.3 ATD Control Register 2 (ATDCTL2)

This register controls power down, interrupt and external trigger. Writes to this register will abort current conversion sequence but will not start a new sequence.

	7	6	5	4	3	2	1	0
R	ADPU	AFFC	AWAI	ETRIGLE	ETRIGP	ETRIGE	ASCIE	ASCIF
W								▨
Reset	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

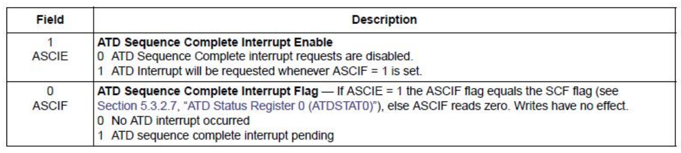
Figure 5-5. ATD Control Register 2 (ATDCTL2)

Read: Anytime

Write: Anytime

Table 5-5. ATDCTL2 Field Descriptions

Field	Description
7 ADPU	**ATD Power Up** — This bit provides on/off control over the ATD block allowing reduced MCU power consumption. Because analog electronic is turned off when powered down, the ATD requires a recovery time period after ADPU bit is enabled. 0 Power down ATD 1 Normal ATD functionality
6 AFFC	**ATD Fast Flag Clear All** 0 ATD flag clearing operates normally (read the status register ATDSTAT1 before reading the result register to clear the associate CCF flag). 1 Changes all ATD conversion complete flags to a fast clear sequence. Any access to a result register will cause the associate CCF flag to clear automatically.
5 AWAI	**ATD Power Down in Wait Mode** — When entering wait mode this bit provides on/off control over the ATD block allowing reduced MCU power. Because analog electronic is turned off when powered down, the ATD requires a recovery time period after exit from Wait mode. 0 ATD continues to run in Wait mode 1 Halt conversion and power down ATD during wait mode After exiting wait mode with an interrupt conversion will resume. But due to the recovery time the result of this conversion should be ignored.
4 ETRIGLE	**External Trigger Level/Edge Control** — This bit controls the sensitivity of the external trigger signal. See Table 5-6 for details.
3 ETRIGP	**External Trigger Polarity** — This bit controls the polarity of the external trigger signal. See Table 5-6 for details.
2 ETRIGE	**External Trigger Mode Enable** — This bit enables the external trigger on one of the AD channels or one of the ETRIG3–0 inputs as described in Table 5-4. If external trigger source is one of the AD channels, the digital input buffer of this channel is enabled. The external trigger allows to synchronize sample and ATD conversions processes with external events. 0 Disable external trigger 1 Enable external trigger **Note:** If using one of the AD channel as external trigger (ETRIGSEL = 0) the conversion results for this channel have no meaning while external trigger mode is enabled.



Field	Description
1 ASCIE	**ATD Sequence Complete Interrupt Enable** 0 ATD Sequence Complete interrupt requests are disabled. 1 ATD Interrupt will be requested whenever ASCIF = 1 is set.
0 ASCIF	**ATD Sequence Complete Interrupt Flag** — If ASCIE = 1 the ASCIF flag equals the SCF flag (see Section 5.3.2.7, "ATD Status Register 0 (ATDSTAT0)"), else ASCIF reads zero. Writes have no effect. 0 No ATD interrupt occurred 1 ATD sequence complete interrupt pending

This register is used to power up the A/D module, and to configure interrupts. You may want conversion complete interrupts, so you may parameterize your initialization function to optionally turn them on.

The A/D offers a 'fast clear' option to acknowledge interrupts, so we will use this. Just reading a conversion result will clear the interrupt flag.

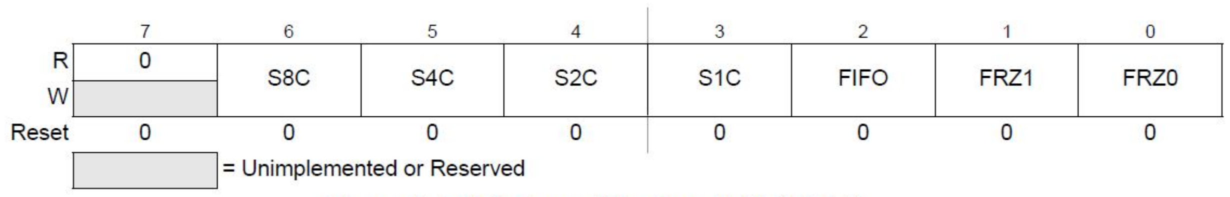
We will not use external triggers.

We will want the A/D to remain on in `WAI`. Normally you would power off the A/D to save power in wait, but we want stability over efficiency during testing.

Your initialization should set the `ADPU` and `AFFC` bits. You may optionally set the `ASCIE` bit if you want the A/D to generate interrupts on a completed conversion.

After the A/D is powered on, you must inject a >50us delay allowing it to become stable. Although it's such a short delay it may be irrelevant!

Next, you will configure the A/D to complete 8 conversions per sequence (scan all channels). This is done with the `AD0CTL3` register (5.3.2.4). We also don't want to use the FIFO feature, and we want the conversion to continue in freeze.



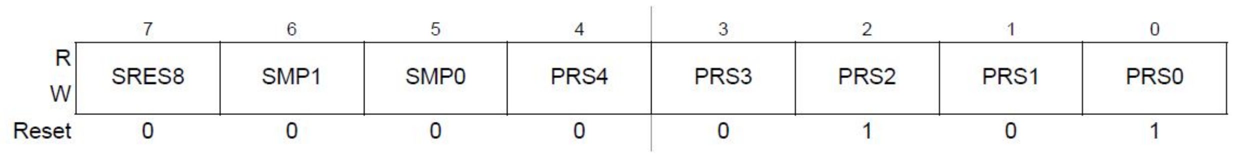
	7	6	5	4	3	2	1	0
R	0	S8C	S4C	S2C	S1C	FIFO	FRZ1	FRZ0
W	▨							
Reset	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 5-6. ATD Control Register 3 (ATDCTL3)

This means you need to write `0b01000000` to `ATD0CTL3`.

The next register (`ATD0CTL4`) determines clock parameters for the A/D. The A/D can only run at 2MHz max, so we need a prescale of 10 to clock our 20MHz bus down to 2MHz. (5.3.2.5)



	7	6	5	4	3	2	1	0
R W	SRES8	SMP1	SMP0	PRS4	PRS3	PRS2	PRS1	PRS0
Reset	0	0	0	0	0	1	0	1

Figure 5-7. ATD Control Register 4 (ATDCTL4)

At 10-bit resolution, this means a conversion will take:

- 2 A/D clock cycles for the 1st phase (fixed)
- 2 A/D clock cycles for the 2nd phase sample time (SMP0:1)
- 1 A/D clock cycle per bit (10).

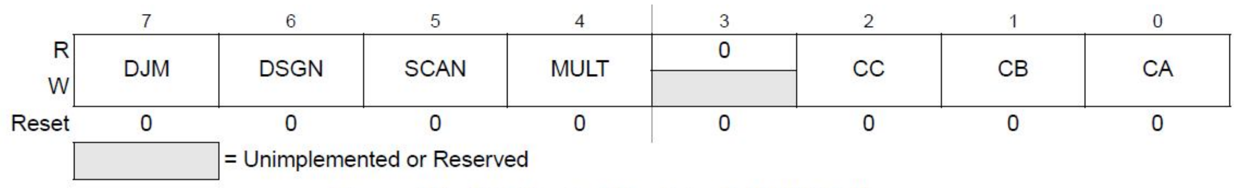
14 x 0.5µs = 7µs * 8 conversions = 56µs total per sequence.

To achieve this, we write `0b00000100` to `ATD0CTL4`. (for fast samples)

Note: 56µs per conversion is fast relative to our bus speed. In fact, to maintain this rate your code could do very little without overrunning this time interval. You could increase the sample time in `ATDCTL4`, which would both increase accuracy of the sample, and purchase time for longer code.

For example, using a conversion time of 16 A/D clocks would mean a value of `0b01100100` for `ATD0CTL4`, and a sample time of (2 + 16 + 10 x 0.5µs * 8 conversions) = 112µs total per sequence. This does not buy you a lot of time, but you might be able to do *something* between samples.

The last register (`ATD0CTL5`) determines how the data is presented, and how the conversion operates. We want right-justified data (`XXXXXX98 76543210`), unsigned results, with continuous scan on multiple channels, using `AN0` as the starting channel.



	7	6	5	4	3	2	1	0
R	DJM	DSGN	SCAN	MULT	0	CC	CB	CA
W					▨			
Reset	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 5-8. ATD Control Register 5 (ATDCTL5)

To achieve this (5.3.2.6), we would write `0b10110000` to `ATD0CTL5`.

You will need to adjust the `VRef` trimmer on your board to provide a precise reference of 5.12V. If you have no ability to verify this, it *should* have been done when fabricated/tested. Your AD2 should be sufficient for calibrating this, in afterthought.

You *should* have pins soldered into your board for `AN0` and `AN1`.

Your A to D library will likely only need one function, and one ISR.

```
// a to d library header

// assumes that interrupts are available on the A/D
// interrupt VectorNumber_Vatd0 void XXXXX (void);
void AtoDInit (int iEnableInterrupt);
```

Reading the results of the conversions is done by performing 16-bit reads on the individual channels (`ATD0DR0` through `ATD0DR7`).

Polling

If you are polling for results, you will need to wait for `ATD0STAT0_SCF`. This flag goes high when a conversion is complete, and with the fast flag clearing mechanism, a read from any result register will clear the flag:

Blocking:

```
    while (!ATD0STAT0_SCF)
      ;
```

Loop polling:

```
if (ATD0STAT0_SCF)
```

A read is then possible from any result register to clear the flag and obtain the result value:

```
unsigned int uiResult = ATD0DR0;
```

Results will be available for each pin as `ATD0DR0` to `ATD0DR7`.

