July 2020

<Bits & Bytes Newsletter/>

Email:
support@ccsinfo.com

sales@ccsinfo.com
Ph: 262-522-6500

www.ccsinfo.com

New Compiler Versions
>Downloads>Compiler Software

Easy License Renewal
>Support>Renewals

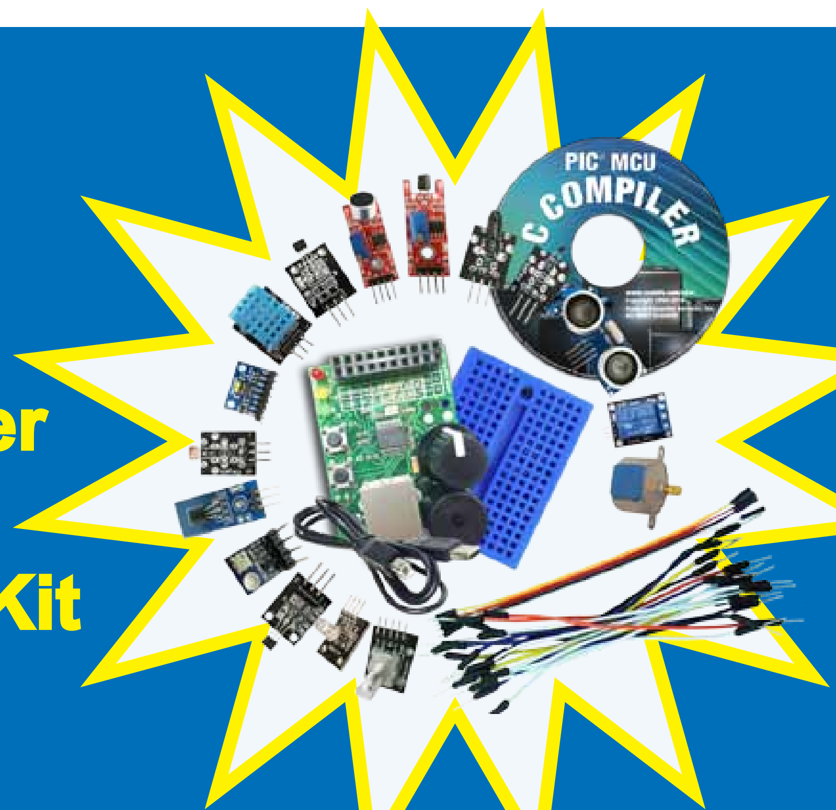
TECH NOTE: Unrolling Loops with a Duff's Device

TECH NOTE: Compiler Feature Focus

TECH NOTE: Built-in Local Interconnect Network (LIN) Bus Support

Product Spotlight

C Workshop Compiler & E3 Sensors Explorer Kit



Unrolling Loops with a Duff's Device

Do you remember when you learned programming languages and your teacher or instruction materials told you to never use a GOTO because it could lead to code that was difficult to understand? Here is something that you might not have been warned about - Duff's device. Duff's device was created by Tom Duff, and its design was to unroll and speed up loops by removing conditional statements from the loop. By doing this, the execution time of the loop is decreased. A Duff's device is basically a switch-case statement, with the break operations removed so the code can continue by falling to the next case. Let us look at how it works and look at some real measurements of speed increases on a PIC18F MCU.

Before jumping into a Duff's device, it would be useful to review the switch-case statement in C:

```
switch(state)
{
   case 0:
      doState0();
      break;
   case 1:
      doState1();
      break;
   case 2:
      doState2();
   case 3:
      doState3();
      break;
}
```

`switch()` reviews the value passed to it, the variable 'state' in the above example, and jumps to matching case statement that matches this variable. For most cases, the C compiler will create a jump table at the `switch()` to jump to the variable that matches. That means for a `switch()` statement, most of all the branching logic happens at the `switch()` statement. The break statement tells the C compiler to jump out of the current `switch()`.

