StatiC v1.0

Language Reference Guide

(Motorola DSP56F80x Microcontrollers)

Updated: 25 January 2005

Peter F Gray
(petegray@ieee.org)

1. Table of Contents

2. Overview

The StatiC compiler supports both traditional sequential and Finite State Machine language methodologies. This feature is automatically enabled when the compiler detects FSM constructs within the source. Dual-methodology support allows software engineers to code, using an identical syntax (but different language constructs) in either the ‘generic’ sequential mode or the inherently multitasking FSM mode.

The sequential language is based on the familiar language constructs of C, Basic and Pascal – with a unified and simplified syntax. The FSM language is the same as the sequential language, except that it has additional FSM extensions. In addition, a command line switch permits the use of C-style syntax and operators.

The compiler has been designed from scratch, specifically for the embedded domain, and includes the features required to support both the sequential and FSM modes of operation. In addition, the languages themselves have been enhanced to remove ‘clutter’ (such as ambiguous operators) found in other languages, as well as incorporating some features more suited for embedded software development.

The StatiC compiler can be hosted under Windows or Linux, and currently targets the Motorola DSP56F80x microcontrollers.

2.1. Compiler Parameters and Switches

The compiler is invoked as follows –

```
static  sourcename  [switches]
```

by convention, StatiC FSM-mode programs have a filetype of .fsm and non-FSM programs have a filetype of .nsm

Switches control the compiler operation, and are as follows…

-c	Use C-style syntax (pointer / address-of operators, assignments, variable declarations, loops, conditional branching)
-s$xxxx	Specifies the initial SP value, in Hex (defaults if unspecified)
-m$xxxx	Specifies the initial Machine SP value, in Hex (defaults if unspecified)
-a	Allows loops and procedure within a transition state ‘causes’ block (relaxes the strict FSM rules)

For example, to produce assembly language for source myprog.nsm –

```
static  myprog.nsm
```

To produce assembly language for the FSM source myotherprog.fsm –

```
static  myotherprog.fsm
```

Output is placed in the file postopt.txt

3. Program Structure – Sequential Mode

A typical sequential program structure follows.

- Comments
- Directives
- Constant Declarations
- Global Variable Declarations
- Procedures and Interrupt Routines
- Program

All items except the Program are optional. Comments can appear anywhere. Most Directives can appear anywhere. Constants and Global Variables must appear prior to being referenced (i.e. referenced from a Procedure or the Program). Procedures must appear before the Program.

The Procedure and Program Block Structures follow. Optional elements are shown between square braces "[" and "]". Keywords (reserved words) and symbols are **bold**-ed.

3.1. Procedure Block Structure

Defines the name, parameters, and code of a call-able routine.

```
procedure procedurename ( [parameter_list] )
begin
  [local variable declarations]
  statements
end
```

Note that forward declarations of procedure block (i.e. prototypes) are allowed, and take the following form –

```
procedure procedurename ( [parameter_list] )
```

3.2. Interrupt Routine Block Structure

Defines the name and code of an interrupt routine.

```
interrupt procedurename ( )
begin
  statements
end
```

3.3. Program Block Structure

Defines the code of the main program.

```
program
begin
  [local variable declarations]
  statements
end
```

4. Program Structure – FSM Mode

A typical FSM program structure follows.

- Comments
- Directives
- Constant Declarations
- Global Variable Declarations
- Machine List
- State List
- Transitions
- Procedures and Interrupt Routines
- Program

As you can see, many of the components of a FSM program structure are present in the Sequential program structure. The extensions required for FSM mode are the Machine List, State List, and the Transitions, which must appear in the order shown above.

4.1. Machine List

Simply lists the machines used in the application.

```
machines  machinename1, machinename2, … machinenameN
```

4.2. State List

Simply lists the allowable application machine states.

```
states statename1, statename2, … statenameN
```

4.3. Transition Block Structure

Defines the conditions required for a state change, and the actions to perform when those conditions are met.

```
transition statename
begin
  condition expression
  causes
    statements
  endcondition
end
```

5. General Language Topics

5.1. Comments

StatiC comments are C-style, with a leading “/* and a trailing “*/”.

```
/*  This is a StatiC language comment  */

/*
This is another StatiC language comment,
spread over multiple lines
*/
```

StatiC single-line comments are C-style, with a leading “//” and terminating at end-of-line.

```
// this is a StatiC single-line comment
```

5.2. Compiler Directives

These tell the compiler to perform a specific action or check a specific condition. Most of the common compiler directives are fairly generic, and so it is with StatiC.

