WHIMSICAL

Release 1.8

J. R. SPRAY

CONTENTS

DISCLAIMER

Whimsical Developments makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Whimsical Developments reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Whimsical Developments to notify any person of such revision or changes.

WHIMSICAL OBJECTIVES

Writing software for a microprocessor has become a costly and time consuming exercise which can be out of proportion to that for the hardware development. High level languages have a big advantage over assembly language because they allow the programmer to concentrate on his problem without also having to be distracted by all the fine details of its implementation. Programs written in a high level language are also far easier to check and are more readily maintained by persons other than the original author. The advantages of modern block structured high level languages are so considerable that many feel the inferior speed of these languages should not be overcome by going back to assembly language, but by using a faster, better microprocessor. The 6809 is that microprocessor. When Motorola designed it, they incorporated many features which would enable it to efficiently support block structured languages.

Whimsical is a "new" language in that it does not strictly conform to the syntax of any other single language. Rather it combines many excellent constructs of other languages, such as modules, with aspects which particularly suit the 6809 and a few new features such as the unified Input/Output statements and special types for hex. The three lex levels of Whimsical are chosen to be efficiently supported by the 6809's complement of index registers. Also 8 bit arithmetic and fast unsigned multiply are supported directly in the language because the processor supports these operations with fast instructions.

One of the most important design philosophies with Whimsical is that although it is a neat and concise high level language, it does not hide the programmer completely from the machine. Where other languages may try to do the right thing most of the time, and offer overrides or exceptions in the cases where the feature is not required, Whimsical takes the simple approach and the programmer is required to explicitly code his intentions. For example, in a WRITE statement, most languages output a carriage return, line feed automatically and spaces either side of numbers. You have to know all these things. In Whimsical you remember one thing; what you code is what you get. You have to output every space and every line feed yourself. You don't have to remember what type mixing is allowed and what automatic type conversions take place in Whimsical because no type mixing is allowed and type conversions must always be explicitly programmed.

There is no complicated runtime system involved with Whimsical. It produces a simple self contained executable file. If the target machine has a foreign environment, then Whimsical has the versatility to produce programs to run in it. The I/O may be redefined so that the READ and WRITE statements are still useful. Variables may be declared at a specified address and interrupt procedures may be written. Runtime errors may be trapped and handled by user written routines. In addition the code is ROMable, reentrant and relocatable.

Whimsical is a recursive descent compiler which requires just one pass of the source code, and it produces fast native 6809 code directly. This enables a very fast turn-around time in the edit-compile-test cycle.

WHIMSICAL BACKGROUND

The need for a language like Whimsical for the 6809 was realized soon after the chip became available. It was obvious that the designers had given much thought to software and had done their part by providing us with a machine that had all the stack manipulation, addressing modes and registers that were needed.

The possibility of writing all code without significant use of assembly language was potentially a great advantage to the 6809 processor. So Whimsical was born and in keeping with the whole philosophy of the language, the compiler itself could be written in Whimsical.

The project was started at the University of Auckland on a Burroughs B6700 computer. Initially the language contained only sufficient features to allow the compiler itself to be written in Whimsical. In order to compile the compiler a second version was written in Algol. Since Whimsical and Algol are very much alike this translation process was not difficult. Before Whimsical was successfully compiling itself, circumstances forced the project to be moved to a SWTPc computer running the FLEX operating system. Pascal replaced Algol as the boot-strapping language and the project proceeded slowly on a part time basis.

Eventually Whimsical successfully compiled itself and the first milestone was reached. Being independant of Pascal speeded up the development process considerably. For example, the six pass Pascal compiler took 45 minutes to compile the Pascal version. The native code Pascal version then took 25 minutes to compile the Whimsical version. Whimsical then took just 5 minutes to compile itself! Even on the B6700 at the university, the Algol version used to take 3 minutes to compile Whimsical.

The project has continued, each version of Whimsical being used to compile the next slightly better version. At each stage the compiler must be able to produce a code file identical to itself before the next stage is begun.

Initially, the features essential to the easy use of the compiler were added. This included file I/O and command line processing and the like. Then some optimization was added to reduce the size of compiled programs. In particular the compiler itself decreased in size from about 20K to 16K. Then many more features and optimizations were added increasing its size again to 32K. Meanwhile the size of the source file had increased from about 2000 lines to over 6000 lines.

WHIMSICAL FEATURES

* Fast single pass compiling.
  No separate assembly or linkage stage.
  Produces fast, efficient native 6809 machine code.
* No separate runtime package.
  Object file is completely self contained.
* Object program can run under any environment.
  Ideal for embedded computer systems.
* Code is ROMable, relocatable and reentrant.
* Modern Block structured style as in PASCAL, "C" or ADA.
  Structured statements e.g. WHILE DO ; IF THEN etc.
* Module structure.
  May be pre-compiled and then linked into main program.
* Heavily typed with full type checking at compile time.
* Primitive data support - bytes and double bytes.
  These types are compatible with hexadecimal constants.
* Integer sizes of 8, 16 and 32 bits.
  These types are compatible with decimal constants.
* Real number type.
* Character and boolean types.
* Arrays may be any size.
  Constant arrays.
* Fully recursive procedures.
  Parameter passing by value or reference.
* Unified Input/Output statements.
  User definable Input/Output.
* User definable error handlers and error traps.
* Versatile interfacing to assembly language routines.
  Ability to call external routines.
* Interrupt handling.
* Disk Input/Output support.
* AND, OR, XOR etc for BYTES as well as BOOLEANS.
* Constant expressions evaluated at compile time.
* Any length variable names.
* Three lex levels
  Procedures can be declared inside procedures, to one level.
  Procedures not limited in size.
* Include source file directive.
  Page formatting control.
  Version number facility.
* Directives to set up stack pointer, code origins etc.
  An alternative data segment can be set up.
* Declaration time initialization of variables.
  Variables automatically initialized to zero by default.
* Conditional compiling.
* Global symbol table and cross reference generation.

