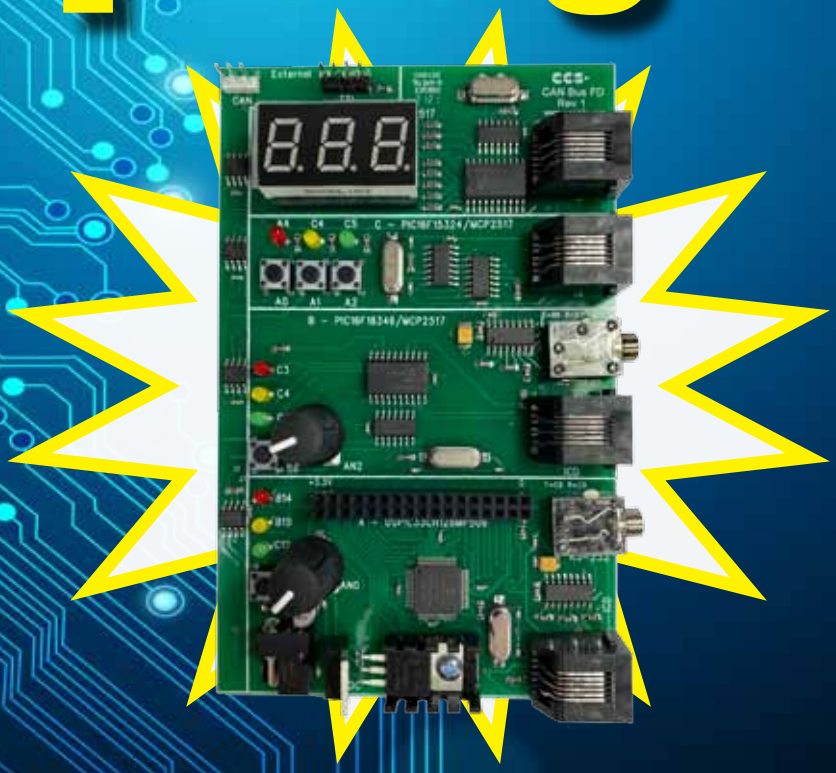
Quarter 3
2022

www.ccsinfo.com
262-522-6500

<Bits & Bytes> Newsletter

INSIDE THIS ISSUE:

Product Spotlight



CAN BUS FD PROTOTYPING BOARD

Node 1: dsPIC33CH128MP506 which includes an integrated CAN FD peripheral.
Node 2: PIC16F18346 which includes a MCP2517FD to connect to the CAN FD bus.
Node 3: PIC16F15324 which includes a MCP2517FD to connect to the CAN FD bus.
Node 4: PIC16F18346 which includes a MCP2516FD to connect to the CAN FD bus.

support@ccsinfo.com sales@ccsinfo.com

PIC16F1614 Family

The PIC16F1614 family of devices is currently the only PIC® MCU family that has a built in Math Accelerator with Proportional-Integral-Derivative (PID) Module. This module is a mathematics module that can perform a variety of operations, most prominently acting as a PID controller.

The module accomplishes the task of calculating the PID algorithm by utilizing user-provided coefficients along with a multiplier and accumulator. The main benefit of using the hardware module is for doing the PID calculation much faster than it can be done in software. For example when running the PIC® from the 32 MHz internal oscillator the built-in HW PID function was measured to take 12.8 us, compared to a software PID function which was measure to take 216 us. This is approximately 1/16 of the time.

The following are the built-in functions that have been added for the Math Accelerator with PID module:

- **setup_pid()** - used to setup the PID module and set the user-input coefficients.
- **pid_get_result()** - used to input the set point and feedback from the external system to the PID module, start the calculation, and to retrieve the result to input in the external system.
- **pid_read()** - used to read various PID module registers.
- **pid_write()** - used to write various PID module registers.
- **pid_busy()** - used to check if PID module is busy or not-busy, finished, with calculation.

The Math Accelerator with PID module can be setup for three basic functions PID calculation, 16-bit unsigned add and multiple and 16-bit signed add and multiple. Both of the add and multiple modes can also be setup to accumulate the output.

When setup for PID mode the user-input coefficients, K1, K2 and K3, are calculated from the three classic PID coefficients Kp, Ki and Kd with the following equations:

$$K1 = Kp + Ki \cdot T + \frac{Kd}{T}$$

$$K2 = -\left(Kp + \frac{2\mathrm{Kd}}{T}\right)$$

$$K3 = \frac{Kd}{T}$$

T is the sampling period.