#define DEFINITION_NAME DEFINITION_VALUE
causes the compiler to replace ‘DEFINITION_NAME’ with ‘DEFINITION_VALUE’ wherever it appears in the source code. For the StatiC compiler, this is used for variable name substitution and conditional compilation.

#ifdef DEFINITION_NAME
#endif
causes the compiler to compile the source code between the ‘#ifdef’ and ‘#endif’ directives, if the specified definition name has been defined (using #define).

#asm
#endasm
causes the compiler to pass the code between the ‘#asm’ and ‘#endasm’ through to the assembler.

#inline “Text”
causes the compiler to pass the text between the double-quotes through to the assembler.

#include filename
causes the compiler to “pull” the specified file into the source, at the location of the #include directive. Included files can have a total maximum depth of 9 levels.

5.3. Variables

StatiC variables, like C, have what’s known as ‘scope’. This means that their visibility to different parts of the source (procedures / program) depend upon the location of their declaration. Global variables are declared prior to any procedure or program block. Local variables are declared inside the block to which they are local. StatiC variables have a data type which is most appropriate for the memory architecture and instruction set of the target system. For the DSP56F80x, it is a signed word.

```
word  variablename  size
```

Notice that variables have a ‘size’. This will be discussed later, when we cover arrays. It should be noted that in FSM mode, all variables have global scope. For the DSP56F80x target, **int** and **char** are synonyms for **word**.

If the C-style switch was used, the following syntax is required –

```
word variable

word variable[size]
```

5.4. Constants

Constants, in StatiC, are used to define an unchanging collection of data values. For example, a ‘welcome’ message used to display text via a serial communications interface.

```
const  char  msg1a  “Welcome to the StatiC Demo program”
const  int  msg1b  13,10,0
```

5.5. Procedures

These are blocks of functional code that may be call-ed from other blocks of code and allow the optional passing of parameters. A special kind of procedure is supported using the keyword ‘**interrupt**’. This is, not surprisingly, a procedure that services an interrupt.

Prototyping – the forward declaration of a procedure – is allowed in StatiC.

5.6. Statements

Statements include fundamental language elements such as assignments, loops, conditional branches and calls to procedures.

5.6.1. Assignments

These take the general form

```
LHS = RHS
```

Where LHS represents the Left Hand Side of the assignment (the target) and RHS represents the Right Hand Side of the assignment (the source).

5.6.2. Loops

The C-style ‘while’ loop and ‘for’ loop are the StatiC language loop mechanisms. All other fundamental loops (for, do, etc.) can be constructed from the while loop – the ‘for’ loop is simply provided as a convenience.

```
while (expression)
```

```
  statements
endwhile

for (assignment; condition; final_assignment)
 statements
endfor
```

5.6.3. Calls

This is the subroutine invocation mechanism of the StatiC language. Procedure parameters are optional, but must match the number of parameters specified in the formal procedure definition. The ‘call’ keyword is optional, as procedures can be invoked by referencing their name.

Examples

```
call  myproc()

call  anotherproc (p1, p2, p3)

dothisproc()
```

5.6.4. Branching

All branching in StatiC is conditional, and takes the traditional if..then..else format, with single or compound statements.

Examples

```
if  (a > b)  max = a  endif

if  (a > b)  max = a  else  max = b  endif

if  (a > b)
  max = a
else
  max = b
endif
```

The general form being

```
if expression
  statements
else
  statements
endif
```

5.6.5. Bit Shifting

Logical Shift Left and Logical Shift Right operations are available via the **<<** and **>>** operators. These operate on global and local variables, and parameters.

Example

```
var1 = var2 >> 1
```

shifts var2 one binary digit to the right, placing the result in var1.

5.7. Operators

5.7.1. Math Operators

Math operators include the fundamental math operators, + - / and * (add, subtract, divide and multiply).

5.7.2. Boolean Operators

Boolean operators include | & ~ and ! (or, and, xor and not).

