<Bits & Bytes Newsletter/>

June 2020

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

Product Spotlight

Touch Display 4.3 Development Kit!

How to Block Critical Code from Interrupts

It is quite common for an interrupt handler to save or alter data that can be accessed from your non-interrupt code. Doing this does require some care, otherwise the interrupt code might alter the data in the middle of the non-interrupt code dealing with the data. Look at following example:

```
WORD g_Timer1;

#int_timer1
void isr_timer1(void)
{
   g_Timer1++;
}

int1 check_timer(void)
{
   int1 ret = 0;
   if (g_Timer1 > 500)
   {
      g_Timer1 = 0;
      ret = 1;
   }
   return ret;
}
```

WORD is a type that takes multiple instructions of the architecture to modify, for example an int16 on an 8-bit architecture. If the Timer1 interrupt fired in the middle of check_timer() clearing g_Timer1, would result in a state where the interrupt would increment a half cleared g_Timer1. A common solution for this problem is to pause interrupts while handling this data. Here is an example of check_timer() updated to temporarily pause interrupts while handling data that could be modified in the ISR:

```
int1 check_timer(void)
{
   int1 ret = 0;
   disable_interrupts(GLOBAL);
   If (g_Timer1 > 500)
   {
      g_Timer1 = 0;
      ret = 1;
   }
   enable_interrupts(GLOBAL);
   return ret;
}
```

Unfortunately, the solution presented in the above modification to check_timer() can fail in a few situations:

- If check_timer() is called in another interrupt, where interrupts have been disabled by the hardware, then it will re-enable the interrupt while inside an interrupt. This can cause a disaster where an interrupt interrupts an interrupt, not something all microcontroller architectures can handle.
- If check_timer() is called when interrupts were disabled, then check_timer() would accidentally enable the interrupt when it is not wanted to be enabled. This can easily happen if check_timer() is used in a library shared in several projects, including projects where interrupts are not used.
- If check_timer() is called when interrupts were already paused for another series of instructions, check_timer() would then re-enable them before the calling function would have finished their operations expecting interrupts to be disabled.

CCS has a library for pausing interrupts that solves all three of the above problems: critical.h. Here is an example of using critical.h with the previous example:

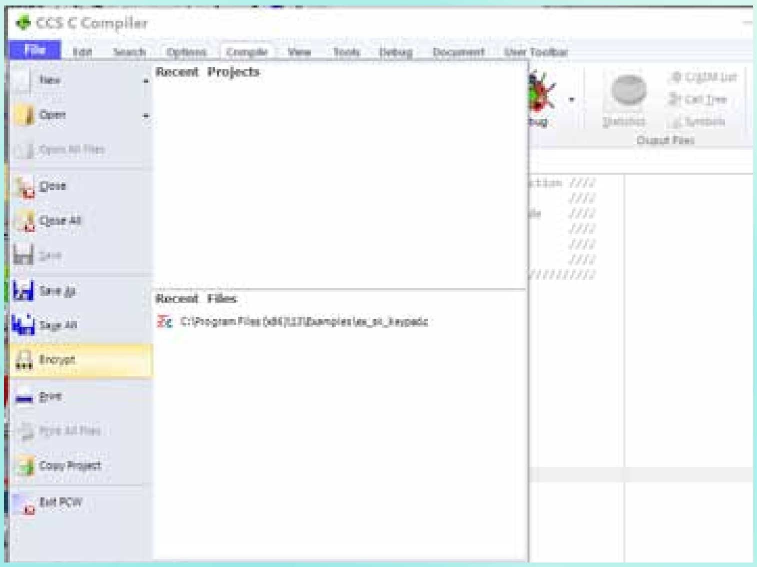
```
#include <critical.h>

int1 check_timer(void)
{
  int1 ret = 0;
  CRITICAL_SECTION_ENTER();
  If (g_Timer1 > 500)
  {
    g_Timer1 = 0;
    ret = 1;
  }
  CRITICAL_SECTION_EXIT();
  return ret;
}
```

CRITICAL_SECTION_ENTER() saves the previous state of the global interrupt enable and is restored by the CRITICAL_SECTION_EXIT(). That means if CRITICAL_SECTION_ENTER() is called when interrupts were not enabled, then CRITICAL_SECTION_EXIT() does not have enable interrupts. The CCS C Compiler does something similar internally to prevent recursion on architectures that do not allow recursion. When this occurs, a warning that states “Interrupts disabled to prevent recursion.” Now this method can be applied to critical sections of code where data is being modified by the interrupts that also need access to outside of the interrupt.

CCS Inc COMPILER
FEATURE HIGHLIGHT

Keep the contents of your files protected!