Remember that the result will only contain ten valid, right-aligned bits, so it will be a value from 0 to 1023 (`0x0000` to `0x003FF`).

Remember: a full conversion sequence will only require 56µs to occur. This is a *really* short period of time, relative to the length of our display functions.

Interrupts

If you enable the A/D interrupt, the interrupt will occur once a conversion sequence is complete. With the 'fast clearing' flag on, simply reading any of the result registers will clear the interrupt flag.

Since a conversion only takes 56µs, it also means you will be performing 17857 interrupts (or samples) per second (if your code in main is fast enough)!

There is only one ISR for A/D module `0`, and it services all channels:

```
/********************************************************************/
//    Global Variables (A/D Results Registers)
/********************************************************************/
volatile uint ADVal[8] = {0};

…

interrupt VectorNumber_Vatd0 void INT_AD0 (void)
{
  // read channel values (reading any clears interrupt flag)
  ADVal[0] =  ATD0DR0;
  //ADVal[1] =  ATD0DR1;
  //ADVal[2] =  ATD0DR2;
  //ADVal[3] =  ATD0DR3;
  //ADVal[4] =  ATD0DR4;
  //ADVal[5] =  ATD0DR5;
  //ADVal[6] =  ATD0DR6;
  //ADVal[7] =  ATD0DR7;
}
```

Since the main code will likely be processing the values asynchronously, we would mark the array (or any other variable used this way) as `volatile`. There is an overwhelming possibility that the values will be updated while reading them, so the compiler will not make assumptions about the state of the variables in either read or write operations.

We could alternatively modify the code in main to setup a critical section around use of a result variable, by temporarily inhibiting interrupts long enough to copy the results into another variable. This is beyond the scope of this course but is certainly something to consider when we are taking samples this fast on a micro that is running this slow.

Fortunately, interrupts don't stack!

You may enable interrupts and simply passively read the results variables at your leisure, confident that the results array is being updated by the interrupt with current values.

If you intend to use interrupts, you should probably consider adjusting the sample time to something a little less aggressive, or polling might be more appropriate.

Sample A/D Library Header

```
// a to d library header (no docs)

// assumes that interrupts are available on the A/D
// interrupt VectorNumber_Vatd0 void XXXXX (void);
// iEnableInterrupt == 0, no interrupts, otherwise interrupts
void AtoD_Init (int iEnableInterrupt);

// read the desired channel, assumes fast flag clearing from init
// returns 0 on bad channel request
unsigned int AtoD_Read (unsigned int chan);
```

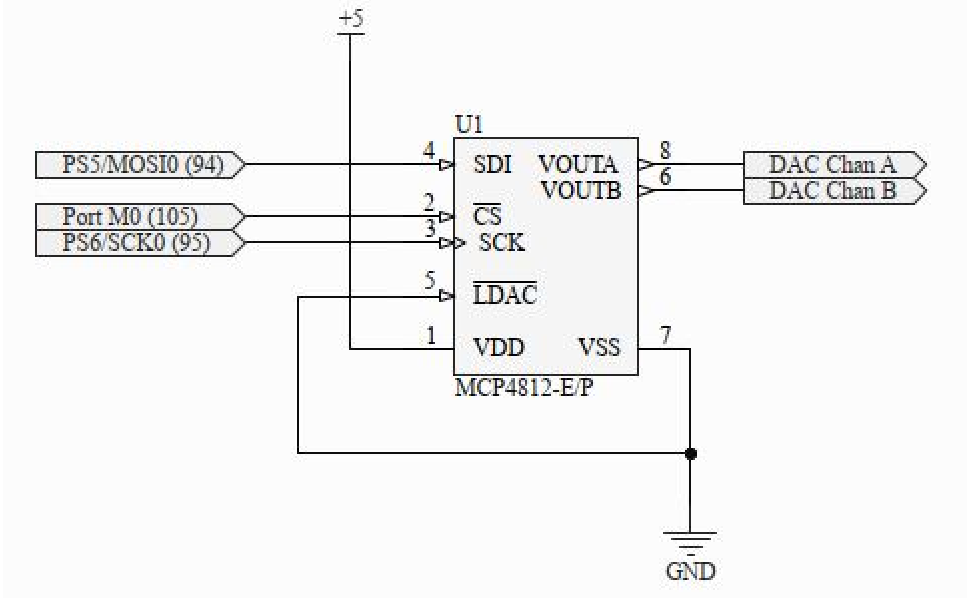
SPI - Serial Peripheral Interface

SPI communications are used typically for chip-to-chip communications over short (intra-board) distances. This synchronous protocol is relatively high-speed and is one of two synchronous communications protocols you will use in this course (I^2C being the other).

The SPI module is described in chapter 12 of Big Pink. SPI *can* be quite complex due to the highly configurable signaling settings, but in this course, we are only going to use it to communicate with a write-only device, and with a few variables taken out of the equation. Having the road paved a little will make this more enjoyable. Awareness over mastery applies here!

You should be in possession of a DIP version of the `MCP4812E/P`, two channel, 10-bit DAC with `2.048V` internal voltage reference. When presented with a suitable 10-bit digital value (`0x000-0x3FF`), this device will produce a proportional voltage from (`0.000V-2.048V`), so `2mV`/step. The SPI protocol is used to communicate with a wide variety of devices, we will be using this one, as it is simple to operate.

You will wire up the DAC on a breadboard, using the following diagram as a guide. The way the device is connected will provide some insight into how the SPI is being used. Note also that all signals to the device are write-only. If need be, you may solder some pins into your micro to make the circuit easier to wire-up. Follow standard safety procedures to protect you, your board, and the device – in that order.

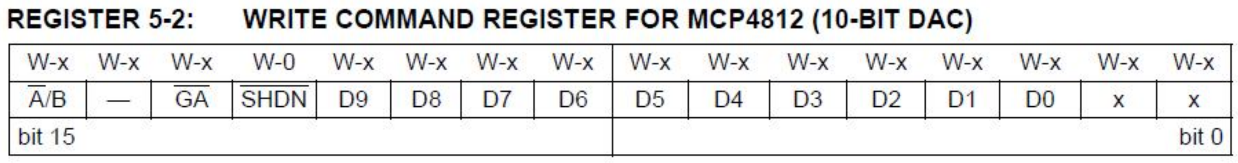


MCP4812E/P Schematic, to be implemented in a breadboard connected to your micro board

This DAC features a chip select signal, which enables clock and data functions. In some instances where a single device is being used, `*CS` may be tied low to have the device be 'always on'. In this case, we will use a GPIO pin to manually control the `*CS` line. This is a better design, has the potential to use less power, and would make the design future-proof if more devices were added in a bus/multiplex configuration.

The `SCK` signal is the clock that drives synchronous data transfer and is produced by the micro. The micro will be the 'master' in this configuration, so it will initiate communications, and will drive the clock signal.

The `SDI` signal is 'serial data input' and will be connected to the `MOSI` (Master Out Slave In) pin on the micro. This signal will carry the data to the DAC. In the case of this DAC, the data will be command information, and the DAC output value:



REGISTER 5-2: WRITE COMMAND REGISTER FOR MCP4812 (10-BIT DAC)

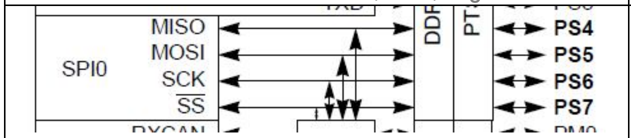
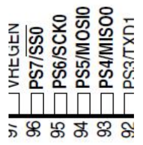
W-x	W-x	W-x	W-0	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x
$\overline{\text{A}}$/B	—	$\overline{\text{GA}}$	$\overline{\text{SHDN}}$	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	x	x
bit 15															bit 0

From the datasheet for the DAC, we can see the format of the data sent to the device. The most significant bits contain command information:

- `*A/B` – what DAC channel is the command for
- `*GA` – output gain select (a nice feature, permits output voltage to be x1 or x2)
- `*SHDN` – device shutdown (turn off the output)

The remaining bits are the oddly left-aligned DAC value. Devices that come in a variety of bit capabilities often use a similar mechanism for bit formatting. If you look at the datasheet for this DAC, you can see how the data bits are aligned for the 8-bit and 12-bit versions of this chip (common design / implementation).

Your micro contains several SPI ports, but you will use `SPI0`:

Module location in PIM, P36, Big Pink	Pin numbers, P50, Big Pink
 SPI0 MISO MOSI SCK SS PS4 PS5 PS6 PS7	 VREGEN PS7/SS0 PS6/SCK0 PS5/MOSI0 PS4/MISO0 PS3/TXD1 96 95 94 93 92

As you can see, the SPI port contains four signals, but we are only using two of them (`SCK0`, and `MOSI`). If we were using bidirectional chip communications* and module authority over the external device, we could additionally use the `MISO` and `*SS` signals as well. Obviously, other chips and different implementations may use the SPI port in a different way.

*We will actually be using the SPI to communicate bi-directionally, as this is how the micro expects it to operate. Since the device is write-only, nothing coherent will be received by the micro, so during this phase, the received bytes are ignored.

Of note from Big Pink:

12.4.3 Transmission Formats

During an SPI transmission, data is transmitted (shifted out serially) and received (shifted in serially) simultaneously. The serial clock (SCK) synchronizes shifting and sampling of the information on the two serial data lines. A slave select line allows selection of an individual slave SPI device; slave devices that are not selected do not interfere with SPI bus activities. Optionally, on a master SPI device, the slave select line can be used to indicate multiple-master bus contention.