5.7.3. Relational Operators

Relational operators include >, <, =, >=, <=, and <> (greater than, less than, equal to, greater than or equal to, less than or equal to and not equal to). In addition, != is a synonym for <>.

5.7.4. Special Operators

Special operators include @ (address of) and ^ (pointer). If the C-style switch is used, * and & are the address-of and pointer operators.

5.8. Identifiers

An identifier is the name of something - a variable, constant, procedure, etc. All identifiers must have a unique name, the one exception being global and local variable names. Global and local references are determined by the rules of scope.

Identifiers must start with alphabetic character (a-z and A-Z), and can contain alphanumeric characters (a-z, A-Z, 0-9) and '_'. Identifiers can be up to 16 characters in length.

5.9. Syntax Notes

5.9.1. Case Sensitivity

Identifiers, directives and keywords are case sensitive.

5.9.2. Ambiguous Operators and Hieroglyphics

Unlike C, StatiC does not contain ambiguous operators. Also, many of the C 'hieroglyphics' are not present in the StatiC language - for example, the statement-ending semicolon and the condition-related parentheses. Obviously, this is not true when the C-style switch is used.

5.9.3. Block Clarity

This has been improved – for the benefit of code support and enhancement – such that each different code block has a terminator which distinguishes it from other block terminators. Never again will you waste time trying to figure out which '{' a particular '}' is associated with - the end of a statement block, a compound 'if' statement, a 'while' block, etc.

5.10. Arrays

Arrays are a collection of contiguous memory locations, referenced by a common name and a subscript identifying an array element.

Example: a 4 word array called myarray –

```
word  myarray  4
```

creates an array with elements myarray[0], myarray[1], myarray[2], and myarray[3].

If the C-style switch is used, the declaration becomes –

```
word  myarray[4]
```

The general form for referencing an array element is –

arrayname[expression]

so, examples of valid assignment statements would be –

```
temp = xarr[count + 1]

A1[ref]  =  A2[pos]
```

5.11. Pointers

Pointers are declared thus –

```
datatype  ^ pointername
```

for example

```
word ^ptr1
```

declares a pointer to a word data type. If the C-style switch is used, the declaration becomes

```
word *ptr1
```

The use of a pointer simply means "the contents of whatever I'm pointing at". Hence, the following code…

```
word myvar 1
word ^myptr

myptr = &myvar
^myptr = 7
```

…sets the variable 'myvar' equal to '7'.

6. FSM-specific Language Topics

FSM mode allows the use of Finite State Machine constructs, which are inherently multitasking. An application typically consists of multiple machines which – at any point in time – exist in a particular state. A good analogy would be that a machine is like a thread running in a process, or, a machine is like a program running in a multitasking operating system.

The implementation of the FSM methodology requires that the software engineer lists the allowable states of the machines in an application, defines the conditions whereby a machine state transition occurs, and declares the name and initial state of each machine.

Each state and each machine has a unique name (they are, after all, identifiers).

State transitions are used to determine and execute a machine transition from one state to another. They achieve this through the assignment of the reserved word '**nextstate**' (in conjunction with the machine name or '**thismachine**') optionally executing additional code during the transition.

Due to the very nature of state machines, a transition should not include loops or calls (i.e. anything which may cause a transition to 'hang').

The StatiC compiler has limited FSM capabilities – 2 machines and 12 states – which is enough to compile the FSM mode example program.