CCSC provides an Encrypt feature in the File menu that will encrypt an include file so it can be distributed and used by others without disclosing the file contents.

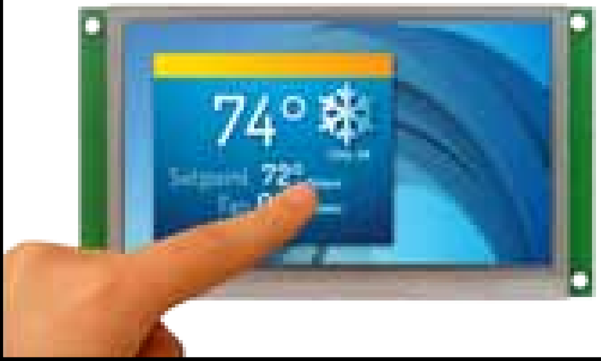
Product Spotlight
Touch Display 4.3 Development Kit

Impressive Graphics on a PIC® MCU!

EVERYTHING YOU NEED! ONLY $149

www.ccsinfo.com/product_info.php?products_id=touch-kit

Easily develop a Graphical User Interface (GUI) using a graphics LCD and touch display



Development Kit includes:

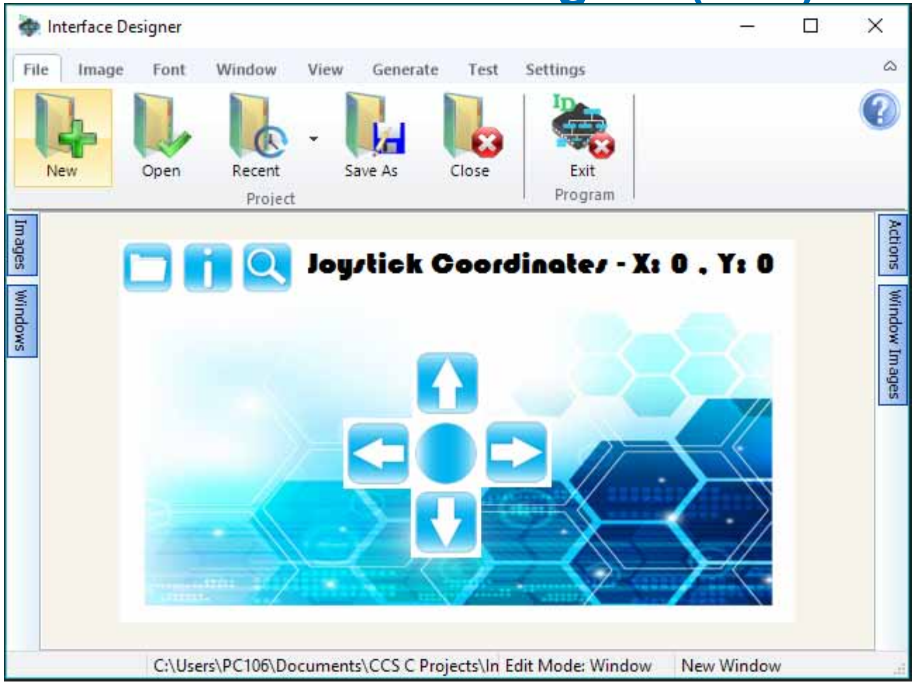
- Full-Featured Single-Chip IDE C Compiler
- Touch Display PRototype Baord
- ICD-U Programmer/Debugger
- Exercise Book and Cables

The new Touch Display 4.3 Development Kit is all that is needed to develop a GUI using the Graphics and Touch Library. Powered by a PIC24EP512GU810 it has a 4.3" 480x272 TFT display with a resistive touch-screen, a 256 megabyte flash, capable of storing hundreds of 480x272 images and custom fonts, a USB connection and 8 digital I/O pins that can be used for external inputs and outputs. No device programmer required - includes onboard programming capabilites.

Interface Designer (GUI)

Included is a software IDE to draw out a GUI and the C library for drawing graphics and handling touch.