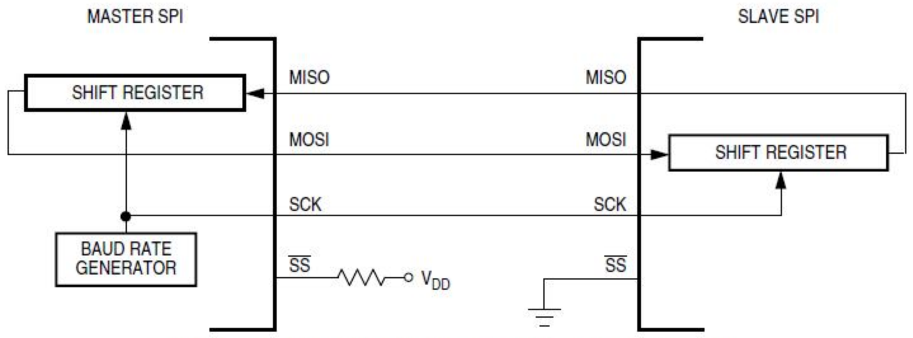
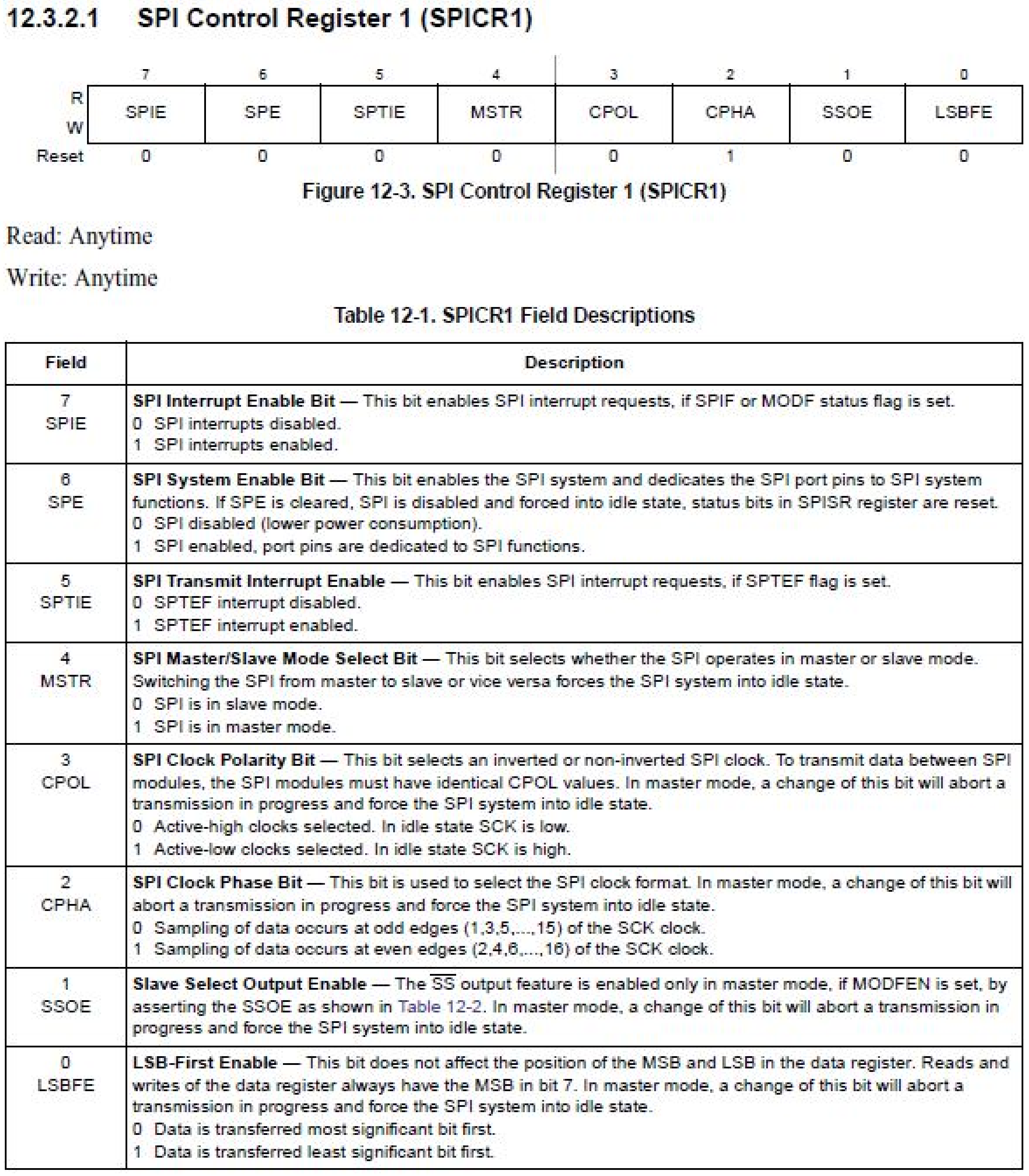


Figure 12-10. Master/Slave Transfer Block Diagram

While we are not using MISO, data will still appear, and it needs to be serviced.

To bring the SPI module up, there are a number of required configuration steps.

The SPI is principally configured through two control registers, `SPI0CR1` and `SPI0CR2`:



12.3.2.1 SPI Control Register 1 (SPICR1)

	7	6	5	4	3	2	1	0
R / W	SPIE	SPE	SPTIE	MSTR	CPOL	CPHA	SSOE	LSBFE
Reset	0	0	0	0	0	1	0	0

Figure 12-3. SPI Control Register 1 (SPICR1)

Read: Anytime

Write: Anytime

Table 12-1. SPICR1 Field Descriptions

Field	Description
7 SPIE	**SPI Interrupt Enable Bit** — This bit enables SPI interrupt requests, if SPIF or MODF status flag is set. 0 SPI interrupts disabled. 1 SPI interrupts enabled.
6 SPE	**SPI System Enable Bit** — This bit enables the SPI system and dedicates the SPI port pins to SPI system functions. If SPE is cleared, SPI is disabled and forced into idle state, status bits in SPISR register are reset. 0 SPI disabled (lower power consumption). 1 SPI enabled, port pins are dedicated to SPI functions.
5 SPTIE	**SPI Transmit Interrupt Enable** — This bit enables SPI interrupt requests, if SPTEF flag is set. 0 SPTEF interrupt disabled. 1 SPTEF interrupt enabled.
4 MSTR	**SPI Master/Slave Mode Select Bit** — This bit selects whether the SPI operates in master or slave mode. Switching the SPI from master to slave or vice versa forces the SPI system into idle state. 0 SPI is in slave mode. 1 SPI is in master mode.
3 CPOL	**SPI Clock Polarity Bit** — This bit selects an inverted or non-inverted SPI clock. To transmit data between SPI modules, the SPI modules must have identical CPOL values. In master mode, a change of this bit will abort a transmission in progress and force the SPI system into idle state. 0 Active-high clocks selected. In idle state SCK is low. 1 Active-low clocks selected. In idle state SCK is high.
2 CPHA	**SPI Clock Phase Bit** — This bit is used to select the SPI clock format. In master mode, a change of this bit will abort a transmission in progress and force the SPI system into idle state. 0 Sampling of data occurs at odd edges (1,3,5,...,15) of the SCK clock. 1 Sampling of data occurs at even edges (2,4,6,...,16) of the SCK clock.
1 SSOE	**Slave Select Output Enable** — The $\overline{SS}$ output feature is enabled only in master mode, if MODFEN is set, by asserting the SSOE as shown in Table 12-2. In master mode, a change of this bit will abort a transmission in progress and force the SPI system into idle state.
0 LSBFE	**LSB-First Enable** — This bit does not affect the position of the MSB and LSB in the data register. Reads and writes of the data register always have the MSB in bit 7. In master mode, a change of this bit will abort a transmission in progress and force the SPI system into idle state. 0 Data is transferred most significant bit first. 1 Data is transferred least significant bit first.

In order of MSB to LSB, we want: no interrupts, SPI enabled, SPI module in master mode, active high clock, odd-edge sampling, no *SS signal, and normal bit ordering (MSB first). That means that `SPI0CR1` should have `0b01010000` written to it, one way or another.

12.3.2.2 SPI Control Register 2 (SPICR2)

	7	6	5	4	3	2	1	0
R	0	0	0	MODFEN	BIDIROE	0	SPISWAI	SPC0
W	▨	▨	▨			▨		
Reset	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 12-4. SPI Control Register 2 (SPICR2)

Read: Anytime

Write: Anytime; writes to the reserved bits have no effect

Table 12-3. SPICR2 Field Descriptions

Field	Description
4 MODFEN	**Mode Fault Enable Bit** — This bit allows the MODF failure to be detected. If the SPI is in master mode and MODFEN is cleared, then the $\overline{SS}$ port pin is not used by the SPI. In slave mode, the $\overline{SS}$ is available only as an input regardless of the value of MODFEN. For an overview on the impact of the MODFEN bit on the $\overline{SS}$ port pin configuration, refer to Table 12-4. In master mode, a change of this bit will abort a transmission in progress and force the SPI system into idle state. 0 $\overline{SS}$ port pin is not used by the SPI. 1 $\overline{SS}$ port pin with MODF feature.
3 BIDIROE	**Output Enable in the Bidirectional Mode of Operation** — This bit controls the MOSI and MISO output buffer of the SPI, when in bidirectional mode of operation (SPC0 is set). In master mode, this bit controls the output buffer of the MOSI port, in slave mode it controls the output buffer of the MISO port. In master mode, with SPC0 set, a change of this bit will abort a transmission in progress and force the SPI into idle state. 0 Output buffer disabled. 1 Output buffer enabled.
1 SPISWAI	**SPI Stop in Wait Mode Bit** — This bit is used for power conservation while in wait mode. 0 SPI clock operates normally in wait mode. 1 Stop SPI clock generation when in wait mode.
0 SPC0	**Serial Pin Control Bit 0** — This bit enables bidirectional pin configurations as shown in Table 12-4. In master mode, a change of this bit will abort a transmission in progress and force the SPI system into idle state.

For `SPI0CR2`, you want `MODFEN` off, as we are not using `*SS`, `BIDIROE` off, as it won't apply, `SPISWAI` off to keep the SPI running in wait mode (optional), and `SPC0` off as we don't want bidirectional pins. If you look at table 12-4, you can see that the SPI offers highly configurable I/O options.

