Flexis® Wiring Framework API

05/19/11

Jim Donelson

Table of Contents

Appendix

Index of Tables

1 Introduction

The framework is for those who are ready for the next step. Engineers, students and hackers that are ready to learn to program using real ANSI C, the language of the MCU. Recent surveys have indicated that ANSI “C” is the predominate language used in this domain. And for a good reason, it is a very efficient and a powerful weapon. It is really a substitute for assembly language and if used well it can end up being more efficient and has a high degree of portability.

Using the Firebird32 Family and Codewarrior from Freescale provides an unparallelled opportunity to learn MCUs inside out with a well supported, full symbolic debugger at a very low entry cost. A single click will flash the device and prepare the environment. As you single step through the code you can not only view all variables, but also the view the actual assembly language as well as all the registers.

While C++ offers some real advantages in other domains, it is really not for this type of development. For one thing, there is a certain amount of overhead to using real object oriented design. Also, to really leverage the power of C++, dynamic memory allocation is required, and this is generally not a good idea on a machine that has so little ram. I personally don't even use malloc.

Although the term “C/C++” is used quite often, be aware that these are really two different languages that only share a degree of syntax similarity.

Assuming are you are not an artist interested in flashing LEDS your goal is to have faultless code that never ever crashes and can run virtually indefinitely with out a fault. To do this requires careful consideration of how resources are being used (the main reason I don't use malloc), attention to detail and a fair amount of testing.

Like all powerful weapons, you are able to shoot your self in the foot, so if you are unfamiliar with “C”, as you proceed please take the time to really learn “C”. The internet is rich with information on this subject.

Many of the wiring framework functionality is derived from the standard “C” libraries and offers a richer selection than the standard wiring “language” does. Again, very well tested and document libraries.

1.1 Why a Framework?

A framework provides an easy pre-tested path to getting your application up and running sooner. As with all frameworks, some compromises must be made to facilitate both ease of use and broad flexibility. Too flexible and it becomes difficult to configure and use. Not flexible enough and it will not have a broad application base.

This framework is loosely based on the Wiring/Processing library, but does not use and C++ at all. C++ introduces large penalties in storage and time and is generally not used in this type of application (“bare metal” hard realtime).

It is designed and tested with Freescales Eclipse based Code Warrior, which as of this writing is at version 10.1.

It is assumed in this document that reader has used the “C” programming language and no attempt is made in this document to explain the “C” language.

The framework is provided as “C” and “H” files, that are to be included in the Eclipse project, but not copied into the workspace. If you decide to modify the frameworks, be aware that this may cause a problem with future updates and merging. There follows a more detailed explanation of how this all works.

Some of the functionality is provided by the standard “C” libraries. For example “math.h” is used to provide trig functions and other math functions.

1.2 Project Structure

This section explains the directory structure and components required to create a framework project from scratch. ***If you are using a provided sample project, you need not do any of this.***

The projects relies on the files being in certain places and use relative paths so that the Framework directories are above workspaces. This makes it easy for several workspaces to use them.

As this framework is similar to the Wiring/Processing language, the functions **`loop`**() and **`setup`**() are used. The function **`setup()`** is called at the start of the program, and each time **`loop`**() returns it is called again. You are certainly not obliged to use this set up, but the examples are geared to this. I have found it is really quite convenient to use this structure.

There is also a third function I have added which must be defined, **`RTCTickCallback()`**, which is called at 1 millisecond intervals. This is quite handy for performing periodic tasks, but note that it is called in the context of an interrupt and there are do's and don'ts that should be followed. It must be understood that since it is called at 1 millisecond intervals, if the function took 1 millisecond to execute it would consume 100% of the cpu's time, and lock out other interrupts, 500us would take 50% and so on.

As will be mentioned later, it is important to understand that there are two basic way a file can be added to a project. They can be added using copy or as links. Some files should be added as links, like the framework proper files, as this way when you update the framework all you projects will see the new files. Other files, such as config.h, should be copied as if you modify it for the project, would not want to change the default one.

Also if you are starting a project from a sample file, you would copy it as again you would be modifying it.

Another important thing to know that that if linked files are deleted from a project, only the link is deleted not the file. If a file that is in a project directory is deleted, the file is actually deleted. Files that are copied into a project directory are automatically added to the project.

After adding or removing a file from a project, the project must be cleaned and rebuilt, or you will get errors about the make file. When you clean the project, the makefiles are re-created.

Always Clean and Rebuild the project after adding or removing files or include paths!

1.2.1 Directory Structure

The framework uses a certain directory structure:

--Project Directory (e.g C:\CWProjects)

|--- Framework (the framework files C:\CWProjects\Framework)

|---Framework/Headers (C:/CWProjects/Framework/Headers)

|---Framework/Sources (C:/CWProjects/Framework/Sources)

|---Framework/Samples (C:/CWProjects/Framework/Samples)

