# Quadsim Version 2.1
# Student Manual

Marwan Shaban
shaban@cs.bu.edu

*February 21, 1993*

**BU-CS Tech Report # 93-003**

Computer Science Department
Boston University
111 Cummington Street
Boston, MA 02215

Abstract:

Quadsim is an intermediate code simulator. It allows you to "run" programs that your compiler generates in intermediate code format. Its user interface is similar to most debuggers in that you can step through your program, instruction by instruction, set breakpoints, examine variable values, and so on.

The intermediate code format used by Quadsim is that described in [Aho 86]. If your compiler generates intermediate code in this format, you will be able to take intermediate-code files generated by your compiler, load them into the simulator, and watch them "run." You are provided with functions that hide the internal representation of intermediate code. You can use these functions within your compiler to generate intermediate code files that can be read by the simulator.

Quadsim was inspired and greatly influenced by [Aho 86]. The material in chapter 8 (Intermediate Code Generation) of [Aho 86] should be considered background material for users of Quadsim.

# Contents

# 1 Getting Started

The first step in using the simulator is generating your intermediate code. The next section describes intermediate code and how to incorporate the intermediate-code I/O functions into your compiler. The following sections describe how to use the simulator to step through your program, what the supported quadruples are, and how to use the sample compiler that accompanies Quadsim.

The simulator executable and the compiler executable can be found in "`~shaban/quadsim/bin`". The intermediate code I/O source files can be found in "`~shaban/quadsim/public`". The test programs can be found in "`~shaban/quadsim/tests`".

# 2 How To Generate Intermediate Code

Intermediate code consists of four parts:

1. Quadruples generated by your compiler. These correspond to machine instructions.
2. Variable locations (and their initial contents) which are generated by your compiler. These correspond to run-time data memory locations. Each of these locations holds either an integer, a real number, or a character which is part of a character string.
3. Line number information that is used by the simulator to show your source code and quadruples in the proper interleaved fashion.
4. Symbolic information that is used by the simulator to refer to variables by their variable names.

## 2.1 Using the Intermediate-Code I/O Interface

The intermediate-code interface functions are described in Appendix B. To use these functions, first initialize the intermediate-code (IC) system by calling ICInitialize(). Next, create a new intermediate code file by calling ICCreate(). Then, write your quads and variables using ICWriteQuad() and ICWriteVar(). Finally, close the intermediate code file by calling ICCloseFile().

Notice that you can also open an existing intermediate code file (using ICOpenFile()) and read its contents (using ICReadFile(), ICReadQuad(), and ICReadVar()). You might want to read an existing intermediate-code file in case you are writing a multi-pass compiler that performs optimization on intermediate code that was generated during an earlier pass.

## 2.2 Writing Quadruples

When calling ICWriteQuad(), you must pass the quadruple that you have already constructed. The structure of a quadruple is as follows:

```
typedef struct {
  int iLocalOrGlobal; /* LOCAL_VARS or GLOBAL_VARS */
  int type;           /* usage depends on the quad */
  int index;          /* " */
  int label;          /* " */
} operand;

typedef struct {
  int label;          /* currently unused */
  int opCode;         /* opcode of the current instruction */
  operand op1;        /* first operand */
  operand op2;        /* second operand */
  operand result;     /* result operand */
} QUAD;
```

- label: Currently unused.
- opCode: Contains the opcode of this quadruple. See Appendix A for information on supported quadruples.
- op1: First operand (usage depends on the opcode). Usually this refers to a memory location.
- op2: Second operand (usage depends on the opcode). Usually this refers to a memory location.
- result: Result operand (usage depends on the opcode). Usually this refers to a memory location.

You must call ICWriteQuad() to write each quadruple that you generate. You may call ICWriteQuad() more than once with the same quadruple location (e.g. when you backpatch a quad that you generated previously).

## 2.3 Writing Variables

When calling ICWriteVar(), you must pass the variable that you have already constructed. The structure of a "variable" is as follows:

```
typedef struct {
  int type;           /* type of the memory cell
                       * (INTEGER_TYPE, REAL_TYPE, or
                       * CHAR_TYPE) */
```

```
  long integer;       /* integer value */
  double real;        /* real value */
} VAR;
```

- type: Type of this variable (INTEGER_TYPE, REAL_TYPE, or CHAR_TYPE).
- integer: The integer value (if this is an integer variable).
- real: The real (floating-point) value (if this is a real variable).

A variable can be initialized by the compiler (by writing the proper value to the "integer" or "real" structure member). This is useful for generating run-time constants or initialized variables.