7. Code Example – Sequential Mode

Here is a complete Sequential Mode StatiC program, which performs simple terminal I/O and allows the user to turn LED's on and off. The target system is NMI's PlugaPod, which is based on Motorola's DSP56F803 chip.

```
// port A definitions for GPIO (LEDs)
#define PAPUR   $0FB0
#define PADR    $0FB1
#define PADDR   $0FB2
#define PAPER   $0FB3
#define PAIAR   $0FB4
#define PAIENR  $0FB5
#define PAIPOLR $0FB6
#define PAIPR   $0FB7
#define PAIESR  $0FB8

// SCI0 definitions for terminal (RS232) interface
#define SCI0BR  $0F00
#define SCI0CR  $0F01
#define SCI0SR  $0F02
#define SCI0DR  $0F03


// output a null-terminated string to SCI0
procedure sci0output (word ^ optr)
begin
  word ostat 1

  while ^optr
    ostat = ^SCI0SR
    while ( ostat & $C000 ) <> $C000
      ostat = ^SCI0SR
    endwhile
    ^SCI0DR = ^optr
    optr = optr + 1
  endwhile
end


// read a character from SCI0
procedure sci0input (word ^ rchar)
begin
  word ostat 1

  ostat = ^SCI0SR
  while ( ostat & $3000 ) <> $3000
    ostat = ^SCI0SR
  endwhile
  ^rchar = ^SCI0DR
end


// the main program
program
begin
  word ichar 1
  ^SCI0BR = 260              // baud 9600
  ^SCI0CR = 12               // 8N1
  ^PAIAR = 0                 // enable LEDs
  ^PAIENR = 0
  ^PAIPOLR = 0
  ^PAIESR = 0
  ^PAPER = $00F8
  ^PADDR = $0007
  ^PAPUR = $00FF
  sci0output("LEDs on/off 1/2=Green 3/4=Yellow 5/6=Red.\n")
  ^PADR = 0                  // LEDs off

  while 1                    // loop forever
    sci0input (@ichar)
    if ( ichar = '1' ) ^PADR = ^PADR | $0004 endif
    if ( ichar = '2' ) ^PADR = ^PADR & $00FB endif
    if ( ichar = '3' ) ^PADR = ^PADR | $0002 endif
    if ( ichar = '4' ) ^PADR = ^PADR & $00FD endif
    if ( ichar = '5' ) ^PADR = ^PADR | $0001 endif
    if ( ichar = '6' ) ^PADR = ^PADR & $00FE endif
  endwhile

end
```

This program displays instructions, and then changes the LEDs (on/off) depending upon what the user types at the keyboard. So, lets examine some of the code in detail.

From within the **program** block, you'll see the following statement –

```
word ichar 1
```

This declares a one word variable, ichar which is local to the **program** block. Next, the statement –

```
^SCI0BR = 260              // baud 9600
```

This loads the SCI (Serial Communications Interface) baud rate register with the value '260', which sets the baud rate of the chip's SCI module to 9600. The statement works this way because

a) We defined SCI0BR, near the top of the program, to be $0F00 – which is the address (in Hex) of the baud rate register for the PlugaPod.
b) We use the ^ operator

This could be thought of as meaning 'load the contents of $0F00 with 260. It could also have been written like this –

```
^$0F00 = 260
```

which would have achieved the same thing. However, it's good practice to
substitute definitions for register addresses because the registers do not always have the same address within the same family of chips. Using definitions means that if you port your code to another chip, which doesn't have the same register address as the original, you'll only need to

change the program in one place – in the #define directive. Besides, SCI0BR is a little more meaningful than $0F00.

The program then sets the various GPIO (General Purpose Input Output) line control registers, which are tied to the LEDs on the PlugaPod. Next, the statement –

```
sci0output ("LEDs on/off 1/2=Green 3/4=Yellow 5/6=Red.\n")
```

calls the SCI output routine, passing the address of the string as the parameter. sci0output is coded to process the string passed to it, displaying the characters (via the SCI) to the terminal.

The program then enters a never-ending loop, reading the keystrokes and adjusting the LEDs accordingly. The statement –

```
sci0input (@ichar)
```

simply calls the SCI input routine, passing the address of the local variable ichar as the parameter. The sci0input routine is coded to wait for keyboard input and return what was typed in the parameter passed to it.

Next, the character returned from the input routine is tested, and the LEDs are adjusted. The statement –

```
if ( ichar = '1' ) ^PADR = ^PADR | $0004 endif
```

performs a 'logical OR' operation on the contents of the GPIO data register, if the user typed a '1' at the keyboard. OR-ing the data register with $0004 sets bit 2 'high' which turns the green LED 'on'.

Let's finish up by looking at one of the procedures, sci0input –

```
ostat = ^SCI0SR
while ( ostat & $3000 ) <> $3000
  ostat = ^SCI0SR
endwhile
^rchar = ^SCI0DR
```

This simply waits until the SCI status register (SCI0SR) indicates that a character has been entered, and then puts the character into wherever rchar is pointing at.

You'll recall that we passed @ichar to the routine, and the formal declaration of the routine was –

```
procedure sci0input (word ^ rchar)
```

so the statement –

```
^rchar = ^SCI0DR
```

actually stores the contents of the SCI data register (SCI0DR) into ichar.

