CCS C Compiler

Functional changes from version 3 to version 4:

New Windows IDE Features:

Download Manager

The download manager checks your system for all CCS files and determines what needs updating from the web. The program uses multiple ports for fast downloads and is able to continue downloads that are interrupted.

In addition to the regular compiler software, this utility will check for reference programs such as the TCP/IP stack and USB PC code. Of course, all drivers and utilities are also checked, and it can even update Microchip PIC® MCU datasheets if desired. The PCW IDE allows the user to configure how often the utility check for new updates.

Updates are automatically downloaded and installed and if desired old versions can be automatically archived.

Modern IDE Look

The PCW IDE uses the newest Windows styles. The traditional menu bar has been replaced with the Office 2007 style Ribbons (shown below), navigation bars are provided as slide out windows and all the dialog boxes use the newest controls including integrated help.

For those long-time customers that prefer the Windows 98/2000/XP look over the new Windows Vista, you can easily change the IDE back by executing the following:

1. Go to Options tab and choose the Editor Properties icon.
2. Click on Display icon on the left-side of the window.
3. At the bottom of the window, click on Classic Menu under Menu Scheme

Vista Ready

The PCW IDE has a solid integration with Vista for those users with the new operating system. Compatibility with Vista is guaranteed for all Version 4 owners.

If you have Version 4 and there is an operational problem with Vista, you will get an update for free even if your download rights have expired.

Project Navigation by File

A navigation bar shows all project related files. This allows for quickly opening or compiling a file from the navigation bar. The navigation bar can even be used to track non-C files associated with the project such as project documentation files.

Project Navigation by Identifier

Another navigation bar shows all project functions and identifiers. Double click on an identifier to move the editor cursor to the location where the identifier is declared.

Code Aware Editor

The source file editor has been improved to make it a more powerful tool for editing source files. Editor macros allow keystrokes to be recorded and played back. For example you could search for a function name, then search for a comma twice, then insert a FALSE,. Recorded as a macro you could use this to insert a new third parameter to each call to the function by repeatedly pressing the playback button. This is great for importing external data that needs to be molded into code, such as a text file with a list of strings that need to be formatted as a valid C initializer.

Code folding allows a function to be reduced to just one line in the editor to aid in readability of the code. Change bars show code that was recently changed. Editor window tiling allows side by side files to be viewed.

Technical Support Wizard

The PCW IDE includes a technical support wizard that may be used to report problems or ask questions of technical support. Customers with active maintenance that use this wizard get their requests marked at a higher priority at CCS. The wizard offers an easy option to zip up your project files to be sent to CCS for analysis.

Project Wizard

The Project Wizard includes code generation for higher level functionality such as generating a CAN Bus application or making a custom bootloader.

C-Metrics Calculator

The compiler evaluates the code cyclomatic and computational complexity using the McCable and Halstead Complexity measures. The derived Maintainability Index gives a good insight into the structural and textual complexity of the source code.

Automatic Documentation Generator

The compiler recognizes comments that have specially marked tags as containing information that should be exported for program documentation. The compiler associated comments with variables and functions plus the compiler accumulates information on its own about functions, variables and types. The new documentation generator uses a user generated template in .RTF format and merges in the project information from the source files and generates a .RTF output file as source code documentation.

Multiple Compilation Units

Version 4 includes support for separately compiling source code files and then linking them together for the final build. The PCW IDE has tools to easily define the units, figure out what units need to recompiled on a build and automatically calling the linker.

Better Windows Integration for File Associations

File types (like .C) can be associated with PCW as a user preference. Doubling clicking on a file brings it up in the PCW editor. The user preference screen allows files to be re associated with the program they were associated with before PCW for easy control over the associations. All PCW file types can be configured and output files (like .HEX) can be assigned directly to the ICD software.

PCW allows the establishment of a project directory and makes it easy to navigate to the project and CCS examples directory.