Writing a `0` to `SPI0CR2` (or leaving it as the reset default) will serve our needs.

Finally, the clock for the SPI module needs to be set.

12.3.2.3 SPI Baud Rate Register (SPIBR)

	7	6	5	4	3	2	1	0
R	0	SPPR2	SPPR1	SPPR0	0	SPR2	SPR1	SPR0
W	▨				▨			
Reset	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 12-5. SPI Baud Rate Register (SPIBR)

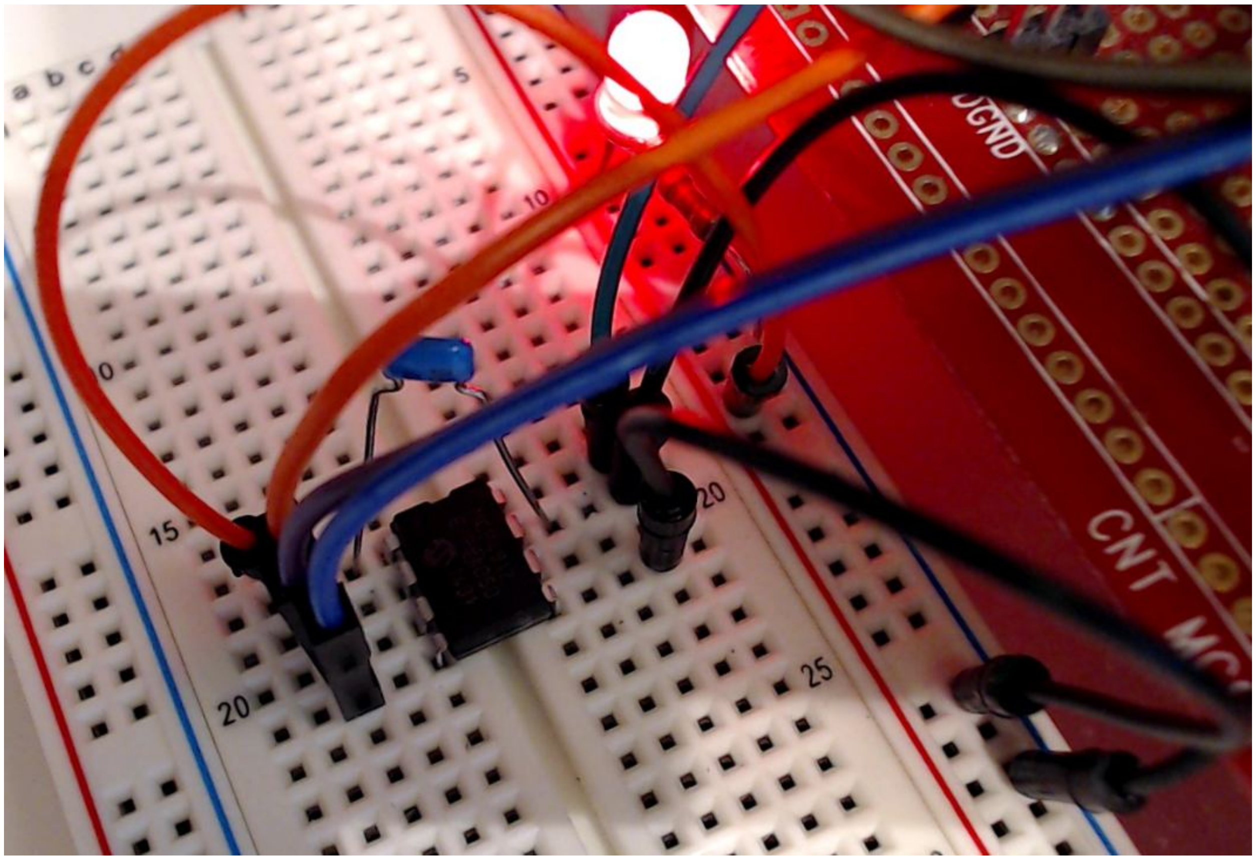
Read: Anytime

Write: Anytime; writes to the reserved bits have no effect

The formula in this section explains how the bus clock is divided through the preselection and selection bits, but a table is also provided! The SPI can run at relatively high rates, but since we are breadboarding our solution, a reasonable rate is recommended. In testing, a value of 'divide by two' yielded no problems. Our bus rate is 20MHz, so divide by two means the SPI clock is 10MHz. This is astonishing fast compared to the SCI!

To achieve a divide by two, you would plug in the value `0` for `SPI0BR`. If you are using very long wires to go from the micro to your breadboard, or your breadboard skills are questionable, you may want to initially use a higher divisor. Once it is working, you can rachet up the speed.

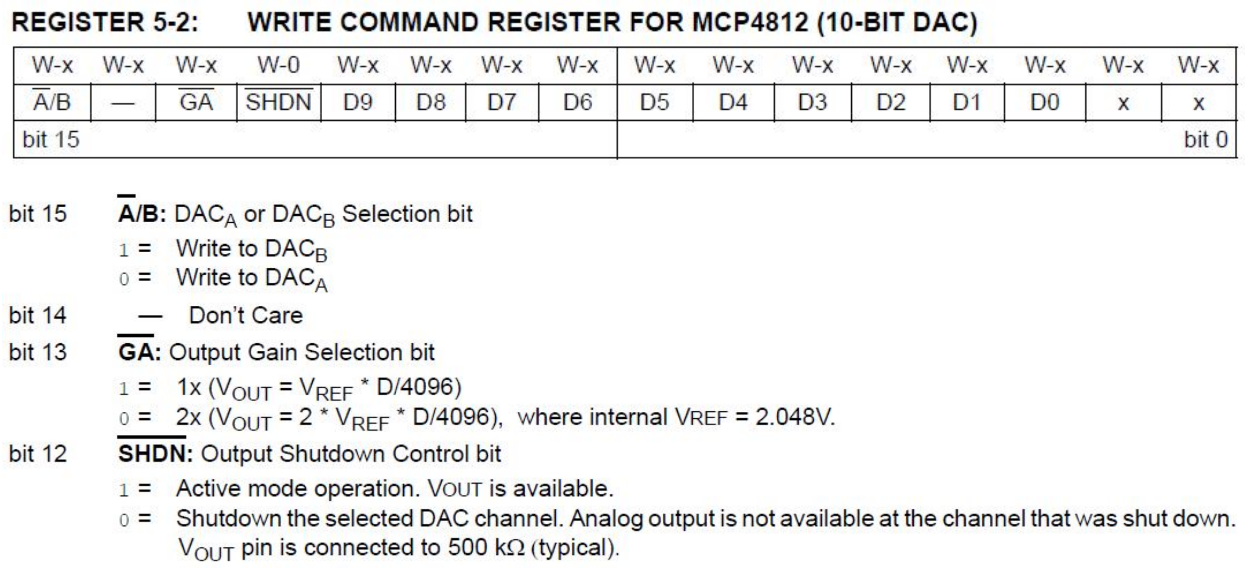
It may also prove useful to place a bypass capacitor (0.1µF) between the power pins (pins 1 and 7) of the DAC, as close as physically possible to the chip:



Since we are controlling the DAC enable line with `PM0`, you will need to use standard GPIO initializations to *safely* bring this pin up as an output, initially high. We have done this in many instances to create the LED, Segs, and LCD libraries, and this is no different.

Before you send data to the DAC, you need to prepare the 16-bit value that contains the correct command and data. Initially, you should probably test the DAC with predictable values, so you know that it is working. For example, you could use the DAC data value of `0x100` as a test. This value, at `2mV` per step, with a gain of `1` should produce (`256` x `2mV`) = `0.512V` on the targeted channel.

Referring back to the DAC datasheet:



REGISTER 5-2: WRITE COMMAND REGISTER FOR MCP4812 (10-BIT DAC)

W-x	W-x	W-x	W-0	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x
$\overline{A}/B$	—	$\overline{GA}$	$\overline{SHDN}$	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	x	x
bit 15															bit 0

bit 15	**$\overline{A}/B$:** DAC_A or DAC_B Selection bit 1 = Write to DAC_B 0 = Write to DAC_A
bit 14	— Don't Care
bit 13	**$\overline{GA}$:** Output Gain Selection bit 1 = 1x ($V_{OUT} = V_{REF}$ * D/4096) 0 = 2x (V_{OUT} = 2 * V_{REF} * D/4096), where internal VREF = 2.048V.
bit 12	**$\overline{SHDN}$:** Output Shutdown Control bit 1 = Active mode operation. VOUT is available. 0 = Shutdown the selected DAC channel. Analog output is not available at the channel that was shut down. V_{OUT} pin is connected to 500 kΩ (typical).

A 16-bit value of `0b0011010000000000`, or `0x3400` would contain a command of channel A, gain = 1x, no shutdown, and a DAC value of `0x100`, where red is command, blue is DAC data, and purple is alignment 'don't care'.

As you manipulate your DAC data values, you will need to use bit manipulations to pack them into a coherent payload to send to the DAC.

