

Quarter 2  
2023



[www.ccsinfo.com](http://www.ccsinfo.com)  
262-522-6500

# <Bits & Bytes> Newsletter

## INSIDE THIS ISSUE:

Pg 2-3  
CAN Bus on a PIC® MCU

Pg 4  
Bootloaders for Field Up-  
Gradable Programs

Pg 5  
Column Editing IDE Feature

Pg 6-7  
Type Conversions

# Product Spotlight



## **MACH X** OTP DIP-CHIPS, IN-CIRCUIT PROGRAMMER & DEBUGGER

The Mach X Programmer is a full-featured device programmer that programs all Microchip PIC® MCU, Flash devices, and One-Time-Programmable (OTP) devices. Program various DIP chips (8-40 pins) directly from the on-board 40-pin ZIF socket. The unit also provides ICD & ICSP™ functionality and debug support covers all targets that have debug mode when used in conjunction with the C-Aware IDE Compilers.

[support@ccsinfo.com](mailto:support@ccsinfo.com)

[sales@ccsinfo.com](mailto:sales@ccsinfo.com)

PIC® is a registered trademark of Microchip Technology Inc.

# CAN Bus on a PIC® MCU

CAN bus is a message-based protocol allowing individual systems, devices, and controllers within a network to communicate. In general, a bus is a multi-node communication system that transfers data between components. A Controller Area Network allows for robust, low-latency, data transfer between sensors and compute units in a system. Without CAN Bus, wiring harnesses could contain miles of wire, with bundles of wires required to carry various signals to and from interconnected systems. In contrast, CAN bus utilizes a high-speed (25kbps – 1Mbps) twisted pair wiring system, greatly reducing the amount of wire necessary to allow system components to communicate.

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 than one device transmits at the same time the highest priority device is able to continue while the other devices 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 is 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 option 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 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.

The third development kit CCS has is CAN Bus FD which features a dsPIC33CH128MP506 on the primary node and a PIC16F18346 on the secondary node. The primary node the PIC® MCU uses 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.

## 8-Bit AVR<sup>®</sup> Support for Programmers



Programming support for all 8-bit AVR<sup>®</sup> microcontrollers. LOAD-n-GO, Prime8 and ICD-U80 supported. Programming adapter and cables available as separate purchase.

**8-bit AVR<sup>®</sup>  
Programming Adapter**  
53505-1867 | \$25.00

sales@ccsinfo.com  
262-522-6500 EXT 35  
www.ccsinfo.com/NL0422



# Bootloaders for Field Up-Gradable Programs

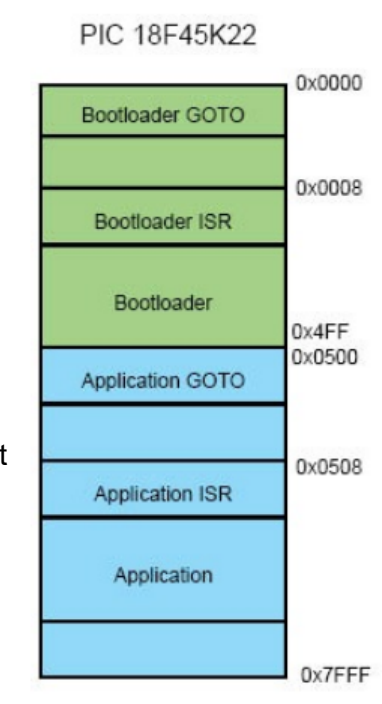
One of the most difficult things to deal with is upgrading a products firmware to fix a bug for products that are already in the field. It can be expensive and time consuming to do a recall of the products or send technicians to update the firmware. One option is to add a bootloader to the product. By using a bootloader it is possible to update a products firmware automatically or by the end user. One of the easiest types of bootloaders to implement is a serial bootloader.

A serial bootloader uses a serial connection, RS232 for example, to transfer the new firmware from a PC to the product, which is then programmed onto the product by a small program that runs on the device. To aid in quickly developing a serial bootloader, the CCS C Compiler has bootloader code that can be included in your project, as well has a PC program that can be used to transfer the firmware to product.

The CCS C Compiler provides the following bootloader examples, `ex_bootloader.c` and `ex_pcd_bootloader.c`. The first is an example of a serial bootloader for PIC16 and PIC18 devices, PCM and PCH compilers, and the second is an example of a serial bootloader for PIC24, dsPIC30 and dsPIC33 devices, PCD compiler. Both are an example of a standalone bootloader. Standalone bootloaders are small programs that run on the device that are responsible for both receiving the firmware and for programming it onto the device. In general, standalone bootloaders do not require the application for them to work. The size of a serial bootloader program depends on the device they are being used on, for example the CCS serial bootloader for PIC18 devices use 1280 instructions or 2560 bytes of ROM and always remains at the same location in ROM. Some PIC® MCUs allow you to specially code protect the bootloader area in ROM. Additionally the CCS C Compiler provides the following bootloader applications, `ex_bootload.c` and `ex_pcd_bootload.c`. Both are examples of applications that can be bootloaded onto a device using the `ex_bootloader.c` and `ex_pcd_bootloader.c` bootloaders. The key difference between a standard application and one that can be bootloaded is that the bootloadable application reserves an area of ROM for the bootloader. Frequently that area includes the reset and interrupt vectors so the application will use an alternate area that the bootloader can link to. In general #including the same `bootloader.h` file that the bootloader uses is all that needs to be done to build an application that is compatible with the bootloader.

