June 2021

CCS Inc

www.ccsinfo.com

262-522-6500

<BITS & BYTES> Newsletter

INSIDE THIS ISSUE:

Product Spotlight

RAPID 18 DEVELOPMENT KIT

This easy-to-use PIC18 family development kit has a built-in bootloader for loading code and a USB port for text communication to the running program. Microchip's PIC18F4523 provides advanced peripherals including a 12-bit ADC. The development kit includes the powerful PCWH Integrated Development Environment with compiler support for Microchip's PIC10, PIC12, PIC16 and PIC18 families and an ICD-U64 in-circuit programmer/debugger that supports C-aware real time debugging.

Extreme Code Optimizer

By: CCS Staff

CCS, Inc. If you have a version 5 compiler there is an optimization feature you may not be aware of. For many chips ther is a aggressive code optimizer available, optimizing for space instead of speed. The optimizer is able to search the entire compiled program to find repeating blocks of code whereby reducing all those repeating blocks into one shared sub-routine. Optimizer is executed during the final phase of the compile which presents the ability to cross a function boundary when performing the optimization. This can only be done on parts with a flexible call instruction and available stack, for example the PIC18 and PIC24 families of devices.

This optimization level can be achieved by adding this line of code into your project.

```
#opt compress
```

The average size reduction of program memory is approximately 15%. In some cases we have seen program memory reduced by 60%. Provided below are examples of compression levels:

Optimization Level Efficiency	**Not Compressed**	**Compressed**	
File / Processor	(Program Memory Bytes)		**Reduction%**
ex_modbus_master.c **PIC16F1937**	3926	3068	22.00%
ex_j1939.c **PIC18F4580**	7552	6166	18.00%
ex_st_webserver2.c **PIC18F4550 + ENC28J60**	60282	52664	12.00%

Consider this example. When a program that passes &n seven times to two functions and the code needed to do that takes 6 instructions. On the right we show the normal optimization and on the right #opt compress.