Configurable Desktop

The desktop layout is very configurable and the PCW IDE allows desktop configurations to be saved and loaded. In addition, the default layouts can be specified separately for editing and debugging.

Special Viewers

PCW recognizes all the standard file types and has special viewers to view the files in an optimum format. Improved viewers include the statistics file, call tree and hex files.

Special Function Register Reference

The PCW IDE allows the viewing of all SFR's for a given part and has the ability to generate custom .h files with the register and bit definitions for specific functions.

Device Errata Database Viewer

The compiler has always kept track of errata and has compensated where possible. With version 4, you can view the errata database to see what errata is associated with a part and if the compiler has compensated for the problem.

Device Selector

The device selector makes it easy to find parts with the right number of pins, memory and other criteria. Grids can be sorted by any column or easily printed. Individual rows can selected to find more details on the part and to check distributor price and availability.

Ability to Generate Customized Include Files with SFR Definitions

Automatically generate a custom include file according to the device parameters you need.

Ability to Generate C Constant Declarations from a Hex or Binary File

The PCW IDE has a tool to import a binary or hex file and create from it a C declaration to make it easy to include images or a bootloader into your C program.

Flow Chart Editor

A flow chart editor is included with PCW for quick and easy charting. This tool can also be used to generate simple graphics including schematics. This is all part of our version 4 effort to make documentation a more integral part of the design process.

RTF Documentation Editor

PCW comes with a full featured RTF editor to make integration of documentation into the project easier.

Integration with External Tools

External programs can be defined to appear on the toolbar, in the programmer options or debugger options. This allows easy integration with a number of other programmers and debuggers. In addition, tools can be set up to run whenever PCW starts or after a compilation.

Debugger Script Execution for Automated Testing

The PCW debugger can be controlled from a C like scripting language. This allows automated tests to be generated and easily executed and re-executed at will. The scripting language allows for results to be logged to a PC disk file and/or evaluated with the script.

Spellchecker

The Version 4 IDE has the ability to spell check all words in comments and/or quotes.

New C Language Features:

Relocatable Objects / Multiple Compilation Unit*

Prior to Version 4, the compile step and linking step were combined, and the user didn't have the ability to compile a unit into a relocatable object to be linked together with other objects. Dividing up your code into several units will not only increase the maintainability of your projects, but it can speed up compilation as only units that have been changed since last build need to be compiled again.

Version 4 has several methods of allowing you to create relocatable objects and link them together:

1. Inside PCW, the CCS IDE, you can use the file navigator to add units to your project. When you press Build it will compile all your units and link them together.
2. #import() and #export() preprocessor commands allow you to create units and link units at the source code level.

   /*
   The following preprocessor command will make the compiler compile this code into a relocatable object file (.o).

RELOCATABLE will create a CCS relocatable object file, and is the default format when #EXPORT is used.

FILE specifies the name of the output file, and is optional.

The ONLY option tells the compiler to only make the symbol GetString (which could be a function or variable) visible to modules that import/link this module.

Another option, EXCEPT, has the opposite effect - all symbols except the listed symbols are visible to modules that import/link this module.
*/
#export(RELOCATABLE, FILE=getstring.o, ONLY=GetString)

/*
The following preprocessor command will make the compiler link other objects into this unit (in this example it will link unit.o).

FILE, ONLY, and EXCEPT have the same operation as detailed in the previous example.

COFF is the reciprocal of RELOCATABLE, and tells the compiler to link/import an .o created by MPASM. C18 and C30. COFF can only be used with #import, it cannot be used with #export.
*/
#import(RELOCATABLE, FILE=unit.o)

As shown in the previous examples #import and #export have options for defining scope, but the standard C qualifiers static and extern can also be used for defining scope.