|---WORKSPACE

|---ECLIPSE PROJECT DIRECTORY

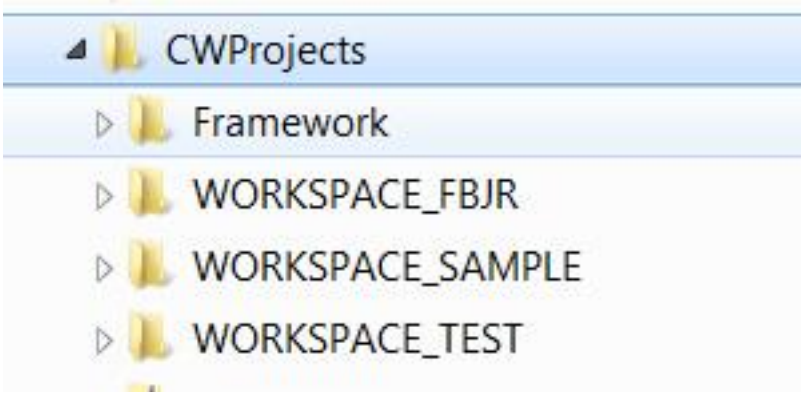

Illustration 1: Directory Structure

1.2.2 CONFIG.H and ALL.H

The file CONFIG.H must be in the projects Project_Headers folder, and should be added as a **copied** file. There are standard ones provided or it can be customized. Please see comments in the file as new features will be added.

ALL.H is an include file that includes all the other framework include files. It should be included in all source files that use the frameworks. The added include paths will find this file.

1.3 I/O Channels

The Framework uses I/O channels for digital I/O, analog I/O and interrupts. The digital channels are configurable both internally and by the user. Analog and interrupt channels are predefined, and assigned to to the MCU pins that support there function. Since almost all MCU pins can also be digital I/O there is a flexible method of assignments of logical channel to digital MCU pins.

The assignments for digital I/O to physical MCU pins is accomplished using PIO (port i/o) tables.

1.3.1 Digital Channels

The assignments for digital I/O to physical MCU pins is accomplished using PIO (pin i/o) tables. There are three predefined tables, for the Firebird32, the Nano32 and the Mini32. The term “pin” actually refers to a digital channel, which in turn maps to a MCU pin.

See the appendix for you board.

```
// NO NEED TO DO THIS FOR STANDARD BOARDS.
// For customizing only!!!!

BEGIN_PIO_TABLE(mytable)
PIO_ENTRY(D_PORTE,BIT2) // PTE1 TX
PIO_ENTRY(D_PORTE,BIT0) // PTE0 RX
PIO_ENTRY(D_PORTG,BIT3) // PTG3
PIO_ENTRY(D_PORTE,BIT2) // PTE2 TPM1CH0
END_PIO_TABLE(mytable)
```

Then, to cause the framework to use this table:

```
      setPIOTable(PIO_TABLE(mytable));
      pinMode(0,OUTPUT);  // Use the first channel in my table.
      digitalWrite(0,1);
```

It is important to use these macro as it will insulate the code from future implementation changes in the internal pio table structure.

See Also Table *PWM Channels* and Table *Interrupt Channels* for a listing of these per-set channels.

1.3.2 Analog Channels

The analog channels are pre-set to the analog channels supported in the MCU. There is no advantage to changing these. Please see the Table in chapter 2 in the description of the Analog Read API function.

The number of bits returned from an analog read can be configured in CONFIG.H, however the frame always reads 12 bits.

1.3.3 PWM Channels

The PWM channels are preset to the MCU timer channels that support PWM. Please the read the description in the Analog Write API description.

1.4 setup(), loop() and RTCTickCallback()

These three functions are the starting point for all applications. Below is the preferred method for blinking the LED, as delay() stalls loop() and prevents other perhaps critical code from running.

```
/*
 * loop_blink_fb32.c
 */

#include <hidef.h>      /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */

// All.h includes all the include files for the framwork.
#include "ALL.H"

// Function Prototypes
void RTCTickCallback(void);
```

```
void setup(void);
void loop(void);

int counter = 0;          // Determine when 1/2 a second is up.
int  blink_state=0;       // Toggles with XOR

// Called every millisecond
void RTCTickCallback(void)
{
      ++counter;          // Increment the 1 ms counter.
      if(500 == counter ) // 1/2 second yet ?
      {
            counter = 0;        // reset the counter so we start over
            blink_state ^= 1;   // Use the "C" xor operator to toggle

            // write to the LED.
            digitalWrite(13,blink_state);
      }
}
// Called first to perform and one time initialization
// Interrupts are already enabled.
void setup(void)
{
      pinMode(13,OUTPUT);
}
void loop(void)
{
   // No need to do anything here!
}
```

2 API Reference

2.1 Digital I/O

Digital I/O is used to read and write bits and ports. Bits are referenced to by their pin numbers, and must first have their direction set (called their mode, as in pinMode). A port is a group of pins, and current two pre-defined ports are supported, 0 and 1. And enhanced function has been added that allows defining a group of bits as a port. This is very useful as often a complete hardware port is not available due to special functions that may be on a given port.

2.1.1 pinMode() - Set the direction of a pin.

`void pinMode(int pin, int mode)`	
int pin	Selects which pin
int mode	Set the mode, INPUT or OUTPUT.

The pin refers to a logical pin number which in turn maps to a phsyical pin number. See the Appendix A for pin assignments. This is done to set up an I/O bit, and is required before use. After the mode has been set, `digitalRead()` and `digitalWrite()` may be used. Pins set for output may also be read from. Pins set for input will have the pull resistor enabled. See the Appendix for a definition of pins to port bit.

Example:

```
pinMode(0,OUTPUT);
digitalWrite(0,1);
```

2.1.2 pinModeEx() - Advanced pin mode setting for Flexis.