## 2.4 Writing Symbols

The information in this section is relevant only if you want to propagate symbol names through the intermediate code into the simulator, so you can do "source level" simulation of your compiled program.

When calling ICWriteSymbol(), you must pass the symbol that you have already constructed. The structure of a "symbol" is as follows:

```
typedef struct {
  char rchName [MAX_SYMBOL_NAME];  /* ascii name of symbol */
  int iType;              /* symbol's type */
  int iCardinality;       /* array, single item, etc. */
  int iLocation;          /* memory location of symbol */
  int iArrayLowerBound;   /* if entry is an array */
  int iArrayUpperBound;   /* if entry is an array */
  int iLocalOrGlobal;     /* LOCAL_VARS or GLOBAL_VARS */
} SYMBOL;
```

- rchName: Ascii name of the symbol.
- iType: Type of symbol (INTEGER_TYPE, REAL_TYPE, CHAR_TYPE, PROC_TYPE, or FUNC_TYPE).
- iCardinality: ARRAY_TYPE if array.
- iLocation: Address of starting location for the variable.
- iArrayLowerBound: If array, lower bound.
- iArrayUpperBound: If array, upper bound.

You must call ICWriteSymbol() in the order that your symbols are parsed in the source code. This is because the local variables (and parameters) of a subprogram (procedure or function) are identified as belonging to that subprogram by virtue of their symbols being defined after the definition of the subprogram's name. The subprogram's name is defined by following these two steps:

1. calling ICWriteSymbol() with symbol type "PROC_TYPE" or "FUNC_TYPE".

2. calling ICWriteLocalSymbolInfo() when a new subprogram is about to be parsed (and before the subprogram's name's symbol is written to intermediate code using ICWriteSymbol()). This puts a mark in the intermediate code to show where the new subprogram's symbols begin. Note that ICWriteLocalSymbolInfo() requires one integer argument which is the quadruple position of the start of the subprogram. This position can be obtained by calling ICCurrentQuadPosition() as you are about to parse the subprogram.

So, for the simulator to recognize that symbol "A" is local to a procedure "PROC1", you must call ICWriteSymbol() to define the procedure name "PROC1" before you call ICWriteSymbol() to define the variable "A", and you must call ICWriteLocalSymbolInfo() to indicate the starting quadruple of the procedure.

To define a symbol as local, set the "iLocalOrGlobal" structure element to LOCAL_VARS. To define a symbol as global, set it to GLOBAL_VARS.

For subprogram symbols, "iLocation" must be set to the absolute quad address of the subprogram.

For global variables, the "iLocation" structure element must be set to the absolute variable address of the variable. For local variables or parameters, the "iLocation" element must be set to the variable address of the variable or parameter relative to the top of the subprogram's stack frame. In Quadsim, the stack grows upward, and the top of a subprogram's stack frame is defined as the first variable address above the stack frame. So, for a subprogram with parameters "A" and "B", and local variables "X" and "Y", the relative address of "A" would be 4, and the relative address of "Y" would be 1.

Note that the simulator's symbol handling ability assumes that the language being compiled doesn't allow nested procedures. This is consistent with the language defined by Appendix A of [Aho 86].

## 2.5 Writing Line Numbers

The information in this section is relevant only if you want to propagate line number information through the intermediate code into the simulator, so you can do "source level" simulation of your compiled program.

When calling ICWriteLineNumber(), you must pass the line number structure that you have already constructed. The structure of a “line number” is as follows:

```
typedef struct {
  int iLine;              /* source file line number */
  int iQuad;              /* starting quad for that line */
} LINE_NUMBER;
```

- iLine: Source file line number.
- iQuad: Starting quad for that line.

Each source line that generates one or more quads should cause a call to ICWriteLineNumber() to cause the simulator to properly display it before the quads belong to it within the program's listing (a program listing can be generated within Quadsim using the “List” command).

# 3 Running Quadsim

The Quadsim executable can be found in “`~shaban/quadsim/bin`”. It is called “qsim”. You should add this directory to your search path in order to run Quadsim. Alternately, you can use an alias to map the command “qsim” to include the full path.

The following commands can be typed at the Quadsim prompt:

## 3.1 Help

Gives some online help.

## 3.2 Quit

Terminates the simulator.

## 3.3 List [<loc1> <loc2>]

Lists the program (the quadruples) or a range of them. A specified location can be the number of a quad, or a source file line number preceded by an ‘L’. For example, “list L1 L4” lists the quads corresponding to the first four source lines in the program.

