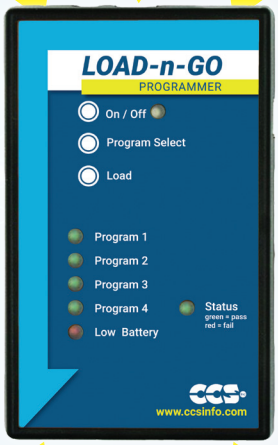
March 2021

www.ccsinfo.com
262-522-6500

<Bits & Bytes> Newsletter

Product Spotlight



Load-n-Go

Our LOAD-n-GO Programmer is the ultimate solution for programming productivity. Simply load the program into the battery powered device with a USB, connect to that target chip and program.
IT'S THAT EASY!

INSIDE THIS ISSUE:

Dealing With Out of ROM Errors

By: Mark Siegesmund

Often times the "Out of ROM, a segment or the program is too large" error pops up near the end of a project and users are now forced to deal with this new problem. This article will help to understand the causes and some simple solutions to get back to the project.

About Segment Sizes

The first thing to realize is that users may get this error and have plenty of ROM left. The reason is the CCS C Compiler does not split a function across a ROM page boundary. This error message always produces some informational lines in the .err file to help understand what is going on. Here is an example output for the case where a function is too large for a segment.

```
C:\PIC\main.c:1231:1:  Error#71  Out of ROM, A segment or the program is too large
process_data
  Seg 00000-00002, 0000 left, need 009C7 Reserved
  Seg 00003-00003, 0001 left, need 009C7
  Seg 00004-00055, 0000 left, need 009C7 Reserved
  Seg 00056-007FF, 003D left, need 009C7
  Seg 00800-00FFF, 042E left, need 009C7
  Seg 01000-017FF, 0800 left, need 009C7
  Seg 01800-01FFF, 0800 left, need 009C7
  Seg 02000-027FF, 0800 left, need 009C7
  Seg 02800-02FFF, 0800 left, need 009C7
  Seg 03000-037FF, 0800 left, need 009C7
  Seg 03800-03FFF, 0800 left, need 009C7
```

The first thing to notice is the "process_data." This is the name of a function in the users program that caused the linker to stop. Then look at the number after "need", this is how many words would be needed for that function (process_data). Sometimes it is different in different areas of memory because extra instructions are needed to jump to other pages.

The ROM in the chip is split into segments. Each segment is shown in this list. The first and third are marked as reserved, this is the reset jump and interrupt handler. These segments cannot be relocated. Of the remaining segments you will notice there is never more than 800 (2K) available (left). This is because on this chip the page size is 0x800.

There is plenty of ROM available on this chip but the function takes 9C7 and that does not fit into the 800 segment size. The simple solution is to split the function into two.

We most commonly see this error in technical support with main() as the function and there are no other functions in the program. It is always good to adapt a coding style where there are some limits on function size to aid in readability, maintainability and to keep sanity. It is recommended that a function should fit on a screen or at least a page.

It is possible to get an error like the above when users have a well structured program with lots of small functions. Consider the following example:

```
void func_b(void) {
...
}

void func_c(void) {
...
}

void func_d(void) {
...
}

void func_a(void) {
       funct_b();
       funct_c();
       funct_d();
}
```

If funct_b(), funct_c() and funct_d() are each only called once then the compiler to save valuable stack space will copy the code from those functions to funct_a(). This makes funct_a() way bigger that it appears.

To solve the problem users can tell the compiler to never inline the function like this:

```
#separate
void func_b(void) {
...
}
```

Note that when prototyping the function there must also have the #separate there.

Here are the page (max segment) sizes for each family:

PCB	0x200	0.5K
PCM	0x800	2K
PCH	0x8000	32K
PCD	0x8000	32K

Fragmentation

Another problem maybe the way the segments are being used. Users can see something like this:

```
Seg 00000-00002, 0000 left, need 00261 Reserved
Seg 00003-00003, 0001 left, need 00261
Seg 00004-00055, 0000 left, need 00261 Reserved
Seg 00056-007FF, 003D left, need 00261
Seg 00800-00FFF, 022E left, need 00261
Seg 01000-017FF, 00F7 left, need 00261
Seg 01800-01FFF, 0005 left, need 00261
Seg 02000-027FF, 0193 left, need 00261
Seg 02800-02FFF, 0073 left, need 00261
Seg 03000-037FF, 011B left, need 00261
Seg 03800-03FFF, 0175 left, need 00261
```

In total you have enough RAM for your function of size 261 but not all in the same segment. The linker does try to shift functions around and make room, but if this is not possible you will get the error. The best way to solve this is to again split some larger functions into smaller functions. If there are more smaller functions the linker will be able to shuffle things around and make more room in one of the segments for your 261 function.