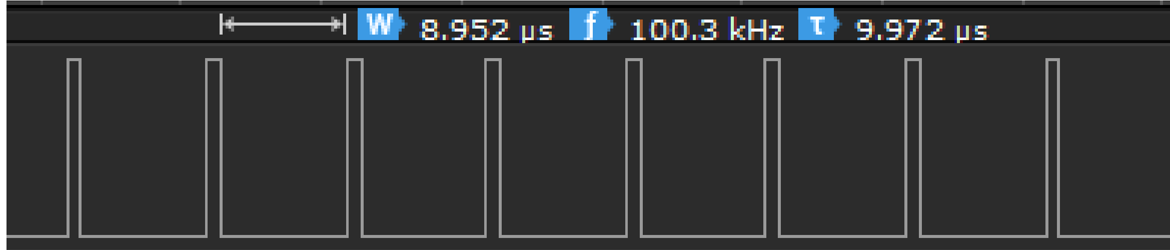
Notice anything odd about the case 2 in the above example? It does not have a break at the end! Without this break, you are telling the compiler that you want to continue execution. That means in the above example, when the case 2 has finished it will keep rolling into the case 3. Many compilers (including CCS C) and lint tools will generate a warning that you forgot a break statement, because for many control loops the developer only intended one block of code to execute for each case. But in a Duff's device we are going to leverage the ability to continue execution to the next case.

Let us look at a simple loop, maybe being used to send pulses or clock data on the IO:

```
#define PULSE() output_toggle(PIN_PULSE);   \
                output_toggle(PIN_PULSE)

void send(int n)
{
   while(n--)
   {
      PULSE();
   }
}
```

This is a simple function that generates n pulses, using a while loop to decrement n and exit when n is 0. That means for every pulse, the value of n has to be decremented and checked against 0. Look at the timing of pulses generated on a PIC18F MCU running at 4MHz (one instruction every 1us):



This generated a 100KHz signal. You will notice the discrepancy of the duty cycle; the high time is 1us but the low time is 9us. That's because it took 8us to execute the while(n--) portion of the loop.

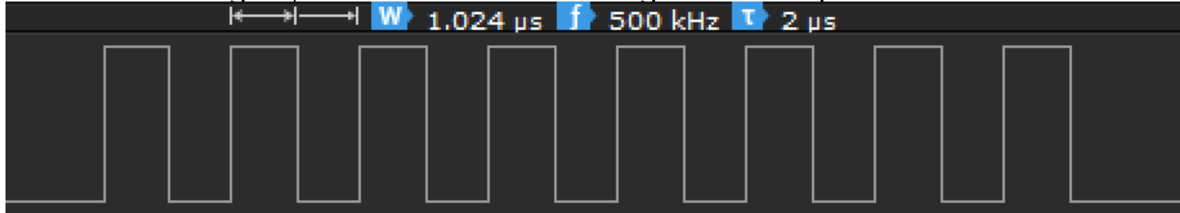
Now let us replace it with a Duff's device:

```
#define PULSE() output_toggle(PIN_PULSE);   \
                output_toggle(PIN_PULSE)

void send(int n)
{
   switch(n)
   {
      case 8:
         PULSE();
      case 7:
         PULSE();
      case 6:
         PULSE();
      case 5:
         PULSE();
      case 4:
         PULSE();
      case 3:
         PULSE();
      case 2:
         PULSE();
      case 1:
         PULSE();
   }
}
```

In the above example, the switch(n) creates a jump table of up to 8 pulses and jumps to position with n pulses. After the jump is calculated and executed, the following pulses run without any other conditional code. Lets look at the timing of pulses generated on a PIC18F running at 4MHz (one instruction every 1us):

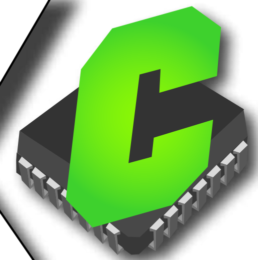
This creates a 500KHz signal, 5 times faster than using a while loop. Also of interest is the duty cycle of the signal, which is now 50% (even time high and low) because the pulses execute without any other conditional checks. If you look at the LST file to view the instructions generated, the reason for this is clear (BTG instruction is used to perform a bit toggle, in this case toggling the register that controls this GPIO pin):




```
....................        case 8:
....................           PULSE();
00016:  BTG    3FBD.0
00018:  BTG    3FBD.0
....................        case 7:
....................           PULSE();
0001A:  BTG    3FBD.0
0001C:  BTG    3FBD.0
....................        case 6:
....................           PULSE();
0001E:  BTG    3FBD.0
00020:  BTG    3FBD.0
```

The next time you have a loop and you were looking to optimize it for speed, the Duff's device is a great method for accomplishing this!