## 3.4 Dump [<loc1> <loc2>]

Lists the variable locations and their contents, or a range of them. A variable location can be specified by the variable's name (if it has a name in the symbol table) or by its location in memory.

### 3.5 Break set <location>

Sets a breakpoint at the specified location. A specified location can be the number of a quad, or a source file line number preceded by an 'L'. For example, "break set L4" sets a breakpoint at line 4 of the source program.

### 3.6 Break clear <location>

Clears the breakpoint at the specified location. A specified location can be the number of a quad, or a source file line number preceded by an 'L'. For example, "break clear L4" clears a breakpoint from line 4 of the source program.

### 3.7 Break List

Lists all breakpoints.

### 3.8 Go

Starts the simulation.

### 3.9 Step

Executes one quadruple, stepping into subroutines.

### 3.10 Next

Executes one quadruple, stepping over subroutines.

### 3.11 Load <file-name>

Loads the intermediate code in the specified file.

### 3.12 Trace [on | off]

Turns trace mode on or off. Trace mode is off by default.

### 3.13 Change [PC | SP] <number>

Changes the value of the program counter or the stack pointer. The stack pointer should be changed with caution since it may affect control flow in subtle ways.

### 3.14 View [PC | SP | Trace]

Shows the value of the program counter, the stack pointer, or the trace setting (or all if no argument is given).

### 3.15 Region [<number>]

Lists the program from quad PC - <number> to quad PC + <number>. If <number> is omitted, it defaults to 4.

### 3.16 Restart

Reloads the intermediate code and resets the instruction pointer.

## 4 The Quadsim Compiler

You are provided with a "correct" compiler which you can use for various troubleshooting purposes.

The compiler can be found in "`~shaban/quadsim/bin`". It is called "qcompile". You should add this directory to your search path in order to run the compiler. Alternately, you can use an alias to map the command "qcompile" to include the full path.

This compiler translates the subset of pascal found in appendix 1 of [Aho 86] with the following exceptions:

1. The grammar presented in appendix 1 of [Aho 86] has a problem in that it does not allow reading from elements of an array. In other words, array references cannot be r-values but they can be l-values, so "ARR[1] = 32" is legal, but "A = ARR[1]" is illegal, according to the textbook's grammar. We fixed it by modifying the grammar. Specifically, the "factor -> id" production was changed to "factor -> variable".

2. The language has been extended to allow simple I/O operations of the following forms:

   ```
   READ (<variable-name>);
   WRITE (<variable-name>);
   WRITE ("literal string");
   ```

   The above read and write operations allow writing and reading of user variables of types integer and real. The literal string may contain escape sequences of the form "\n" to cause a newline character to be embedded in the literal string.

3. The compiler allows use of arrays as local variables of subprograms, but doesn't allow passing arrays as parameters to subprograms.

You can give the compiler the following command-line switches:

- q: This switch produces a listing of the quads generated by the compiler.

- s: This switch produces some scan-phase diagnostic output.
- t: This switch produces a listing of the symbol table.
- p: This switch produces a listing of the parse tree.
- y: This switch turns on yacc diagnostic output.
- h: Prints help information.
- ?: Prints help information.

You may want to use this sample compiler in the following ways:

1. If you need clarification on the correct scan-phase output, parse behavior, quadruple output, or symbol table contents of a particular source program construct, you can compile the program with this compiler and turn on the appropriate switches (from the above list) to study the resulting tables or output. This is particularly useful for clarifying the intended use of certain quad types which may not have been adequately explained in this manual (e.g. to find out how your compiler should use the stack-pointer-manipulation quads, write a simple pascal program that contains a procedure call, and run it through the quadsim compiler, noting the quads generated by the compiler).
2. You can verify that your compiler generates the correct quadruples and symbol table by running your test cases on both your compiler and the quadsim compiler, and compare their outputs.

Your instructor can choose to provide some or all of the source code to the quadsim compiler (for example, part of the compiler's yacc grammar, the symbol table implementation, etc.) to assist you in writing your compiler or provide you with an example implementation of a correct compiler.

The compiler currently has the following minor problems:

1. The compiler's symbol table can't handle overloading of variable identifiers, so if an identifier name is used as a global variable, it can't be re-used as a parameter name or a local variable's name within a subprogram.
2. The compiler's handling of parameter lists to subprograms is limited in that only one list of identifiers can be present. So, the following is legal: "function gcd (a, b : integer)", but the following is illegal: "function gcd (a : integer, b : integer)". The second function prototype above is illegal because it contains more than one identifier list ("a" and "b" are contained in separate identifier lists).