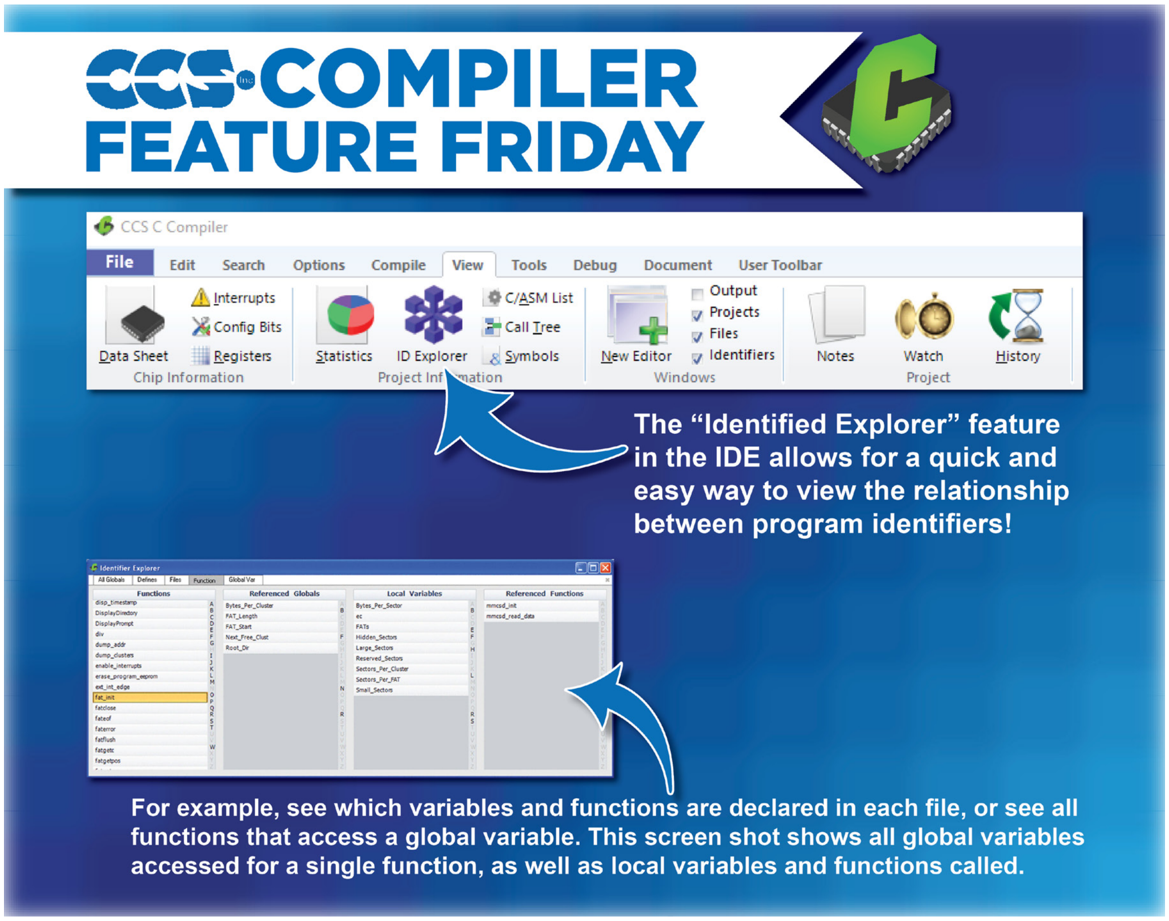
```
................ input_number_volts(&n);
781C:  MOVLW  01
781E:  MOVLB  1
7820:  MOVWF  xBB
7822:  MOVLW  B6
7824:  MOVWF  xBA
7826:  MOVLB  0
7828:  CALL   661C
     . . .
................ input_number_amps(&n);
7882:  MOVLW  01
7884:  MOVLB  1
7886:  MOVWF  xBB
7888:  MOVLW  B6
788A:  MOVWF  xBA
788C:  MOVLB  0
788E:  CALL   67F4
     . . .
................ input_number_amps(&n);
78CA:  MOVLW  01
78CC:  MOVLB  1
78CE:  MOVWF  xBB
78D0:  MOVLW  B6
78D2:  MOVWF  xBA
78D4:  MOVLB  0
78D6:  CALL   67F4

           (four more not shown)
```

```
6E92:  MOVLW  01
6E94:  MOVLB  1
6E96:  MOVWF  xBB
6E98:  MOVLW  B6
6E9A:  MOVWF  xBA
6E9C:  MOVLB  0
6E9E:  RETURN 0
     . . .
................ input_number_volts(&n);
6B5C:  RCALL  6E92
6B5E:  CALL   5DDC
     . . .
................ input_number_amps(&n);
6BB0:  RCALL  6E92
6BB2:  CALL   5F7C
     . . .
................ input_number_amps(&n);
6BE8:  RCALL  6E92
6BEA:  CALL   5F7C
     . . .
................ input_number_amps(&n);
6C20:  RCALL  6E92
6C22:  CALL   5F7C

              (three more not shown)
```

For more information on other advanced Version 5 features please visit: www.ccsinfo.com/version5.



The "Identified Explorer" feature in the IDE allows for a quick and easy way to view the relationship between program identifiers!

For example, see which variables and functions are declared in each file, or see all functions that access a global variable. This screen shot shows all global variables accessed for a single function, as well as local variables and functions called.

Optimize Code Using Fixed-Point instead of Floating point with the CCS C Compiler

By: CCS Staff

Most embedded microprocessors and microcontrollers, including the Microchip PIC® MCU families, do not contain a hardware implemented floating-point calculator. This means that all float calculations must be implemented in software using integer arithmetic, which is very resource intensive. In turn, performance heavy applications and resource limited platforms may be unable to utilize floating point numbers in their implementation. The solution for this is to instead use **fixed-point arithmetic**, which is simplified by the CCS C Compiler's fixed type feature.