`void pinModeEx(int pin, int config)`	
int pin	Selects which pin
int config	See table

This is only for Flexis parts. Note that mode bit that are not enabled will be reset.

Define	Function
INPUT	Set pin to input
OUTPUT	Set pin to output
HI_DRIVE	When high drive is selected, a pin is capable of sourcing and sinking greater current. Even though every I/O pin can be selected as high drive, the user must ensure that the total current source and sink limits for the MCU are not exceeded. Drive strength selection is intended to affect the DC behavior of I/O pins. However, the AC behavior is also affected. High drive allows a pin to drive a greater load with the same switching speed as a low drive enabled pin into a smaller load. Because of this, the EMC emissions may be affected by enabling pins as high drive
SLEW_RATE	Slew control limits the rate at which an output can transition to reduce EMC emissions. Slew rate control has no effect on pins that are configured as inputs
INPUT_FILTER	Enable a input low pass filter (not available on S08)
PULL_UP	Enable internal pull up. The pull-up device is disabled if the pin is configured as an output

Table 1: PinModeEx modes

Example:

```
     pinModeEx(3,OUTPUT | HI_DRIVE | SLEW_RATE);  // Set Hi drive and Slew
rate
   pinModeEx(3,OUTPUT);                          // Rest them
```

2.1.3 digitalWrite() - Set the state of a pin

void digitalWrite(int pin,int data)	
int pin	Selects which pin is affected
int data	0 or 1 (`LOW` or `HIGH`)

Outputs the data to the pin specified. Pin must have been set to output using `pinMode` previously.
Example:

```
int led_state = 0;
pinMode(13,OUTPUT);
digitalWrite(13,led_state);
```

2.1.4 digitalRead() Read the state of pin

int digitalRead(int pin)	
int pin	Selects which pin is read.
Returns:	0 or 1

Returns the state of the pin. The pin must have previously been set to `INPUT` using `pinMode`. A pin set to output can also be read, but it well just read back the last value written. Note that pins set to output can still be read back at any time.

Example:

```
byte switch;
pinMode(3,INPUT);
switch = digitalRead(3);
```

2.1.5 portMode() - Initialize an 8 bit port

void portMode(int port, int direction)	
int port	Selects which port is affected 0 or 1
int data	INPUT or OUTPUT

Port 0 is pins 0-7 while port 1 is pins 8-15. The port operations allow reading and writing of multiple bits.

```
portMode(0,OUTPUT); // bits 0-7 to output
portWrite(0,0xff);  // Write all 8 bits to 1
```

2.1.6 portRead() - Read an 8 bit port

int portRead(int port)	
int pin	Selects which port is read.
Returns:	0-255

Reads 8 bits from port 0 or 1.

```
byte indata;

portMode(1,INPUT);
indata  = portRead(1);
```

2.1.7 portWrite() - Writes 8 bits to a port.

void portWrite(int port, int value)	
int pin	Selects which port 0 or 1
int value	0 - 255

Write 8 bits to port 0 or 1.

```
portMode(0,OUTPUT); // bits 0-7 to output
portWrite(0,0xff);  // Write all 8 bits to 1
```

2.1.8 portModeEx() - Initialize a user defined port.

void portModeEx(byte* pins,int direction, int bits)	
byte* pins	Address of an array that defines which pins are used for this logical port.
int direction	INPUT or OUTPUT
int bits	How many bits are used.

The user defined array names which pins are in the port. The pins go from LSB to MSB in the array.

Example:

```
      // Define the "port"
      byte my_port[] ={3,4,6,7};              // pin 3 is the LSB, pin 7 is
the MSB

      portModeEx(&my_port[0],OUTPUT,4); // There are 4 bits in the port.
   portWriteEx(&my_port[0],8,4);     // This would set bits 3,4,6 to '0' bit 7
to '1'
```

2.1.9 portWriteEx()

void portWriteEx(byte* pins,int data,int bits)	
byte* pins	Address of an array that defines which pins are used for this logical port.
int data	Data to be written.
Int bits	How many bits are used.

Write to a user defined port defined by an array of bytes.

```
      // Define the "port"
      byte my_port[] ={3,4,6,7};              // pin 3 is the LSB, pin 7 is
the MSB

      portModeEx(&my_port[0],OUTPUT,4); // There are 4 bits in the port.
   portWriteEx(&my_port[0],8,4);     // This would set bits 3,4,6 to '0' bit 7
to '1'
```

2.1.10 portReadEx()

int portReadEx(byte* pins, int bits)	
int pins	Address of an array that defines which pins are used for this logical port.
int bit	How many pins are in the array.
Returns:	The value of the port.

2.2 Analog IO

2.2.1 analogRead() - Input an ADC Channel

int analogRead(int ch)	
int ch	Selects which logical channel is read.
Returns:	0 to maximum.

Certain pins are analog input pins, and reading them will return the current value of the voltage on that pin. Note that analog pins can also be used as digital I/O, but if the `analogRead` call is made any previous `pinMode` setting will be forgotten and it will be used as an analog input.
By default 10 bits are read, but this can be changed in config.h. 0 is 0 volts and 1023 is 5 volts.