## 5 Sample Programs

Quadsim and the quadsim compiler have been tested using a few programs written in the pascal subset from Appendix 1 of [Aho 86]. You can use these tests to test your compiler's correctness. These sample programs are located in "`~shaban/quadsim/tests`".

The sample programs test most aspects of the language including arrays, functions, procedures, expression evaluation, etc. Note that the Quadsim compiler generates shortcut evaluation for boolean expressions.

## 6 Credits

Nagendra Mishr wrote the original simulator user interface. Sulaiman Mirdad and Liwen Reardon wrote the original intermediate code input/output functions. Marwan Shaban wrote the simulator core.

Marwan Shaban wrote the "example" compiler.

Finally, thanks are due to Professor Wayne Snyder for lending moral and technical support to the simulator project.

# A Quadruple Reference

The following are the currently supported quadruples. Most of these are direct implementations of the suggested three-address statements on page 467 of [Aho 86]. The material in chapter 8 (Intermediate Code Generation) of [Aho 86] should be considered background material for users of Quadsim.

## A.1 Two-operand Instructions

These are assignments of the form x := y *op* z, where *op* is a binary arithmetic or logical operation. To use these quads, set the following quad structure elements:

- opCode: one of the following opcodes:
  - ADD: Addition.
  - SUB: Subtraction.
  - MULT: Multiplication.
  - DIV: Division.
  - MOD: Remainder.
  - OR: Logical "or".
  - AND: Logical "and".
  - INT_EQ: Integer equality.
  - INT_NE: Integer non-equality.
  - INT_LT: Integer less-than.
  - INT_LE: Integer less-than or equal.
  - INT_GT: Integer greater-than.
  - INT_GE: integer greater-than-or-equal.
- op1.index: address of the first parameter.
- op1.iLocalOrGlobal: tells whether the first parameter address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).
- op2.index: address of the second parameter.
- op2.iLocalOrGlobal: tells whether the second parameter address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).
- result.index: address of the result.
- result.iLocalOrGlobal: tells whether the result address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).

## A.2 One-operand Instructions

These are assignment instructions of the form x := *op* y, where *op* is a unary operation.

- opCode: one of the following opcodes:
  - NOT: Logical Negation.
  - UNARY_MINUS: Unary minus operation.
  - SHIFT_LEFT: Left shift by one bit.
  - SHIFT_RIGHT: Right shift by one bit.
  - INT_TO_FLOAT: Integer to real number conversion.
  - FLOAT_TO_INT: Real number to integer conversion.
- op1.index: address of the first (and only) parameter.
- op1.iLocalOrGlobal: tells whether the first parameter address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).
- result.index: address of the result.
- result.iLocalOrGlobal: tells whether the result address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).

## A.3 Assignment Instruction

Assignment of the form x := y. The opcode for this instruction is ASSIGN.

- op1.index: address of the source.
- op1.iLocalOrGlobal: tells whether the source address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).
- result.index: address of the result.
- result.iLocalOrGlobal: tells whether the result address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).

## A.4 Unconditional Jump

The opcode for this instruction is JMP.

- result.label: quad address to jump to.

## A.5 Conditional Jumps

These are of the form "if x *relop* y goto L". The opcodes for these instructions are:

- opCode: one of the following opcodes:
  - EQ: Jump if equal.
  - NE: Jump if not equal.
  - LT: Jump if less than.
  - LE: Jump if less than or equal.
  - GT: Jump if greater than.
  - GE: Jump if greater than or equal.
- op1.index: address of the first parameter.
- op1.iLocalOrGlobal: tells whether the first parameter's address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).
- op2.index: address of the second parameter.
- op2.iLocalOrGlobal: tells whether the second parameter's address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).
- result.label: address to jump to.

## A.6 Function Calling Instructions

These are quadruples that assist in calling procedures and functions, and passing them parameters. The reference ([Aho 86]) leaves a lot of these details up to the implementation, so some of this information cannot be found in that reference.

The opcodes for these instructions are:

- PARAM: Push a parameter onto the stack. Each parameter pushed occupies one memory location. When the function returns, a POP_STACK quad must be invoked by either the caller or callee to adjust the stack pointer to the point it was at before parameters were pushed.
  - op1.index: address of the parameter to be pushed.
  - op1.iLocalOrGlobal: tells whether the parameter's address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).
- CALL: Call a function or procedure. Saving the return address is not necessary. It is done internally by the simulator. The return address is retrieved by the RETURN command.