CCS COMPILER FEATURE FOCUS



#opt compress

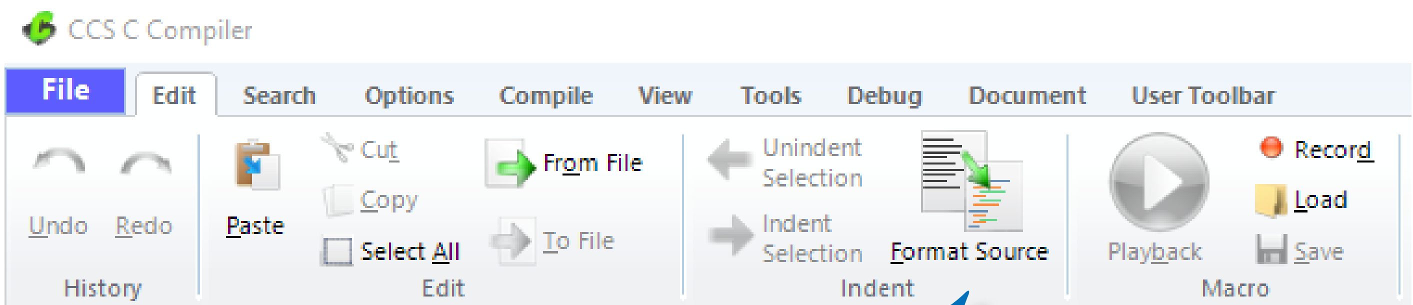
Add this line of code to your program to utilize the aggressive code optimizer in the CCS C Compiler! This optimizer works for space instead of speed. This tool searches the entire compiled program to find repeating blocks of code and reduce them into one shared sub-routine.

Optimization Level Efficiency	Compiler Versions		
	V4.141	V5.006	
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%

This new optimization level is supported on Enhanced PIC16 and PIC18 microcontrollers.

The average size reduction of program memory is approximately 15%. In some cases, we have seen program memory reduced by 60%. The chart above features examples of compression levels:

Format Source Button



Forgot to indent your code?
Use the Format Source Button!

Our Automatic Source code formatting is a great tool in the CCS C Compiler. The tool is intelligent enough to look over your code and indent it in an easy-to-read format!

Built-in Local Interconnect Network (LIN) Bus Support

The CCS C Compiler has added built-in Local Interconnect Network (LIN) bus support to the #use rs232() library. LIN bus is an inexpensive serial communication protocol used primarily in the automotive industry to complement the existing CAN Bus network. The LIN bus network contains one master node and up to 15 slave nodes for a total of 16 nodes, and supports bit rates up to 20 kbit/s.

The LIN bus protocol's message frame consist of two parts, the header and the response. The header is always sent by the master node, meaning all communication is initiated by the master node. After the header is sent only one node sends the response. The header consists of three main fields, the break field, the sync field and the identifier field. The break field is used to get the attention of all LIN slave nodes on the network. The sync field is the hexadecimal value 0x55 used by the slave nodes to determine the current bit time of the bus. Finally the identifier field is used to determine which node will respond during the response part of the frame. The response consist of two fields, the data field and the checksum field. The data field contains 0 to 8 bytes and the checksum field is one byte. Depending on the LIN bus protocol specification being used the checksum is either the checksum of the data field bytes, or the identifier field and the data field bytes.

The following options have been added to the CCS C Compiler's #use rs232() library to enable LIN bus master or slave protocol support, LIN=MASTER and LIN=SLAVE. Additionally the options LIN_CHECKSUM=LEGACY or LIN_CHECKSUM=ENHANCED can be used to select which checksum type is used by default by the functions. Legacy checksum only uses the data field when calculating the checksum, and enhanced uses both the identifier field and the data field when calculating the checksum.