3. Command-line options have been added to create units (instead of a compile/link in one step) and link all units into one final HEX. The following example shows compiling two .C files into seperate .O files and then linking them into one .HEX file:

   C:\project\ccsc +FH +EXPORT main.c
   C:\project\ccsc +FH +EXPORT uart.c
   C:\project\ccsc +FH LINK="main.hex=main.o,uart.o"

4. Inside MPLAB® IDE, if you add more than one .C in the project manager all the .C files will be compiled seperately into relocatable objects, and in the final step all relocatable objects are linked into one HEX.

*** Command-Line compiler customers can only link, they cannot create relocatable objects. Relocatable Objects/Linker functions work only with the CCS C Windows IDE.**

Generation of multiple hex files for chips with external memory

Two additional preprocessors are included in Version 4: #export and #import. You have may have already noticed these being used in the above examples for creating relocatable objects, but it has other options for specifying how to export and import HEX files. You will find these commands useful if creating bootloaders or using a device with external memory (CPU or EMCU mode):

```
/*
The compiler will create two HEX files. One contains all the ODD addressed program memory, the
other contains all the EVEN addressed program memory. You will find this useful if you are using the
16-bit byte write mode on the 18F family external memory interface.
*/
#export(HEX, file=odd.hex, ODD)
#export(HEX, file=even.hex, EVEN)

/*
When the compiler creates the HEX file for this project, offset all addresses up 0x800 bytes.
*/
#export(HEX, file=application.hex, offset=0x800)

/*
The previous example was an application where the final HEX file had it's addresses offset by 0x800.
If we wanted to import the loader HEX into the application this could have been done instead:
*/
#import(HEX, file=loader.hex, range=0:0x7FF)
```

Variable length constant strings

Prior to Version 4, if you had an array of constant strings it was usually inefficient. Examine this example of an ineffecient array of constant strings:

```
const char strings[3][15] =
{
  "HELLO",
  "WORLD",
  "EXTRALONGERWORD"
};
```

In the above example we had to make the maximum length of each string be 15 characters because of the length of "EXTRALONGERWORD". But since "HELLO" and "WORLD" are only 6 characters (don't forget null termination), 9 bytes are wasted for each.

To alleviate this problem, use this method for variable length constant strings:

```
const char strings[][*] =
{
  "HELLO",
  "WORLD",
  "EXTRALONGERWORD"
};
```

Note: this is done by adding extra intelligence to the indexing of the constant data table. Because of this you cannot create a pointer to a variable length constant string.

More flexible handling of constant data

Version 4 has a few ways of allowing pointers to constant data. First, Version 4 adds pointers to constants:

```
/*
```

A simple example showing the assignment of a pointer to a constant with the address of a constant string:

```
*/
const char version[] = "PRODUCT ID V1.01";
const char *ptr;

ptr = &version[0];
```

```
/*
A more complex example that creates an array of pointers to constant strings:
*/
const char *strings[] =
{
  "HELLO",
  "WORLD",
  "CONST",
  "STRINGS"
};

/*
Access the above const pointers
*/
const char *ptr;
while (i = 0; i < (sizeof(strings) / sizeof(const char *)); i++)
{
  ptr = strings[i];
  printf("%s", ptr);
}
```

In addition, constant strings can be passed to functions that are normally looking for pointers to characters:

```
/*
The following enables the ability for the compiler to copy constant strings into RAM when being passed as a parameter to a function:
*/
#device PASS_STRINGS=IN_RAM

/*
An example of using this new feature:
*/
if (stricmp(buffer,"ATDT\r")==0)
{
  //do something
}
```

Note: The const qualifier in CCS always means that the data will be placed in program memory, and that the data is 'read-only'. It does not follow the ANSI definition which simply states that const is 'read-only'.

addressmod capability to create user defined address spaces in any kind of memory device