To find out how much ROM each functions used in the IDE use COMPILE > STATISTICS. There needs to be an error free compile to see this however.

Code Optimization

If users are really out of ROM then they need to consider optimizing your code. The general rule is to find groups of lines that are the same or similar and move them to a function. The compiler does some optimization between statements, but most of the optimization is done on a single statement. Skimming through the .LST file to identify C lines that generate a lot of assembly can help to identify areas that users may want to check for the possibility to move to a function. Here are some examples:

1. Even though this appears to be a simple initialization you will save ROM by moving the four lines to a function:

```
next_in=0;
next_out=0;
count=0;
total=0.0;
read_from_history();
   ...
next_in=0;
next_out=0;
count=0;
total=0.0;
read_from_device();
```

2. Unless using #opt compress (see below) the compiler does not optimize the subscript expression used here:

```
weight = weight_lookup[width*height+cal_offset];
time   = time_lookup[width*height+cal_offset];
```

When possible do this:

```
volume = width*height+cal_offset;
weight = weight_lookup[volume];
time   = time_lookup[volume];
```

3. When reviewing the LST file users will see printf's can take a lot of space. For code like this:

```
printf("Max position = %lu:%lu\r\n", max_x, max_y);
printf("Ave position = %lu:%lu\r\n", ave_x, ave_y);
printf("Min position = %lu:%lu\r\n", min_x, min_y);
```

Consider:

```
void printxy( rom char * label, int16 x, int16 y) {
    printf("%s position = %lu:%lu\r\n", label, x, y);
}
   ...
printxy("Max", max_x, max_y);
printxy("Ave", ave_x, ave_y);
printxy("Min", min_x, min_y);
```