Logical Analog Channel	MCU Function	ADC CH.
0	PTB0/ADP0	A0
1	PTB1/ADP1	A1
2	PTB2/ADP2	A2
3	PTB3/ADP3	A3
4	PTB4/ADP4	A4
5	PTB5/ADP5	A5
6	PTB6/ADP6	A6
7	PTB7/ADP7	A7
8	PTD0/ADP8/ACMP+	A8
9	PTD1/ADP9/ACMP-	A9
10	PTD3/ADP/10KBIP3	A10
11	PTD4/ADP11	A11
26	Temp Sensor	A26
27	Internal Band Gap	A27

Table 2: Analog Channel Assignments

Temp = 25 - ((VTEMP -VTEMP25) / m)

```
#define POT_PIN     0   // Use channel 0.
      int val;
      val = analogRead(POT_PIN);
```

2.2.2 analogWrite() - PWM output

void analogWrite(int ch,int data)	
int pin	Selects which pin is affected
int data	0 to PWM_MAX

Writes to a PWM pin and sets the duty cycle.

Channel	MCU Function	FB32	NANO32	MINI32
0	PTE2 TPM1-CH0	D3	E2	D3
1	PTE3 TPM1-CH1	D4	E1	
2	PTF0 TPM1-CH2	D5	F0	D5
3	PTF1 TPM1-CH3	D6	F1	D9
4	PTF2 TPM1-CH4	D7	F2	
5	PTF3 TPM1-CH5	D11	F3	
6	PTF4 TPM2-CH0	D8	F4	D8
7	PTF5 TPM2-CH1	X		X

Table 3: PWM Channels

```
// Set PTE2 to 50% duty cycle
    analogWrite(0, 512);
```

2.3 Time functions

2.3.1 micros() - Return the clock in microseconds

`long micros(void)`

Returns: A 32 bit value that is the clock time in microseconds.

The clock is a counter that starts at reset.

```
    long startime = 0;
```

```
loop()
{
       // something like this is much better that delay!
       if(0 == starttime)
              starttime = micros();
       // Always use >= or you might miss the exact time.
       if( (micros() - startime) >= 500 )
       {
              starttime = micros();
              // do what you needed to do every 500 us.
       }
}
```

2.3.2 millis() - Return the clock in milliseconds

`long millis(void)`

Returns:	A 32 bit value that is the clock time in milliseconds.

The counter is return in milliseconds.

```
long startime = 0;
loop()
{
       // something like this is much better that delay!
       if(0 == starttime)
              starttime = millis();
       // Always use >= or you might miss the exact time.
       // Wait for 500 milliseconds
       if( (millis() - startime) >= 500 )
       {
              starttime = millis();
              // do what you needed to do every 500 milliseconds
       }
}
```

2.3.3 delay() - Delay in milliseconds

void delay(long delay)	
long delay	Delay in milliseconds

This does not return until the delay is up, so it will stall the thread. Not advised on interrupt threads.

```
int ledPin = 13;                 // LED connected to digital pin 13
// NOTE: This works fine but is not the recommended method.
void setup()
{
  pinMode(ledPin, OUTPUT);      // sets the digital pin as output
}

void loop()
{
  digitalWrite(ledPin, HIGH);   // sets the LED on
  delay(1000);                  // waits for a second
  digitalWrite(ledPin, LOW);    // sets the LED off
  delay(1000);                  // waits for a second
}
```

2.3.4 delayMicroseconds() - Delay in microseconds

void delayMicroseconds(int delayus)	
long delayus	Delay in microseconds

This does not return until the delay is up, so it will stall the thread. Not advised on interrupt threads.

```
int outPin = 8;                 // digital pin 8

// NOTE: This works but is not recommended

void setup()
{
  pinMode(outPin, OUTPUT);      // sets the digital pin as output
}

void loop()
{
  digitalWrite(outPin, HIGH);   // sets the pin on
  delayMicroseconds(50);        // pauses for 50 microseconds
  digitalWrite(outPin, LOW);    // sets the pin off
  delayMicroseconds(50);        // pauses for 50 microseconds
```

```
}
```

2.3.5 pulseOut() - Generate a pulse stream

void pulseOut(int pin,int frequency,int nPulses)	
int pin	Can only be pin 8
int frequency	Frequency in Hertz
int nPulses	How many pulse 0 = infinite.

This function uses TMP2 CH0 to generate frequencies and pulse bursts. This is found on output D8. It can generate frequencies from 1 Hz to about 4.5 MHz, however at higher frequencies repetitive pulse bursts can consume too much overhead as in the counted mode (ie. NPulses > 0) some interrupt overhead is used.

```
// Send 10 1Khz pulse out
pulseOut(8,1000,10);
```

2.3.6 pulseStop() - Stop the pulse stream

void pulseStop(int pin)	
int pin	Can only be pin 8

This will shut off the the pulse stream.

```
// Start a continuous pulse stream.
pulseOut(8,1000,0);
// End it
pulseStop(8);
```

2.3.7 pulseRunning() - Return the remaining pulses

int pulseRunning(int pin)	
int pin	Can only be pin 8
Returns:	Number of pulses remaining 0 is infinite.

Returns the number of pulse remaining when a count is given.

```
// Send 100 1Khz pulses out
pulseOut(8,1000,100);

   // wait for them to all go out.
   while(pulseRunning(8) )
     ;
```

2.3.8 pulseIn()

Currently not implemented.

2.3.9 singleshotPulse() - Send a pulse

void singleshotPulse(int ch, dword time_us)	
int ch	Any PWM channel (see chart in analogWrite)
int time_us	Amount of time to send the pulse.