Part of the IEEE Embedded C standard (ISO/IEC TR 18037), addressmod allows you to create custom qualifiers to create variables in any kind of memory device. The identifier can be used with any data types, including structures, unions, arrays, and pointers. Review the following example, which uses addressmod to create variables that are located in external memory:

```
/*
Syntax for addressmod is:
addressmod (identifier,read,write,start,end)
  identifier - your new custom identifier name
  read/write - the read/write functions to access the external memory
  start/end - the range of addresses this identifier can access
*/
addressmod(extram, readextram, writeextram, 0, 0xFFFF)

/*
Create a large array, the actual contents of this array will be stored in the external memory.
*/
extram largeBuffer[2000]

/*
Create a pointer to the external memory. The pointer itself will be stored in the PIC's memory.
*/
extram char *extramPtr;

/*
Some examples of usage
*/
//direct access
largeBuffer[0] = 5;

//assign pointer, get address
extramPtr = &largeBuffer[0];

//access external memory indirectly
*extramPtr = 5;
```

Function Overloading

Version 4 has borrowed a C++ feature called function overloading. Function overloading allows the user to have several functions with the same name, with the only difference between the functions is the number and type of parameters.

```
/*
Here is an example of function overloading: Two functions have the same name but differ in the types
of parameters. The compiler determines which data type is being passed as a parameter and calls the
proper function.
*/
```

```
void FindSquareRoot(long *n)
{
/*
This function finds the square root of a long integer variable (from the pointer), saves result back to pointer.
*/
}

void FindSquareRoot(float *n)
{
/*
This function finds the square root of a float variable (from the pointer), saves result back to pointer.
*/
}

/*
FindSquareRoot is now called. If variable is of long type, it will call the first FindSquareRoot() example. If variable is of float type, it will call the second FindSquareRoot() example.
*/
FindSquareRoot(&variable);
```

Default Parameters

Default parameters have also been borrowed from C++ and have been included in Version 4. Default parameters can be specified in your functions, and if you don't pass the parameter to your function the default will be used.

```
int mygetc(char *c, int n=100)
{
/*
This function waits n milliseconds for a character over RS232. If a character is received, it saves it to the pointer c and returns TRUE. If there was a timeout it returns FALSE.
*/
}

//gets a char, waits 100ms for timeout
mygetc(&c);

//gets a char, waits 200ms for a timeout
mygetc(&c, 200);
```

Variable number of Parameters

You can use functions with a variable number of parameters in Version 4. This is found most commonly when writing printf and fprintf libraries.

```
/*
stdarg.h holds the macros and va_list data type needed for variable number of parameters.
*/
```

```
#include <stdarg.h>

/*
A function with variable number of parameters requires two things. First, it requires the ellipsis (...),
which must be the last parameter of the function. The ellipsis represents the variable argument list.
Second, it requires one more variable before the ellipsis (...). Usually you will use this variable as a
method for determining how many variables have been pushed onto the ellipsis.

Here is a function that calculates and returns the sum of all variables:
*/
int Sum(int count, ...)
{
  //a pointer to the argument list
  va_list al;

  int x, sum=0;

  //start the argument list
  //count is the first variable before the ellipsis
  va_start(al, count);

  while(count--) {
    //get an int from the list
    x = var_arg(al, int);

    sum += x;
  }

  //stop using the list
  va_end(al);

  return(sum);
}

/*
Some examples of using this new function:
*/
x=Sum(5, 10, 20, 30, 40, 50);
y=Sum(3, a, b, c);
```

CCS Backwards Compatibility

CCS provides a method to attempt to make sure you can compile code written in older versions of CCS with minimal difficulty by altering the methodology to best match the desired version. Currently, there are 4 levels of compatibility provided: CCS V2.XXX, CCS V3.XXX, CCS V4.XXX and ANSI. Notice: this only affects the compiler methodology, it does not change any drivers, libraries and include files that may have been available in previous versions.

