February 2021

www.ccsinfo.com
262-522-6500

<Bits & Bytes> Newsletter

CAN Bus Development Kit

The CANbus Development kit enables users to begin CAN network development with Microchip's PIC® PIC18 family.

INSIDE THIS ISSUE:

CAN Bus Variations

The CAN protocol is a serial communication protocol that is used in the automotive industry for communicating between devices inside of a vehicle, the engine control unit and dashboard for example.Data is sent in frames and is done in such a way that if more then one device transmits at the same time the highest priority device is able to continue while the other device back off.There are two CAN standards that are in use today, CAN 2.0 and CAN FD. CAN 2.0 is the older of the two protocols and has two parts; part A is for the standard format with an 11-bit identifier, commonly called CAN 2.0A, and part B if for the extended format with a 29-bit identifier, commonly called CAN 2.0B.Both parts can transmit data with bit rates up to 1MBit/s with up to 8 data bytes.

CAN FD is a newer protocol that has flexible data-rate, an options for switching to a faster data rate, up to 5 Mbits/s, after the arbitration bits, which is limited to 1Mbits/s for compatibility with CAN 2.0, and it increases the max number of data bytes that can be transmitted in a frame to 64.CAN FD is compatible with existing CAN 2.0 networks so new CAN FD devices can coexist on the same network with existing CAN 2.0 devices.

There are several PIC® microcontrollers that have a built-in CAN 2.0 or CAN FD modules.For these devices, the CCS C Compiler comes with drivers for communicating with these protocols.There are separate drivers depending on the device being used. Additionally, the CCS C Compiler comes with several external CAN controllers.The following are a list of can drivers that are currently available in the CCS C Compiler:

- can-pic18f_ecan.c – PIC18 CAN 2.0
- can-pic24_dsPIC33.c – PIC24 and dsPIC33 CAN 2.0
- can-dspic30f.c – dsPIC30 CAN 2.0
- can-mcp2515.c – External MCP2515 controller CAN 2.0
- can-dspic33_fd.c – dsPIC33 CAN FD
- can-mcp2517.c – External MCP2517 controller CAN FD

J1939 is an upper level protocol that specifies how to send messages in a vehicle using the CAN 2.0 and CAN FD protocols. J1939 is maintained by SAE and the full J1939 specifications can be obtained from them.The J1939 is broken into several layers including, but not limited to, the Data Link Layer, Network Layer and Application Layer. These layers contain information about how to communicate on the network, how to claim an address, the format of messages, how often a message can be transmitted, etc. The CCS C Compiler comes with a J1939.c driver which is a library for the Data Link Layer running on a CAN 2.0 protocol network. The library has functions for claiming an address, responding to address claim messages, transmitting J1939 messages and receive J1939 messages.

Additionally, CCS also has several CAN development kits that can be used to aid in developing CAN Bus and J1939 projects. Each development kit has four nodes on it that can communicate with each other, as well as headers allowing the kit to be connected to an external Bus.

The first development kit CCS has is the CAN Bus development kit which has a PIC18F45K80 on the primary node and a PIC16F1938 on the secondary node. The primary node the PIC® MCU uses its built-in CAN peripheral for communicating on the Bus. The secondary node the PIC® MCU uses an external MCP2515 CAN controller for communicating on the Bus.

The second development kit CCS has is the CAN Bus 24 development kit which has a PIC24HJ256GP610 on the primary node and a dsPIC30F4012 on the secondary node. Like the previous kit, the primary node the PIC® MCU uses is its built-in CAN peripheral for communicating on the Bus and the secondary node PIC uses an external MCP2515 CAN controller for communicating on the Bus.

Finally coming soon, CCS with have the CAN Bus FD development kit which features a dsPIC33CH128MP506 on the primary node and a PIC16F18346 on the secondary node. The primary node the PIC® MCU uses is its built-in CAN FD peripheral for communicating on the bus, and the secondary node the PIC® MCU uses an external MCP2517 CAN FD controller for communicating on the Bus.