- result.label: quad address of the procedure or function being called.

- RETURN: Return from a function or procedure.
- PUSH_STACK: Increment the stack pointer so that a new space on the stack is created for a local variable.
  - op1.index: type of the parameter being pushed.
- POP_STACK: Decrement the stack pointer to pop a stack frame and restore a previous stack environment.
  - op1.index: number of parameters and local variables to pop from the stack. The stack pointer is decremented by this amount. Note that the simulated stack grows upward.
- PUT_FUNCTION_RESULT: Writes the result of a function to a special register.
  - op1.index: address of the result to be written.
  - op1.iLocalOrGlobal: tells whether the address in op1.index is a global or local address (GLOBAL_VARS or LOCAL_VARS).
- GET_FUNCTION_RESULT: Reads the contents of the special register containing the result of a function.
  - op1.index: address of the location to receive the result.
  - op1.iLocalOrGlobal: tells whether the address in op1.index is a global or local address (GLOBAL_VARS or LOCAL_VARS).

## A.7 Indexed Assignments

The opcodes for these are:

- READ_FROM_ARRAY: This is an assignment of the form x := y[i]. "y" is in op1, and "i" is in op2.
  - op1.index: address of the first element of the array.
  - op1.iLocalOrGlobal: tells whether op1.index refers to a local or global array (LOCAL_VARS or GLOBAL_VARS).
  - op2.index: address of a variable which holds the index into the array.
  - op2.iLocalOrGlobal: tells whether op2.index refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).
  - result.index: address of the result.
  - result.iLocalOrGlobal: tells whether the result's address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).

- WRITE_TO_ARRAY: This is an assignment of the form x[i] := y. "i" is in op1, and "y" is in op2.
  - op1.index: address of the source variable.
  - op1.iLocalOrGlobal: tells whether op1.index refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).
  - result.index: address of the first element of the array.
  - result.iLocalOrGlobal: tells whether result.index refers to a local or global array (LOCAL_VARS or GLOBAL_VARS).
  - op2.index: address of a variable which holds the index into the array.
  - op2.iLocalOrGlobal: tells whether op2.index refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).

## A.8 Address and Pointer Assignments

The opcodes for these are:

- READ_DATA_ADDRESS: This is an assignment of the form x := &y.
  - op1.index: value of the address to be written to the destination.
  - result.index: address of the destination.
  - result.iLocalOrGlobal: tells whether the destination's address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).
- READ_DATA_INDIRECT: This is an assignment of the form x := *y.
  - op1.index: address of the source parameter (y).
  - op1.iLocalOrGlobal: tells whether the source parameter's address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).
  - result.index: address of the result (x).
  - result.iLocalOrGlobal: tells whether the result's address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).
- WRITE_DATA_INDIRECT: This is an assignment of the form *x := y.
  - op1.index: address of the source parameter (y).
  - op1.iLocalOrGlobal: tells whether the source parameter's address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).
  - result.index: address of the result's address (x).
  - result.iLocalOrGlobal: tells whether the (x)'s address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).

## A.9 Input/Output Instructions

These allow machine-level I/O into and out of memory locations.

- READ: Performs user input into a memory location(s). The "op1.index" structure element of the quadruple decides the type of the operation:
  - STRING_IO: Reads a string of ascii characters into successive memory locations.
    - result.index: address of the location to write the string into.
    - result.iLocalOrGlobal: tells whether the above address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).
  - INTEGER_IO: Reads an integer.
    - result.index: address of the location to write the integer into.
    - result.iLocalOrGlobal: tells whether the above address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).
  - REAL_IO: Reads a floating-point number.
    - result.index: address of the location to write the real number into.
    - result.iLocalOrGlobal: tells whether the above address refers to a local or global variable (LOCAL_VARS or GLOBAL_VARS).
- WRITE: Performs output from a memory location(s). The "op1.index" structure element determines the type of the data written (see above), the "result.index" structure element determines the location read from, and the "result.iLocalOrGlobal" element determines whether the above address refers to a local or global variable (as in the READ operation).

## A.10 Miscellaneous Instructions

- HALT: Tells the simulator that the end of the program has been reached. A halt instruction must be placed at the end of the program.
- NOP: The simulator performs no operation.