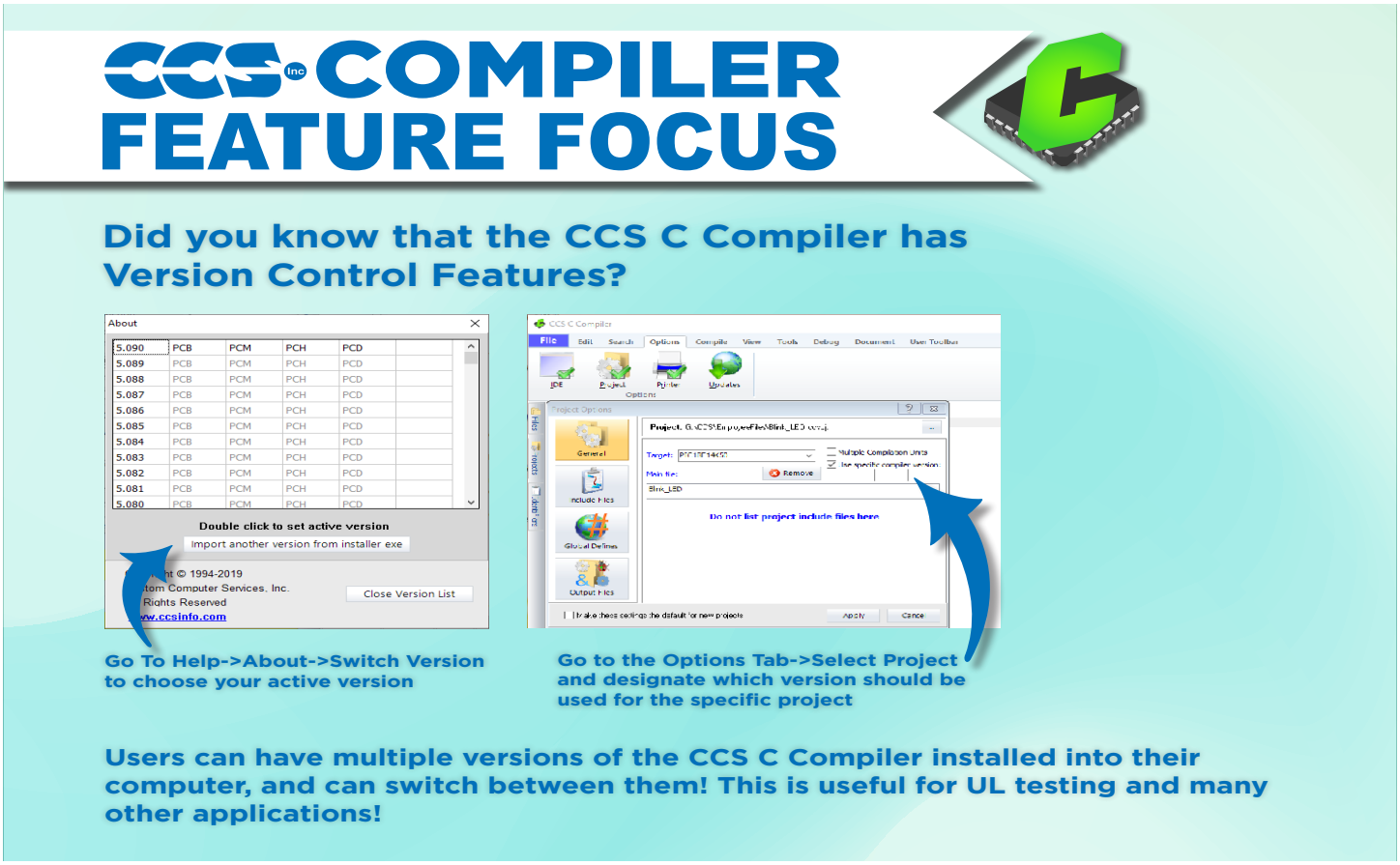
Compiler Optimization

By default the compiler uses optimization level 9. This provides a good level of optimization that has been well tested. If the optimization is set down then now is a good time to get it back up to 9. If there are any optimization problems do report then to support so we can fix them.

Some chips have a more extreme level of optimization available. To invoke it use:

```
#opt  compress
```

This optimization will save ROM but will take longer to run. What the compiler does is to go through the whole program and any sequence of instructions that is repeated are made into a function that is called each time it is needed. This will make debugging and even reading the .LST file more difficult. The savings can be substantial and may prevent users from moving to a larger chip.



Use the IDE Column Editing Feature to Quickly Edit Repetitive Text

By: Mohammed Alzakariya

There are situations where 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.

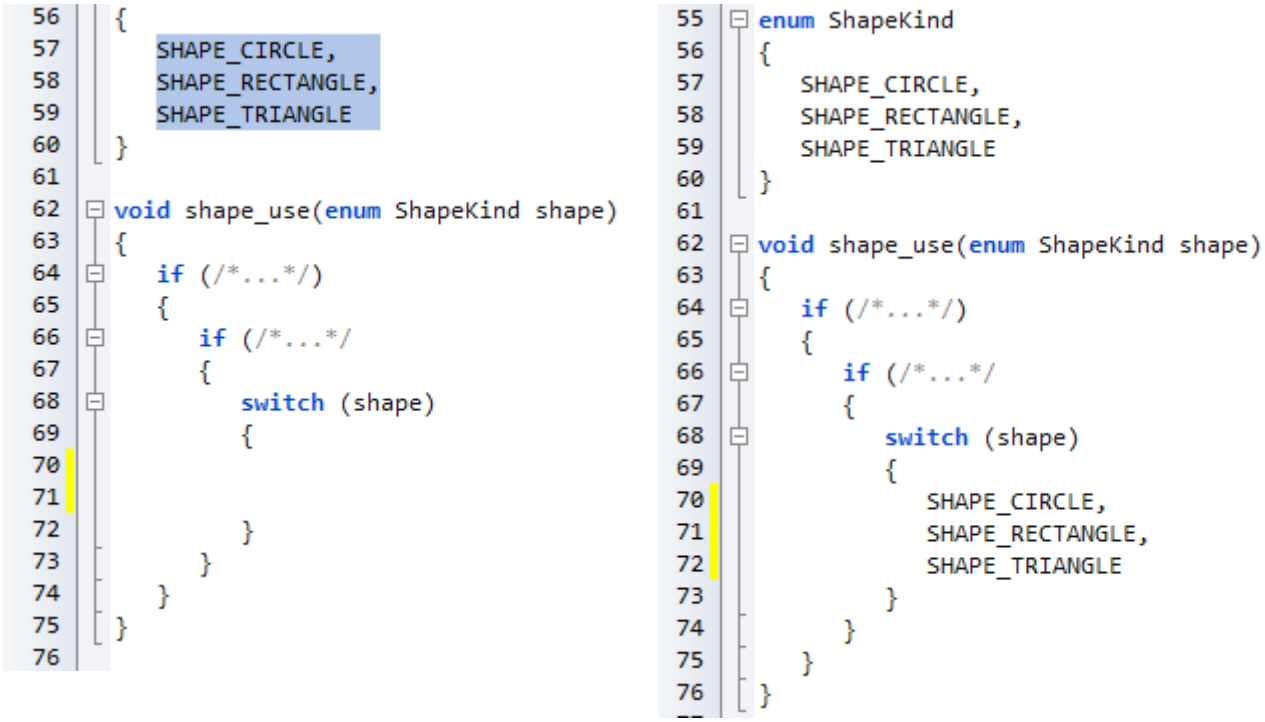
Access this feature by pressing the CTRL key while making a selection with the left mouse button. This enables selecting a rectangular region to be able to type to replace its contents, paste over it, or delete it.

