

***VIRTUAL C MACHINE AND
INTEGRATED DEVELOPMENT ENVIRONMENT
FOR ATMS CONTROLLERS***

Newell Crookston
Brian Whipple
William Grenney

Utah State University

UDOT Report #00.07

April 2000

Disclaimer

The contents of this report reflect the views of the authors, who are responsible for the facts and accuracy of the information presented herein. This document is disseminated under the sponsorship of the Department of Transportation, University Transportation Centers Program, in the interest of information exchange. The U.S. government assumes no liability for the contents or use thereof.

TABLE OF CONTENTS

INTRODUCTION.....	1
Background.....	1
OBJECTIVES.....	2
PREPARING WITH OS-9 AND FASTRAK.....	4
Working with the 2070 Controller	4
Using FasTrak.....	4
The 2070 Controller and OS-9.....	5
TRAFFIC LIGHT SIMULATION TESTS.....	8
Traffic Light Simulation Using the Controller’s Front Panel.....	8
Traffic Light Simulation Using the Graphical Process Model	10
BUILDING THE INTERPRETER, C COMPILER, AND IDE.....	11
The Interpreter.....	12
The Compiler	13
CONCLUSIONS.....	19
RESOURCES	20
APPENDIX A: GRAMMAR.....	21
APPENDIX B: SOURCE CODE.....	Available on Request

INTRODUCTION

Background

Many digital controllers are in use today, and the number is rapidly increasing as additional elements of Intelligent Transportation Systems (ITS) are implemented. Digital controllers perform functions in accordance with the software code implemented in the controller. When modified or extended functionality is required, new software code must be loaded into each individual controller. An even more difficult problem is caused by the fact that different controllers use different proprietary operating systems, therefore, custom code must be developed for a variety of controller types.

Because of the proprietary characteristics of the software environments, modifying programs and interfacing different brands of controllers is an expensive and difficult task. Due to their nature, controllers normally used in transportation are linked by electronic networks. Considerable economies could be achieved by implementing a virtual machine on the controllers that process uniform byte code, which can be accessed by means of a wide area network.

It is difficult to implement a virtual machine on current controllers because of limitations of the central processing unit and operating system. For example, attempts to install Java, the popular virtual machine developed by Sun Microsystems™, have not been successful because of many requirements for computer resources, which are not available on the controllers.

Achieving a practical virtual machine that fits on current controllers and provides the functionality needed for complex traffic operations could produce substantial savings for traffic control operations.

OBJECTIVES

The overall objective of this project is to develop a prototype virtual machine that fits on current ATMS controllers and provides functionality for complex traffic operations. The virtual machine will be developed and tested using the popular controller: the Matrix Model 2070 ATMS Controller. Pictures of the controller are shown in Figure 1. The controller operates under Microware OS-9 real-time operating system. Development tools are provided by Microware FasTrak tools including a library of C functions, the Makefile Editor, debugger, and Target System Tool.

Objective 1:

Obtain a Matrix Corporation Model 2070 ATMS Controller with OS-9 operating system and C compiler for conducting the study.

Objective 2:

Study and learn the features of the controller, operating system, and compiler so effective software can be developed.

Objective 3:

Investigate candidate virtual machines, such as Java, for implementation on the controller.

Objective 4:

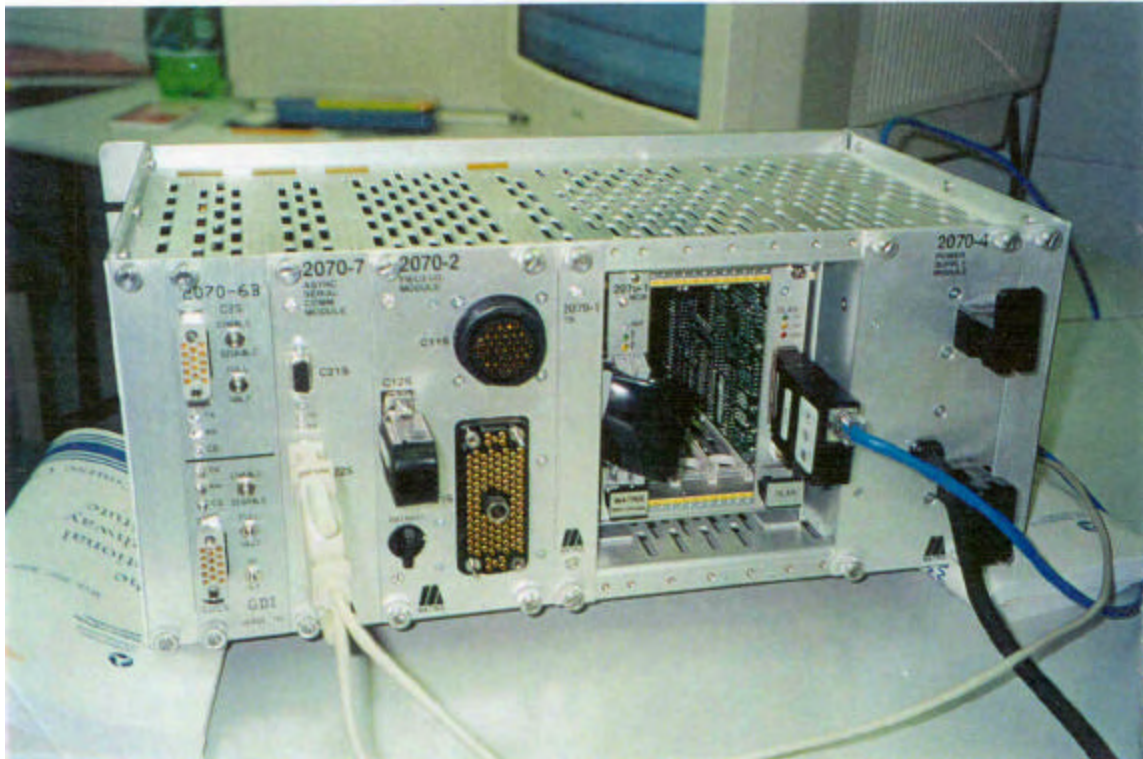
Develop or modify an existing virtual machine for prototype implementation on the controller.

Objective 5:

Develop or modify an existing Integrated Development Environment (IDE) for the prototype virtual machine.

Objective 6:

Demonstrate the prototype virtual machine and software development environment.



PREPARING WITH OS-9 AND FAS TRAK

Working with the 2070 Controller

Gardner Transportation Systems, Inc. and the Utah Department of Transportation provided a Matrix Model 2070 ATMS Controller, Microware OS-9 operating system, and Microware FasTrak software tools for the study. The equipment was loaned to Utah State University for the project's duration.

Using FasTrak

FasTrak is a library of C functions provided by Microware, Inc. It consists of three primary tools:

- The Makefile Editor
- The Debugger
- The Target System Tool

Codewright Professional text editor is bundled into the FasTrak package. The Makefile Editor tool uses user developed source code to generate procedure files called "makefiles," which describe the relationship between the final product and the source files that build the product.

After the make file is generated an executable module can be compiled for the DOS environment using the command:

```
Os9make -f makefilename.mak
```

The compiler has a useful set of error messages to assist the user with debugging the source code.

Executable modules also can be compiled in the MS 9x/NT environment using the "Build" command. However if the compilation fails under MS 9x/NT, no error messages are displayed – the .exe simply is not created.

The Target System Tool is used to interact with the 2070 controller. The OS-9 executable modules created through OS9 Make, as explained above, will run on the controller, and processes can be monitored by the Target Tool. Also, the activities of the processes running on the controller can be changed through inter-process communication. The input command of

the client process determines the procedure sequence, and so the activity of the controller can be controlled remotely using the FasTrak Target System Tool.

The 2070 Controller and OS-9

The Transportation Electrical Equipment Specifications (TEES) were used to understand the operation of the controller. Also the book “Using Microprocessors and Microcomputers – the Motorola Family” provided excellent reference material. Excellent information for the operation of OS-9 was found in the “OS-9 Primer” at the c:\mwos\doc\FTW68K\home.pdf documentation file supplied with the software.

1. **Development Environment:** in our project, we use a PC-hosted development environment. Matrix "2070 DOS Development Package" (PCBridge) installed on a PC, served as a bridge to allow development of an OS-9 system on a standard IBM compatible computer running DOS or an equivalent. The PCBridge let us compile our programs and create OS-9 modules on PC. When networking features are not configured, PCBridge communicates with OS-9 through a serial interface between the PC and OD-9 systems. Using a hyper-terminal (obtained from the accessory of windows), which interacts with the controller's embedded OS-9, modules are transferred using the kermit protocol.

Also there is a one-time process to configure networking on the controller. Once this has been done, we just need to run a procedure file to get the networking up and running. The procedure file "loading" is enclosed in Appendix (see Appendix 1.). Using ping to test if the network connection is established. Then, start the FasTrak's target tool, connect to the controller (in our project, ip address: 129.123.9.78), and run the modules created on PC remotely (see chapter 1 about FasTrack target tool). The target system window acts as a dumb terminal. It allows us to seamlessly execute commands that reside on the host as if they were resident on the OS-9 system.