## B Intermediate-Code I/O Interface

You are provided with the source code to the following functions, which generate intermediate-code files suitable for reading by the simulator. The main functions here are ICWriteQuad(), ICWriteVar(), ICWriteSymbol(), and ICWriteLineNumber(). These functions write a quadruple to the intermediate-code file, write a new variable to the intermediate-code file, write a new symbol to the intermediate-code file, and write a new line number record to the intermediate-code file, respectively. These functions are located in the file "ic.c". The structure definitions are located in "simulator.h". You can copy these files from the directory "`~shaban/quadsim/public`".

The possible return codes for the below functions are SUCCESS, FAILURE, or OUT_OF_MEMORY.

```
#include "simulator.h"

------------------------------------------------------------

int ICInitialize ();

Initializes the Intermediate Code File Manager.  Must be
called before any other IC function.

------------------------------------------------------------

int ICOpenFile (char * fileName, quadLoc ** fileHandle);

Opens an existing intermediate code file.  The second
parameter returns a pointer to the stream associated
with the file, which will serve as the handle to that
file.

------------------------------------------------------------

int ICCreate (char * fileName, quadLoc ** fileHandle);

Creates a new intermediate code file.  The second
parameter returns a pointer to the stream associated
with the file, which will serve as the handle to that file.

------------------------------------------------------------
```

```
int ICReadFile (quadLoc * fileHandle);

Reads the contents of an intermediate code file.  This
should be done after calling ICOpenFile() to cause the
contents of the file to be read into memory.

------------------------------------------------------------

int ICCloseFile (quadLoc * fileHandle);

Closes an intermediate code file.  This should be done
after a file is no longer needed.

------------------------------------------------------------

int ICSwitchStream (quadLoc * fileHandle);

Switches the current file to the file with the specified
stream.  The functions ICCurrentQuadPosition(),
ICWriteQuad(), ICWriteVar(), ICReadQuad(), and ICReadVar()
always operate on the current file.  The functions
ICOpenFile() and ICCreateFile() cause the current stream
to be the file opened or created.

------------------------------------------------------------

int ICCurrentQuadPosition (int * currentQuadPosition);

Returns the sequential number of the current quad in the
current file. This position may be later used in ICReadQuad()
and ICWriteQuad() calls to facilitate backpatching.

------------------------------------------------------------

int ICCurrentVarPosition (int * currentVarPosition);

Returns the sequential number of the current variable in
the current file.  This position may be later used in
ICReadVar() and ICWriteVar() calls to facilitate backpatching.

------------------------------------------------------------
```

```
int ICWriteQuad (int position, QUAD * quad);

Writes a quad to the current file.  If the 'position'
parameter is AT_END_OF_FILE, the quad is written after
all existing quads.

------------------------------------------------------------

int ICWriteVar (int position, VAR * var);

Writes a variable to the current file.  If the 'position'
parameter is END_OF_FILE, the variable is written after all
existing variables.

------------------------------------------------------------

int ICReadQuad (int position, QUAD * quad);

Reads a quad from the current file.  If the 'position'
parameter is NEXT_ENTRY, the next sequential quad is read
from the file.

------------------------------------------------------------

int ICReadVar (int position, VAR * var);

Reads a variable from the current file.  If the 'position'
parameter is NEXT_ENTRY, the next sequential variable is
read from the file.

------------------------------------------------------------

int ICWriteLineNumber (LINE_NUMBER * psLineNumber);

Adds a line number record to the intermediate code file.

------------------------------------------------------------
```

```
int ICGetLineNumbers (LINE_NUMBER ** ppsLineNumber,
      int * piNumLineNumbers);

Returns a pointer (in the first parrameter) to the array
of line number records in the intermediate code file.  The
second parameter returns the number of elements in the
array.

------------------------------------------------------------

int ICWriteSymbol (SYMBOL * psSymbol);

Adds a symbol record to the intermediate code file.

------------------------------------------------------------

int ICGetSymbols (SYMBOL ** ppsSymbols, int * piNumSymbols);

Returns a pointer (in the first parrameter) to the array
of symbol records in the intermediate code file.  The second
parameter returns the number of elements in the array.

------------------------------------------------------------

int ICWriteLocalSymbolInfo (int iLocation);

Puts a mark in the intermediate code which indicates the
start of a new subprogram, and the quad location at which
the subprogram begins.  This allows the simulator to know
(when a CALL quad is executed) which set of local symbols
to refer to when responding to ``dump'' commands.

------------------------------------------------------------
```

# References


[Aho 86] Aho, Alfred V, Ravi Sethi, and Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools*; Reading, Addison-Wesley Publishing Company, 1986.