

Making Waves

By John Dobson

Introduction

In last month's article on making a sound waveform we used the E-blocks SPI bus D/A and memory board to generate a sound waveform of around 400Hz. In using Flowcode I read that the flowchart is first converted into C and then into assembly code, and that you can embed C into your Flowcode programs to speed the operation up. In this article I describe my attempts use C to speed up the sound waveform generator. At the same time I discovered how you can use Flowcode to learn C programming.

Background theory

To generate code for the PICmicro microcontroller Flowcode processes a flow chart in a number of steps. Lets have a look at how this works:

Step 1

Flowcode initially takes your flow chart and generates C from it. As an example of this lets look at a simple counter program (COUNTER1.FCF) in Flowcode and its C equivalent.

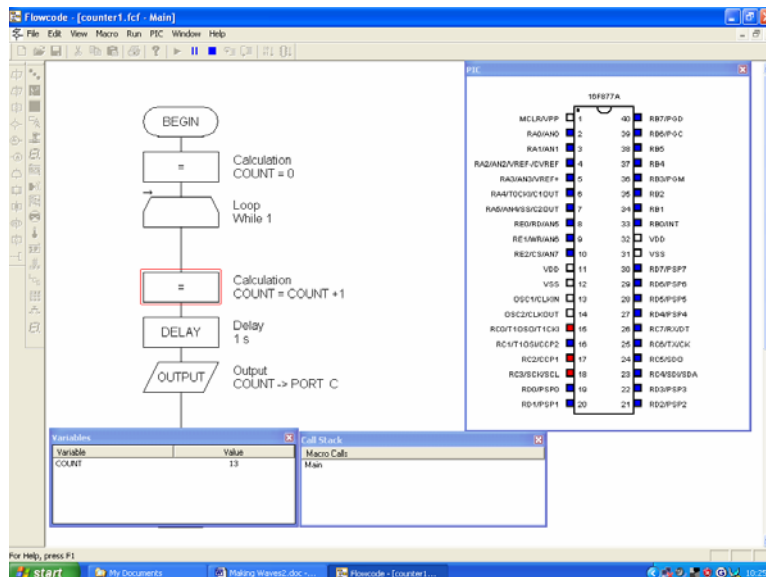


Figure 1 – simple Flowcode counter

In Figure 1 you can see the counter in Flowcode: we declare the value of variable COUNT as 0, then we have a loop icon. The Loop WHILE 1 declares an endless loop as '1' is always true. Inside the loop we have icons to increase the value of COUNT by 1, wait for one second and then output the value of COUNT to port C. I have a 16F877 PICmicro with LEDs counting up in a binary sequence on port C.

Step 2

If you open Windows Explorer and look in the directory where the COUNTER1.FCF Flowcode file is saved you will find that Flowcode has generated a number of files one of

which is COUNTER1.C. This is the C equivalent of the COUNTER1 flow chart program. If you open this file in Notepad you will see the following program:

```

counter1.c - Notepad
File Edit Format View Help
//Defines for microcontroller
char PORTC@0x07;
char TRISC@0x87;
char PORTD@0x08;
char TRISD@0x88;
char PORTE@0x09;
char TRISE@0x89;

//PIC Functions
#pragma CLOCK_FREQ 19660800
#define P16F877A
#include <system.h>
#define MX_EE
#define MX_EE_TYPE2
const char MX_EE_SIZE = 256;
#define MX_SPI
#define MX_SPI_C
#define MX_SPI_SDI 4
#define MX_SPI_SDO 5
#define MX_SPI_SCK 3
#define MX_UART
#define MX_UART_C
#define MX_UART_TX 6
#define MX_UART_RX 7
#define MX_I2C
#define MX_MI2C
#define MX_I2C_C
#define MX_I2C_SDA 4
#define MX_I2C_SCL 3
#define MX_PWM
#define MX_PWM_PORT portc
#define MX_PWM_TRIS trisc
#define MX_PWM_CNT 2
#define MX_PWM_0 2
#define MX_PWM_1 1

//Macro function declarations

counter1.c - Notepad
File Edit Format View Help
#define MX_PWM
#define MX_PWM_PORT portc
#define MX_PWM_TRIS trisc
#define MX_PWM_CNT 2
#define MX_PWM_0 2
#define MX_PWM_1 1

//Macro function declarations

//variable declarations
char FCV_COUNT;

//Macro implementations

void main()
{
//PIC Initialisation
adcon1 = 0x07;

//Interrupt initialisation code
option_reg = 0xC0;

    FCV_COUNT = 0 ;
    while( 1 )
    {
        FCV_COUNT = FCV_COUNT + 1 ;
        delay_s(1);

        TRISC = 0x00;
        PORTC = FCV_COUNT;

    }
mainendloop: goto mainendloop;
}

```

Figure 2 – C equivalent of COUNTER1.FCF.

Here we have shown the program in two parts, side by side – to save space on this page. Firstly Flowcode has defined some of the constants the C compiler needs with statements like ‘char PORTC@0x07;’. This defines the C variable ‘PORTC’ as being hex (that’s what the ‘0x’ stands for) address 07. Similarly TRISC is defined as being hex 87. TRISC is the data direction register on the PICMicro and the TRISC register on the 16F877 I am using is at hex 87. : Similarly many of the other PICmicro and circuit specific functions – such as the clock speed, the pins for the internal USART etc. are defined. (Note that in C when a line begins ‘//’ this indicates that the line is a comment and not a line of code.) There are no macros or subroutines in our program so these sections are blank, but there is a variable defined as type CHAR called FCV_COUNT. This is our COUNT variable from the Flowcode program. This is the first hint in how you can use the C icon in Flowcode to embed code into a Flowcode program: all variables in Flowcode are prefixed by ‘FCV_’ when transferred to the C compiler. This means that when referring to a Flowcode variable in a C icon you must also prefix the variable with ‘FCV_’. (Incidentally FCV stands for FlowCode Variable.)