OS-9 system is multi-tasking, many users on the system can run shell simultaneously, or one user can run many processes concurrently.

2. **OS-9 shell is similar to Unix.** The following are some common shell commands and utilities that we found to be useful for the project:

setenv – declare the variable and set the value of the variable

unsetenv –clear the value and remove the variable from storage
printenv – print the variables and their values to standard output
dir – get a listing of the files in the current directory
chd <path> – change the current data directory to the directory specified by the path
pd – print the absolute location of the current directory
kill <proc ID> – abort the process specified by <proc ID>
profile <path> – read input from a named file and then return to the shell's original input source
procs – lists all processes currently running on OS-9
mdir – lists all the modules in memory
wait – wait for all child processes to terminate
devs – lists all initialized devices on the system
iniz <device descriptor name> – initialize the device
load – to place the modules from a file into memory

3. **Environment variables can be retrieved within a program** (see Appendix A.).

4. **Paths and Processes:**

The number of each path represents a 256 byte system data structure called a path descriptor. OS-9 using this number to hold information pertaining to the particular path. This information includes the actual device descriptor/driver/file manager related to the path, the number of processes using the path, access modes the path was opened with, and the memory location of storage area for the particular path's driver. So the idea of imbedded system programming simply is write to a specific address and then read from that location.

5. **Basic C Calls for Paths:**

_os_open(): open a path to an existing file or device
_os_create(): create a new file, and open it at the same time
_os_dup(): duplicate one of its open paths
_os_close(): close an access to a path
_os_read(): reading data from a path that has been opened
_os_write(): data are written to a device
_os_exec() and _os_fork: to spawn a child process, a potential parent make a call

_os_exec(), and _os_fork create a new process

6. **There are six serial ports on the controller.** The global variables used for their path id are all defined in file: c:\mwos\os9\cpu32\ports\mscpu360\src\config.h :

```
#define FIELDIO_PORT    "/sp5"    /* sp5 , scc3 , /t2 */
#define FRONTPANEL_PORT "/sp6"    /* sp6 , scc4 , /t3 */
#define GP_TIMER_PORT   "/tims"
#define SRAM_DISK       "/r0"
#define FLASH_DISK      "/f0"
#define LED_PORT        "/led"
#define SP1_PORT        "/sp1"    /* sp1 , scc1 , /t0 */
#define SP2_PORT        "/sp2"    /* sp2 , scc2 , /t1 */
#define SP3_PORT        "/sp3"    /* sp3 , smc1 , /t4 */
#define SP4_PORT        "/sp4"    /* sp3 , smc1 , /t4 */
```

Port 4 (SP4_PORT) was used during the project to interact with the controller.

Validation Suit (TEES 9-2-0): provided a working example of how to program all functions, and all of the 2070 controller's unit functions were tested. Execution from the shell will commence by typing "simpletest" from the prompt. The specific unit will be tested according to the command (A, B... Q) keyed in:

- <A> Display the Time of Day from the hardware TODC
- Set the Time of Day.
- <C> Set the Kernel TOD from the hardware TODC
- <D> Show Field I/O DataKey Contents
- <E> Verify Linesync Interrupts
- <F> Display the Inputs from the Field I/O Module
- <G> Set the Outputs on the Field I/O Module
- <H> Serial Port Tests
- <I> General Purpose Timer Tests
- <J> Display text on the Front Panel
- <K> Read input from the Front Panel
- <L> Flash Disk Test
- <M> SRAM Disk Test
- <N> CPU Activity LED Test

- <O> Activate the CPU Reset
- <P> 2070 System Test
- <R> Continuous 2070 System Test
- <S> OS9 Command Shell
- <Q> Quit

7. **Inter-process Communication and Signals:**

Signals are software generated interrupts. When a process has a signal sent to it, that process will put its current code location on hold, execute a signal handling function, and after the handler is done, continue executing at the place put on hold.

Signal intercept routine:

`_os_intercept()` installs a routine, specified as one of the input parameters, as the process' signal handler.

`_os_send()` will send the signal to a specific process.

8. **Combining Assembly Language with C Routines:** in the Ultra C compiler, there is a built-in function `_asm()` to allow assembly instructions to be placed in a C source file.

TRAFFIC LIGHT SIMULATION TESTS

Traffic Light Simulation Using the Controller's Front Panel

The goal of this task was to learn how to send instructions through a client process to a server process using the FasTrak Target Tool residing on the host PC. The Target Tool was used to control the activities of the modules running on the controller. (Code details are included in Appendix A).