CCS COMPILER FEATURE FOCUS

Need to make a comment in a hex file?
Use #hexcomment pre-processor directive for this purpose.

```
40  #define KBD_ROW1  PIN_C1
41  #define KBD_ROW2  PIN_C2
42  #define KBD_ROW3  PIN_B4
43  #define KBD_ROW4  PIN_B5
44
45  #include <kbd3.c>
46
47  #hexcomment  Version 3.1 - Requires 20Mhz crystal
48  void main(void)
49  {
50     char c;
51
52     port_b_pullups(0x30);
```

Puts a comment in the hex file.
As shown, the comment is put at the top of the hex file.
CCS device programmers will display those comments before a chip is programmed.

Add a \ before the comment to put it at the end of a hex file. Those comments are not displayed and will not confuse some device programmers that do not understand comments.

The CCS pre-processor extensions specific to the PIC® processor can be used with or without the leading pragma. If the pragma is used other C compilers will ignore or flag a warning for the PIC® directives. For example: #pragma hexcomment V1.23

PIC® MCU Address Spaces -- Part 2

In part one, we covered the two hardware address spaces in the PIC® MCU and how they can be used from C.This article will explain how a developer defines virtual address spaces. These user defined spaces can be used to add a layer of abstraction between “normal” C variables and the actual hardware memory implementation. We will detail implementations for three example address spaces:

- External serial EEPROM
- Display memory in a graphic LCD module
- Dual copy safety variables for a safety critical application

External Serial EEPROM

For the first example, we will use an external serial EEPROM that uses I^2C to access the data. CCS provides a simple driver that gives users write_ext_eeprom() and read_ext_eeprom() for byte access to the external memory. To use that memory like they would use regular RAM users need to create a new address space that the variable can be declared in.

Before users can define the address space, users need to provide some functions to read and write to the memory. For example:

```
void DataEE_Read(int32 addr, int8 * ram, int bytes) {
  for(int i=0;i<bytes;i++,ram++,addr++)
    *ram=read_ext_eeprom(addr);
}

void DataEE_Write(int32 addr, int8 * ram, int bytes) {
  for(int i=0;i<bytes;i++,ram++,addr++)
    write_ext_eeprom(addr,*ram);
}
```

When the embedded C ANSI specification was first proposed they called this typemod and by the time the specification was approved it was called addressmod with a new syntax. The CCS C compiler accepts both syntax forms.This is everything needed to define the space:

```
addressmod (DataEE, DataEE_read,DataEE_write,0,0x3ff);
```

The first parameter give the new space a name, then we have the read and write functions and finally we have the address range of the new space. Here is how it can be used:

```
DataEE int32 serial_number;

serial_number=0x123456789;

printf("S/N = %8LX\r\n", serial_number);
```

In this case, the variable serial number actually resides in the external EEPROM. Suppose the user wants to make the EEPROM access more efficient.They could crate a simple RAM array with a copy of all the EE data that gets loaded on power up.Then the **DataEE_Read()** function reads from the RAM array and **DataEE_Write()** writes to the array while marking a dirty bit. In the background as time permits some task could copy the dirty data back to the EE.The details of this implementation are separate from the main program and the algorithm can be changed without affecting the main program.For example, maybe the user want to keep a CRC of the EE data in the last EE location, or they want to encrypt the data or keep three copies of each byte in the EE in case of corruption.This can be completely handled in **DataEE_Write().** Data can be declared with the **DataEE** qualifier with arrays, structures or any other data type.

Display Memory in a Graphic LCD Module

Consider an array where each element in the array represents a color of a pixel on a display.If the processor shares memory with the display then this is a given. However, for displays with a serial interface or requires commands to access the display users can use a user defined space to have the same capability.The functions might look like this:

```
void LCD_Read(int32 addr, int8 * ram, int bytes) {
  glcd_ReadPixels(addr*LCD_WIDTH*sizeof(color_t), // X
      addr % (LCD_WIDTH*sizeof(color_t)),       // Y
      bytes/sizeof(color_t),1,ram);
}

void LCD_Write(int32 addr, int8 * ram, int bytes) {
  glcd_DrawPixels(addr*LCD_WIDTH*sizeof(color_t), // X
      addr % (LCD_WIDTH*sizeof(color_t)),       // Y
      bytes/sizeof(color_t),1,ram);
```

```
}

addressmod (LCD_screen, LCD_read,LCD_write,0,0x3ff);


LCD_screen  char  screen[180][240];
```