What would be the result of the following DAC payloads:

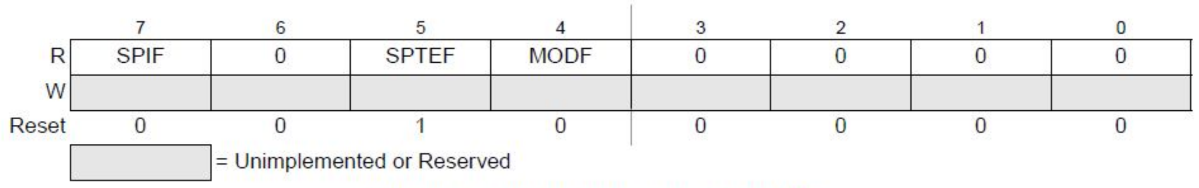
`0b1111101010101011`

`0x0F00`

`62000`

The status is the SPI is found in the `SPI0SR` register:

12.3.2.4 SPI Status Register (SPISR)



	7	6	5	4	3	2	1	0
R	SPIF	0	SPTEF	MODF	0	0	0	0
W	▨	▨	▨	▨	▨	▨	▨	▨
Reset	0	0	1	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 12-6. SPI Status Register (SPISR)

Read: Anytime

Write: Has no effect

Table 12-7. SPISR Field Descriptions

Field	Description
7 SPIF	**SPIF Interrupt Flag** — This bit is set after a received data byte has been transferred into the SPI data register. This bit is cleared by reading the SPISR register (with SPIF set) followed by a read access to the SPI data register. 0 Transfer not yet complete. 1 New data copied to SPIDR.
5 SPTEF	**SPI Transmit Empty Interrupt Flag** — If set, this bit indicates that the transmit data register is empty. To clear this bit and place data into the transmit data register, SPISR must be read with SPTEF = 1, followed by a write to SPIDR. Any write to the SPI data register without reading SPTEF = 1, is effectively ignored. 0 SPI data register not empty. 1 SPI data register empty.
4 MODF	**Mode Fault Flag** — This bit is set if the $\overline{SS}$ input becomes low while the SPI is configured as a master and mode fault detection is enabled, MODFEN bit of SPICR2 register is set. Refer to MODFEN bit description in Section 12.3.2.2, "SPI Control Register 2 (SPICR2)". The flag is cleared automatically by a read of the SPI status register (with MODF set) followed by a write to the SPI control register 1. 0 Mode fault has not occurred. 1 Mode fault has occurred.

We will not be using interrupts (yet), but the `SPIF` flag still serves to indicate data has been received. Even though we are only writing to the DAC, the SPI operates with a bidirectional transfer of data. We can ignore the data coming from the DAC, as there isn't any, but we can still use this flag to indicate the end of the transfer operation, and consequently go through the motions of pulling the dummy data from the buffer.

The `SPTEF` flag indicates that we are clear to send a byte through the SPI. Note the description on how this flag is read and cleared! You will always check to ensure that you are clear to send a byte through the SPI before you write to `SPIDR`.

Since we are not using `*SS`, we can effectively ignore `MODF`.

The procedure then, for sending a 16-bit payload to the DAC is the following:

1) Prepare 16-bit payload value that contains the appropriate command and data bits
2) Enable the DAC by lowering the `PM0` line (manually)
3) Wait for `SPTEF` (clear to send)
4) Send the MSB of the payload to `SPI0DR` (the SPI data register)
5) Wait for `SPTEF`
6) Send the LSB of the payload to `SPI0DR`
7) Wait for `SPIF*`
8) Read discard value from `SPI0DR`
9) Wait for `SPIF`
10) Read discard value from `SPI0DR`
11) Disable the DAC by raising the `PM0` line (manually)

*Note: steps 7-8 can be injected between steps 4 and 5, if the pattern of TX/RX, TX/RX is more comfortable for you, but the above technique is slightly (~15%) faster, as you do not have to wait for one of the received bytes to arrive.

Note: steps 7-10 do not read any useful values, as the DAC does not produce any data. These steps simply acknowledge the flags and associated receiver buffers in a timely way. Failure to do so will block subsequent transmission. If the connected device did produce data, these steps would produce the data from that device.

You may put a 0.1µF capacitor on the output channel to ground to stabilize the output, since the measurement device will have very high impedance.

While the procedure for packing the payload with meaningful information may be done in the main loop, pushing a 16-bit value to the DAC through the SPI should be placed in a function, as this is a generic operation that will be used in subsequent activities.

You may wish to parameterize your DAC write function to include an option for the gain setting and output channel selection, since these are both very useful options.

We would normally put this in a compilation unit specific to the device, not the SPI, as the operation of the SPI is not abstracted, and has very specific use of the module.

Sample MCP4812 Header

```
// MCP4812 header

// wire pattern:
// PM0 (Micro P105) -> *CS (MCP4812 P2)
// PS5 (Micro P 94) -> SDI (MCP4812 P4)
// PS6 (Micro P 95) -> SCK (MCP4812 P3)

// *LDAC, VSS -> GND
// VDD +5V

// DAC Chan A out (MCP4812 P8)
// DAC Chan B out (MCP4812 P6)

typedef enum MCP4812_Chan_Sel
{
  MCP4812_ChanA, // channel A (pin 8)
  MCP4812_ChanB, // channel B (pin 6)
} MCP4812_Chan_Sel;

typedef enum MCP4812_Gain_Sel
{
  MCP4812_Gain1x, // 2.048V full scale (2mV/step)
  MCP4812_Gain2x, // 4.096V full scale (4mV/step)
} MCP4812_Gain_Sel;

// init MCP4812 for use on SPI0, and PM0
void MCP4812_Init (void);

// write a 10-bit value to the DAC on the specified channel, with the specified gain
void MCP4812_Write (unsigned int uiVal, MCP4812_Chan_Sel chan, MCP4812_Gain_Sel gain);
```

I^{2}C Bus

The I^{2}C bus (“I-Squared-C”) was invented by Philips Semiconductor in 1982. The terms “IIC” and “I2C” generally refer to the same thing. It is a synchronous, half-duplex protocol that only requires a clock and data signal for operation. It is primarily intended to be used for short distance inter-device (intra-board) communications. By comparison to other modern communications protocols, it is relatively slow, but is cheap to implement in designs. This protocol is generally fixed at 0.1, 0.4, 1.0, 3.4, or 5.0 Mbit/s, depending on device mode and capability. In our class, we will operate the I^{2}C bus at 400KHz, as the micro and all devices have this capability (‘Fast Mode’).

The clock (SCL) and data (SDA) lines are open collector and must be pulled high with pullup resistors for the bus to operate correctly. This bus configuration is limited by capacitance, low noise immunity, and physical layer requirements, and as a result, it only works over very short distances.

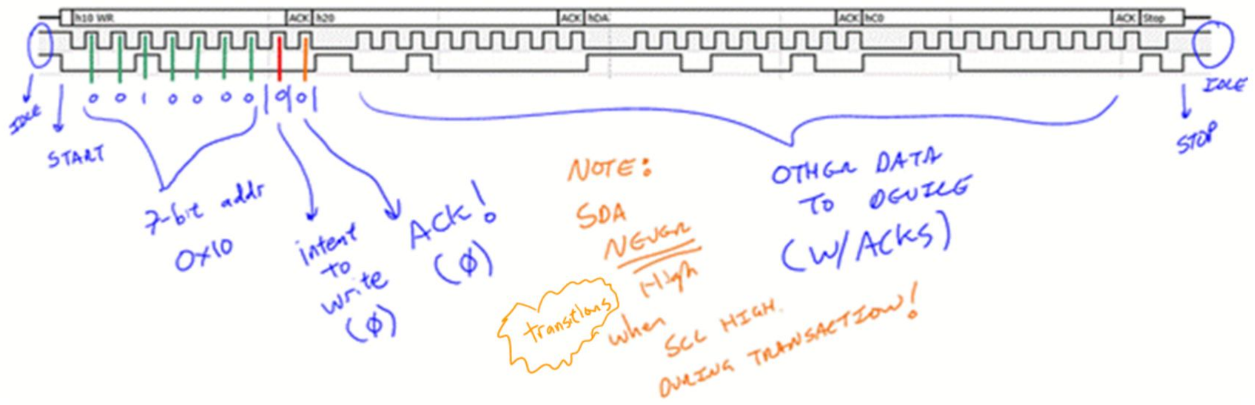
Note: I^{2}C documentation, to this day, continues to use the terms “Master” and “Slave” for devices participating on the I^{2}C bus. Effectively, only one device can control the bus (initiating the transaction), while one other responds if addressed correctly. These are terrible terms, and they will be replaced with **initiator** and **responder**, respectively, for the remainder of this document.

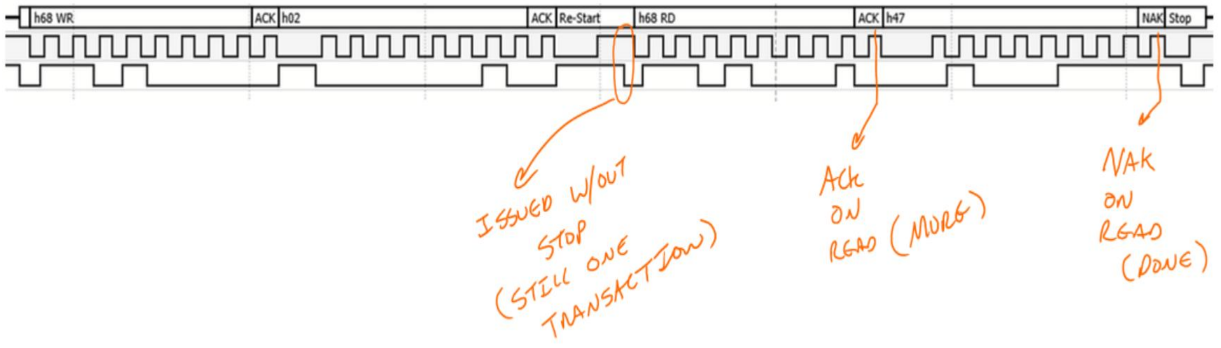
The fine details of the physical layer of the I2C bus protocol is beyond the scope of this document, but some elements are useful to understand. For instance, the clock (SCL) is always driven by the initiator and is used as a reference for the data signal (SDA). The level of the SDA line with respect to the SCL line marks events on the bus, including a START condition, a STOP condition, ACK, or data. When not in use, the bus idles high.

Transactions on the I2C bus begin with a START and end with a STOP condition. The initiator typically waits for the bus to be idle, as the bus could be in use by another initiator. In our case, the micro will always be the initiator, and all the other I2C devices on the board will be responders.