Two modules, the client and the server, are created on host PC. The FasTrak Target Tool was used to run these two modules remotely by connection to the controller, which also is a network node. At the prompt from the client process:

<1> traffic lights cycles simulation.

<2> Led flash five times.

<3> QUIT.

On entering 1, traffic lights simulation will appear on the front panel:

South-West	North-East
Red	Red
Yellow	Yellow
Green	Green

Blinking text indicates that a light is on.

The original cycle is:

<u>South-West</u>		<u>North-East</u>
45s	Red	Green
8s	Red	Yellow
4s	Red	Red
	35s	Green
8s	Yellow	Red
4s	Red	Red

command 1 – scale the original cycle down to 1:10, with the active light off;

command 2 – scale the original cycle down to 1:5, with the active light on;

command 3 – exit the simulation.

The step-by-step procedure follows:

- (1) Open the FasTrak target tool and connect to controller (ip = 129.123.9.78).
- (2) Load the module "server" from "Action" menu bar.
- (3) At the prompt, type "client" to run client module.
- (4) Respond to the prompt according to the menu.
- (5) When testing is done, unlink the server module.

Each time a module is modified, the new module should be reloaded into the memory.

A second test was conducted using an interrupt process. The light cycle was put into an infinite loop. The user can trigger an interrupt from the console or from a networked PC. The interrupt handlers reside in the body of each simulation procedure.

Traffic Light Simulation Using the Graphical Process Model

A graphical process model was developed on the PC using Borland's Delphi™ 4.0. The code details are included in Appendix A. The simulated traffic light cycle is the same as described above. The program uses threads to control the "start," "suspend," and "resume" status of the light. Sockets are used to do the inter-process communication. The procedure for running the module follows:

- (1) Run project2.exe - form 2 will pop up.
- (2) Click ShowClient button and another form 1 will pop up.
- (3) Type in an integer scale in the scale field.
- (4) Type in "0" in the text field of the Client form to start the light cycle simulation. The text field will display the command for confirmation of Client's request.
- (5) Type in 1 in the text field of the Client form to suspend the lights cycle simulation. Text field of form 2 will echo and show the real command of Client's request.
- (6) Type in 2 in the text field of the Client form to resume the lights cycle simulation. Text field of form 2 will echo and show the real command of Client's request.
- (7) In form 1, start, suspend, and resume all can be tested independently; the cycle frequency also can be changed during the testing.
- (8) Use the menu to terminate the program.

BUILDING THE C COMPILER, IDE, AND INTERPRETER

This section of the report describes a virtual machine and integrated development environment for the Matrix 2070 ATMS controller.

First we reviewed the availability and feasibility for implementing alternative virtual machines for this controller. From the beginning we were told that this product was desirable for two reasons. The first reason was the cost of a programming seat on “codewrite,” the current compiler, is high. The second reason was that frequent changes are anticipated by UDOT, and a product was needed to support downloads over a wide area network. The first suggestion was to use a Java virtual machine. Sun Microsystems and microware recently have developed an embedded Java for OS9. Although this possibility was considered, the price was not reasonable. A single seat on Sun’s embedded server would cost \$3500 (<http://performance-computing.com/news/981005.shtml>). After further consideration of using Java, it became apparent that this was probably not the best route. Looking into other languages, it was decided that it would be more feasible to write our own language that would be a subset of C and would include all the functionality that was required for traffic controller projects.

We also looked into other languages and put together how we would construct our language. We wanted the syntax to be the same as, or very close to, C. This approach would have some benefits over Java. First, it would allow previous code to compile and run on our interpreter. Second previous programs were written in C and the programmers are familiar with it. Third, C is an established language if we used its syntax, over on designed by us, we would be able to avoid ambiguities in the syntax.

The task of designing the language was delegated to Newell Crookston, head programmer. Newell looked into many interpreters already on the market for ideas on his design. The one that he found most useful was called Haskell. Haskell is a purely functional language designed at Yale. One of Haskell’s features that we liked was how it handled arrays. One of the

major differences of our language and C is that all arrays are dynamic. For example: if you call for a value outside the arrays original bounds it will automatically resize the array to fit the size requested. This feature simplified many of the other problems that were encountered when writing a language. For example, all functions are stored in the byte code as an array. In the debugger, looking into the array will typically show the function stored in the array. When debugging your code this is a useful thing to know.

