August 2020

<Bits & Bytes Newsletter/>

Email:
support@ccsinfo.com

sales@ccsinfo.com
Ph: 262-522-6500

CCS Inc
www.ccsinfo.com

New Compiler Versions
>Downloads>Compiler Software

Easy License Renewal
>Support>Renewals

TECH NOTE: Notifications From the Serial Library on Data Reception

TECH NOTE: Variables That Seem to Change on Their Own

SPECIAL OFFERS: Fall Discount

Product Spotlight

Prime 8 Production Programmer

Notifications From the Serial Library on Data Reception

The CCS C 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:

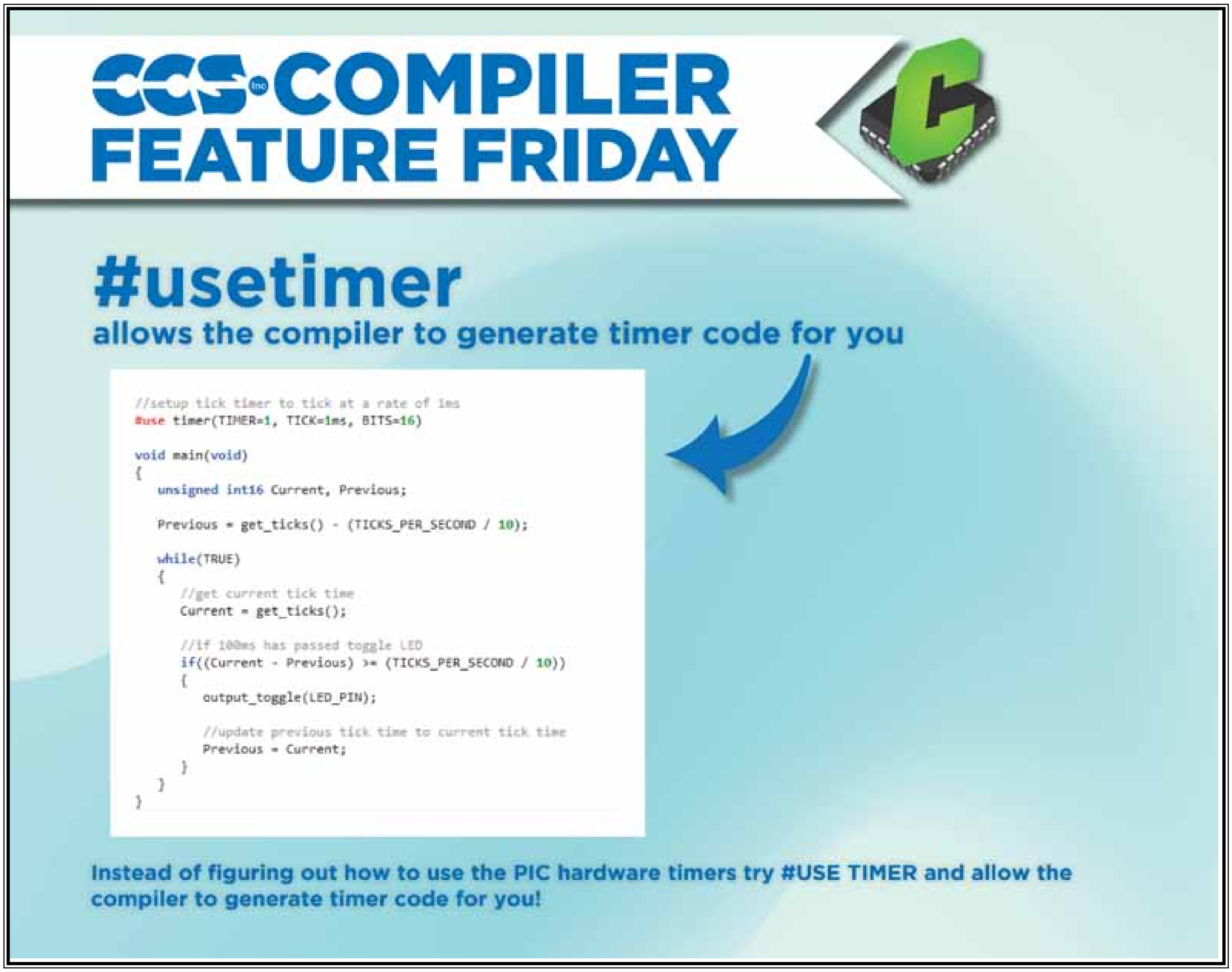
```
#use rs232(UART1, baud=9600, receive_buffer=16, stream=STREAM_UART1, \
            callback=Uart1On
```

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);
}
```


CCS COMPILER FEATURE FRIDAY

#usetimer

allows the compiler to generate timer code for you

```
//setup tick timer to tick at a rate of 1ms
#use timer(TIMER=1, TICK=1ms, BITS=16)

void main(void)
{
   unsigned int16 Current, Previous;

