Boot Loaders and MPLAB8

One of the challenges you face with any microcontroller is how to get the program onto the chip.

One way to program a PIC chip is to use an external programmer, such as PICStart-Plus

PICStart Plys from www.Digikey.com

Once you compile your program with MPLAB (coming next), you place your PIC chip into the programmer and select *program.* That easy. (This is how the boot-loader on your PIC chips was programmed if you're curious.)

The problem with external programmers is you have to remove yor PIC chip from the board, program it, and put it back into your board. This is inconvenient, breaks pins, takes time, etc.

A boot-loader is a small program resident on your PIC chip which runs on reset. This program waits to see if you are trying to download a new program.

- If so, it takes data sent on the serial port and writes it to program memory.
- If not, it runs whatever program is in program ROM

The boot-loader on your PIC chip takes up the lower 0x300 words of program memory. On reset, it sends the message

```
3 2 1 0 >
```

to the serial port at 9600 baud.

- If it gets to zero before you hit the return button on the PC keyboard (ascii 13), it then executes whatever program is in memory, starting at address 0x300
- If you hit the return key, however (ascii 13), the boot loader clears program memory and waits to receive a new program on the serial port at 9600 baud.

Note that this means you need to start your programs at 0x300

MPLAB8 and Assembler Programming

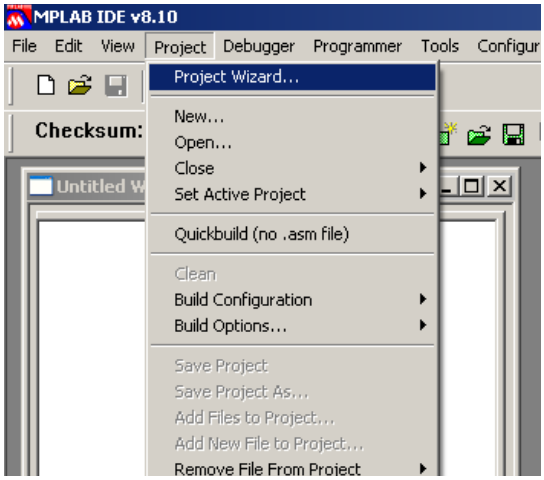
To write a program in assembler in MPLAB8

Step 1. Create a new directory. I prefer using your Z: drive with a folder Z:\ECE376\ASM\Count

Step 2. Start MPLAB8

Step 3. Click on File New Project

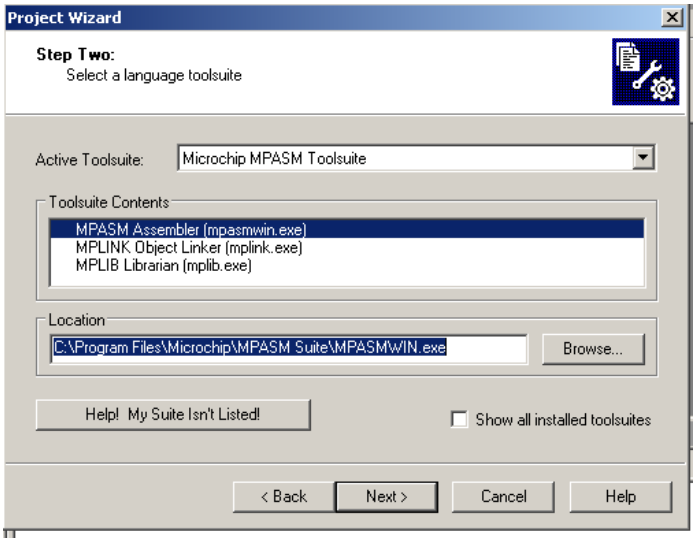
Project Wizard if this is a new project



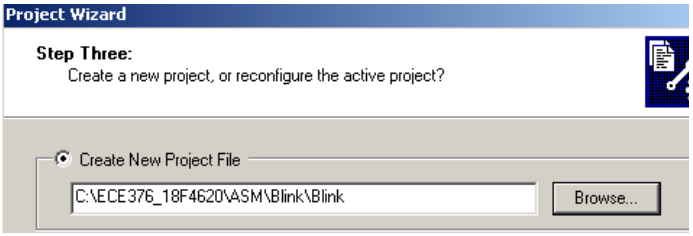
This takes you through the process of starting a new project (i.e. a new program). Click OK

Device = PIC18F4620 (next)

Program Language is MPASM

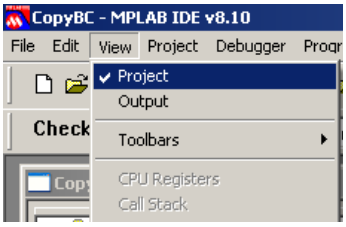


Directory for Files: Select the directory on your Z-drive (I don't have a Z-drive so I'm using my C drive.)