A function to draw a line might look like this:

```
void draw_line( int16 x, int16 y, int16 length, color_t color){
        for(int16 i=x; i<(x+length); i++)
           screen[y][i]=color;
    }
```

Dual Copy Safety Variables for a Safety Critical Application

Consider an application where to get certified users are required to save all critical variables in two memory locations and to verify they match before using them. Users can add logic to the hundreds of places they access the variables. It would be easier with macros however it will still make the code hard to read and if the variables considered critical changes then it can be a mess to update the code. Instead try this:

```
void Safety_Read(int32 addr, int8 * ram, int bytes) {
  for(int i=0;i<bytes;i++,ram++,addr++)
    if(safety_ram1[addr]!=safety_ram2[addr])
       trigger_system_fault('Bad RAM');
     else
       *(ram)=safety_ram1[addr];
}

void Safety_Write(int32 addr, int8 * ram, int bytes) {
  for(int i=0;i<bytes;i++,ram++,addr++) {
    safety_ram1[addr]=*(ram);
    safety_ram2[addr]=*(ram);
  }
}
```

Then the critical variables can be declared like this:

```
SafetyRAM  int16 motor_speed;
SafetyRAM  int16 recent_presures[10];
SafetyRAM  struct {
           int32 target_voltage;     // tenths of a volt
              int16 time_since_change;  // seconds
              int8  last_adjustment;    // tenths of a volt
            } drive_data;
```

In the above discussion, we covered how to implement user defined address spaces. The next part details how some PIC® MCUs have their own alternative address spaces and how the C compiler deals with them. For example the enhanced PIC16 parts have both a physical address space in the traditional memory banks and they have an alternate linear address space that includes only the user RAM locations and no Special Function Registers. Some PIC18's and PIC24's also have alternate addressing schemes.

Using the CCS C Compiler Under MacOS

The CCS C compilers are a great tool for programming Microchip PIC® microcontrollers however, there is not a version of the compiler available for use on macOS. Despite this, there is a way for Mac users to utilize the CCS C compilers by using a virtual machine. A virtual machine is like a fully functioning computer but it is running in software on a host computer. By using a virtual machine, a macOS user can run a Windows 10 computer from their Mac. This allows the CCS C compilers to be run on a macOS computer.

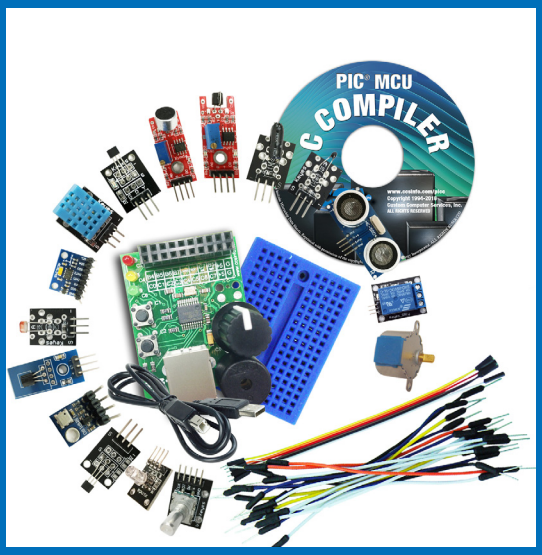
There is an article on the CCS website that includes a step-by-step guide to using CCS C compilers on macOS (a link to this guide is at the end of this article). The article walks through installing and editing a driver created by FTDI which allows a Mac machine to communicate with the ICD that is used when programming/debugging MCUs. In the guide, it is recommended to use a free app called Xcode that is available on the app store, which makes editing the driver easier and more user friendly.

After the driver from FTDI is installed, the guide outlines how to create a virtual machine on a Mac computer. It starts with installing VirtualBox, a virtualization tool that allows for the creation and management of a virtual machine. The guide walks through how to create a virtual machine through VirtualBox, as well as how to download an ISO image of Windows 10 so the operating system can be installed on the virtual machine.

Once a virtual machine has been created and an ISO image has been attached to it, the machine can be booted and the Windows 10 installation process begins. The guide provides steps to format the virtual hard drive that is used by the virtual machine so that the files for Windows 10 can be copied over correctly and installed. After the OS is installed, the user can follow the Windows 10 setup and enter their own preferences.