The following are some examples.

1. Editing identical variable types

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



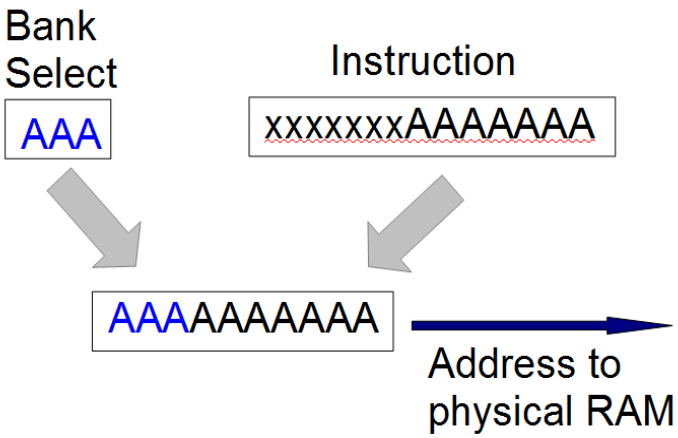
Above shows a selected rectangle consisting of all of the enum variants to avoid copying the spacing. Pasting it into the switch statement maintains its tabbing. Next insert “case” into each line by simply column select the space before “SHAPE” and type “case”.

In part one we covered the two hardware address spaces in the PIC® MCU and how they can be used from C. Part two explained how a developer defines virtual address spaces. In this article we will detail how some PIC® MCUs have their own alternative address spaces and how the C compiler deals with them.

First, we must have a basic understanding about the fundamental addressing at the hardware level. Consider a PIC16F887 part and the instruction to add a memory value to the working register and put the result in the working register. The 14 bit instruction looks like this:

0 0 0 1 1 1 0 a a a a a a a

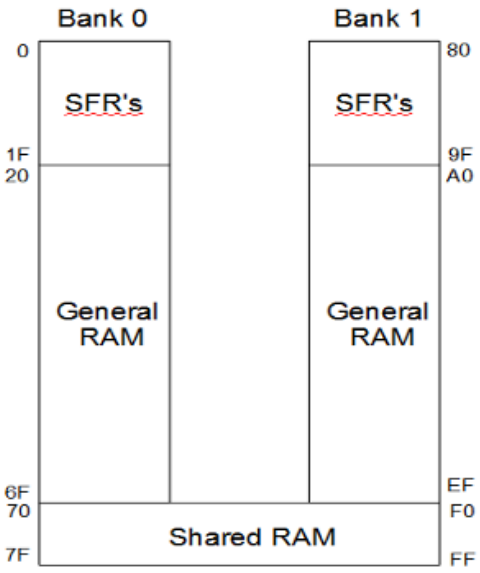
The aaaaaaa is the address in memory to get the data from. Notice there are only 7 bits so the address range is 0-127. In order to gain access to the rest of memory, there are some bank select bits. If the bank select bits are set to 010 then the instruction with aaaaaaa set to 0000001 will actually access memory location 0100000001. Fortunately, the compiler (at least the CCS C compiler) takes care of setting the right bank select bits before any instruction that needs the bits changed. There is a similar method used for program memory where the bits are called the page select bits. That comes into play with goto's and calls.