This is an enhancement to the wiring specification. It is intended for driving RC servos, but of course can be used for any application that requires a precision repeatable pulse. Each time it is called it will emit a single pulse of the specified duration. See `analogWrite` for a table of channels.

```
// Send a 1.5 ms pulse on channel 0
   singleshotPulse(0, 1500);
```

2.4 Interrupts

Interrupts can be used to count pulses, measure pulses and signal external events.

2.4.1 attachInterrupt() - Configure an interrupt

void attachInterrupt(int intno,PFNCALLBACK cb,int mode)	
int intno	Interrupt number (see table)
PFNCALLBACK cb	Pointer to callback function
int mode	Mode (see mode table)

Sets up to cause an interrupt on an input event, edges and levels on the pins shown in the table.

Number	Function	Modes	FB32	NANO	MINI
0	PTE2 TPM1-CH0	R,F,C	Pin 3		D3
1	PTE3 TPM1-CH1	R,F,C	Pin 4		D5
2	PTF0 TPM1-CH2	R,F,C	Pin 5		
3	PTF1 TPM1-CH3	R,F,C	Pin 9		D9
4	PTF2 TPM1-CH4	R,F,C	Pin 10		
5	PTF3 TPM1-CH5	R,F,C	Pin 11		
6	PTF4 TPM2-CH0	R,F,C	Pin 8		
7	PTF5 TPM2-CH1	R,F,C	Speaker		
8	PTG3	H,L,R,F	Pin 2		
9	PTG2	H,L,R,F	Pin 4		
10	PTG1	H,L,R,F			
11	PTG0	H,L,R,F			
12	PTD2	H,L,R,F			D4
13	PTD3	H,L,R,F			
14	PTB5	H,L,R,F			A2
15	PTB4	H,L,R,F			A1

Table 4: Interrupt Channels

LOW	Logic 0
HIGH	Logic 1
CHANGE	Both Edges
RISING	Rising Edge
FALLING	Falling Edge

Table 5: Interrupt Modes

```
// Measure the period of a pulse.
long lastcount=0;
long period;
void interruptCallback(void)
{
```

```
      long m = micros();
      period = m - lastcount;
      lastcount = m;
}
void setup(void)
{

      // Apply a pulse to pin 3 to test this.
      attachInterrupt(0,interruptCallback,RISING);

}
void loop(void)
{
}
```

Interrupt Example

2.4.2 detachInterrupt() - Remove interrupt

int detachInterrupt(int intno)	
int intno	The same interrupt number that was used in the attach.

Removes the interrupt from the interrupt channel.

2.4.3 interruptMode() - Change the mode of an interrupt

void interruptMode(int intno,int mode))	
int intno	The same interrupt number that was used in the attach.
int mode	See table of interrupt modes.

Modifies the mode of an interrupt that was previously attached to.

2.4.4 ENTER_CRITSEC() EXIT_CRITSEC() Macros

void ENTER_CRITSEC(void)

ENTER_CRITSEC()/EXIT_CRITSEC() is the preferred method for disabling interrupts when mutual exclusion is required for two reasons:

- It will restore the previous state of the interrupts. So if interrupts had been

disabled before a function requiring mutual exclusion the will remain disabled.

- It nests, meaning that the interrupts will not be re-enabled until the outer EXIT_CRITSEC() is executed. If you simply disabled the interrupts and were to call a function that also required mutual exclusion, it would return with interrupts re-enabled an the exclusion would have been lost.

It calls cli() and sti():

```
volatile byte SR_level = 0x00;
volatile word saved_SR = 0x00;
void CLI(void)
{
   // If this is the first call, the save SR and disable int.
   if (++SR_level == 1)
   {
      asm
      {
       move.w SR,D0;
       move.w D0,saved_SR;
       ori.l #0x700,D0;
       move.w D0,SR;
       }

   }
}

void STI(void)
{
   // If this is the last call, then restore SR.
   if (--SR_level == 0)
   {
      asm
      {
        move.w saved_SR,D0;
        move.w D0,SR;
      }

   }
}
```

In the following example, vdata can be modified by an interrupt so we need to insure not concurrent access to it when we modify it.

```
// vdata can be modified in an interrupt.
myfunction(struct* vdata)
{
      ENTER_CRITSEC();
      if( vdata->flag )
      {
            vdata->flag = 0;
```

```
            strcpy(vdata->str,savedata);
      }
      EXIT_CRITSEC();
}
```

2.4.5 interrupts() - Enable MCU interrupts

```
void interrupts(void)
```

This enables the MCUs interrupts. See **ENTER_CRITSEC().**

2.4.6 noInterrupts() - Disable MCU interrupts

```
void noInterrupts(void)
```

Disable the MCU's interrupts. See **EXIT_CRITSEC()**.

3 Extension API

3.1 Serial Ports

Both SCI1 and SCI2 (ports 0 and 1) are supportd. Each port you plan to use must first be initialized. The driver is fully interrupt driven, both send and receive and uses 80 byte buffers.

3.1.1 Serial_Init()

void Serial_Init(int port)	
int port	Selects which is initialized. 0 or 1

This initializes the ports and sets the baud rate to 9600 8,1.

```
Serial_Init(SCI1);
```

3.1.2 Serial_setbaudrate()

byte Serial_setbaudrate(int port,byte baud)	
int port	Selects which port is affected 0 or 1
int baud	Baud rate selector e.g. `BAUD_115200`
Return:	0 if no error, `BAD_BAUD_RATE` if undefined baud rate.