Fixed-point arithmetic is an implementation that uses a scaling factor to represent decimal numbers in integer form. The CCS C Compiler implements this using 16 or 32 bit integers and a scaling factor of 10-n. A fixed type can be declared as follows:

int16 _fixed(n) foo; where 0 < n < 6
int32 _fixed(n) foo; where 0 < n < 11

The value n determines the number of decimal places of accuracy the variable will contain, as well as the maximum representable value. Since n is part of the type and determines what instructions are generated, it must be given as a constant at compile time.

For int16 _fixed(2) : Max = 65,535 * 10-2 = 655.35
For int32 _fixed(5) : Max = 4,294,967,295 10-5 = 42,949.67295

The binary/hexadecimal representation for any number can then be determined by multiplying by the inverse of the scaling factor and converting.

237.16 * 10-2 = 23716 = 0x5CA4

The fixed type is compatible with another fixed type of the same n for the 4 basic arithmetic operations. They are not compatible with fixed types with a different n.

```
int16 _fixed(1) f1 = 5.5;
int16 _fixed(1) f2 = 2.5;
f1 + f2; //evaluates to 8.0
f1 - f2; //evaluates to 3.0
f1 * f2; //evaluates to 13.7
f1 / f2; //evaluates to 2.2
```

It can also perform arithmetics with literals and cast integers.

```
int16 _fixed(2) f1 = 22.14;
int16 i1 = 7;
f1 + 19.52; //evaluates to 41.66
f1 - (int16 _fixed(2)) i1; //evaluates to 15.14
```

The increment and decrement operations function the same in binary. This equates to adding or subtracting 1 times the scaling unit.

```
int16 _fixed(3) f1= 5.234;
f1++; //f1 = 5.235
```

Casting a fixed type to an integer will truncate the decimal places. This can also be used to isolate the digits after the decimal by subtracting the integer cast from the original.

```
int16 _fixed(2) f1 = 6.94;
int16 noDec = (int16) f1; //noDec = 6
int16 _fixed(2) decOnly = f1 - noDec; //decOnly = 0.94
```

The CCS C Compiler also supports using printf, sprintf, etc. with the fixed-point type using the “%w” format flag. It will print the value with no leading zeroes and digits after the decimal equal to the precision.

Code:

```
int16 _fixed(2) f1 = 1.5;
int16 _fixed(2) f2 = 22.78;
int16 _fixed(3) f3 = 5.21
printf(“%w - %w - %w”, f1, f2, f3);
```

Output:

```
1.50 - 22.78 - 5.210
```

Floating-point numbers have an inescapable error when representing decimal in which expressions that should evaluate to be equal will be off at a very low decimal point. This happens because some decimal numbers in base10, such as 0.1, cannot be perfectly represented in base2, thus causing a rounding error. This is similar to how 1/3 cannot be represented in base10 and is then rounded to 0.33... to whatever precision is needed. Since the CCS C Compiler's implementation of fixed-point uses a decimal scalar, there is 100% precision in base10. This makes it perfect for handling money and other values where this precision is necessary.

Fixed-point operations yield significant performance increases over floating-point. In order to quantify this, benchmarks were performed using a PIC18F45K22 and the CCS C Compiler. One of the on-chip timers was used to approximate the amount of time an arithmetic operation took. For both floating-point and 16 bit fixed-point at 2 places accuracy, each operation was timed and averaged 50 times on a spread of values. The average times for each could then be compared to generalize performance.

The benchmarking results are as follows:

- Add (+): Fixed ~19.6 times faster than float.
- Subtract (-): Fixed ~19.4 times faster than float.
- Multiply (*): Float ~2.7 times faster than fixed.
- Divide (/): Fixed ~3.3 times faster than float.

The only operation that floating-point performs better is multiplication. This is logical since floats are stored in a multiplicative form (significand x baseexponent). However, fixed-point performs better on the other three, particularly on addition and subtraction.