8. Code Example – FSM Mode

Here is a complete FSM Mode StatiC program, which performs simple terminal I/O. Characters entered on the keyboard are received by the microcontroller and echoed on a PC running a terminal emulator. The target system is NMI's PlugaPod, which is based on Motorola's DSP56F803 chip.

```
// definitions for SCI (RS232)
#define SCI0BR  $0F00
#define SCI0CR  $0F01
#define SCI0SR  $0F02
#define SCI0DR  $0F03

// global variables
word appstate 1
word appchar 1

// application control definitions
#define APPSTATEINPUT 1
#define APPSTATEOUTPUT 2


// define the machines
machines inputmachine,outputmachine


// list the states
states waitappinput,waitinput,waitappoutput,doappoutput


// describe the transitions
transition waitappinput
begin
  condition appstate = APPSTATEINPUT
  causes
    nextstate[thismachine] = waitinput
  endcondition
end

transition waitinput
begin
  condition ( ^SCI0SR & $3000 ) = $3000
  causes
    appchar = ^SCI0DR
    appstate = APPSTATEOUTPUT
    nextstate[thismachine] = waitappinput
  endcondition
end

transition waitappoutput
begin
  condition appstate = APPSTATEOUTPUT
  causes
    nextstate[thismachine] = doappoutput
  endcondition
end

transition doappoutput
```

```
begin
  condition ( ^SCI0SR & $C000 ) = $C000
  causes
    ^SCI0DR = appchar
    appstate = APPSTATEINPUT
    nextstate[thismachine] = waitappoutput
  endcondition
end

// a procedure used at start-up, to display welcome message
procedure sci0output (word ^ optr)
begin
  word ostat 1

  while ^optr
    ostat = ^SCI0SR
    while ( ostat & $C000 ) <> $C000
      ostat = ^SCI0SR
    endwhile
    ^SCI0DR = ^optr
    optr = optr + 1
  endwhile
end


// the main program
program
begin

  ^SCI0BR = 260                  // baud rate 9600
  ^SCI0CR = 12                   // 8N1
  sci0output ("StatiC FSM SCI Demo Ready.\n")
  appstate = APPSTATEINPUT       // the initial app state
  init inputmachine waitappinput 10
  init outputmachine waitappoutput 10

end

// at this point, all of the defined machines are 'running'
```

Skipping the parts we've already covered in the Sequential Mode example, let's examine the code in detail.

Firstly, you'll notice that this application consists of two machines – inputmachine and outputmachine. The 'main' part of the program…

```
      ^SCI0BR = 260                  // baud rate 9600
      ^SCI0CR = 12                   // 8N1
      sci0output ("StatiC FSM SCI Demo Ready.\n")
      appstate = APPSTATEINPUT       // the initial app state
```

…simply sets up the SCI (Serial Communications Interface), displays a message, and sets the global variable 'appstate' to be 'APPSTATEINPUT'. This particular application is designed in such a way that the two machines are cooperative, and the setting of 'appstate' determines their transition to / from one state to another. Machines don't have to behave this way, but it's useful, for demonstration purposes.

Once the program block – as described above – has been executed, all machines are activated. That is to say, they're put into their 'initial state' as determined by the machine definition statements –

```
init inputmachine waitappinput 10
init outputmachine waitappoutput 10
```

So, inputmachine is put into 'waitappinput' state, and outputmachine is put into 'waitappoutput' state. Once in these states, they will remain in these states until the state transition conditions have been satisfied. So, inputmachine is initially in 'waitappinput' state, which is described in the transition block, thus –

```
transition waitappinput
begin
  condition appstate = APPSTATEINPUT
  causes
    nextstate[thismachine] = waitinput
  endcondition
end
```

However, appstate was defined as 'APPSTATEINPUT' in the main program block, so the inputmachine's transition condition is satisfied. This causes inputmachine to change states to 'waitinput'.

You'll notice also that outputmachine's initial state is 'waitappoutput', which is described in the transition block, thus –

```
transition waitappoutput
begin
  condition appstate = APPSTATEOUTPUT
  causes
    nextstate[thismachine] = doappoutput
  endcondition
end
```

Unlike the inputmachine, outputmachine's transition condition has not been satisfied, so no state change takes place, and outputmachine remains in the 'waitappoutput' state.