The following topics cover special features in some PIC® MCU parts that allow for a more effective way to access memory.

Mid-range Shared RAM

Even the old PIC chips had some special function registers duplicated in every bank, so there is no need to switch banks to access a popular register, like the status register. Many of the newer PIC® MCUs also have some general purpose RAM duplicated in all the banks. For example, on the PIC16F887 locations 0x70-0x7F reference the same RAM locations regardless of the bank bit setting. That means 0x70 and 0x1F0 are the same location. It is for this reason the compiler scratch locations begin at 0x70.



Mid-range Linear Address Space

On the mid-range parts the special function registers typically are at the start of each bank. For example 0x00 to 0x1F are SFR and 0x20-0x7F is general RAM. Then 0x80 to 0x1F are SFR and 0xA0 to 0xFF has general RAM. This means the largest data structure a user can have in RAM is 96 bytes or 80 bytes if there is shared RAM. If the user needs a larger array for example, they would need to split it into two smaller arrays.

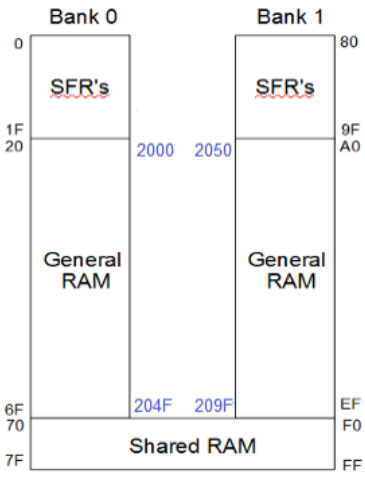
On some mid-range PIC® MCUs there is an extended instruction set (4 and 5 digit parts after the F). In these parts there is a second way to access RAM called linear addressing. The linear address space has all the general purpose RAM locations and none of the SFR's. The entire general purpose RAM is put at location 0x2000 and up. Location 0x2000 is the same as 0x20. Location 0x02050 is the same as 0xA0. Now users can have data structures as large as memory can hold. The only way to address using the linear method is using indirection via the FSR registers. In the CCS C Compiler anytime a user accesses a data item using a variable index the linear addressing method is used if available on the part.

PIC18 Access Space

On the PIC18 parts most instructions have what is called an access bit to select what address space is used in the instruction. In the normal mode, the addressing works like the PIC16 using bank select bits somewhere else to fully specify the address. However, if the access bit is 0 then the address in the instruction is used to access only certain registers. The add instruction on the PIC18F4520 looks like this:

0 0 1 0 0 1 0 x a a a a a a a a

When x (the access bit) is 1 the address in the instruction is combined with the bank select bits to form the physical address.



When x is 0 then the physical address depends on the value of the address. With lower addresses then the remaining address bits are used as-is to get the physical address without using the bank select bits. For example, 00000001 will give users physical address 1 regardless of the bank select bit setting. This is kind of like the shared RAM feature in the PIC16. When the address is higher (0x80 and up on the PIC18F4520) then 0xF80 is added to the address over the cutoff to get the physical address. This allows for access to special function registers regardless of the bank select bit setting. Be aware however that different PIC18 parts have a different split between general purpose RAM and the SFR's in the access space. The PIC18F4520 has 128 bytes in each however the PIC18F4550 has only 96 bytes in general RAM and the rest in SFR's.

It should also be pointed out the PIC18 has a special double word instruction that can copy from any RAM address to any other RAM address.

PIC18 XINST

Most PIC18 parts have a XINST bit in the configuration fuses. When this bit is set the way the general purpose RAM is accessed when the access bit is set is modified. Instead of accessing the RAM directly the address is first added to the contents of another register. The intent with this feature is to make the using local variables easier in C code where users could have variables on a stack. This would allow for recursive functions. Since this feature is either on or off for the whole CCS has decided not to add support in the compiler for XINST because of the rare need for recursion and the inefficiencies caused by using this mode.