The final step outlined in the guide is setting up the Windows 10 environment so that it can run the CCS software. This includes installing USB drivers from CCS so that the virtual machine can detect the ICD when it is passed through from macOS. The guide also outlines creating a shared folder between the host macOS machine and the Windows 10 virtual machine. Creating a shared folder allows for the CCS C compiler installation files to be transferred to the VM so the software can be installed. Once you have completed the step-by-step guide to using CCS C compilers on macOS, you will be able to program Microchip PIC® MCUs without the use of a Windows computer.

For more information, view the full article here:
https://www.ccsinfo.com/faq.php?page=compiler_mac

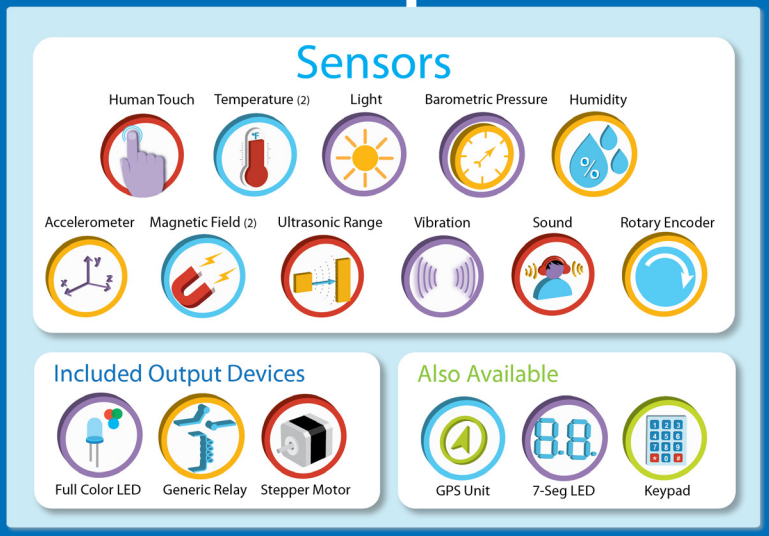


C Workshop Compiler

Limited to 13 Devices with the IDE

$99

Sensors Explorer Kit



Supported Devices:

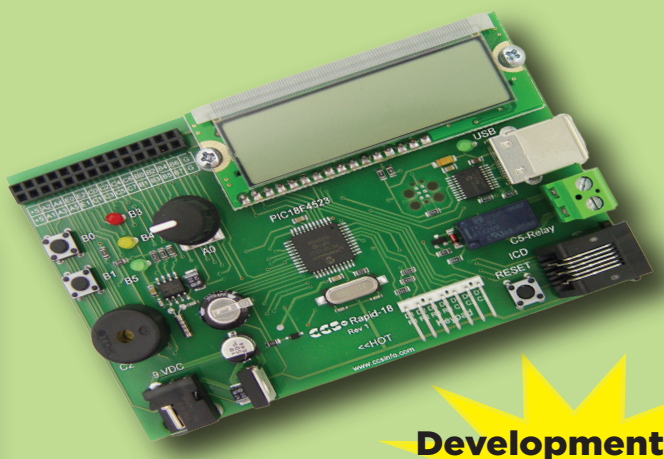
8-bit: PIC10F222, PIC12F1822, PIC16F84A, PIC16F818, PIC16F877A, PIC18F13K50,PIC16F1459, PIC18F24J11, PIC18F4520

16-bit: PIC24F16KM102, PIC24FJ128GA006, dsPIC30F3010, dsPIC33EP128MC202

RAPID-18 DEVELOPMENT BOARD

FEATURES:

- *PIC18F4523 Microchip PIC® MCU
- *24 I/O Pins (11 can be 12-bit analog)
- *One Potentiometer
- *Two Pushbuttons
- *Three LEDs
- *USB Connector (with USB to PIC® UART interface)
- *ICD Jack
- *Keypad Socket (for 3x4 keypad)
- *16x2 Character LCD
- *Piezo Speaker/Buzzer
- *Relay and Dry Contacts
- *Real Time Clock/Calendar with Supercap



Development Kits Starting at $50.00

PIC® MCU is a registered trademark of Microchip Technology Inc

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.

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.