At this point in time, outputmachine is waiting for its transition condition to be satisfied, and inputmachine has changed state to 'waitinput'. So, looking at the 'waitinput' transition block –

```
transition waitinput
begin
  condition ( ^SCI0SR & $3000 ) = $3000
  causes
    appchar = ^SCI0DR
    appstate = APPSTATEOUTPUT
    nextstate[thismachine] = waitappinput
  endcondition
end
```

We see that inputmachine will remain in this 'waitinput' state until a key has been pressed at the keyboard (i.e. in the terminal emulator, running on the PC). The outputmachine is still waiting for its transition conditions to be satisfied.

When a key is pressed in the terminal emulator, inputmachine's transition conditions are satisfied, a character is read from the SCI data buffer into the global variable, appchar, the appstate is set to 'APPSTATEOUTPUT' and inputmachine performs a state change back to 'waitappinput'.

At this point, outputmachine's state transition conditions have been satisfied (because appstate was set to 'APPSTATEOUTPUT' by inputmachine), so outputmachine experiences a state change from 'waitappoutput' to 'doappoutput'. Looking at the 'doappoutput' transition block –

```
transition doappoutput
begin
  condition ( ^SCI0SR & $C000 ) = $C000
  causes
    ^SCI0DR = appchar
    appstate = APPSTATEINPUT
    nextstate[thismachine] = waitappoutput
  endcondition
end
```

We see that the outputmachine will wait until the SCI is ready to send, then it loads the SCI data register with the global variables, appchar, sets the appstate to 'APPSTATEINPUT', and performs a state change back to 'waitappoutput'. While this is all happening, inputmachine does nothing, because its state transition conditions have not been satisfied.

At this point in time, both machines are back in their initial states, and the whole cycle starts again.

8.1. Why FSM ?

Now, you may be thinking "Why on earth would anyone want to code like this?" and it's a perfectly reasonable question. The answer is, because it's inherently multitasking. For example, let's say that you've coded the above example, and you want to also have your application monitor an ADC reading and set a GPIO line high if the reading goes above a certain point… all you have to do is add another machine. Want to send PWM signals? Add another machine.

There's no difficult "Where on Earth do I put this new code so that the existing code still works?" – machines run independently from each other (unless, of course, you deliberately design them to be cooperative).

You simply create machines, as required, to perform the tasks you desire. Each machine runs and changes state when its transition conditions are satisfied. All of the machines you define are running at the same time – the same as a multitasking operating system – and performing whatever function you've designed them to do. This is the true power of FSM programming.

9. Performance and other considerations

9.1. *Variables and Procedures*

Stack usage is costly, from both a speed and code size perspective. Local StatiC variables – as with most compilers / languages – are located on the stack. Hence, if you want faster and smaller code, avoid using local variables, i.e. use global variables wherever possible.

Additionally, it makes no sense to pass a global variable as a parameter to a procedure. Global variables have global scope, so passing them to a procedure is simply a waste of stack space and generates unnecessary assembly code.