As discussed above it was decided that a subset of the C language would best fit our purposes. The next major hurdle was deciding what features would be included in our language. With little experience working with the controller, and the grad student who did most of the work leaving for another job, this was a difficult task. After meeting with the technical advisory committee we had a much better idea of what functionality they wanted and expected to include most of it in the final prototype.

The Interpreter

Running byte code on our interpreter is simple. First, choose “Compile Make File” from the Tools menu. This will produce byte code with extra information for the debugger. If this byte code is to be used on the 2070 controller you must then save the byte code with no debug, this option also is found in the tools menu. Once you have the byte code with no debug information, you are ready to download in onto the Controller. We chose to call the virtual machine interpret so once the byte code file is on the controller you would type interpret <ByteCodeFile.ext>. Our interpreter does not support multiple threads, so if multiple programs must run at the same time, one or more of them must run in the background.

Along with the virtual machine for the Controller we were to include a virtual machine for a PC and a Compiler. Currently the virtual machine for the PC is in the form of a debugger, in the debugger you can watch your code being executed and see the values of the buffers. The

compiler comes with a development environment and was demonstrated at our last meeting with Gardner.

The Compiler

Upon first opening the Compiler you will be greeted by an empty Window (Figure 1). From here you can open documents, start new documents, or create and edit make files. You will have the Open, New, and Exit options enabled from the File menu. These will allow you to open a "C" file for editing, or create a new edit window for creating a new "C" file.

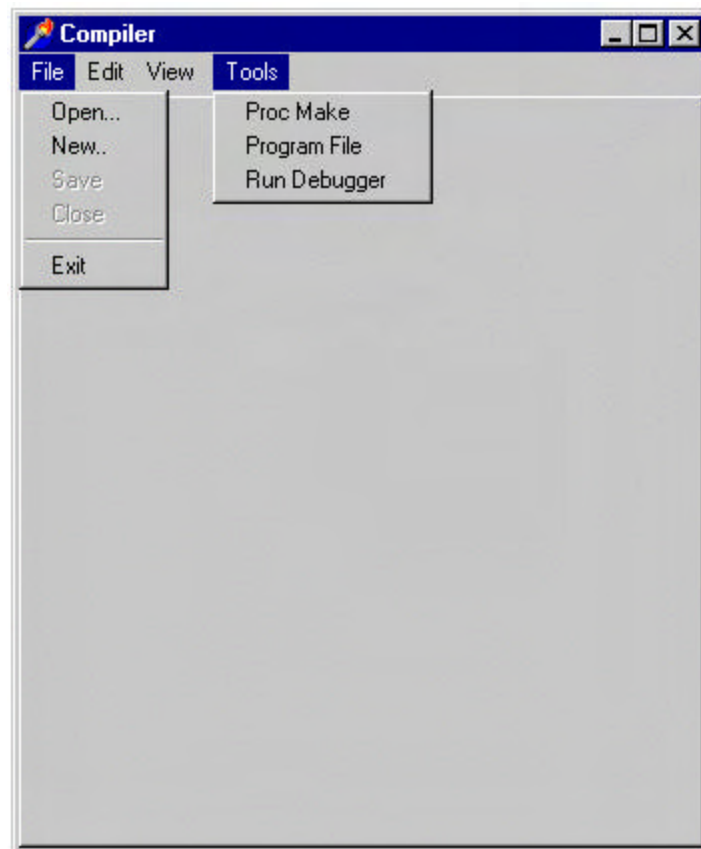
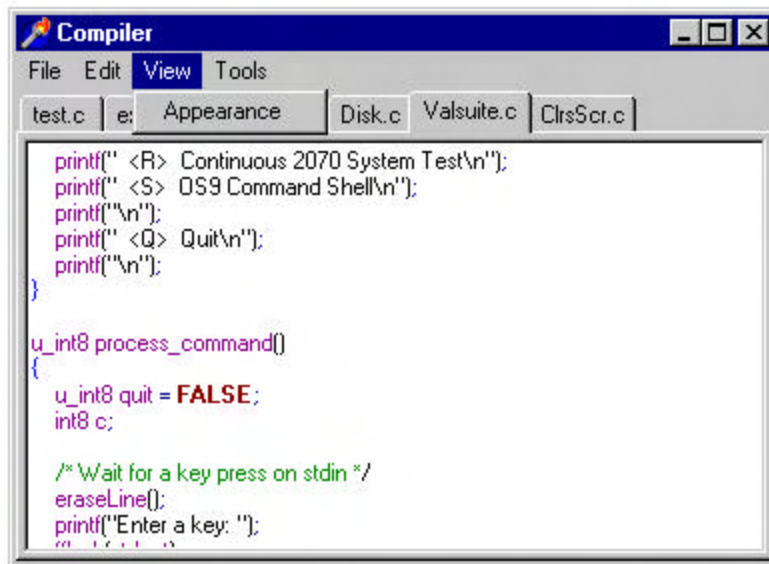