- #device CCS2

    - ADC default size is set to the resolution of the device (#device ADC=10, #device ADC=12, etc)
    - boolean = int8 is compiled as: boolean = (int8 != 0)
    - Overload directive is required if you want to overload functions
    - Pointer size was set to only access first bank (PCM *=8, PCB *=5)
    - var16 = NegConst8 is compiled as: var16 = NegConst8 & 0xFF (no sign extension)
    - Compiler will NOT automatically set certain #fuses based upon certain code conditions.
    - rom qualifier is called _rom
- #device CCS3
    - ADC default is 8 bits (#device ADC=8)
    - boolean = int8 is compiled as: boolean = (int8 & 1)
    - Overload directive is required if you want to overload functions
    - Pointer size was set to only access first bank (PCM *=8, PCB *=5)
    - var16 = NegConst8 is compiled as: var16 = NegConst8 & 0xFF (no sign extension)
    - Compiler will NOT automatically set certain #fuses based upon certain code conditions.
    - rom qualifier is called _rom
- #device CCS4
    - ADC default is 8 bits (#device ADC=8)
    - boolean = int8 is compiled as: boolean = (int8 & 1)
    - You can overload functions without the overload directive
    - If the device has more than one bank of RAM, the default pointer size is now 16 (#device *=16)
    - var16 = NegConst8 is will perform the proper sign extension
    - Automatic #fuses configuration (see next section)
- #device ANSI
    - Same as CCS4, but if there are any discrepancies are found that differ with the ANSI standard then the change will be made to ANSI
    - Data is signed by default
    - const qualifier is read-only RAM, not placed into program memory (use rom qualifier to place into program memory)
    - Compilation is case sensitive by default
    - Constant strings can be passed to functions (#device PASS_STRINGS_IN_RAM)

Automatic #fuses configuration

Version 4 configures some of the configuration bits (#fuses) for you automatically based upon your code:

- By default, the NOLVP fuse will be set (turn off low voltage programming)
- By default, the PUT fuse will be set (turn on the power-up timer)
- If there is no restart_wdt() in your code, it will set the NOWDT fuse. If there is a restart_wdt() fuse in your code then it will set the WDT fuse.
- The oscillator config bits will automatically be set based upon your #use delay() (see next section)
- If you have the debugger enabled in the PCW IDE, the DEBUG fuse will be set.

With the basic #fuses now being set automatically many programs will not need a #fuses directive.

This feature can be disabled by using the CCS3 backwards compatability (see previous section).

#USE DELAY() Improvements

Commas, periods and speed postfixes now supported for delay parameter

By using commas, periods and speed prefixes, the delay parameter is now easier to read. The following speed postfixes are supported: M, MHZ, K, KHZ. See the next section for an example.

#use delay() has new parameters: OSCILLATOR (or OSC), CRYSTAL (or XTAL), RC (or RC), INTERNAL (or INT)

By using these new parameters you can automatically configure your #fuses (configuration bits) for the proper oscillator type and PLL. Here are some examples:

```
/*
We are using a 12MHz crystal. This will set the HS fuse:
*/
#USE DELAY(CLOCK=12MHZ, CRYSTAL)

/*
We are using an external RC oscillator at 4000345 Hz. This will set the RC fuse:
*/
#USE DELAY(RC=4000345)

/*
We are using an external crystal at 10MHz, but because of the HS PLL the actual system will run at 40 MHz. This will set the H4 fuse:
*/
#USE DELAY(CLOCK=40MHZ, CRYSTAL=10MHZ)

/*
We are using an internal oscillator at 8 MHz. This will set the INTRC_IO fuse and configure the internal oscillator to run at 8 MHz:
*/
#USE DELAY(INTERNAL=8000000)
```

#USE SPI()