There is also a program memory usage difference between implementing fixed-point and floating-point. The actual number of instructions it takes to implement the arithmetic operations is significantly different for both. The following program was compiled using both options for the PIC18F45K22.

```
#ifdef USE_FIXED
int16 _fixed(2) a, b, c;
#else
float a, b, c;
#endif

void main() {
a = 2.25;
b = 0.85;
c = a + b;
printf("Add: %w", c);
c = a - b;
printf("Subtract: %w", c);
c = a * b;
printf("Multiply: %w", c);
c = a / b;
printf("Divide: %w", c);
}
```

The compiled program’s memory statistics were as follows.
Fixed Option:

- 364 instructions
- 0.84KB ROM usage
- 2.6% ROM usage

Float Option:

- 787 instructions
- 2.03KB ROM usage
- 6.3% ROM usage

Switching from the fixed implementation to float increased the instruction count by over 400. This may or may not be significant on the PIC18F45K22, depending on program complexity. However, at 32KB of ROM, it is on the higher end in terms of program memory. On the more limiting units in the PIC18 family, this would be an extremely significant amount of space or may not be significant on the PIC18F45K22, depending on program complexity. However, at 32KB of ROM, it is on the higher end in terms of program memory. On the more limiting units in the PIC18 family, this would be an extremely significant amount of space.

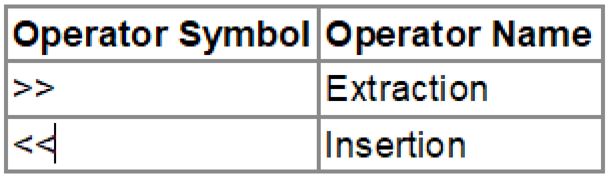
C++ Like Console Stream Operator Support

By: Mark Siegesmund

In standard C, basic I/O are handled by functions like `getc()`, `putc()` and `printf()` and the formatting of data is handled by functions like `atoi()`, `atof()`, `strotul()` and `sprintf()`. For example, reading a floating point number from the user over RS232 would require a combination of `gets()` followed by `atof()`. While CCS includes an input.c library that accomplishes many of these tasks, the input.c library uses a fixed RS232 stream and does not work with Keypad/LCD or USB without modification. CCS has added some support for the C++ stream operator to make it easier to handle routine user input and output.

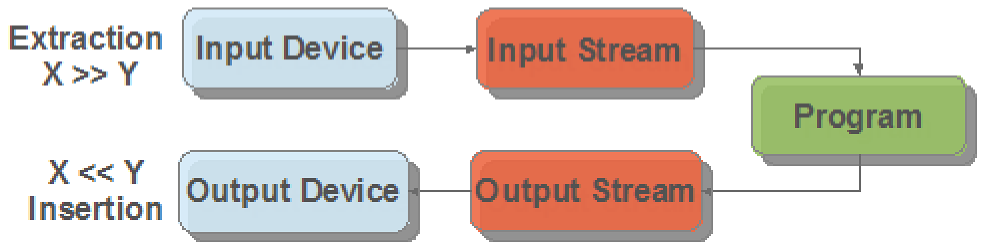
One of the key features of this new feature in the CCS C Compiler is the way it automatically handles the conversion based upon the data types of the variables passed. These conversions are done automatically, no other helper functions like atof() need to be called. For example, if a variable is of float type the compiler will properly convert it from string to float on an input or convert float to string on an output.

The two new operators added are the extraction operator and the insertion operator:
Operator Symbol
Operator Name



Operator Symbol	Operator Name
>>	Extraction
<<	Insertion

When used, these operators show the direction of data. For example:



The beauty of these operators is that the x and y in the above examples can be any combination of function, RS232 serial stream, variable, string and more.