A key consideration for bootloaders is deciding when to bootload. The bootloader program starts when the chip starts. If there is no application program in memory then it goes into bootload mode. That is the easy case. For reloading, a button could be used, for example hold that button down, power up and the bootloader sees the button down and starts the loading process. The application itself could trigger a bootload by writing a value to EEPROM and then resetting, the bootloader would see the special value and could force a bootload.

Finally CCS provides a PC program, CCS Bootloader, that can be used to transfer firmware (a .hex file) from a PC to a device that is running a CCS C Compiler bootloader. The CCS Bootloader program is a command line utility that may be distributed as part of the user's end product.



# Column Editing IDE Feature

The Editor in the V5 IDE has a column editing feature. This is useful if there are several lines that start or contain the same block of text but need to be replaced or edited. To use this feature, press the CTRL key on the keyboard while using the left mouse button on the mouse to select a block of text. Pressing DEL will delete that block of text, or typing will replace the text within the block with new text you type on each line.

This can also be used to simply insert the same text at a given spot on a group of lines. To do that select a thin column where you want the text inserted.

There are situations wherein there is a need to make repetitive but identical changes to each line, or copy a block of text and then add the same text or spacing to each line when editing source code. Column editing allows a way to enter or delete text on multiple rows at once.

Access this feature by pressing the CTRL key while making a selection with the left mouse button. This enables selecting a rectangular region to be able to type to replace its contents, paste over it, or delete it.

The following are some examples.

## 1. Editing identical variable types



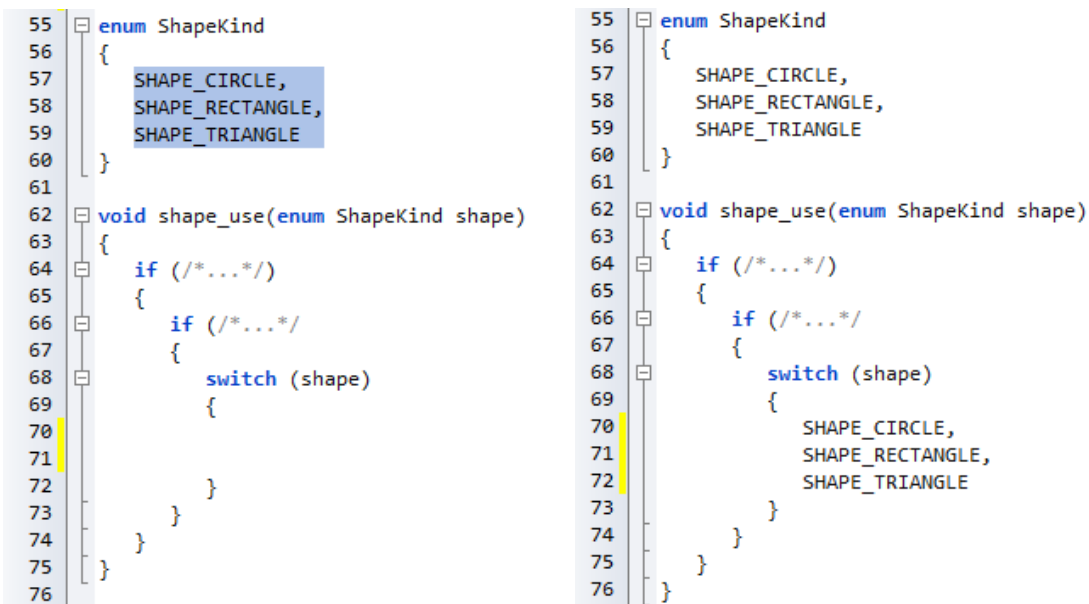
```
48 int x_seq;
49 int y_seq;
50 int count;
51
```

→

```
48 unsigned int x_seq;
49 unsigned int y_seq;
50 unsigned int count;
51
```

In the above illustration, column select the “int” type, and then simply type “unsigned int”, to all 3 lines at once.

## 2. Working with Enums



```
55 enum ShapeKind
56 {
57     SHAPE_CIRCLE,
58     SHAPE_RECTANGLE,
59     SHAPE_TRIANGLE
60 }
61
62 void shape_use(enum ShapeKind shape)
63 {
64     if (/*...*/)
65     {
66         if (/*...*/)
67         {
68             switch (shape)
69             {
70                 SHAPE_CIRCLE,
71                 SHAPE_RECTANGLE,
72                 SHAPE_TRIANGLE
73             }
74         }
75     }
76 }
```

Above shows a selected rectangle consisting of all of the enum variants to avoid copying the spacing. Pasting it into the switch statement maintains its tabbing. Next insert “case” into each line by simply column select the space before “SHAPE” and type “case”.