The initiator begins the transaction by sending a START condition followed by the 7-bit address of the responder it wishes to communicate with. The address component includes a desire to read (1) or write (0) as the LSB. If the target device exists on the bus, it will respond with an ACK*.

Data is sent MSB first, and a START condition is indicated by high to low transition of SDA (while SCL high). A STOP condition is indicated by a low to high transition of SDA while SCL is high (back to idle*). All other transitions of SDA occur with SCL low (during a transaction)***:





*Some write-only devices, like the LTC2633, do not ACK on read and may evade bus scans!

**The exception to this is a RESTART condition. The initiator may issue a RESTART condition prior to a STOP condition, usually to change the direction of data travel, as seen above. This is the only instance where SCL and SDA are high at the same time during a transaction.

Our micro handles this by declaring the bus busy between START and STOP conditions, not by simply sampling the SCL and SDA lines.

This super-simple mechanism provides robust, and easy to interpret synchronous communications over only two wires.

Using I^{2}C

This document is to be used in conjunction with the course resources for CMPE2200. The physical wiring and hardware components of the I^{2}C bus will not be discussed in this section, rather the abstracted behaviors of I^{2}C devices will. It is assumed that you have a basic understanding of what I^{2}C is.

You should construct your libraries with an abstraction (driver) component, and a device-specific component, separating each device into a separate compilation unit. Each device compilation unit will leverage the I^{2}C abstracted (driver) compilation unit.

Program Layer (Uses device specific libraries)				
LTC2633 (device lib)	24AA512 (device lib)	MPL3115 (device lib)	LSM303 (device lib)	M41T81 (device lib)
I2C Driver Layer (Basic I/O functions for I2C)				

All devices that participate on the I^{2}C bus in the CMPE2250 context will have a specific address (or addresses) to which they respond. You will need to know these addresses when you are writing an implementation library for that device. Note that device addresses are 7 bits in length and are left-aligned in the initiation byte. The 7-bit left-aligned nature of the address may be confusing, as the address will appear double in value when viewing the device address in command byte form vs. datasheet values.

Your board contains the following devices, and the datasheets for these devices are found in Moodle:

Device	**Description**	**7-bit I2C Address**
LTC2633	Dual 12-bit DAC	0x10 (8-bit = 0x20)
24AA512	512Kbit serial EEPROM	0x50 (8-bit = 0xA0)
MPL3115	Precision Altimeter	0x60 (8-bit = 0xC0)
LSM303	3D Accelerometer 3D Magnetometer	0x19 (8-bit = 0x32) 0x1E (8-bit = 0x3C)
M41T81	Real-Time Clock	0x68 (8-bit = 0xD0)

NOTE: Adafruit has cobbled together a somewhat complete list of known I2C address used by many devices. If you want to select multiple devices to connect to the same bus, this can assist in preventing address collisions:

https://learn.adafruit.com/i2c-addresses/the-list

All I^{2}C devices will require the same basic operations with the I^{2}C bus. It would not make sense to repeat this code in every device library. Instead, you will create (or be provided) a driver library that performs these generic bus operations, and the specific device libraries will call the driver functions as prescribed for that device.

Your generic I^{2}C driver library will contain the following functions:

I2C_WaitForBus

The I^{2}C bus is multi-device but is coordinated by only one controlling device. In our case, the micro will be the device that controls and coordinates all activity on the bus. In order to start any operations on the bus, the bus must not be busy. The busy/not busy condition is signaled by the `IBB` bit of the `IBSR` register (9.3.2.4). The bus is not busy when both SDA and SCL are high, but IBB is set on START and cleared on STOP. When this bit is set, the bus is busy. Note: if a bus operation is disturbed (possibly by reset) while a device is answering the micro, the bus may stay busy indefinitely.

The `I2C_WaitForBus` function will inspect the `IBB` bit until it clears, or a timeout loop count is reached. The calling code should check the return value of this function before continuing with subsequent activities.

I2C_Init0

The I^{2}C bus is initialized for use (and “unstuck”) with this function.

The version provided is parameterized for the desired bus rate – either 100KHz or 400KHz. The implementation provided will attempt to teardown any previous bus activity, and then initialize the bus for use. Because the bus may have been disturbed during a transmission, this function will clock out a series of pulses on `SCL` using GPIO operations to complete any pending transfer.

In the initialization phase the clock rate for the bus is set through `IBFD` (9.3.2.2). The bits written here determine the bus rate via a divisor, and the various hold rates for SDA and SCL. With a micro bus rate of 20MHz, the clock rate must be reduced by a factor of 50 for 400KHz operation or by 200 for 100KHz operation. The function is parameterized to allow this selection, and you would likely select the maximum rate allowed by the devices you wish to use. All the devices on our board permit 400kHz operation.

Next, the bus must be enabled through `IBCR` (9.3.2.3). The `IBEN` bit must be set before any other bits in the `IBCR`. After this, interrupts may be switched on or off and the module may be configured to continue to run in wait mode.

Using interrupts may be helpful, as the module will report errors that may be useful for debugging.

General Device Library Operation

Operation of the bus begins with `I2C_SendAddressRW`*, indicating that the micro wants to initiate a transaction with a start condition (in `IBCR`). Next, the address of the target is written to the `IBDR`, with the address as the top 7 bits, and R/W* indication in the least significant bit.

All devices will acknowledge their address on write, so presence/absence of an acknowledge on address announcement will determine if a device is responding properly.

After this, the `I2C_WriteByte`*, `I2C_RXByte`*, and `I2C_IssueRestart`* functions, or combinations thereof are used to operate the target device. The use of these functions is dependent on the type of device, so the datasheet of the target device must be consulted.

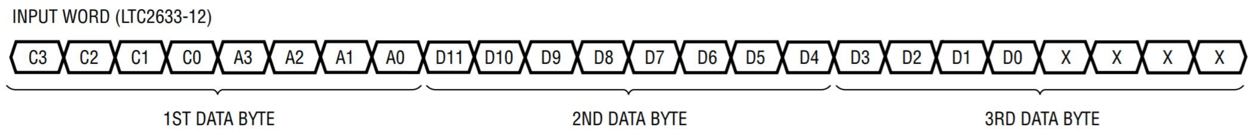
Ultimately the transaction is terminated with a STOP condition. The above functions support this where appropriate.

***All** I^{2}C devices may be operated with these four functions.

The LTC2633 Dual DAC

Before operating a device, you need to be keenly aware of the device address, the max. bus speed, and the bus format for communications. In other words, before you proceed, you would read the entire relevant sections of the datasheet (focus on page 18ish for this device). You should do this now.

A brief look at the datasheet for the LTC2633 will reveal that the max. bus rate is 400KHz (Page 1, 9), the device address is 0x10 (7-bit) (Page 17), and the bus format is (page 18):



This device is a write-only DAC, so it is simple in operation (all writes, no reads). This device is a little odd, in the fact that it does not ACK reads at all, but since we can't read from it, this makes *some* sense.

The library header and implementation for this device has been provided to you, at the end of this document. The code for this device will serve as a template for general I2C operations. You will be required to have more of a hand in creating the libraries for the other devices on the board.

Following along in the header, you need only one function to operate the device. This function will take the 12-bit value you wish to write, and the target channel. You may optionally include an address offset, as this device permits writing to multiple devices (but not in our case).

In the function you would:

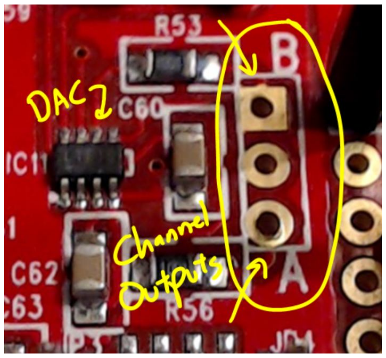
- Use the `I2C_SendAddressRW` function to send the device address in 8-bit form (`0x20`), along with write intention, waiting on the bus to be free.
- Use the `I2C_WriteByte` function to send the command byte (customized for the selected channel), without stop (more bytes are on the way).
- Use the `I2C_WriteByte` function to send the left-aligned, 8-bit MSB data (without stop (more bytes are on the way)).
- Use the `I2C_WriteByte` function to send the left-aligned, 4-bit LSB data (with stop, done).

```
/////////////////////////////////////////////
// LTC2633 - Dual 12-bit I2C DAC Library
// 7-bit device address 0x10, but 0x20 as command
// this device can go to 400kHz
/////////////////////////////////////////////

// this is a 12-bit, 4.096V (1mV/step), two channel DAC
// write time per value is ~118.5us @ 400KHz bus (measured)
// general form (write only):
// <ADDR><C3:C0,A3:A0><D11:D4[DAC Value]><D3:D0[0]>

// C3 C2 C1 C0
//  0  0  0  0 - write to register n
//  0  0  0  1 - update (power up) DAC, register n
//  0  0  1  0 - write to input register n, update (power up) all
//  0  0  1  1 - write to and update (power up) DAC register n
//  0  1  0  0 - power down n
//  0  1  0  1 - power down chip
//  0  1  1  0 - select (power up) internal reference
//  0  1  1  1 - select external reference
//  1  1  1  1 - no op
// NOTE: left-aligned data

// A3 A2 A1 A0
//  0  0  0  0 - DAC A
//  0  0  0  1 - DAC B
//  1  1  1  1 - all DACs

// command form (8-bit) of address is 0x20 (0x10 << 1)
#define LTC2633ADDR 0x20

// channel decode tags for LTC write function
typedef enum LTC2633_CHAN_SELECT
{
  LTC2633_CHAN_A,
  LTC2633_CHAN_B,
  LTC2633_CHAN_BOTH
} LTC2633_CHAN_SELECT;

// write a channel
int LTC2633_WriteChan (unsigned int Value, LTC2633_CHAN_SELECT chan);
```

```
/////////////////////////////////////////////
// LTC2633 - Dual 12-bit I2C DAC Library
/////////////////////////////////////////////
#include <hidef.h>
#include "derivative.h"
#include "i2c.h"
#include "LTC2633.h"

int LTC2633_WriteChan (unsigned int Value, LTC2633_CHAN_SELECT chan)
{
  // address the device
  if (I2C_SendAddressRW(LTC2633ADDR, I2C_WRITE, I2C_WAIT))
    return -1;

  // send command (write chan, power up all)
  if (chan == LTC2633_CHAN_A)
    (void)I2C_WriteByte (0b00100000, I2C_NOSTOP); // P18, datasheet
  else if (chan == LTC2633_CHAN_B)
    (void)I2C_WriteByte (0b00100001, I2C_NOSTOP); // P18, datasheet
  else // assume all channels
    (void)I2C_WriteByte (0b00101111, I2C_NOSTOP); // P18, datasheet

  // send msb data (data is 12 bits, oddly, left aligned, P18, datasheet)
  (void)I2C_WriteByte ((unsigned char)(Value >> 4), I2C_NOSTOP); // 0x0123 becomes 0x12

  // send lsb data
  (void)I2C_WriteByte ((unsigned char)(Value << 4), I2C_STOP);   // 0x0123 becomes 0x30

  return 0; // good condition
}
```

With this single function you may write values to the DAC and they will be immediately expressed on the desired channel.