Figure 1 :Open Window for IDE

After a File is opened the View and Edit menus are enabled. A text edit area is opened in the main window, which will allow for modifying any text document. Text will be colored according to C statement. Figure 2 shows a sample source code file. In this example, comments are green, keywords in bold blue, numbers in light blue, and symbols in red. The coloring can be modified in the View Appearance menu option (Figure 3).



```
printf("<R> Continuous 2070 System Test\n");
printf("<S> OS9 Command Shell\n");
printf("\n");
printf("<Q> Quit\n");
printf("\n");
}

u_int8 process_command()
{
    u_int8 quit = FALSE;
    int8 c;

    /* Wait for a key press on stdin */
    eraseLine();
    printf("Enter a key: ");
```

Figure 2: Edit Window, example source code

You can have as many open text areas as you want. To switch in between text areas you can use the tabs at the top of the window, as shown. Choosing the Appearance option will display the following (Figure 3). This window will allow the programmer to choose the colors that best fit personal preferences.

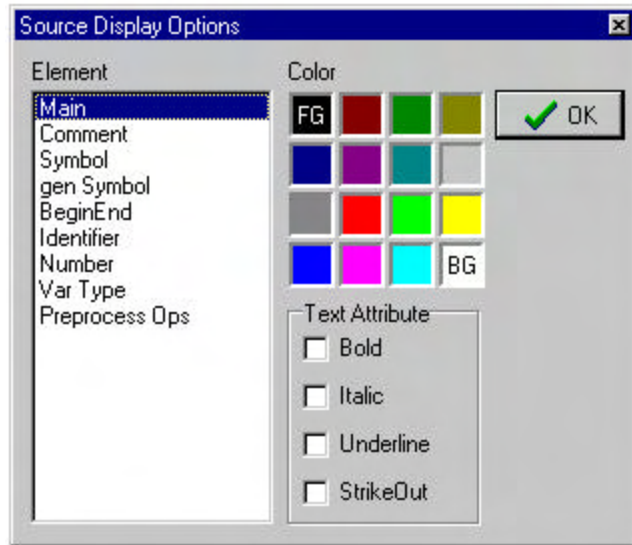


Figure 3: Appearance window

From the Tools menu, there is the option of "Proc Make" (process make) this will prompt for a make file. Opening the make file will take that make file and generate the byte code from the source files listed in the make file. It will then display the program Info window (Figure 4).

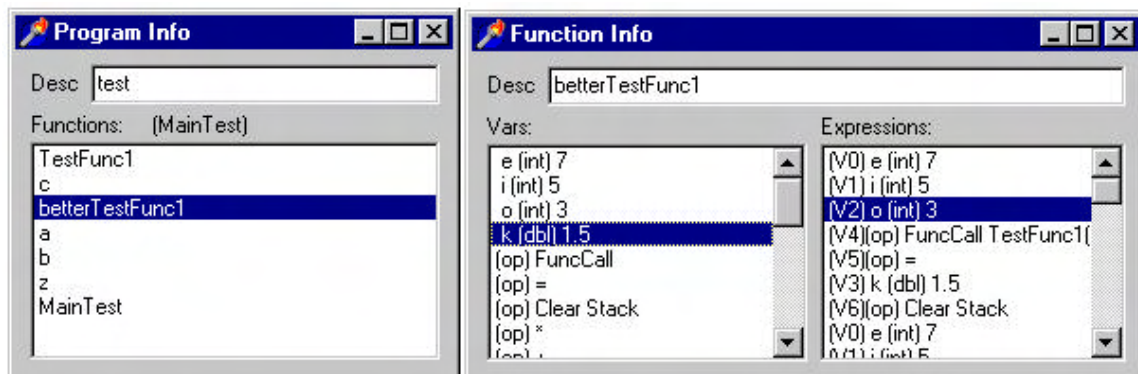


Figure 4: Program and function information window

The Program Info window will show a list of all the functions in the program. Double clicking on any of the functions will take you to the Function Info window, shown here on the right. This will

give a list of variables and a list of byte code instructions, which will prove useful in the debug process. The debug option will give you a similar window and will allow you to step through the program. The way this runs is specifically made for use in developing for the 2070 controller. The byte code is developed and debugged on a PC for implementation on the Controller. This can cause some difficulty in debugging and testing the program. Consequentially we developed our debugger to step you through the byte code, showing you the values of the variables and stack as each expression is executed. This will allow the programmer to see what the program is doing before testing it on the 2070 controller. Eventually functionality will be added where the programmer will be able to step through the source code line by line. Currently, the programmer only is allowed to step through the byte code produced by the source code. We also hope to add a full virtual machine that will run on the PC. This virtual machine would run on the PC exactly as the virtual machine would run on the 2070 controller.