   Previous = get_ticks() - (TICKS_PER_SECOND / 10);

   while(TRUE)
   {
      //get current tick time
      Current = get_ticks();

      //if 100ms has passed toggle LED
      if((Current - Previous) >= (TICKS_PER_SECOND / 10))
      {
         output_toggle(LED_PIN);

         //update previous tick time to current tick time
         Previous = Current;
      }
   }
}
```

Instead of figuring out how to use the PIC hardware timers try #USE TIMER and allow the compiler to generate timer code for you!

Software developers will sometimes notice a variable has an unbelievable value. In a debugging situation, the value is checked right after it is written, and the value is good. If every place it is written is checked, then it appears as though somehow the variable value is changing not as a result of intentional C code. This application note covers some basic techniques to locate the problem.

1. The first step is to open the .SYM file and locate the variable in question. This memory map shows where the variable sits in relation to other variables. The compiler will share the same memory location between variables that should not be active at the same time. For example, the main program calls function A and it calls function B. The A function has a local variable called LA and B has a local called LB. Because function A and function B are not running at the same time, the compiler might put LA and LB in the same memory location.

Check the other variables in the same memory location and confirm those variables are not active at the same time. Remember that local variables that are not marked static may have garbage in them each time the function starts. In the rare case the compiler places two variables that could be active at the same time in the same memory location report it to CCS Support.

2. Check nearby variables, especially arrays. An index out of range could cause accessing the array to overwrite another variable. If re-arranging your variables or otherwise changing the code makes a problem come and go, then look at how your problem variable moves in the memory map.

3. Look at the wrong data and see if you recognize it as belonging somewhere else. A bad pointer could place good data in the wrong location.

4. If this is a multi-byte data item and it is accessed inside and outside a interrupt function, then make sure your code logic has protection against the variable being partially updated when an interrupt happens.

5. Check the chip errata. Sometimes chips have bad behavior at certain voltages, temperatures or clock speeds.

6. Check to see if your chip allows for data breakpoints. If so, and your application is debugger capable, you should be able to find the problem using the debugger. Set a data breakpoint at the location of the problem variable. Run the program and each time it breaks, check the variable value. When it is bad, look to see where in the code you are. If you see something like the following:

```
                buffer[bptr] = c;
```

Then it would seem bptr is out of range. Remember that in C, a ten byte array can not have an index over 9.

7. If none of the above solves the problem, you will need to roll up your sleeves and debug the problem more manually. To start, you need some way to identify the variable as being bad. Say the variable is temp and for now the expected value is over 68. When it is bad, it seems to be 0.

A. Write a function something like this:

```
char diag_tag=' ';

void check(char tag) {
  if(temp<68)
    if(diag_tag!=' ')
      diag_tag=tag;
}
```

B. Add throughout your code in places where temp should be valid (the start of main would probably not be such a place unless you init temp to say 68) the following:

```
check('A');
```

In each place change the letter (B, C,...) so each call is unique.

C. Run the code and after the variable goes bad check diag_tag to find out where it was first discovered. You will need some way to output diag_tag if you are not using the debugger.

When the location of first error is found, look back in the code execution path to find where check() was previously called (a good call). Then add more check calls between those two points.

D. Repeat step C until you locate the line that causes the variable corruption.

PRIME8

Programs
8 DEVICES
Concurrently

- **Supports 2.0V to 5V targets**
- **PC or Flash Drive Download**
- **Touchscreen Controls**
- **Ability to program PIC32 Family**
- **Can supply up to 200 mA to each target**
- **Green/Pass, Red/Fail simulated LEDs for each target**

COVID-19 RESPONSE

During this time of global uncertainty and change, we want to assure you that we are taking every precaution to ensure that we can safely support our customers during this time.

Despite these challenges, CCS staff is continuing to provide technical support, as well as processing orders. It is essential customers have the tools they need to provide the development of existing or new products that may be necessary in the fight of Covid-19.

Many of our existing customers are having to work from home and we want to remind everyone of our Software Licensing Agreement. We pre-register all compilers in a user's name. You can install your compiler on your home PC and laptops. If you do not have access to the registration files and installer, contact customer service for assistance.

CCS wants to help further embedded development by customers, and is offering a discount on any new compilers or maintenance plan purchases. The customers that need development boards, and programmers, we are offering Free Ground shipping (to the U.S.48) so you can get the tools you need to continue working from home.

Most importantly, as we work together in this unique and rapidly changing environment, we do so with confidence that we will overcome this challenge. Until then, we hold our enduring commitment to the health and well-being of our employees and customers Please let us know how we can help you. Stay healthy.

Fall Back Into Programming with a C Compiler

Half Price on Additional User License

Add an Additional User to any IDE license and receive 50% the list price!

Use Code: Fall25

More than 25 years experience in software, firmware and hardware design and over 500 custom embedded C design projects using a Microchip PIC® MCU device. We are a recognized Microchip Third-Party Partner.

www.ccsinfo.com

CCS Inc

Follow Us!