The pins for these channels are found near the DAC on the bottom left of the board:



The M41T81S Real-Time Clock

The real-time clock chip is capable of independently keeping track of wall/calendar time. Timekeeping persists during power-off periods by use of a CR2032 battery backup and a separate oscillator circuit. A setup that contains an RTC is nice, as the time is always available from a local source, and the chip manages the confusion of date and time management.

The RTC chip is bidirectional; in addition to reading the clock values, the clock needs to be set, and configuration registers need to be modified. This will be your first exposure to a bidirectional device. This is also the first device you will encounter that has a very rich set of registers:

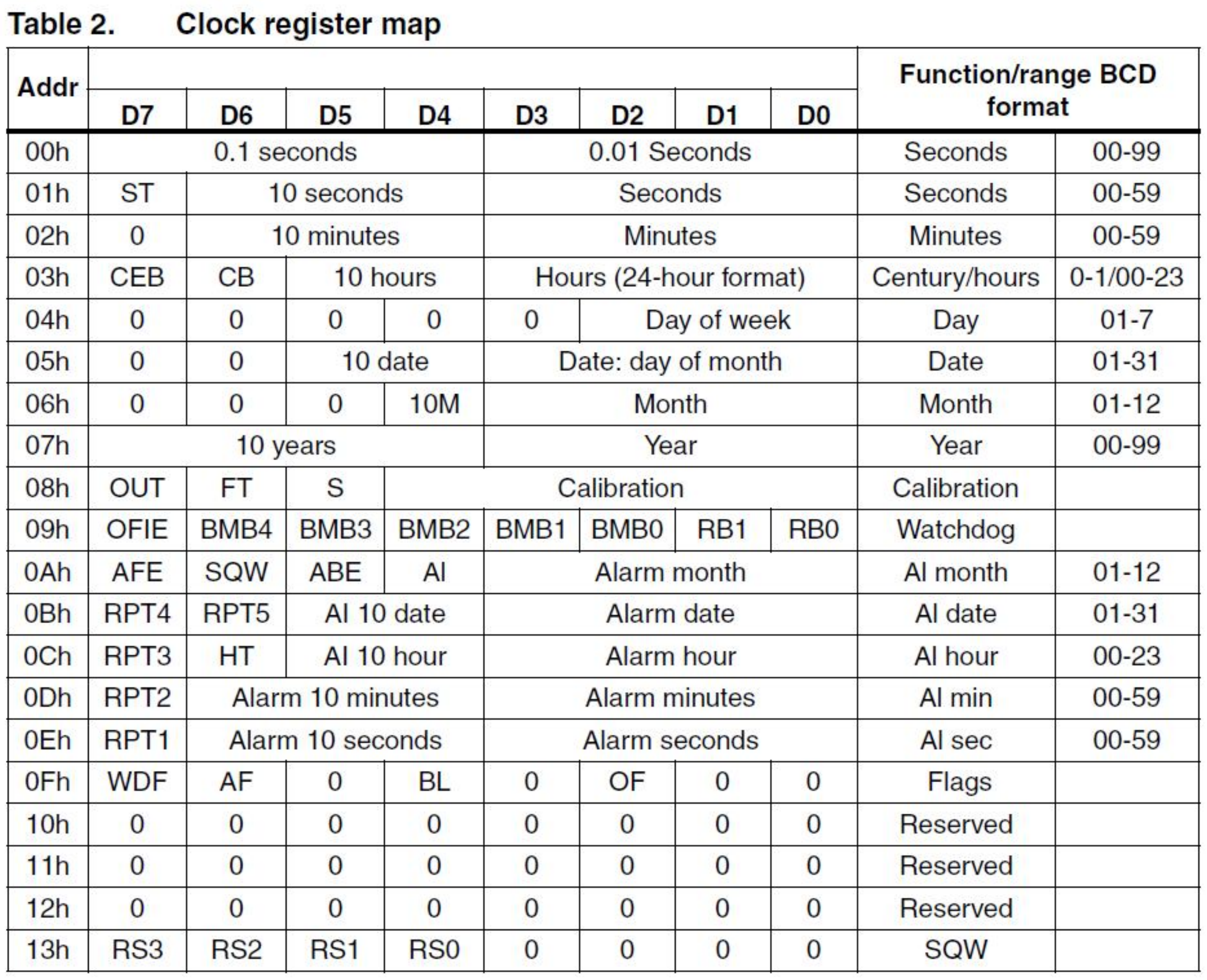


Table 2. Clock register map

Addr	D7	D6	D5	D4	D3	D2	D1	D0	Function/range BCD format	
00h	0.1 seconds				0.01 Seconds				Seconds	00-99
01h	ST	10 seconds			Seconds				Seconds	00-59
02h	0	10 minutes			Minutes				Minutes	00-59
03h	CEB	CB	10 hours		Hours (24-hour format)				Century/hours	0-1/00-23
04h	0	0	0	0	0	Day of week			Day	01-7
05h	0	0	10 date		Date: day of month				Date	01-31
06h	0	0	0	10M	Month				Month	01-12
07h	10 years				Year				Year	00-99
08h	OUT	FT	S	Calibration					Calibration	
09h	OFIE	BMB4	BMB3	BMB2	BMB1	BMB0	RB1	RB0	Watchdog	
0Ah	AFE	SQW	ABE	AI	Alarm month				Al month	01-12
0Bh	RPT4	RPT5	AI 10 date		Alarm date				Al date	01-31
0Ch	RPT3	HT	AI 10 hour		Alarm hour				Al hour	00-23
0Dh	RPT2	Alarm 10 minutes			Alarm minutes				Al min	00-59
0Eh	RPT1	Alarm 10 seconds			Alarm seconds				Al sec	00-59
0Fh	WDF	AF	0	BL	0	OF	0	0	Flags	
10h	0	0	0	0	0	0	0	0	Reserved	
11h	0	0	0	0	0	0	0	0	Reserved	
12h	0	0	0	0	0	0	0	0	Reserved	
13h	RS3	RS2	RS1	RS0	0	0	0	0	SQW	

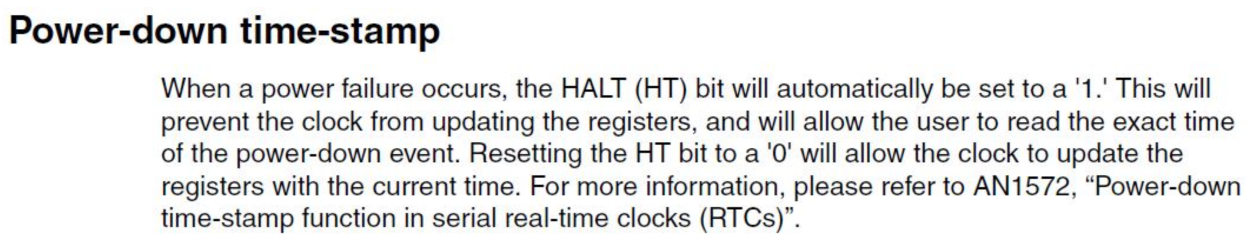
All time components are represented as BCD, and any paradoxical values will only be corrected when that time component is updated. Care must be taken when setting the clock, as unusual values may be accepted!

Once again you should create a separate compilation unit for the RTC, leveraging the driver level I2C library. A header suggestion would be something like this:

```
// RTC Functions
// Nov 2021: corrected osc fail code/function

// general form:
// READ: device supports block reading:
//  <ADDR><data x N, ACK><data NACK STOP>
// OR:
//  <ADDR><REG><RS ADDR><DATA x N ACK><DATA NACK STOP>
// WRITE:
//  <ADDR><REG><Data x N + STOP>
// see datasheet for 20 registers map

// 8-bit address form (datasheet 7-bit address is 0x68)
#define RTC_ADDRESS 0xD0

enum RTC_Days
{
  RTC_SUN = 1, RTC_MON, RTC_TUE, RTC_WED, RTC_THU, RTC_FRI, RTC_SAT
};

// all values in BCD!
typedef struct RTC_TimeBlock
{
  // BCD format
  unsigned char ucSec;      // seconds 00-59
  unsigned char ucMin;      // minutes 00-59
  unsigned char ucHour;     // hours   00-23
  enum RTC_Days eDOW;       // day of week (1-7)
  unsigned char ucDOM;      // day of month (1-31)
  unsigned char ucMOY;      // month of year (1-12)
  unsigned char ucYear;     // year (00-99) -> 2000-2099
} RTC_TimeBlock;

// kill the stop flag if the osc failed
// will clear halt
// will use LED indicators for fault presentation
// returns !0 on error
int RTC_CheckOF (void);

// kill the halt flag
void RTC_ClearHalt (void);

// set all main clock components (BCD)
void RTC_SetClock (RTC_TimeBlock timeblock);

// get all main clock components (BCD)
RTC_TimeBlock RTC_GetClock (void);

// return the time string to the caller (buff must be 21 characters)
void RTC_GetCurrentTimeString (char * buff);

// return the time string to the caller (buff must be 21 characters)
void RTC_GetCurrentTimeStringSecs (char * buff);

// compare time values (are they different?)
int TimeComp (RTC_TimeBlock * A, RTC_TimeBlock * B);

// is the battery low (1) or not (0)
int RTC_BatteryLow (void);

// happy helpers
// front-end for RTC read a byte
//unsigned char RTC_Read (unsigned char Address);
// front-end for write a byte
//void RTC_Write (unsigned char Address, unsigned char Value);
```

The RTC chip may have random values if it has never been set or may have been set (possibly inaccurately) during manufacture and testing. In general, you will need to set the clock. Remember that if you use a block of code to set the clock, it will need to be disabled in subsequent runs or the clock will be continually set to the same time.