__Using the stream operator for output__

```
Simple examples:
     int16  v16;
     v16 = 1234;
     cout << v16;                 // Outputs 1234
     cout << hex << v16 << eoln;  // Outputs 4D2 \r\n
```

Formal definition:
stream << expresion [<< expresion]...
stream in the above example can be one of the following:

- cout - maps to the default #use rs232() stream. This provides compatibility for using existing C++ code that explicitly uses the cout stream class.
- RS232 stream name - A stream identified with the stream=x option of #use rs232().
- char array (string) - Data is parsed from identifier and saved to the char array with a null terminator.
- function - A function that takes a char for it's input variable. For example, lcd_putc() in CCS's lcd.c driver or usb_cdc_putc() in CCS's usb_cdc.h driver. The function is called for each

expresion can be can be an int, float, fixed point decimal, int1 (boolean) or char array (string).
expresion can also be any of the following manipulators (from ios.h):

- **hex** - When converting variable to string, convert it to hex format characters (similar to %x in printf()).
- **dec (default)** - When converting variable to string, convert it to decimal format characters (similar to %d in printf())
- **setprecision(x)** - set number of places after the decimal point.
- **setw(x)** - set number of characters output for numbers
- **boolalpha** - output int1 as "true" or "false"
- **noboolalpha (default)** - output int1 as "1" or "0"
- **fixed (default)** - floating point numbers displayed in decimal format (similar to %f in printf())
- **scientific** - floating point numbers displayed in E notation (similar to %e in printf())
- **iosdefault** - all modifiers back to default settings
- **endl** - output CR/LF

Here is a fuller example usage of this operator:

```
cout << "Price is $" << setw(4) << setprecision(2) << cost*num << endl;
```

This example transmits “Price is $” followed by the result of cost*num with two decimal places followed by the CR/LF. This is transmitted using cout, which is the default RS232 stream.

Here is a change to the above example to display on the LCD using the lcd_putc() function provided in CCS’s lcd.c driver:

```
lcd_putc << "Price is $" << setw(4) << setprecision(2) << cost*num
         << endl;
```

Here is a change to the above example to save the formatting to a string variable called result_string:

```
result_string << "Price is $" << setw(4) << setprecision(2) << cost*num
              << endl;
```

<u>**Using C++ stream operator for input**</u>

stream >> identifier [>> identifier]...
stream in the above example can be one of the following:

- cin - maps to the default #use rs232() stream. This provides compatibility for using existing C++ code that explicitly uses the cout stream class.
- RS232 stream name - A stream identified with the stream=x option of #use rs232().
- function - A function that returns a char. For example usb_cdc_getc() in CCS’s usb_cdc.h driver. This function is called for each character, until a \r is received.
-

identifier can be a variable that is integer, char, char array, float or fixed point integer type. Float type formats can use the E format.
identifier can be any of the following manipulators:

- **hex** - hex format numbers
- **dec (default)** - decimal format numbers
- **strspace** - allow spaces to be input into strings
- **nostrspace (default)** - spaces terminate string entry
- **iosdefault** - all manipulators to default settings.

Here is an example of reading a number from the user and saving it to the variable value:

```
          cout << "Enter Number";
          cin >> value;
```

The above example can be quickly modified to read from the USB virtual COM port using the routines in CCS’s usb_cdc.h driver:

```
usb_cdc_putc << "Enter Number";
usb_cdc_getc >> value;
```

Several values can be read at a time:

```
cin << variable1 << variable2 << variable3;
```

In the above example, the input operator would stop reading into **variable1** and start reading into **variable2** once a character is received that is not valid for that data type. For instance, if **variable1** and **variable2** are both int, it would stop reading into **variable1** and start reading into **variable2** upon the reception of any character that isn't "0" to "9", like a space or new-line.

Data conversion from a string to a variable can also be achieved. This example converts the **`str`** string variable to the val variable. The type of conversion is determined by the data type of val:

```
str >> val;
```

USB Development Kit

Includes

- USB Prototyping Board
- In-Circuit Debugger/ Programmer
- Exercise Tutorial
- 9V AC Adapters and Cables

sales@ccsinfo.com
262-522-6500 EXT 35
www.ccsinfo.com/NL521

Summer is for C Compiler Savings

$25 Off a Full Compiler or Compiler Maintenance

Use Code:Summer2021

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

Follow Us!