Sets the baud rate for the port.
Available baud rates:

BAUD_1200
BAUD_2400
BAUD_4800
BAUD_9600
BAUD_19200
BAUD_38400
BAUD_57600
BAUD_115200

Table 6: Serial Baud Rates

```
Serial_Init(SCI1);
Serial_setbaudrate(SCI1,BAUD_19200);
```

3.1.3 Serial_rxbyte()

byte Serial_rxbyte(int port, byte rxdata)	
int port	Selects which port
Byte rxdata	Pointer to data to receive.
Return:	0 if character available, `RX_EMPTY` if no receive data.

Checks the buffer to see if a character is available. Returns 0 if available and puts the character in to rxdata. Does not wait.

3.1.4 Serial_txbyte()

byte Serial_txbyte(int port, byte data)	
int port	Selects which port
byte data	Pointer to data to receive.
Return:	0 if no error, `TX_FULL` if tx buffer is full.

Serial Port Example Code

```
      // General form of the calls.
      byte rxdata;                          // Variable to receive a
character.
      byte rc;                              // return code.
      Serial_init(SCI1);                    // Must initialize
      Serial_setbaudrate(SCI1,BAUD_115200); // Set the baud rate to 115200
      Serial_txbyte(SCI1, 'A');             // Send the character 'A'
      rc = Serial_rxbyte(SCI1,&rxdata);     // See if an rx character is
there.

      // Was there a character waiting?
      if(0 == rc )
      {
            // Yes, so echo it. Loop until there room in buffer.
            while(TX_FULL == Serial_txbyte(SCI1, rxdata))
                  ;
      }
```

3.2 I2C

3.2.1 I2CInit()

void I2CInit(void)	
Notes: This call also will un-hang any I2C device. With many I2C devices, if the MCU is reset or crashes while the device is sending, the device will still need to finish sends a byte. This call manually clocks away this remaining data.	

The baud rate is initially set to 100KHz. Different devices can handle higher baud rates depending upon the devices specification, Vdd voltage and length of connection wires.

3.2.2 I2CSetBaudrate()

byte I2CSetBaudrate(int ratekhz)	
int ratekhz	Sets the I2C Bus baud to the approximate value requested
Returns	ERROR_NONE, ERROR_BUSBUSY

You can't set the baud rate while in a transaction.

```
// Wait until not busy to set the baud rate.
while (I2CSetBaudrate(300) ==  ERROR_BUSBUSY)
      ;
```

3.2.3 I2CSetDeviceAddress()

void I2CSetDeviceAddress(byte address)	
byte port	Set the I2C Device address. Generally from the spec sheet of the device.

3.2.4 I2CSend()

byte I2CSend(byte* data,word size)	
byte data	Pointer to data to send.
word size	How many bytes to send
Return:	ERROR_NONE,ERROR_BUSBUSY

3.2.5 I2CRecieve()

byte I2CRecieve(byte* data,word size)	
byte* data	Pointer to receive buffer.
word size	How many bytes to send
Return:	ERROR_NONE,ERROR_BUSBUSY

3.2.6 I2CSetDeviceAddress()

void I2CSetDeviceAddress(byte address)	
byte port	Sets the I2C Device address. Generally from the spec sheet of the device and the schematic of the board.

Some I2C Devices have address lines that allow modifying the the devices address, so you need to look at both the specification sheet for the device and the schematic of the board.

3.2.7 I2CGetRemainingSendBytes()

word I2CGetRemainingSendBytes(void)	
Returns:	Bytes left to be sent.

This returns how many bytes are left in the I2C send buffer.

3.2.8 I2CGetRemainingRecieveBytes()

word I2CGetRemainingRecieveBytes(void)	
Returns:	Bytes left to be received.

3.2.9 I2CGetLastError()

byte I2CGetLastError(void)	
Returns:	Returns last error

This call helps because certain errors will not be detected when a call is initially made. It is not cleared until read, so it is a good idea to read it before and after a transaction.

ERROR_NONE	No error
ERROR_BUSBUSY	Busy with a current transaction
ERROR_NAK	Did not receive a ACK (normal if device had no more data to send) Could indicate no device or you used to wrong address
ERROR_ARB_LOSS	Another master was detected.

Table 7: I2C Errors

```
// 24C32 I2C EEPROM - 32 byte pages
// Address line A1 high, A0 and A2 low.
#define I2C_EEPROM_ModuleAddress 0xA4  // One of the address set bit is high.

typedef struct _eeprom_write {
      word address;
      byte data[32];
} EEPROM_WRITE_BLOCK;

EEPROM_WRITE_BLOCK eeprom_write_data;

setup()
{
      I2CInit();
      I2CSetDeviceAddress(I2C_EEPROM_ModuleAddress);
      eeprom_write_data.address = 0;

      for( i=0 ; i<32 ; ++i)
```

```
            eeprom_write_data.data[i] = i;

      interrupts();
      I2CSend((byte*) &eeprom_write_data,sizeof(EEPROM_WRITE_BLOCK));
}
```

3.3 Button

There is a button the Firebird32 connected to PTG0. These APIs use a a callback for each time it changes state. See example at end of section.

3.3.1 BUTTONInit()

void BUTTONInit(void))	
Returns:	Nothing

This initializes the input and sets up the de-bounce timer.