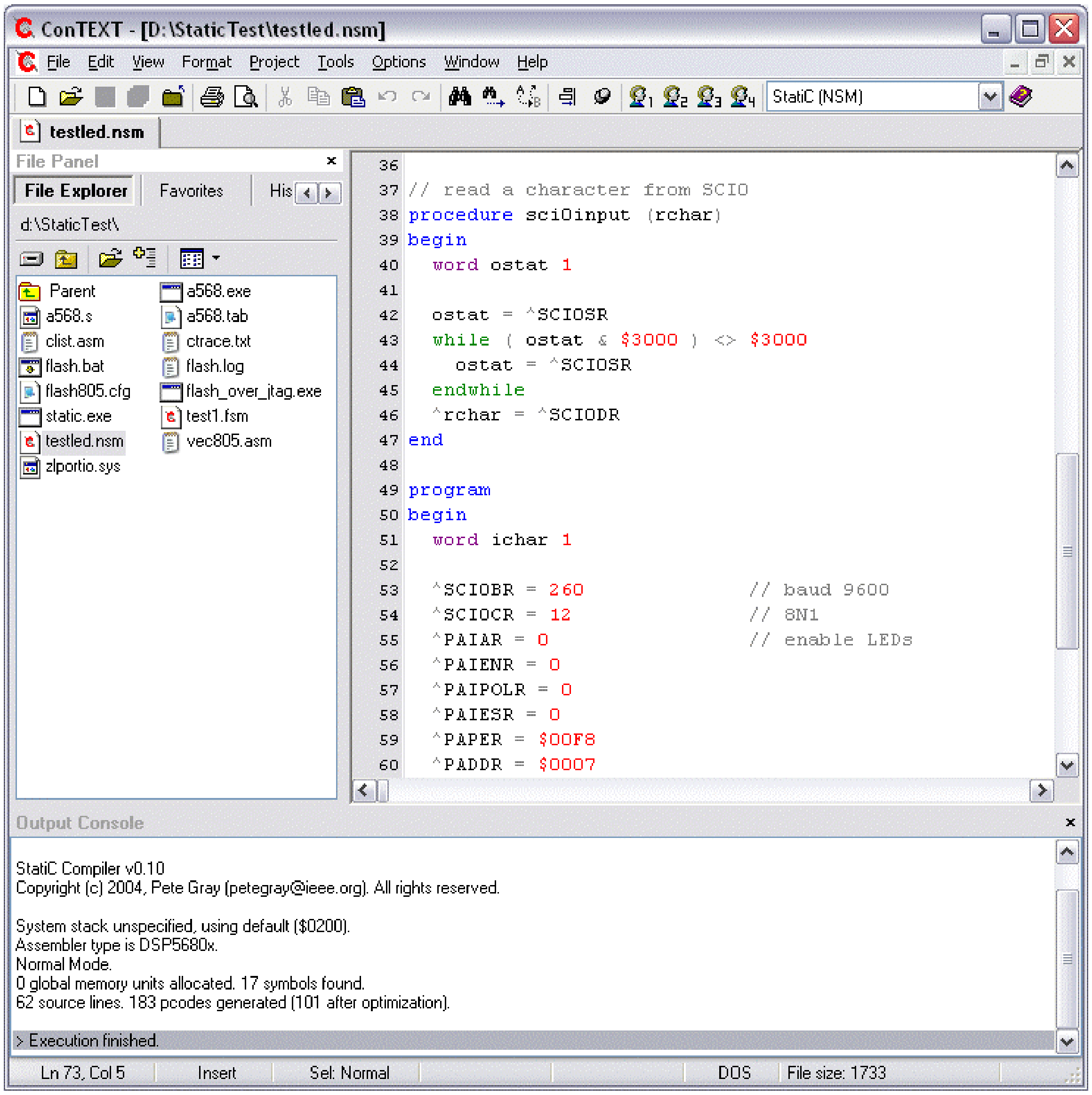
9.2. *Complexity*

Complex statements may be too difficult for the compiler's optimizer to rationalize into efficient code. Additionally, the language parser may not be able to de-construct very complex statements. Hence, more efficient assembly code will be generated if complex statements are split into multiple, simpler, statements.

10. Using a GUI / IDE with StatiC

10.1. Windows - ConTEXT

The use of a highlighting, language-sensitive editor is always beneficial. One such editor, for a Windows host, is ConTEXT (http://www.fixedsys.com/context), which also includes functionality to invoke compilers etc.



```

// read a character from SCIO
procedure sci0input (rchar)
begin
  word ostat 1

  ostat = ^SCIOSR
  while ( ostat & $3000 ) <> $3000
    ostat = ^SCIOSR
  endwhile
  ^rchar = ^SCIODR
end

program
begin
  word ichar 1

  ^SCIOBR = 260               // baud 9600
  ^SCIOCR = 12                // 8N1
  ^PAIAR = 0                  // enable LEDs
  ^PAIENR = 0
  ^PAIPOLR = 0
  ^PAIESR = 0
  ^PAPER = $00F8
  ^PADDR = $0007
```

```
StatiC Compiler v0.10
Copyright (c) 2004, Pete Gray (petegray@ieee.org). All rights reserved.

System stack unspecified, using default ($0200).
Assembler type is DSP5680x.
Normal Mode.
0 global memory units allocated. 17 symbols found.
62 source lines. 183 pcodes generated (101 after optimization).

> Execution finished.
```

10.2. Linux

A large variety of highly configurable editors and IDE's exist for Linux. My personal preference is JED or EMACS.