Whether a device can use the library's built-in LIN bus protocol support depends on the mode being used and configuration. When setup as a LIN bus master both software and hardware configurations are support, the only limitation is that when using the hardware UART peripheral the device is required to have an advanced UART peripheral or an UART peripheral with built-in protocol support. When setup as a LIN bus slave it's only supported when using the hardware UART peripheral on devices that have an advanced UART peripheral or an UART peripheral with built-in protocol support. For devices that have a hardware UART peripheral, most devices have a UART peripheral that will work with the #use rs232() library's built-in LIN bus protocol.

When built for a LIN bus master the following functions are added by the #use rs232() library: linbus_header(), linbus_rx_response(), linbus_tx_response() and linbus_checksum_type(). The linbus_header() function is used to by the master to send the header part of the LIN bus message frame, the parity bits of the identifier field is automatically calculated by the function. The linbus_rx_response() function is used by the master to receive the response from one of the slave nodes during the response part of the LIN bus message frame. The linbus_tx_response() function is used by the master to transmit the response during the response part of the LIN bus message frame, the master node only sends the response when the identifier it sent indicates that it should transmit the response. Finally the linbus_checksum_type() function can be used to change how the linbus_tx_response() function calculates the checksum that it sends when used to send the response part of the LIN bus message frame.

When built for a LIN bus slave the following functions are added by the #use rs232() library: linbus_header_hit(), linbus_header_get(), linbus_rx_response(), linbus_tx_response() and linbus_checksum_type(). The linbus_header_hit() function is used to determine if the header as been received. For devices that have a UART with build-in protocol support this function returns TRUE after the entire header as been received, and for devices with an advanced UART it returns TRUE after the first 8 bits of the

break byte is received. The linbus_rx_header() function is used to retrieve the identifier field of the received header, the parity bits masked off. Additionally for devices with an advanced UART peripheral this functions also sets up the UART peripheral to receive the sync field. The linbus_rx_response() function is used by the slave to receive the response from another node during the response part of the LIN bus message frame. For devices with a UART peripheral with built-in protocol support this function only needs to be called if the received identifier indicates that the message is to be received by that node. For devices with an advanced UART this function should be called for all messages were the received identifier doesn't indicates that the node should transmit the response. The linbus_tx_reponse() function is used by the slave to transmit the response during the response part of the LIN bus message frame. This function should only be called when the received identifier indicates that the node should transmit the response. Finally the linbus_checksum_type() function can be used to change how the linbus_tx_response() function calculates the checksum that it sends when used to send the response part of the LIN bus message frame.

Product Spotlight
C Workshop Compiler and E3 Sensors Kit

Sensors Explorer Kit
Sku: S-205
$69

Sensors
Human Touch, Temperature (2), Light, Barometric Pressure, Humidity
Accelerometer, Magnetic Field (2), Ultrasonic Range, Vibration, Sound, Rotary Encoder

Included Output Devices
Full Color LED, Generic Relay, Stepper Motor

Also Available
GPS Unit, 7-Seg LED, Keypad

C Workshop Compiler
Limited to 13 Devices with the IDE
$99
Sku: 52204-1534

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

16-bit: PIC24F16KM102, PIC24FJ128GA006, dsPIC30F3010, dsPIC33EP128MC202

***Additional chips may be purchased separately**

WORKING FROM HOME?

CCS

TACKLE WORK AND HOBBY PROJECTS WITH A POWERFUL C COMPILER AT A DISCOUNT!

SPECIAL OFFERS

$25 off
ANY FULL COMPILER OR COMPILER MAINTENANCE
Use Code: Home25

FREE GROUND SHIPPING
(TO U.S. 48 ONLY)
ON ALL PROGRAMMERS OR DEVELOPMENT KITS
Use Code: HomeFS

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.

CCS wants to help further embedded development by customers, and is offering a discount on any new compilers or maintenance plan purchases. The customers that need development boards, and programmers, we are offering Free Ground shipping (to the U.S.48) so you can get the tools you need to continue working from home.

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.

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