Some of CCS's most powerful libraries have been the RS-232 and I2C libraries, which give users the flexibility of using multiple RS-232 and I2C ports at once using any set of general purpose I/O pins, and not tying the user to only using the hardware peripheral. In Version 4, SPI libraries are included to give the user: use of any general purpose I/O pins, clocking configuration, any number of data bits, streams, clock rate and more!

```
/*
The #use SPI configures the SPI port. Here is a simple configuration:
*/
#use SPI(
  DO = PIN_B0,
  DI = PIN_B1,
```

```
  CLK = PIN_B2,
  baud = 100000,
  BITS = 8,
  LSB_FIRST,
  SAMPLE_RISE,
  stream = SPI_PORT0
)

/*
Read a byte of data to a 9356 external EEPROM using this new SPI stream
*/
void Read9356(long address, int data)
{
  output_high(EEPROM_9356_SELECT);
  SPI_XFER(SPI_PORT0, 0x18);
  SPI_XFER(SPI_PORT0, address);
  data=SPI_XFER(SPI_PORT0, 0);
  output_low(EEPROM_9356_SELECT);
}
```

#USE RS232() upgrade

The powerful RS-232 library upgrade in Version 4 includes the following options:

- Two additional parameters to #use rs232(): UART1 and UART2. Choosing one of these parameters will automatically set the transmit and receive pin of the CCS RS232 library to the specified hardware MSSP transmit and receive pins of the PIC® MCU.
- A timeout parameter is included, which will cause getc() to timeout within specified number of milliseconds.
- A clock parameter is included, so you can specify the system clock speed instead of using the clock specified in the #use delay. When the clock parameter is not specified, it will use the clock speed specified in the #use delay.
- The number of stop bits can be defined.
- You can use RS-232 in synchronous master or synchronous slave.
- The baud rate option supports commas, periods, and the following prefixes: K, KHZ, M, MHZ. For example, these are now valid: BAUD=9.6k, BAUD=115,200, BAUD=115.2K, etc.

#USE I2C() upgrade

The powerful I2C library upgrade has been included in Version 4. First, you can give your I2C ports different stream identifiers. By giving your different I2C channels a stream identifier, it is easier to differentiate in your code which port is being used.

Second, the i2c_start() function can send an I2C start or I2C restart signal. The restart parameter for i2c_start() may be set to a 2 to force a restart instead of a start. A 1 value will do a normal start. If the restart is not specified or is 0, then a restart is done only if the compiler last encountered a i2c_start() and no i2c_stop().

```
/*
This configures two I2C ports, each has a different stream name.
```

```
*/
#use i2c(sda=PIN_C4, scl=PIN_C3, stream=I2C_HW)
#use i2c(sda=PIN_B1, scl=PIN_B2, stream=I2C_SW)

/*
The following function reads a data byte from a Microchip 24LC16 I2C EEPROM, using the I2C_HW
stream
*/
int Read2416(int address)
{
  i2c_start(I2C_HW, 1); //perform a start
  i2c_write(I2C_HW, 0xA0);
  i2c_write(I2C_HW, address);
  i2c_start(I2C_HW, 2); //perform a restart
  i2c_write(I2C_HW, 0xA1);
  data=i2c_read(IC2_HW, 0);
  i2c_stop(I2C_HW);
  return(data);
}
```

Bit Arrays

You can create an array of bits (or booleans). You cannot create a pointer to an array of bits or to a bit.

```
/*
This will create an array of bits, and initialize their values
*/
int1 flags[]={TRUE, TRUE, TRUE, FALSE, FALSE, FALSE};

/*
Some usages:
*/
bool = flags[1];

if ( flags[2] ) { /* do something */ }

flags[i++] = FALSE;
```

Fixed point decimal

A powerful feature in Version 4 is the ability to represent decimal numbers using a new data type, the fixed point decimal. Fixed point decimal gives you decimal representation, but at integer speed. This gives you a phenomenal speed boost over using float. This is accomplished with a qualifier: _fixed(x). The x is the number of digits after the decimal the data type can hold.

```
/*
Creates a 16 bit variable with a range of 0.00 to 655.35
*/
int16 _fixed(2) dollars;
```

```
/*
Assign 1.23 to dollars. Internally, 123 will be saved to the int16.
*/
dollars=1.23;

/*
Add 3.00 to dollars. Internally, 300 will be added to the int16.
*/
dollars += 3;

/*
printf will display 4.23
*/
printf("%w", dollars);
```

delay_us() and delay_ms() support int16 variable parameters

Prior to Version 4, delay_us() and delay_ms() could only take int16 parameters if they were constant values. Version 4 allows you to use delay_us() and delay_ms() with int16 variable parameters.

```
/*
This function delays s seconds.
*/
void delay_s(int s)
{
  int16 ms;

  ms = s * 1000;

  delay_ms(ms);
}
```