The following is an example of how to setup and use the Math Accelerator with PID module in PID mode:

```
void main(void) {
   pid_struct_t PIDOutput;
   unsigned int16 ADCReading;
   signed int16 K1 = 7, K2 = -6, K3 = 0;
   unsigned int16 SetPoint = 500;
   unsigned int16 PWMDuty;

   setup_pid(PID_MODE_PID, K1, K2, K3);

   //Setup ADC
   setup_adc_ports(sAN3, VSS_VDD);
   setup_adc(ADC_CLOCK_INTERNAL);
   set_adc_channel(3);
```

```
   //Setup PWM 3
   setup_timer_4(T4_CLK_INTERNAL | T4_DIV_BY_32, 249, 1); //1ms period, from 32 MHz
   set_pwm3_duty(0); //0% duty
   setup_pwm3(PWM_ENABLED | PWM_OUTPUT | PWM_TIMER4);

   while(TRUE) {
      delay_ms(50);

      ADCReading = read_adc();
      pid_get_result(SetPoint, ADCReading, &PIDOutput);

      PIDOutput.u &= 0x07;

      if(PIDOutput.u >= 4)          //PIDOutput is negative, set PWMDuty to Minimum
         PWMDuty = 0;
      else if(PIDOutput.u != 0)     //PIDOutput > Maximum, set PWMDuty to Maximum
         PWMDuty = 1000;
      else if(PIDOutput.l > 1000)   //PIDOutput > Maximum, set PWMDuty to Maximum
         PWMDuty = 1000;
      else
         PWMDuty = PIDOutput.l;

      set_pwm3_duty(PWMDuty);
   }
}
```

When the Math Accelerator with PID module is setup for one of the add and multiple mode the operation is preformed as follows:

OUTPUT = (A + B) * C

The multiple value, C, is set with the K1 option that is passed setup_pid() function, and the two add values, A and B, are passed as the set_point and input parameters to the pid_get_result() function. For example the following is how to setup the Math Accelerator with PID module in add and multiply mode:

```
  int16 C = 100;
   int16 A, B;
   pid_struct_t Result;

   //setup for add and multiple mode and set multipler
   setup_pid(PID_MODE_UNSIGNED_ADD_MULTIPLY, C);

   //get add and multiple result
   pid_get_result(A, B, &Result);
```

The PIC16F1614 family of devices is available in the IDE compilers and the PCM command-line compilers starting with version 5.045. They come in 14 and 20 pin packages with flash memory of 4048 or 8192 instructions. Additionally they have four 8-bit timers, three 16-bit timers, 8 or 12 analog inputs, two CCP modules, two 10-bit PWM modules and 1 CWG module, which can be used in conjunction with other modules, CCP or PWM for example, to generate a Half-Bridge or Full-Bridge PWM.

CCS sells a simple Ethernet network integration device which can be embedded into any product or into industrial equipment, appropriately named EZ Web Lynx. Essentially it connects products or equipment to an HTML programmable website with no other protocol language skills needed! The website will allow the user to view the equipment status and the user can receive emails from the device. The emails are triggered by changes in the state of the pins, which are configured through the easy-to-use EZ Web Lynx IDE.

EZ Web Lynx is available in either 3.3 or 5 volt, as well as a 3.3V Wi-Fi versions at a very low cost to make implementation to the Ethernet very affordable. EZ Web Lynx enables products and equipment to become Ethernet-ready without having to design a new circuit board which can drastically increase development time.

HTML is the only programming language needed to program the website. However, if you prefer to program in C, the CCS PCH or PCWH compiler is compatible with EZ Web Lynx. To find out more about this product and for detailed pricing please go to www.ezweblynx.com.

Add Flow Control and Buffering to Serial Routines

A feature of the compiler is the powerful ***#use rs232()*** library that has added transmit buffering, receive buffering, and flow control. While the API for the serial library remains unchanged (***getc(), putc(), printf()***), existing code using this API allows for buffering and flow control by simply modifying the ***#use rs232()*** parameters. A user may specify:

- size of transmit buffer
- size of receive buffer
- interrupt usage or no interrupt usage
- pin for CTS and pin for RTS

Click through to: http:\\www.ccsinfo.com/Version5
to review a usage example of the ***#use rs232()*** and additional details on each new usage. Additional configurations and control options are also available.