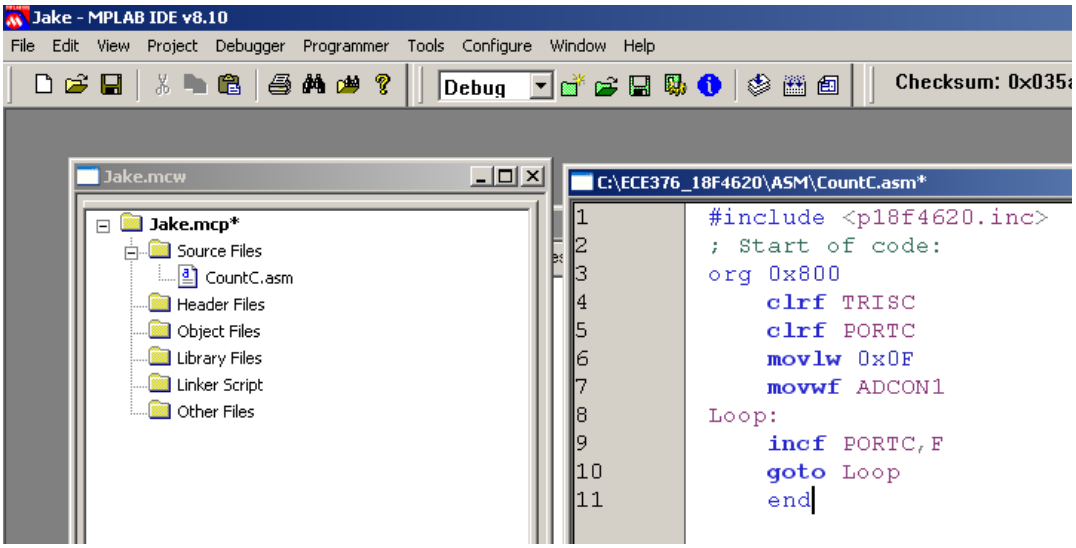


If you have an ASM file, select that file. If not, leave the file name blank and continue.

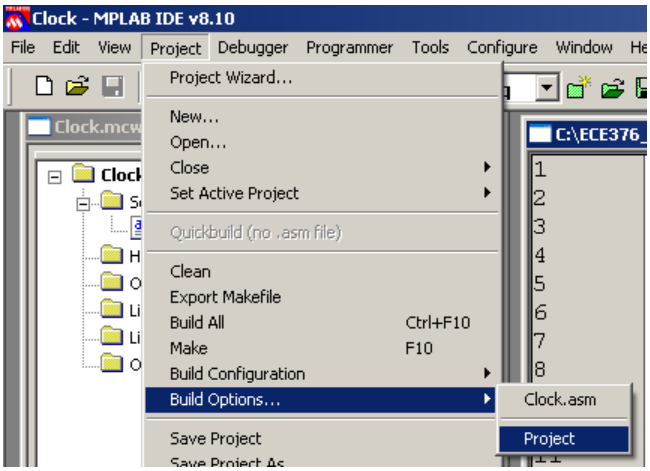
Click on View Project



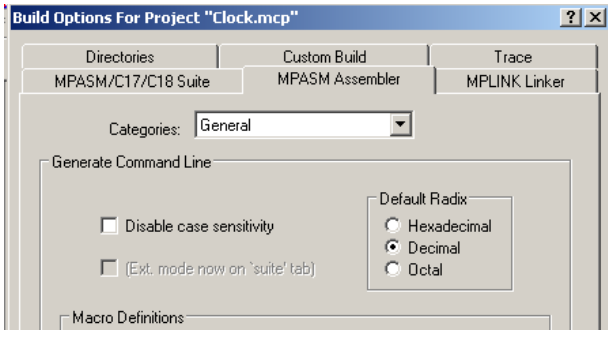
You should see the following:



Change the default to decimal. Click on Project Build Options Project



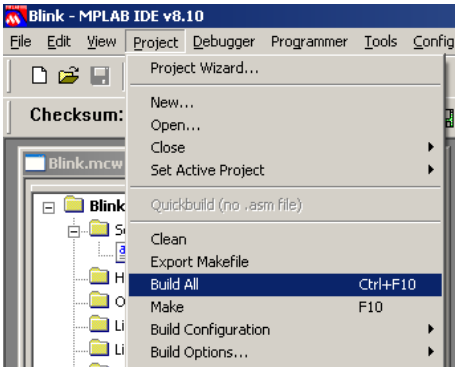
Click on MPASM anc select Decimal. This results in numbers like 100 representing 100 base 10.



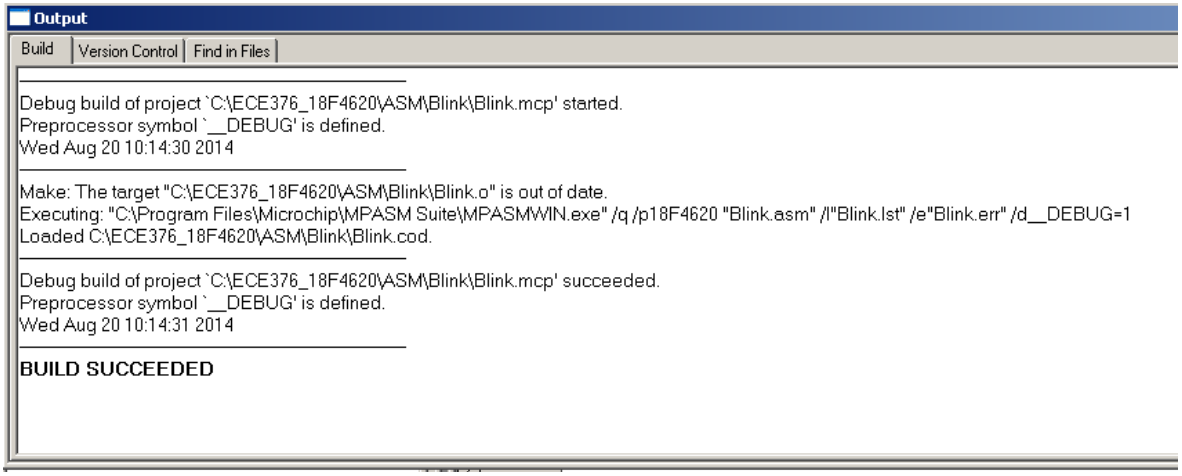
The source file is what you compile.

- If this is blank, right click on Source File and select the ASM file you wish to compile.
- If you don't have an ASM file yet, select File New edit a file, and save it as .ASM

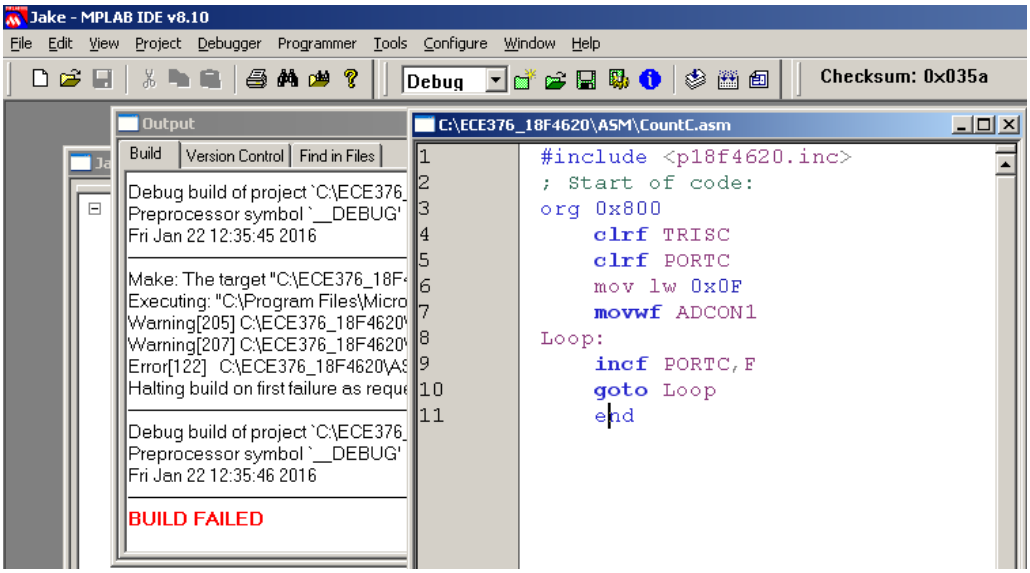
To compile your code, click on Project Bulid All (or hit key F10)