- Draw an Image
- Draw an Object
- Handle Touch



Event-driven Programming Using the Timeouts Library

In a multitasking environment using state machines, it is quite common to poll for events that may have happened, and then go to a state based on which event that happened. For example, a push-button and LCD GUI polling the button to see if it is pressed, or polling to see if a time has expired to change or refresh what is on the LCD screen. If a polling scheme is used, eventually one may find that as the program gets bigger, that it spends most of its time polling for events that rarely happen. By switching to event-driven programming, one can create a callback that is instead called when an event happens that needs to be processed. The CCS C Compiler provides a portable callback library, timeouts.c. The following is a simple LCD GUI example using the the timeouts library:

```
#include <timeouts.c>

// debounce button, on debounce send event
void ButtonDebounceOnTimeout(void *pArg)
{
   if (BUTTON_PRESSED())
   {
   if (++pArg == 4)
      {  // debounced, send event
         TimeoutsRemove(LCDGUIOnTimeout, NULL);
         TimeoutsImmediate(LCDGUIOnTimeout, 1);
      }
   }
   else
      pArg = 0;
   TimeoutsAdd(ButtonDebounceOnTimeout, pArg, 16);
}

// handle button press or periodically refresh the screen
void LCDGUIOnTimeout(void *pArg)
{
   static int screenIndex;

   if (pArg)
   {
      if (++screenIndex > 4)
         screenIndex = 0;
   }

   switch(screenIndex)
   {
   case 0: ShowLCDGUIScreen0();  break;
   case 1: ShowLCDGUIScreen1();  break;
   case 2: ShowLCDGUIScreen2();  break;
   case 3: ShowLCDGUIScreen3();  break;
   }

   TimeoutsAdd(LCDGUIOnTimeout, 0, 333);  //refresh
}

// init button debouncer and screen handler.
void LCDGUIInit(void)
{
   TimeoutsImmediate(ButtonDebounceOnTimeout, 0);
   TimeoutsImmediate(LCDGUIOnTimeout, 0);
}
```

LCDGUIInit() calls **TimeoutsImmediate()**, which places functions onto the timeouts callback stack to be called. The first parameter of TimeoutsImmediate() is the function that is to be called, and must match the prototype void function_name(void *). The library will create a pointer to that function and push it onto the stack. The second parameter of TimeoutsImmedate() is the parameter that is passed to the function when the timeouts library is called. Since this parameter is of type void*, it could either be pointer to state variables, or it could be just used as a int. In the examples shown here it is simply used as an int, to hold an int variable from one call to the next. That means what LCDGUIInit() is doing is pushing ButtonDebounceOnTimeout and LCDGUIOnTimeout to be called, to be passed a value of 0.

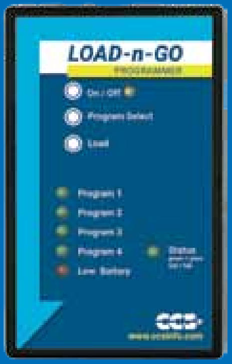
ButtonDebounceOnTimeout() will debounce the button. BUTTON_PRESSED() would be a hardware dependent macro that returns true if the button is currently held down. In this example, if BUTTON_PRESSED() returns true a counter is incremented and if the counter is incremented a fourth time it will then use the timeouts library to call LCDGUIOnTimeout() with a parameter of 1. At the end of ButtonDebounceOnTimeout(), TimeoutsAdd() is then used to push ButtonDebounceOnTimeout() to be called again in 16 milliseconds. That means ButtonDebounceOnTimeout() is called every 16ms, and it takes 64ms of debouncing before a button pressed event is sent. Since the pArg is passed back to the repeat call of ButtonDebounceOnTimeout(), it can be used as a state variable of this function to measure how many times the button was held. If you look ahead to the LCDGUIOnTimeout(), you will see that it pushes itself back onto the timeouts call stack every 333ms to refresh the screen; for this reason ButtonDebounceOnTimeout() is doing a TimeoutsRemove() to remove all other calls to LCDGuiOnTimeout() so the only even that is going to be called is a button press event.

LCDGuiOnTimeout() will either handle a button event or redraw the screen depending on the pArg passed to it. In this example a pArg of 1 is a new button, in which case it goes to the next screen. If pArg is a 0 then it's a screen refresh, just redraw the current screen. At the end of LCDGUIOnTimeout() a TimeoutsAdd() is used to push another call to LCDGuiOnTimeout in 333ms, meaning the screen is redrawn 3 times a second.
What isn't shown in the above example is the main loop, which must call TimeoutTask(). TimeoutTask() looks at the top of the call stack, and if it's expiration has expired it then calls that function. Functions that add to the call stack always keep the call stack sorted so the top element, the one being checked, is always the next to expire. That means TimeoutTask() only needs to poll one task, regardless of how many elements are pushed.

This library can also be used in some other situations:

- Interrupts can push a function to be called on the main loop, to prevent execution of lengthy or low priority code to be running in an interrupt.
- If completely using an ISR and timeouts library based design, the processor could be put to sleep until the next event. The timeouts library has a function, TimeoutsNext(), which tells you how long until the next event. The processor could be put to sleep for that duration, as long as it's configured so interrupts will wake it.
- Diagnostic and metrics could be added to TimeoutTask() to measure how long each event takes, to find or debug certain events that take too long to execute.