Notifications From the Serial Library on Data Reception

The compiler provides an extremely flexibly serial library; it has the ability to use the hardware peripheral or bit bang the pins, to control and monitor flow control, to specify parity, to use a one wire bus, and more. One feature it has is the ability to specify a receive buffer, and the library will automatically use the receive interrupt to buffer incoming characters. Here is an example of creating a stream called STREAM_UART1 on the UART1 hardware peripheral with a 16 byte receive buffer:

```
#use rs232(UART1, baud=9600, receive_buffer=16, stream=STREAM_UART1)
```

Essentially the stream works like a file handle that can be used with C standard I/O functions like fputc, fgetc, etc. Using the stream created above, here is a simple loop that echoes data received on the UART back to the UART:

```
while (kbhit(STREAM_UART1))

{

fputc(fgetc(STREAM_UART1), STREAM_UART1);
}
```

This example shows the flexibility of the #use rs232() library provided by CCS. The ‘receive_buffer’ option creates an interrupt on the UART receive to buffer incoming characters and kbhit() and fgetc() accesses that buffer, but if the ‘receive_buffer’ was removed from the #use rs232(), then kbhit() and fgetc() would instead check for any received data being held by the UART.

The ‘receive_buffer’ example as shown above has no way of notifying the users software that data is available, except by polling the receive buffer status with kbhit(). The 5.095 version of the CCS C Compiler adds a new option called ‘callback’ that allows the user to specify a function to be called when the receive buffer goes from empty to not empty. This could be used to mark a semaphore or enable a routine to start parsing data in the receive buffer. Here is an example of adding this new option:

```
#use rs232(UART1, baud=9600, receive_buffer=16, stream=STREAM_UART1, \
       callback=Uart1OnRx)
```

As stated earlier, this example will call the ‘Uart1OnRx’ function whenever the receive buffer goes from empty to not empty. Here is how the earlier echo example can be changed to use an RTOS with

a semaphore to mark when the receive buffer is ready:

```
#use rtos(timer=0)
      int uart_sem = 0;
      static void Uart1OnRx(void) {
            rtos_signal(uart_sem);
}
#task(rate=10ms)
    static void Uart1Task(void) {
       for(;;) {
            rtos_wait(uart_sem);
            while(kbhit(STREAM_UART1)) {
                  fputc(fgetc(STREAM_UART1), STREAM_UART1);
             }
       }
}
```

Alternatively, a function for parsing data in the receive buffer can be queued for execution with the timeouts library:

```
#include <timeouts.c>

void Uart1OnxTimeout(void* pArgs) {
      while(kbhit(STREAM_UART1)) {
            putc(getc(STREAM_UART1), STREAM_UART1);
      }
}

static void Uart1OnRx(void)  {
      TimeoutsAdd(Uart1OnxTimeout, NULL, 0);
}
```

How can I use two or more ports on one PIC®?

The #USE RS232 (and I2C for that matter) is in effect for GETC, PUTC, PRINTF and KBHIT functions encountered until another #USE RS232 is found.

The #USE RS232 is not an executable line. It works much like a #DEFINE. The following is an example program to read from one port (A) and echo the data to both the first port (A) and a second port (B).

```
#USE RS232(BAUD=9600, XMIT=PIN_B0, RCV=PIN_B1)
void put_to_a( char c ) {
      put(c);
}
char get_from_a( ) {
      return(getc()); }
#USE RS232(BAUD=9600, XMIT=PIN_B2,RCV=PIN_B3)
void put_to_b( char b ) {
      putc(c);
}
main() {
      char c;
      put_to_a("Online\n\r");
      put_to_b("Online\n\r");
      while(TRUE) {
            c=get_from_a();
            put_to_b(c);
            put_to_a(c);
      }
}
```

The following will do the same thing but is more readable and is the recommended method:

```
#USE RS232(BAUD=9600, XMIT=PIN_B0, RCV=PIN_B1, STREAM=COM_A)
#USE RS232(BAUD=9600, XMIT=PIN_B2, RCV=PIN_B3, STREAM=COM_B)
      main() {
            char c;
            fprintf(COM_A,"Online\n\r");
            fprintf(COM_B,"Online\n\r");
            while(TRUE) {
            c = fgetc(COM_A);
            fputc(c, COM_A);
            fputc(c, COM_B);
            }
      }
```