If your program compiles correctly, you get the message 'Succeed'

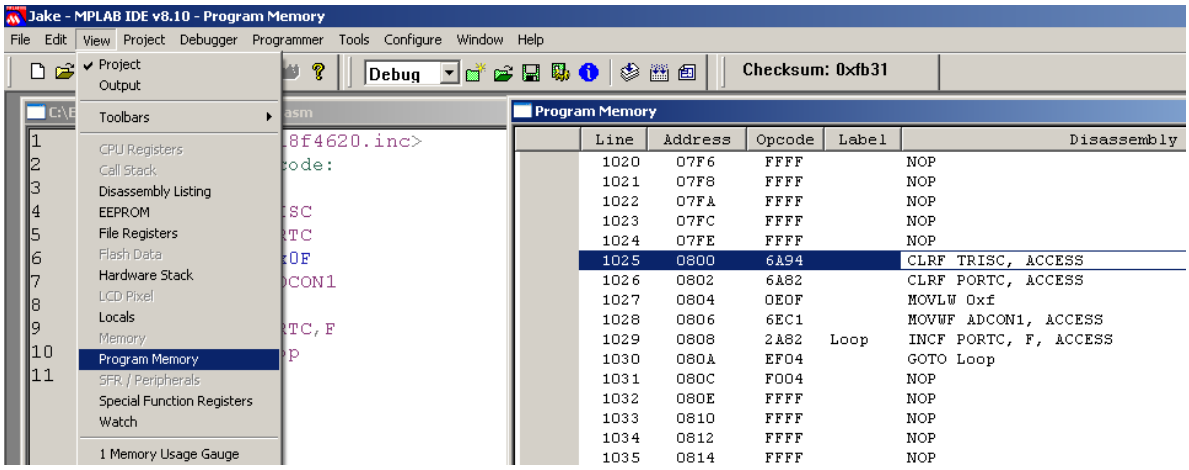


If there is an error in your code (such as a space in line 13 below), you will get an error message along with a notice which line has a problem



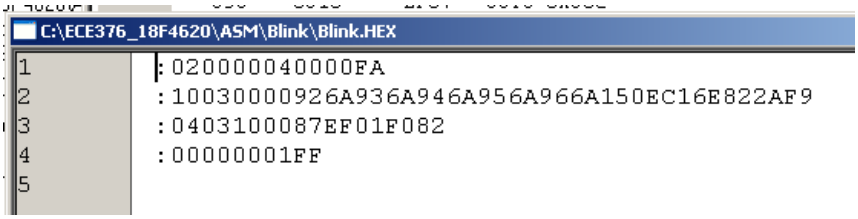
Note that compiling doesn't mean your code is correct - it only means the compiler could understand it.

If you want to see what your program looks like, click on View Program Memory



Note that your program starts at address 0x800 (due to the ORG statement. This keeps it away from the boot-loader). Also note that this is a very small program: it takes 8 lines of code (out of 32,000). A PIC can do more.

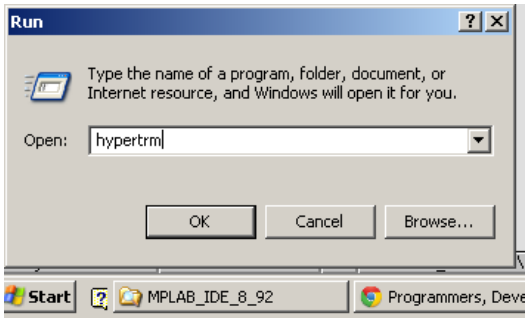
The program is stored in the file .HEX This is a text files that contains the program in machine language (the OP-Code above)