Check out the popular CCS Programmer / Debuggers



CAN FD Support in CCSC

The CCS C Compiler now supports sending and receiving CAN FD messages over the CAN Bus. This support is from the addition of two new drivers that compiler now comes with. The can-dspic33_fd.c driver is for dsPIC33CH and dsPIC33CK devices with a built-in CAN FD peripheral and the can-mcp2517.c driver is for a MCP2517FD external CAN FD controller. The MCP2517FD external CANFD controller uses an SPI interface to communicate, so it can be used with any PIC® microcontroller.

The CAN FD stands for CAN Flexible Data-Rate, the main improvements of CAN FD over CAN 2.0 is that the data rate switches to a faster rate after the arbitration bits are sent, and the maximum data packet size is increased from 8 bytes to 64 bytes. Both of this improvements allow for more data to be transferred in less time, increasing the throughput of the CAN Bus.

The CAN FD driver API was made to be as similar as possible to the current CAN driver provided in the CCS C Compiler. The CAN FD driver API is fully documented in the driver's .h files, however the following function will mostly likely be used in any CAN FD project being developed, can_init(), can_kbhit(), can_getd() and can_putd(). The can_init() function is used to initialize the peripheral and setup the baud rate, filters and objects used by the driver, this function also accepts an optional parameter to select what operation mode the CAN FD peripheral will be operating in when the function call is complete. The can_kbhit() function is used to check if any CAN messages were received by the RX object. The can_getd() function is used to retrieve received messages from the RX object. Finally the can-putd() function is used to load a message into the TX object to be sent on the CAN Bus.

Additionally to make is easier to setup the CAN FD peripheral the CAN FD driver as multiple preproccessor defines that can be made before the driver is included. The full list of the defines that can be made and their descriptions, including the default values, can be found in the driver's .h file. Some of the more common defines that will most likely be used are CAN_NOMINAL_BAUD_RATE, CAN_DATA_BAUD_RATE, CAN_TX_BUFFERS and CAN_RX_BUFFERS. The defines CAN_NOMINAL_BAUD_RATE and CAN_DATA_BAUD_RATE are used to set the bit rate used with CAN FD message frames during the arbitration and data periods respectively. The only requirements are that the CAN_CLOCKS_SPEED define must be evenly divisible by the rates, and the max CAN_NOMINAL_BAUD_RATE is 1,000,000 and the max CAN_DATA_BAUD_RATE is 8,000,000. For the built-in CAN FD peripheral the define CAN_CLOCK_SPEED is used to set the clock being presented to the peripheral. There are multiple ways the clock can be setup, see the can-dspic33_fd.h file for all options. By default the driver is setup to use the auxiliary clock setup for a speed of 80MHz. For the MCP2517FD controller the define CAN_CLOCK_SPEED is made automatically based on the define MCP2517_EXT_CLOCK_SPEED which is the speed of the external crystal connected the controller. Only a 4MHz, 20MHz or 40MHz crystal can be used with it, by default the driver is set to use a 20MHz crystal. The define CAN_TX_BUFFERS sets the size of the TX Queue object used to send messages, the number of messages that can be held in RAM to be sent on the CAN Bus. It can be set from 0 to 32, 0 disables the TX Queue object, the default size if 1. The define CAN_RX_BUFFERS sets the size of the FIFO 1 object which is setup as a receive FIFO by the driver, the number of received messages that can be held in RAM. It can be set from 0 to 32, 0 disables the FIFO 1 object, the default is 32 for the built-in CAN FD peripheral, and 16 for the MCP2517FD controller.

Finally a new CAN FD development kit will soon be available from CCS which contains a node with a dsPIC33CH128MP506 for using the can-dspic33_fd.c driver, and a node with the MCP2517FD external controller for using the can-mcp2517.c driver. In addition to the hardware the development kit comes with an exercise manual that has examples of using more of the CAN FD features, including setting up and using CAN filters, using multiple FIFO objects and using the CAN FD interrupts.

COVID-19 RESPONSE

WORKING FROM HOME?
CCS Inc
TACKLE WORK AND HOBBY PROJECTS WITH A POWERFUL C COMPILER AT A DISCOUNT!
SPECIAL OFFERS
$25 off
ANY FULL COMPILER OR COMPILER MAINTENANCE
Use Code: Home25
FREE GROUND SHIPPING
(TO U.S. 48 ONLY)
ON ALL PROGRAMMERS OR DEVELOPMENT KITS
Use Code: HomeFS

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.

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.

CCS Inc

www.ccsinfo.com

Follow Us!