A feature of the RTC chip we are using is a mechanism that captures the last time of power failure. Essentially, when you power down your board, the RTC chip switches to backup power and records the power failure time. This value is presented in the time registers the time next time the chip is powered up. This mechanism permits you to read the last power failure time, but also hides the actual time until you clear a flag:

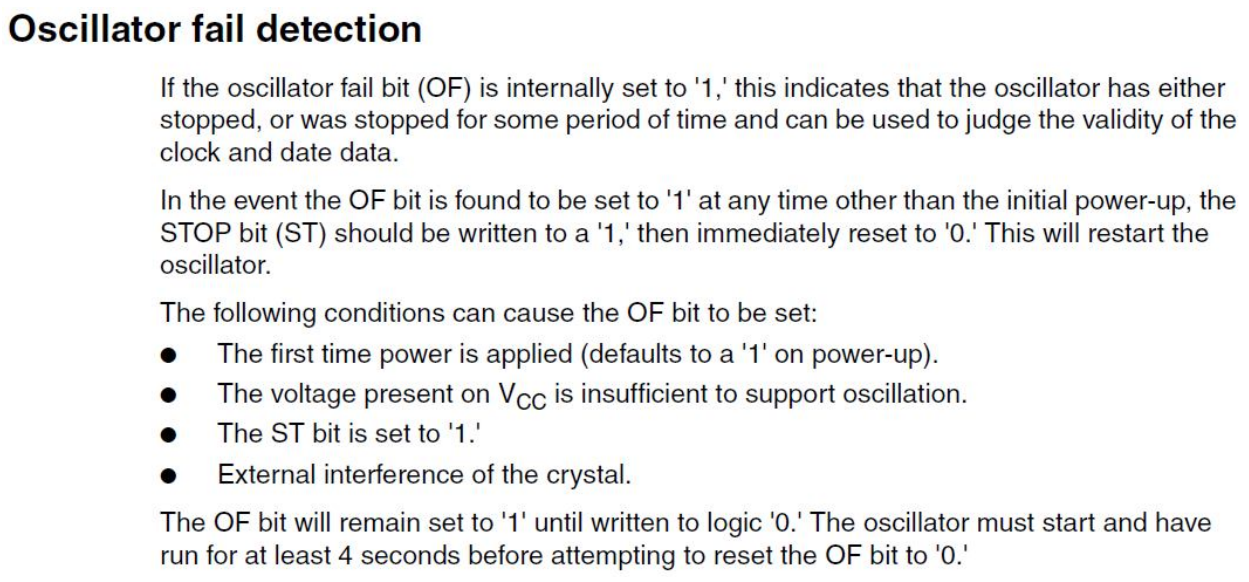


Power-down time-stamp

When a power failure occurs, the HALT (HT) bit will automatically be set to a '1.' This will prevent the clock from updating the registers, and will allow the user to read the exact time of the power-down event. Resetting the HT bit to a '0' will allow the clock to update the registers with the current time. For more information, please refer to AN1572, “Power-down time-stamp function in serial real-time clocks (RTCs)”.

You need to include a function to clear the HT bit to restore normal operation. In the sample header, this function is called `RTC_ClearHalt`.

When the device is birth powered on it will report a failed oscillator. If this flag has not been cleared, you may need to deal with it. This typically only happens if the backup battery has been replaced, or backup power has been lost:



Oscillator fail detection

If the oscillator fail bit (OF) is internally set to '1,' this indicates that the oscillator has either stopped, or was stopped for some period of time and can be used to judge the validity of the clock and date data.

In the event the OF bit is found to be set to '1' at any time other than the initial power-up, the STOP bit (ST) should be written to a '1,' then immediately reset to '0.' This will restart the oscillator.

The following conditions can cause the OF bit to be set:

- The first time power is applied (defaults to a '1' on power-up).
- The voltage present on V_{CC} is insufficient to support oscillation.
- The ST bit is set to '1.'
- External interference of the crystal.

The OF bit will remain set to '1' until written to logic '0.' The oscillator must start and have run for at least 4 seconds before attempting to reset the OF bit to '0.'

You should create a function to deal with an OF condition, regardless of whether you will ever see the condition. You can test for this condition by temporarily popping the backup battery out of the holder, but this is well beyond the scope of this course!

The header provided describes two support functions that are local to the implementation file. These functions are generic register read/write functions that are designed to read or write a single register value. The M41T81S does support multi-byte reads and writes, but you can get away with single register read/writes, albeit less efficiently, potentially.

Writing to a register on the M41T81S is similar to writing to any other I2C device. Start a session, with the device address, with intent to write. Write the register address, no stop. Write the register value, stop.

```
int RTC_Write (unsigned char Address, unsigned char Value)
{
  // send device address, intent to write
  if (I2C_SendAddressRW (RTC_ADDRESS, I2C_WRITE, I2C_WAIT))
    return -1;

  // write register address, more data, so no stop
  if (I2C_WriteByte (Address, I2C_NOSTOP))
    return -1;

  // write data, not more data, so stop
  if (I2C_WriteByte (Value, I2C_STOP))
    return -1;

  return 0;
}
```

The other functions in your library can use this function to write values to registers. Any errors should cause the function to abort.

Reading a register requires that you announce the device address with write intent, write the register address that you want to read from, no stop, then issue a restart so you can change the direction of the data flow. Announce the device address, this time with intent to read. Read a byte, no ACK, issuing a stop:

```
unsigned char RTC_Read (unsigned char Address)
{
  unsigned char bRet = 0;

  // send device address, intent to write
  if (I2C_SendAddressRW (RTC_ADDRESS, I2C_WRITE, I2C_WAIT))
    return 0xFF;

  // write register address, more data, so no stop
  if (I2C_WriteByte (Address, I2C_NOSTOP))
    return 0xFF;

  // restart to change data direction
  I2C_IssueRestart ();

  // send device address with intent to read
  if (I2C_SendAddressRW (RTC_ADDRESS, I2C_READ, I2C_NOWAIT))
    return 0xFF;

  // read the register, reading only 1 byte, so NACK,
  //  also, done, so free bus with stop
  if (I2C_RXByte (&bRet, I2C_NACK, I2C_STOP))
    return 0xFF;

  return bRet;
}
```

This function has been simplified and uses the return value to both indicate an error condition (0xFF), or the data received. This makes the function easier to use in your library but can't distinguish between an actual error and legitimate data. It would be better to use a pointer argument to manage the data, but the function would be slightly more cumbersome to operate. Build it to the level you are comfortable with!

Most I2C devices operate with the same method, or a mild variation of the above. You always need to look at the timing and bus diagrams to ensure that you are following the correct communications procedures. This is especially true for expectations around acknowledgements.

Now that you have the basic register read and write functions, you should be able to create the rest of the library for the RTC.

The MPL3115 Precision Altimeter

The MPL3115 is an interesting device, as it is tiny and very high-resolution. In addition to pressure sensing, it also returns temperature data. This device is configurable to generate interrupts to notify of certain conditions, but you will use it in polling mode.

The MPL3115 is more complex than the LTC2633, as it has many configuration registers, and more complex data to read. In many ways, this chip is like a micro itself.

Unlike the DAC, the MPL3115 needs to be initialized before it can be used. Register 0x26 (Control Register 1) is of interest (Page 32/7.17). We want the part active, with maximum oversampling. Register 0x13 (PT_DATA_CFG) is also of interest (Page 28/7.7.1). We want flags for Pressure/Altitude and Temperature.

The initialization order will be:

- set flags (0x13)
- set oversampling and set active (0x26)

We can check for a ready temperature/pressure reading with the status register (DR_STATUS 0x00/0x06) (7.1.2). Look at the PDR/TDR bits to see if a new sample for pressure/temperature is ready. Pressure/Temperature data is found in registers 0x01 to 0x05, being careful to note position, component, and alignment of the data bits:

[image of returned data]

The format of the data is clever, as the device returns the data in signed and unsigned components. For pressure, the left-aligned data contains 18 bits of signed data for the integer pressure component, then 2 bits of unsigned data for the fractional component in ¼ Pascals.

The temperature component is represented as 8 bits of signed data for the integer temperature component, then 4 bits if unsigned data for the fractional component in 1/16th degrees C.

To make these results useful, we can add the fractional and signed components together with separate expressions to create a float result.

For example, the temperature can be converted to a float by adding R4 as a char to R5 shifted right four positions * 1/16f. This produces a float, which is signed and contains a fractional component.

You may operate this device successfully with a simple single byte read/write scheme. Single byte read/writes will cause more I2C traffic then multi-byte read/writes but makes the library simpler. If you want to write more efficient communications functions, you should do so, but only after you have the device working with the simpler scheme.

As a minimum, you should have the following functions in your MPL3115 library:

[MPL3115 header]

The 24AA512 EEPROM

This device has not yet been formally included in this course.

The LSM303 eCompass (3D Accelerometer + 3D Magnetometer)

T