To download your code to your PIC board,

- Power up your PIC board (i.e. plug it in)
- Connect the serial cable to a PC
- Run a terminal program, such as Hyperterminal

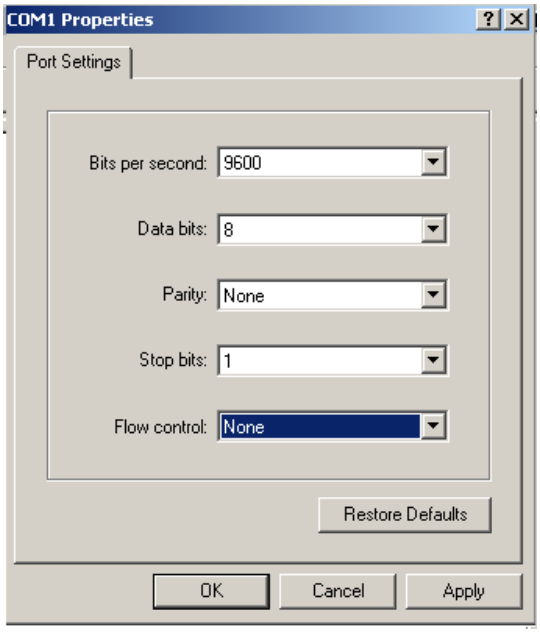
  Click on Start - Run - Hypertrm for the computers in room 235 / 237



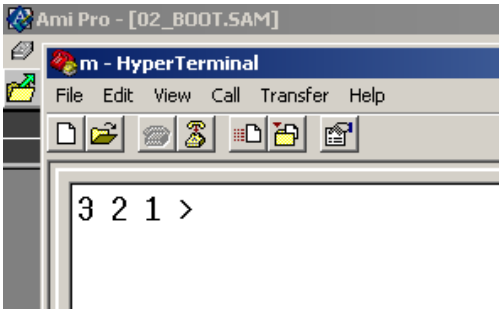
Pick any name (m)

Connect Using: COM1 usually. Sometimes different.

Connect at 9600 baud, flow control None



Hit the reset button on your board. You should see the message '3 2 1 0 >' on hyperterminal



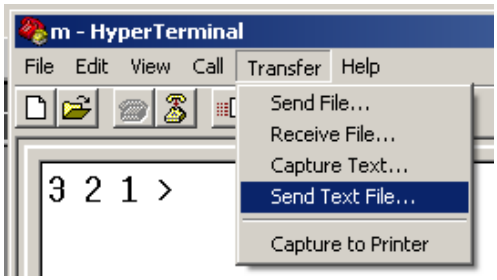
This means you are communicating with your board.

To download your program,

- Hit the reset button on your board
- Before it gets to zero, hit the return key on the keyboard

This clears out the old program and waits for a new one

- Select Transfer - Send Text File



Select the .HEX file you just compiled.

You will see the LED on RA4 blink a few times (each blink is one line in the .HEX file) then your program is running!

First Program: BLINK.ASM

To get started, let's make an LED blink. Only engineers get excited when an LED blinks. Most people think "so what?" Well, a blinking LED means

- Your compiler worked,
- You were able to download your code to your board, and
- Your processor is running

It's a big deal.

Following the previous procedure, when you download your code, you should see PORTC all lit up. Actually, each LED is blinking really fast. If you look at PORTC pin 0 (RC0) on an oscilloscope, you should see the following:

< picture of O-scope >

Note that

- RC0 is a square wave. When you count, the one's bit goes 0 / 1 / 0 / 1 (even, odd) as you count.
- The wave is low for 0.3us, high for 0.3us, etc.

This is what you should espect. Each instruction on a PIC takes one clock (0.1us). The main routine has two instructions - with one being a goto. This results in 3 clocks per loop.