# Type Conversions

In order to evaluate expressions the C compiler uses a complex set of rules to get a result that is consistent and as intuitive as possible. Sometimes the coder needs to add an explicit typecast in order to nudge the compiler to do something specific.

To understand this topic we will start with a simple example that will show a possible problem:

```
int8 x,y;
int16 z;

x=5;
y=100;

z = x * y;
```

There are two expression evaluations here. One is the `*` and the second is the `=`. The order is `*`, and then `=`, and that is controlled by operator precedence, another topic. The two operands for the `*` are eight bit so the eight bit multiply is used, yielding an eight bit result. In this case the result of the multiply will be 244. Then because the `=` operator has an 8 bit operand and a 16 bit operand, the 8 bit operand is automatically converted (Integral Promotion because of Usual Arithmetic Conversion) to 16 bit by the compiler. Then the assignment is done.

Probably the coder expected a 500 in `z`, not 244. To fix the situation we can do a typecast on either 8 bit operand like this:

```
z = (int16)x * y;
```

Now when the multiply is done the second eight bit operand is converted to 16 bit to match the other operand and the multiply and result are now 16 bit (500).

Note that this concept is not an invention of the CCS C compiler although people who have never had this trouble with another compiler might think it is. A C compiler for a PC for example where an int is 32 bit by default may never cause a problem because the numbers used are much smaller than the default type. Because the CCS C compiler starts with an eight bit int by default, problems are more common.

Having said that, because the rules are somewhat complex and not always fully understood and sometimes not very specific in the specification, the CCS C compiler has tweaked the rules over the years. Sometimes to match the latest interpretation of the spec or sometimes to match a GNU compiler or the Microchip compiler to help in code portability.

## Terminology:

### Explicit Conversion, or Explicit Cast, or Explicit Typecast, or Typecast:

This is where the coder has a typecast that will force a conversion from one type to another. The syntax is `(new type)expression` and if the new type is compatible the value will be the same. That is to say this is more than a bit for bit movement, for example a float to an integer will have the same value to the extent possible.

### Implicit Conversion, or Automatic Conversion:

This is where the compiler uses a set of rules to be detailed later to perform a conversion.

### Type Conversion:

Either an Explicit Conversion or Implicit Conversion.

### Integral Promotion:

In this case any type conversion where a smaller integer is converted to a larger one (or superior one). The value does not change when promoted.

### Usual Arithmetic Conversions:

These are implicit conversions that are made according to the rules when an operator and operands are involved. One operand is converted based on the type of the other operand BEFORE the operator is invoked.

### **The Rules:**

The way to read this table is for each group start at the top and go down until you find the first rule that applies. Use that result and stop.

Be aware that by default the CCS C compiler for 24 bit parts is **signed** and the size is 16 bits. For all other compilers the default is **unsigned** and the size is 8 bits. This can be changed using **#type** so this needs to be taken into account when considering the rules.

The syntax in this table is not real C, just pseudo-code that a C coder should be able to understand.

The table as follows:

<b>Integer Constant Types</b>	
Leading 0 in constant	Unsigned
Trailing u in constant	Unsigned
Else if default type is signed    is preceded by a minus	Signed
Else if default type is unsigned	Unsigned
Find the smallest of these types the constant will fit into	int8, int16, int32, int48, int64
<b>Integral promotion anytime X is used in an expression (int is the default type)</b>	
issigned (X) && bitsin(X)<bitsin(int) && (X>max(signed(X))	X=>unsigned int
bitsin(X)<bitsin(int)	X+>int
<b>Any type conversion from signed to unsigned</b>	
bitsin(signed(X)) <=bitsin(unsigned(X))	X+>unsigned(X) sign extend
Else	X+>unsigned(X) truncate
<b>Usual Arithmetic Conversions for Binary operations</b>	
isfloat(X) && !isfloat(Y)	Y=>typeof(X)
isfloat(X) && bitsin(X)>bitsin(Y)	Y=>typeof(X)
Only if both operands are integer:	
bitsin(X)>bitsin(Y) && (X & Y are both signed or unsigned)	Y=>typeof(X)
bitsin(X)>bitsin(Y) && unsigned(X) && signed(Y)	Y=>typeof(X)
bitsin(X)>bitsin(Y) && signed(X) && unsigned(Y)	Y=>typeof(X)
bitsin(X) = bitsin(Y) && unsigned(X) && signed(Y)	Y=>unsigned(Y) X=>typeof(Y)

**\$25 Off a Full  
Compiler or Compiler  
Maintenance**

# CCS C Compiler Savings!



**Use Code: Summer23**



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.



**Follow Us!**



[www.ccsinfo.com](http://www.ccsinfo.com)