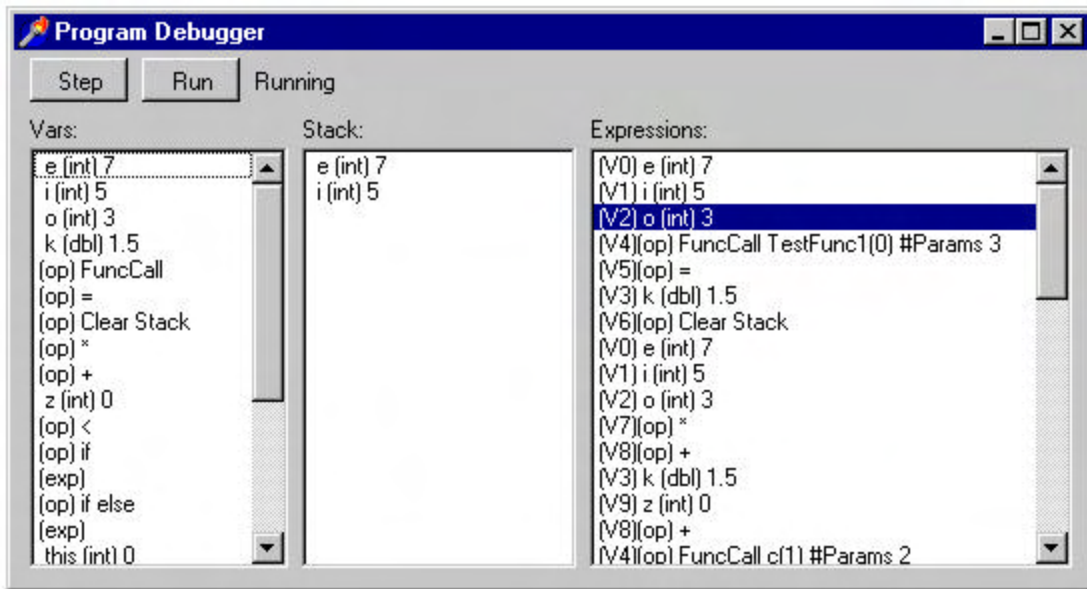


Figure 5: Debug Window

If a main function is not present in the body of the program with the main key word, then the program automatically chooses the last function in the program to be the main function. In the

debug window of Figure 5 we are using the sample code found in Figure 2. This code was prepared to test and demonstrate features of the compiler. In this example there is no main function so MainTest became the new main function. In the debug window of Figure 5 we have just stepped into BetterTestFunction1. BetterTestFunction1 has default values on three of its parameters, which is supported in this language, the values are found on the stack. They were out on the stack as soon as we entered the function. As you can see in Figure 5 we have just assigned the default values, first three lines of expression area, and we are about to step into the function call testFunc1.

To produce byte code you must first create a make file. Creating a make file is simple. A make file is just a text document listing the source files in your program. For our make files each source file has its own line with the source that has the main function listed last. Save this file with a .make extension for use with the debugger. The procedure may best be described by the following steps:

- Step 1: Open a new text area in the main window.
- Step 2: List all the files that are included in your program – each file on its own line.
- Step 3: Save this file with the .Mak extension.

This procedure will give you a make file that the Make Proc option will turn into a BCP (Byte Code program) File. When running the debugger open the BCP file created by the make Proc. Option.

Appendix A defines the grammar for the language. Appendix B contains the source code. Some of the major differences between this compiler C and conventional C are:

- 1) Case insensitive on key words**
- 2) No Type Casting**
- 3) No unions or structs**
- 4) No Octal or Hex numbers**

5) No case statements

6) No Goto

7) No Pointers

8) No bitwise operations

9) No Preprocessor

10) Does not include the following functionality

- a) <<. >>
- b) <<=, >>=
- c) ##
- d) sizeof
- e) &=
- f) ^=
- g) ||=

CONCLUSIONS

A new computer language grammar was created. An Integrated Development Environment and compiler were developed for the grammar. A virtual machine was developed to operate with the OS-9 operating system on the 2070 controller. Several simple demonstration programs were written using the new grammar and successfully implemented on the 2070 controller.