After the declaration of variables you see the first line of C:

```
Void main()  
{
```

This is a function declaration within C to indicate that this is our main program. This is the equivalent to the START icon in Flowcode. The pair of braces indicate that there are no variables passed to this function, and the open curly bracket '{' indicates the start of the main function, and you can see a close curly bracket '}' at the end of the program. After this we have two more declarations – for the A/D converter – to declare these pins as analogue inputs – and to turn the timer interrupt on. These are declared inside the main program loop as they can be altered by the user within the Flowcode program.

After this we have the main program: FCV_COUNT is declared as being 0, corresponding to our first flow chart icon, then there is a `while (1)` statement followed by further code inside curly braces: this means execute the routine inside the curly braces forever. Inside the braces we have the FCV_COUNT increment, and a delay of 1 second. Note that these lines of C code end with a semicolon. Lines of C code are always terminated with a ';'. Then we have the line `TRISC=0x00`. We saw earlier that TRISC is the data direction register. This line of code writes value 0 to TRISC which declares all of the port C pins as outputs. A value of hex FF would declare all the pins as inputs. Our last line of C code in the loop writes the value of our COUNT variable to port C.

The last line here is a safety net: in the case where we do not have an endless loop in our Flowcode program, then the C program will get stuck at this point and execute this line forever. This is the equivalent to the END icon in Flowcode. If you have a C program without an END loop like this then you get a curious effect: the program counter in the PICmicro device keeps incrementing until it rolls over back to address 0000, at which point your program will start to run again.

Well that was not as hard as I had expected: C is a mysterious language but it seems that if we start from a flowchart then, with a little explanation, the C becomes readable. All the declarations are difficult to understand but Flowcode seems to take care of them, so I won't worry too much what they all mean.

Back to the plot

So back to our original objectives of increasing the speed of the waveform generator program I wrote. Well I compiled my program 'SINE WAVE GEN.FCF' from last month and looked at the C produced. What I noticed was that every time Flowcode used a DAC_SEND_CHAR icon it went through a lot of lines of C to declare further variables and address references that the C compiler needs. The reason Flowcode does this is that it has to assume that you don't know what you are doing (thank goodness!) and it has to set all the parameters of Port C for serial communication, each time you use the SPI DAC macro icon, just in case you are using port C I/O pins for other functions. Well maybe we can take out some of these lines of C to speed things up? I decided to tackle this in two stages: firstly replace a DAC_SEND_CHAR icon with the C equivalent to make sure I

had replicated the function of the program, and secondly to then start altering the C code to see if I could make it more efficient.

Going back to the main program I simply replaced one of the DAC_SEND_CHAR icons with a C icon. Then I looked up the equivalent C of the DAC_SEND_CHAR icon and pasted it into the new C icon. Then I adjusted the variable declarations to make sure the C icon picked up my OUTVAL Flowcode variable, and recompiled the program to make sure that it still worked. Eventually it did.

Having got to first base I then commented out all the unnecessary lines of C – of the original 34 lines of C code, which a DAC_SEND_CHAR icon produces only 9 were actually needed! There was a certain amount of trial and error here – I just kept commenting out lines until the program stopped working! The program produced is called SINC3.fcf and you can download it from the Elektor web site.

Unfortunately however I was unsuccessful in making the DAC go any faster – the speed limitation turned out to be the speed of the SPI but itself: let me explain:

Within the code of the DAC_SEND_CHAR Flowcode command the 8 bit SPI data is sent in two bytes. The last four bits of the first byte contain the most significant data nibble, and the 4 most significant bits of the second byte contain the least significant nibble. Other bits in each SPI data byte are reserved for chip configuration etc. The DAC_SEND_CHAR Flowcode icon takes OUTVAL and processes it like this:

```
    dac_val = (FCV_OUTVAL & 0xF0) >> 4;
    sspbuf = dac_val;
    delay_us(3);

    dac_val = (FCV_OUTVAL & 0x0F) << 4;
    sspbuf = dac_val;
    delay_us(3);
```

Looking at the top routine: first we take OUTVAL and AND it with hex F0 (binary 11110000) and then shift left by 4 bits (that's the '>> 4'), then we set the SSPBUF register in the chip with the result and wait 3us. Whatever gets placed in the SSPBUF register will be sent by SPI. The second routine is similar: take OUTVAL, shift left by 4 bits, set SSPBUF and wait 3us. The last statement is the key: SSPBUF is the serial buffer in the PIC device and it will take 3us for the buffer to clear as the information is sent one bit at a time. If you write to the buffer again before 3us has elapsed you will overwrite the data mid way through the serial send operation which produces some very strange results, as the SPI bus ceases to function correctly.

So it turns out that the frequency of our oscillator is determined by the speed of the SPI bus. (For those of you that want the maths: each sample takes 6us, 256 samples per waveform gives a theoretical minimum period of 1.5ms or around 650Hz).

Oh well...time to reduce the number of samples to 64.....

Programs: COUNTER1.fcf
SINC3.fcf