To slow this down, add a wait loop

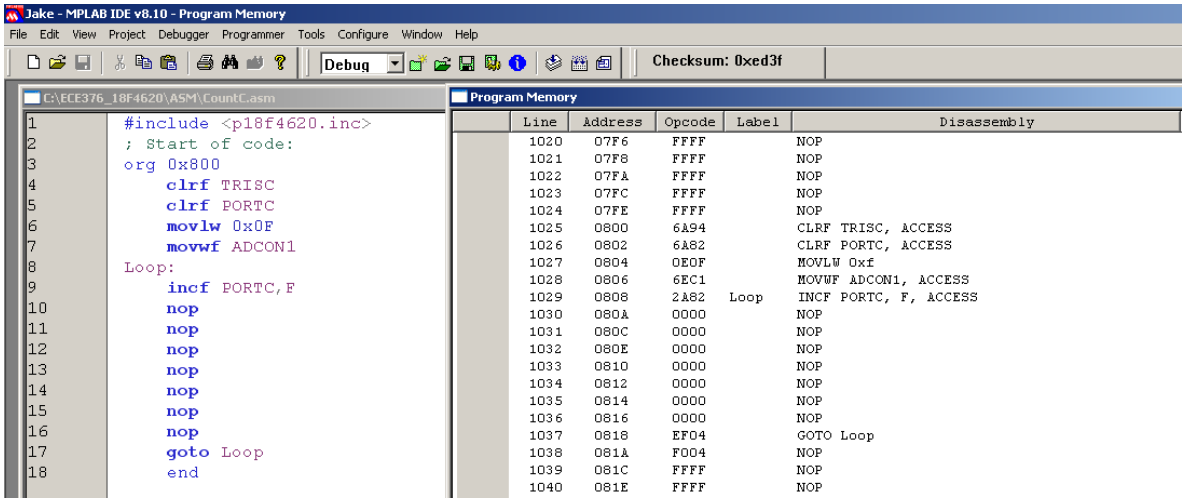
Option 1: In-Line Code with NOP statements.

A NOP statement does nothing but eat up 1 clock (0.1us). If you add 7 NOP Statements, each loop will take

- 3 clocks to increment and loop, plue

- 7 clocks for the NOP statements

making 10 clocks per loop (1us high, 1us low, 2us total or 500kHz)



If you want to play note C4 (261.64Hz),

- The period should be 3.822ms
- 1/2 period (time high and low) is 1.911ms or 19,110 clocks

If you add 19,107 NOP statements, RC0 will be playing node C4.

This use of a series of NOP statements is termed 'In-Line Code'

Option 2: Use subroutines and loops:

A subroutine has several purposes

- It cleans up your code, making it easier to understand and modify
- It allows you to use a routine several places in your code without having to rewrite it over and over

A Wait routine would be useful for the above. To make the wait routine shorter in terms of code size, use a loop

```
CNT0  EQU 1                    ;  save variable CNT0 at address 1

Wait_100us:
          movlw 100
          movwf CNT0
Loop1:
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP
```

```
            decfsz CNT0, F
            goto Loop1

            return
```

This routine will burn 1000 clocks:

- Each time it goes through Loop1, it burns 10 clocks (9 statements plus one for a goto)
- Loop1 executes 100 times
- The total time is then 1000 clocks (10us) (actually 1004 clocks due to setup code)

To make this 19,110 clocks, you can't just change the number 100 to 1911: CNT0 is an 8-bit register which can only store numbers up to 255. For a longer wait, you need a second loop

```
CNT1  EQU 2

C4:
            movlw  19
            movwf  CNT2
LoopC4:
            call   Wait_100us
            decfsz CNT2, F
            goto   LoopC4

            return
```

with the resulting program being

```
#include <p18f4620.inc>

; ---  BLINK.ASM ----
; This program counts on PORTC

; Variables
    CNT0  EQU 1
    CNT1  EQU 2

; Program

    org 0x800

    clrf  TRISA                  ;PORTA is output
    clrf  TRISB                  ;PORTB is output
    clrf  TRISC                  ;PORTC is output
    clrf  TRISD                  ;PORTD is output
    clrf  TRISE                  ;PORTE is output
    movlw 15
    movwf ADCON1                ;everyone is binary

Loop:
    call Wait_1911us             ;  Play note C4 on RC0
    incf PORTC,F
    goto Loop


; --- Subroutines ---

Wait_1911us:
         movlw  19
         movwf  CNT1
Loop2:
         call   Wait_100us
         decfsz CNT1, F
         goto   Loop2
         return

Wait_100us:                      ; Wait 100x10 clocks (100us)
         movlw 100
         movwf CNT0
Loop1:                           ;  Wait 10 clocks (1us)
         NOP
         NOP
         NOP
         NOP
         NOP
         NOP
         NOP
         decfsz CNT0, F
         goto Loop1

         return


    end
```