3.3.2 BUTTONSetCallback()

void BUTTONSetCallback(void(*func)(int state))	
func	Pointer to receive buffer.
Return:	Nothing

The state is 0 when the button is pushed.

```
#define LED        13   // Pin number for the LED

// Our call back function prototype.
void OnButtonChanged(int state);
```

```
// This is called each time the button changes state.
// state is the de-bounced state of the button
void OnButtonChanged(int state)
{
      // Turn the LED off or on based on the state
      // of the button.
      if(!state)  // If pushed, turn button off
            digitalWrite(LED,0);
      else
            digitalWrite(LED,1);
}

// Initialize for the button and LED.
void setup()
{
      BUTTONInit();
      BUTTONSetCallback(OnButtonChanged);
      pinMode(LED,OUTPUT);

}
```

3.4 LCD

If you connect the LCD as follows, it will work with the framework. This is how the LCD is connected on Wytec boards.

This assumes an HD44780 or compatible LCD module, 4 bit interface.

PTA0	RS (data/command) (Pin 4 on LCD module)
PTA1	EN (Write pulse) (Pin 6, also called E)
PTA2	Data 4 (Pin 11)
PTA3	Data 5 (Pin 12)
PTA4	Data 6 (Pin 13)
PTA5	Data 7 (Pin 14)

Table 8: LCD Port A Connection

Pins 1 & 5	should be 0V on the LCD module
Pin 5	is R/W and we only use W so ground it
Pin 3	should go to ground thru a 1K resistor
Pin 15	is the positive side of the back light LED and can be connected to +5
Pin 16	is the ground side of the back light LED, and should go to ground thru a 100 OHM or higher resistor. You could also hook up a PWM signal here thru a transistor for variable back light. (note: The resistor can be on either side of the LED. You MUST have a resistor)

Table 9: Other LCD Connections on the LCD connector (see devices documentation)

3.4.1 LCDInit()

`void LCDInit(int width)`	
int width	The width of each line in the LCD.

This initializes the ports and sets the width. Setting the width will allow the code the clear a line first.

3.4.2 LCDGetWidth()

`int LCDGetWidth(void)`	
Returns:	The width that was set.

3.4.3 LCDWriteLine()

`void LCDWriteLine(char* string, byte line)`	
char* string	Pointer to data buffer.
byte line	Which line to write it to – up tp 4 lines.

3.5 Keypad

These API's support the Firbird32 standard keypad that connects to J5 on the main board.

The standard keyboard is wired like this to Port D:

MCU PIN	Col. 1 PTD0	Col. 2 PTD1	Col. 3 PTD2	Col. 4 PTD3
Row 1 PTD4	**1**	**2**	**3**	**A**
Row 2 PTD5	**4**	**5**	**6**	**B**
Row 3 PTD6	**7**	**8**	**9**	**C**
Row 4 PTD7	*****	**0**	**#**	**D**

Table 10: Keyboard Connections

3.5.1 KPADInit()

`void KBRDInit(void)`	
Returns: Nothing	

This should be called before using the keyboard. It initializes port D and sets up a polling routine.

3.5.2 KPADReadChar()

byte KBRDReadChar(void)
Returns: key if one is waiting or KBRD_NOCHAR

There is no buffer, so the keypad should be polled often.

```
// Echo the keypad to the LCD.
setup()
{
   LCDInit(8,2);
   KPADInit();
}


void loop(void)
{
  char c;
  static char lcd_msg[8];
  static byte lcd_index = 0;

  // Echo the keypad to the LCD
  // Poll the keypad....
  if(KPAD_NOCHAR != (c = KPADReadChar()))
   {
      LCDClear();
      // If there less than 8 chars, then add this one
      if( lcd_index < sizeof(lcd_msg) - 1)
      {
        lcd_msg[lcd_index++] = c;
      }
      else
      {
        // Start over with a new line.
        lcd_index = 0;
        lcd_msg[lcd_index++] = c;
      }
      lcd_msg[lcd_index] = 0;
      LCDWriteLine(lcd_msg, 0);

    }

}
```

3.6 Console printf Support

The printf function can be directed to either serial port, the LCD, the debug console or any combination of them. There are various CONFIG.H setting and the device must be initialized.

```
void SetActiveConsole(int cons);
```

3.7 Speaker Support

The Firebird32 and Nano32 both have speakers.