PIC24/DSPIC Extended RAM

On the 24 bit parts, the bank select bits were removed. Looking at the same add instruction it looks like this:

1 0 1 1 0 1 0 0 0 0 0 a a a a a a a a a a a a a a a

That means the instruction has direct access to RAM locations 0-1FFF (8K). That is a lot but many of these parts have more RAM than that. There are two ways to access RAM above 1FFF. First is to indirectly access the RAM using a 16 bit pointer (64K). This does require an extra instruction or two but is not as bad as it sounds since many instructions allow indirect addressing using one of the working registers. The other method is to use a special instruction to copy to/from any address 0-FFFF (64K) and a working register.

In summary, we have a great way to access 8K of RAM and there is a method to get at RAM over 8K but the method needs to be repeated for each access.

PIC24/DSPIC PSV Mode

Access to program memory (for constants) requires some extra steps to set up some registers and grab the data. The 24 bit parts have a mode that can be used to map the upper 32K of RAM addresses on to a selected area of program memory. This can be used to access constants in program memory as easily as if they were in RAM. When doing this, users loose access to the upper 32K of RAM. By default the CCS Compiler does not use this technique because in general it seems additional RAM is more important than fast access to ROM. A lot of the Microchip example code does use PSV mode so there is support in the CCS compiler to allow for easier porting.

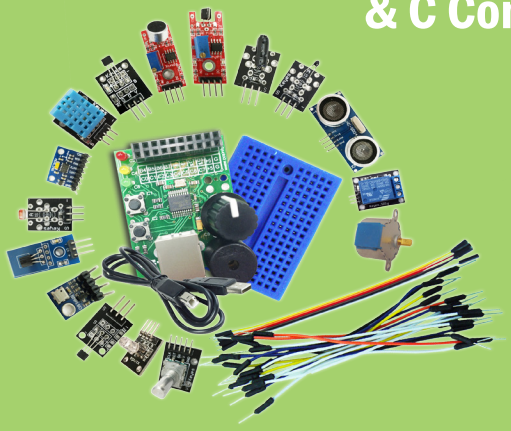
```
#device PSV=16    // Turns on PSV mode

const char serial_number[] = {"123456789"};  // Saved in ROM

...

sn = atol32(serial_number);  // Is accessed by a RAM address
```

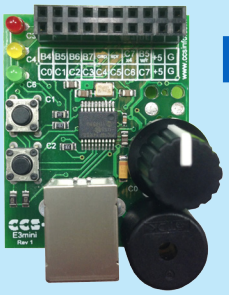
Sensors Explorer Kit

$69

For the Complete Kit!

INCLUDES: E3mini Prototyping Board & C Compiler!





E3mini

Enjoy 16 Different Sensors!

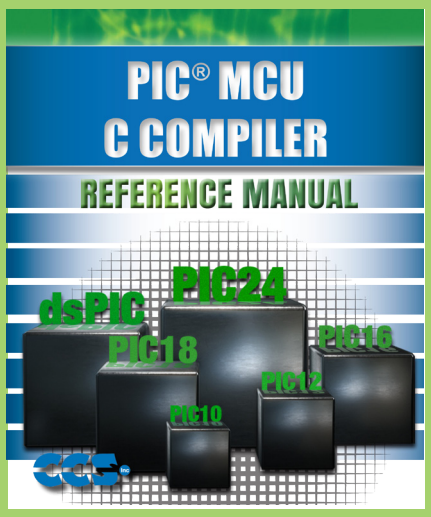
Learn Embedded C with hands-on tools that inspire real world applications!

C Workshop Compiler for PIC® MCU

$99

- **Easily migrate between all Microchip PIC® MCUs devices**
- **Supports any 13 devices**
- **Minimize Development Time with Peripheral Drivers and Standard C Constructs**

FREE 45-Day Demo!



PIC® MCU is a registered trademark of Microchip Technology Inc.

LOAD-n-GO

Handheld Portable Programming for PIC® MCUs

- 2 MB Internal Flash for Storing up to Four Separate Programs
- FREE CCSLOAD Software Allows for Ease of Loading Firmware and Programming
- Power the Target Board at 2.5V, 3.3V or 5.0V with Simple Jumper Setting

Spring into Coding Projects with a C Compiler

$25 Off a Full Compiler or Compiler Maintenance

Use Code: Spring2021

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

CCS Inc

Follow Us!