General Purpose I/O Improvements

output_high(), output_low(), output_toggle() and input() now support variable parameters

The built in functions OUTPUT_LOW(), OUTPUT_HIGH(), OUTPUT_BIT() and INPUT() accept a variable to identify the pin. The variable must have a value equal to one of the constants (like PIN_A1) to work properly. The tristate register is updated unless the FAST_IO mode is set on port A. Note that doing I/O with a variable instead of a constant will take much longer time.

```
/*
This function gets bit n of port D
*/
int GetPinD(int n)
{
  int16 pin;

  pin = PIN_D0 + n;
```

```
  return(input(pin));
}
```

output_drive()

The output_drive() function sets the desired pin's tristate to output. This is the opposite of output_float().

get_tris_x()

The get_tris_X() function returns the current tristate setting for that port (where X is the desired port, ie get_tris_e())

Interrupts Improvements

The interrupt_active(INT_XXX) function returns TRUE if the specified interrupt flag is set, meaning the interrupt condition has triggered. INT_XXX is a valid constant, only valid interrupts on your target PIC® MCU are allowed (ie INT_TIMER0, INT_CCP, etc). This will be useful if you want to poll the flag instead of using interrupts.

Preprocessor Improvements

== and != can be used to evaluate strings at the preprocessor level

The following example is now legal:

```
#if (getenv("DEVICE") != "PIC16F877A")
  #error This target PIC is not supported!
#endif
```

getenv() can return byte and bit addresses of special file registers

The getenv() function has two additional options to get the address of special function registers. The strings SFR:name and BIT:name may be used to get the address of a SFR. For example:

```
/*
On a PIC16F877A, returns a string of the form 0x003 (where the STATUS SFR is located)
*/
#byte status_reg = GETENV("SFR:STATUS")

/*
On a PIC16F877A, returns a string of the form 0x003.0 (where the carry bit is located)
*/
#bit carry_flag = GETENV("BIT:C")
```

The following parameters to getenv() are available in Version 4:

- RAM - Returns the number of bytes of RAM the target PIC® MCU has

#type upgraded

The #type upgrade defines if the default data type is signed or unsigned. By default the CCS C Compiler uses unsigned.

New Peripherals Support

- setup_opampX() - Enable / Disable the integrated operational amplifier on PICmicro® MCUs that have an integrated operational amplifier (see PIC16F785)
- sleep_ulpwu(us) - Sets the ultra-low power wakeup pin high for the set micro-seconds (which will charge the user's capacitor), then put's the PIC® MCU to sleep. When the capacitor has discharged, the PIC® MCU will awaken (this time is determined by the user's RC time constant). Requires a PIC® MCU with the ultra-low power wakeup feature (see the GP0 on the PIC12F683).

Many more example programs and libraries

These are some of the additional libraries and examples included in Version 4:

- FAT - Access/read/write files on a SD/MMC card that has a FAT file system. Run a long term log on your PIC® MCU, saving each entry by appending to a file. Then read the results on your PC by opening the file.
- SIM/SMART Card - Access the contact and phone number information on a SIM/SMART card, commonly found on GSM/GPRS cell phones.
- Frequency Generator - Generate tones, frequencies and DTMF using only one user define general purpose I/O pin. This is done by implementing a software PWM on one pin.
- XTEA Cipher Library - Transmit encrypted data between two PIC® MCUs using XTEA Encryption.
- XML Parser - Parse XML documents using the minimal resources of a PIC® MCU.