Appendix A Firebird32 Channels and Pins

FB32 PIN	DIGITAL CHANNEL	INTERRUPT ANANLOG PWM	FUNCTION
D0	0 (RX)		PTE1 J6-1 SCI or Digital
D1	1 (TX)		PTE0 SCI or Digital
D2	2	I6	PTG3 Digital/Interrupt
D3	3	I0 P0	PTE2 TPM1-CH0 PWM Motor1/RGB LED
D4	4		PTD2
D5	5	I1 P1	PTE3 TPM1-CH1 (PWM)
D6	61 /30	I2 P2	PTF0 TPM1-CH2 PWM (J-D6)
D7	7		PTD1 Digital or Keypad
D8	8	P6	PTF4 TPM2-CH0 PWM J5-1 Digital RGB LED
D9	9	I3 P3	PTF1 TPM1-CH3 PWM
D10	10[1] / 31	I4 P4	PTF2 TPM1-CH4 PWM or /SS1 (J-D10)
D11	11	I5 P5	PTF3 TPM1-CH5 PWM or MOSI1 (J-D11)
D12	12		PTE4 MISO1
D13	13		PTE6 SPSCLK1 LED
A0	14	A0	PTB0 J6-2 X Axis
A1	15	A1	PTB1 J6-4 Y Axis
A2	16	A2	PTB2
A3	17	A3	PTB3
A4	18[1]	A4	PTB4 (J- A4)
A5	19[1]	A5	PTB5 (J- A5)
A6	20	A6	PTB6 (J6-5)
A7	21	A7	PTB7 (J6-6)
	22[2]	A8	
	23[2]	A9	
	24[2]	A10	
	25[2]	A11	
C6	26		PTC6/RXCAN PIN 1 of Power Connector
F7	27		PTF7/TXCAN PIN 2 of Power Connector
A4	28[1]		PTC1/SDA (J-A4)
A5	29[1]		PTC0/SCL (J-A5)
D6	30[1]		PTD0 (J-D6)
D10	31[1]		PTE7 (J-D10)
D11	32[1]		PTF3/MOSI1 (J-D11)
	33[3]		PTF5 TPM2-CH1 On board speaker
	34[3]		PTG1 RGB LED Enable, High True
	35[3]		PTG2 EEPROM WP

1 Jumper Selectable
2 see J4
3 Internal use, not pinned out.

	36-39		RESERVED

Table 11: FB32 Channels

J4 – Keyboard or General I/O

J4 PIN	DIGITAL CHANNEL	ANALOG CHANNEL	INTERRUPT	FUNCTION
1	100	A8		PTD0/ACMP+/ADP8
2	101	A9		PTD1/ACMP-/AD9
3	102		I12	PTD2/KBIP2/ACMPO
4	103	A10	I13	PTD3/KBIP3/ADP10
5	104	A11		PTD4/ADP11
6	105			PTD5
7	106			PTD6
8	107			PTD7
9	X			GND
10	X			5

Table 12: J4 Channels

Appendix B MINI Channels and Pins

Extension API

MINI PIN	DIGITAL PIN	ANALOG CHANNEL	INTERRUPT	FUNCTION	
1 TX1	0	X		PTE0 TXD1	
2 RX1	1	X		PTE1 RXD1	
3 A4	22/24	4		PTB1 or PTF7 (J3)	
4 A5	23/25	5		PTB0 or PTC6 (J4)	
5 D2	2	X		PTG3 KBIP7	
6 D3	3	X	0	PTE2 PWM	
7 D4	4	X	12	PTD2 KBIP2	
8 D5	5	X	1	PTE3 PWM	
9 D6	6	X		PTD0	
10 D7	7	X		PTD1	
11 D8	8	X		PTF4 FREQ/PWM	
12 D9	9	X	3	PTF1 PWM	
13 SCL	11	X		PTC0	
14 SDA	12	X		PTC1	
15 RX2	17	X		PTC5	
16 TX2	14	X		PTC3	
17 D10	10	X		PTE7	
18 MO	15	X		PTE5	
19 MI	16	X		PTE4	
20 SCK (LED)	13	X		PTE6	
21 A0	18	0		PTB5	
22 A1	19	1	15	PTB4 KBIP4	
23 A2	20	2	13	PTB3 KBIP5	
24 A3	21	3		PTB2	
Internal I/O	25	X		PTG2 EEPROM WP	
25 +5V					
26 RESET					
27 GND					

28 VIN					

Table 13: Mini I/O Channels

Appendix C NANO Channels and Pins

NANO PIN	DIGITAL CHANNEL	ANALOG CHANNEL	INTERRUPT	FUNCTION
1 B0	0	A0		PTB0 MISO2
2 B1	1	A1		PTB1 MOSI2
3 B2	2	A2		PTB2 SPSCK2
4 B3	3	A3		PTB3 /SS2
5 3V	X	X	X	
6 B4	4	A4		PTB4 KBIP4
7 B5	5	A5		PTB5 KBIP6
8 B6	6	A6		PTB6
9 B7	7	A7		PTB7
10 D0	8			PTD0 ACMP+
11 D1	9			PTD1 ACMP-
12 VRH				VREF
13 GND	X	X	X	X
14 D2				PTD2
15 D3	10	A10		PTD3 KBIP3 A10
16 D4	23	A11		PTD4
17 G3	2			PTG3 KBIP7
18 C0				PTC0 SCL1
19 C1				PTC1 SDA1
20				RESET

/RST				
21 C5				PTC5 RXD2
22 C3				PTC3 TXD2
23 F0	6			PTF0 TPM1CH1 PWM
24 F1	9			PTF1 TPM1CH3 PWM
25 F2	10			PTF2 TPM1CH4
26 F3	11			PTF3 TPM1CH5
27 F4	8			PTF4 TPM2CH0
28 C6				PTC6 RXCAN
29 F7				PTC7 TXCAN
30 E0	1			PTE0 TXD1
31 E1	0			PTE1 RXD1
32 E2	3			PTE2 TPM1CH0
33 E3	5			PTE3 TPM1CH1
34 E4	12			PTE4 MISO1
35 E5				PTE5 MOS1
36 E6				PTE6 SPSCK1 (LED)
37 E7				PTE7 /SS1
38 5V	X	X	X	
39 GND	X	X	X	
40 VIN	X	X	X	

Table 14: Nano Channels

Index