An attempt was made to develop more sophisticated programs of the type that could be expected for practical applications. The grammar was too restrictive to be of practical value.

Although the approach adopted in this study did not succeed in providing a commercial tool, it did produce several benefits. It eliminated one dead-end approach from future consideration by us or others. Also, the development of the software system had great educational value for the students, and better prepared them for pushing the envelope in the future.

Results of the study indicated that a better approach in the future may be to use a modified version of a JAVA engine.

RESOURCES

1. Transportation Electrical Equipment Specifications (TEES); March 1997.
2. Microware 2070 DOS Development Package Documentation and Source Code, September 25, 1998.
3. Mark A. Heilpern, OS9 PRIMER, 1994.
4. William C. Wray, et. al. Using Microprocessors and Microcomputers – the Motorola Family, Fourth Edition, 1999.
5. G. J. Lipovski, 16-And-32 Bit Microcomputer Interfacing – Programming Examples in C and M6800 Family Assembly Language, 1990.

APPENDIX A

GRAMMAR

Lexemes:

Type-><void>
-><Integer>
-><Real>
-><Char>
-><bool>

Void->

Integer-><Digit> | <Digit><Integer>

Digit->>'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'|'0'

REAL-><Integer> "." <Integer>

Char->>'_'|'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m'|'n'|
'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|'z'|'A'|'B'|'C'|
'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|'N'|'O'|'P'|'Q'|'R'|
'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z'

Identifier-><Char> | <Identifier><Char>

Variable-><Type><Identifier>

Const->'const' <Type><Identifier>

Operators:

ArithmeticOp->'+'|'-'|'*'|'/'|'%'

EqualityOp->>'=='|'!='

IncOp->'++'|'--'|<IncOp>('+'|'-')

LogicalOp->'&&'|'||'|<EqualityOp>

RelationOp->'<'|'>'|'>='|'<='

AssignmentOp->'='|'*='|'%= '|'/='|'+='|'-='

BranchOp->'break' | 'return'

Goal-><Program | Library>

Program-><IncludeSection><FunctionProtoSection><FunctionSection>

IncludeSection->('#include' <IncludeFileName>) |

<IncludeSection>('#include <IncludeFileName>

IncludeFileName-><'<'filename'>'> | <'"'filename'">

FunctionProtoSection-><FunctionDec>';'|

<FunctionDec><FunctionProtoSection>

FunctionDec-><Type><FunctionName>'(<parameters>')

FunctionName-><Identifier>

Parameters-><Parameter> | <Parameters>','<Parameter>

Parameter-><Type>' '<Identifier> | <Type> '&'<Identifier>

```

FunctionSection-><Function> | <FunctionSection><Function>
Function-><FunctionDec><CompoundStmt>

CompoundStmt->'{'<StmtList>}'
StmtList-><Statement>';' | <Statement><stmtList>';'

Statement->(<Variable>'='<Expression>) | <ConditionalStmt>
          <LoopStmt> | <BranchStmt> | <Const> |
          <FunctionalStmt>

Expression-> <BoolExp> | <ArithmeticExp> | <FunctionalStmt>
BoolExp-><BoolIden>(<relationOp> | <LogicalOp>)<BoolIden>
BoolIden-><Identifier> | <Const>

ArithmeticExp-><ArithIden><ArithmeticOp><ArithIden> |
              <ArithmeticExp><ArithmeticOp><ArithIden>
ArithIden-><Variable> | <Const>

FunctionalStmt-><FunctionName>'('<Passingvalues>')'
Passingvalues-><PassingVal> | <PassingVal>','<PassingValues>
PassingVal-><Variable> | <Const> | <FunctionStmt>

ConditionalStmt->'if' <BoolExp> (<Statement> | <CompoundStmt>) |
                'else' <BoolExp> (<Statement> | <CompoundStmt>)

LoopStmt->'do'<compoundStmt>'while'<BoolExp> |
          'for'(' (<Variable> | ) ';' (<BoolExp> | ) ';'
              (<Arithmeticexp> | ) ') '<CompoundStmt>'

BranchStmt->'break' | 'return' <functionalExp> |
           'return' <Arithmeticexp> | 'return' <Const>

Library-><IncludeSection><FunctionProtoSection><FunctionalSection>

```

Key

```

| or
'' whatever is in single quotes must be exact text.
| followed by nothing means there is the option for 0 of the
  other options.
() only one of the items in the parenthesis
<> grammer defined term.

```