This section introduces the basics of the language WHIMSICAL. If you already know a block structured language such as PASCAL or "C" you will find Whimsical very easy to learn and you may skip most of this section. If you don't know a block structured language, then this section will help you to learn the concepts. Knowing Whimsical will be a great asset if in future you have to use another block structured language.

A simple Whimsical program can be very simple indeed. Here is a very small but complete program.

```
BEGIN
  WRITE "^M^JHELLO"
END.
```

It consists of a block, followed by a full stop. A block is made up of a BEGIN - END pair which contains a set of declarations and a set of statements. In the above example there are no declarations, and there is just one statement - the WRITE statement. Every program must have a final full stop which functions to make sure that when the compiler finishes, that it has in fact got to the end of the program. The program writes a carriage return, linefeed and the word "HELLO" to the system's terminal. Note the method of getting control characters into a string. The "^" (carat) symbol causes the next character to become a control character.

As mentioned before, a block consists of a set of declarations and a set of statements. The statements describe the actions of the program. The declarations describe the objects of those actions. One common declaration is to declare a variable for use in a program. The following small program gives an example of such a declaration, and some more statements.

```
BEGIN
  BYTE I;
  FOR I:=$20 TO $7E DO
    WRITE "^M^J  $",I,"     ",DEC(I),"       ",CHR(I);
END.
```

The program writes a table of ASCII codes in hexadecimal and decimal. The variable I is made to successively take on all the values from $20 (hex 20) to $7E. For each value of I, a carriage return, line feed and some spaces are output followed by the value of I in hexadecimal, decimal and its ASCII character.

Declarations can consist of more than just variable declarations. Another example program is given now to illustrate what a larger Whimsical program looks like. It contains some procedure declarations. Try to read through it and understand how it works.

```
% THIS IS A SIMPLE MEMORY EXAMINE/CHANGE PROGRAM

BEGIN
  CHAR COMMAND,CH;
  BYTE CONTENTS;
  DBYTE ADDRESS;
  BYTE ARRAY MEMORY($0000);  %SPECIAL FIXED LOCATION ARRAY

  BOOLEAN PROCEDURE HEXDIGIT(CHAR CH)=
% ------- --------- --------
% PROCEDURE TO CHECK IF A CHARACTER IS A HEXADECIMAL DIGIT
  BEGIN
    HEXDIGIT:=CH>='0' AND CH<='9' OR CH>='A' AND CH<='F';
  END;

  BYTE PROCEDURE ASCIITOHEX(CHAR HEXCH)=
% ---- --------- ----------
% PROCEDURE TO CONVERT A HEXADECIMAL DIGIT TO A BINARY NIBBLE
  BEGIN
    ASCIITOHEX:=IF HEXCH<='9' THEN ASC(HEXCH)-$30
                              ELSE ASC(HEXCH)-$37;
  END;

  % MAIN PROGRAM STARTS HERE
  DO BEGIN
    % WRITE CARRIAGE RETURN, LINE FEED AND PROMPT
    WRITE CHR($0D),CHR($0A),'*';
    READ COMMAND;      % GET COMMAND FROM KEYBOARD
    CASE COMMAND OF
    BEGIN
      'M': WRITE ' ';
           READ ADDRESS;
           DO BEGIN
             CONTENTS:=MEMORY[ADDRESS];
             WRITE '^M^J',ADDRESS,' ',CONTENTS,' ';
             READ CH;
             IF HEXDIGIT(CH) THEN
             BEGIN
               CONTENTS:=ASL(ASL(ASL(ASL(ASCIITOHEX(CH)))));
               READ CH;
               IF HEXDIGIT(CH) THEN
               BEGIN
                 CONTENTS:=CONTENTS+ASCIITOHEX(CH);
                 MEMORY[ADDRESS]:=CONTENTS;
                 IF MEMORY[ADDRESS]=CONTENTS THEN CH:=' ';
               END;
             END;
             IF CH=' ' THEN ADDRESS:=ADDRESS+$0001;
             IF CH='^' THEN ADDRESS:=ADDRESS-$0001;
           END UNTIL CH=CHR($0D);
      'E': ; % DO NOTHING
      ELSE: WRITE ' Invalid Command';
    END;
  END UNTIL COMMAND='E';
END.
```

The following commentary refers to the program on the previous page. The first line is a comment because it begins with a percent sign. The first BEGIN starts the main block which ends at the final END. The variable declarations inside the main block are known as globals. In this program we start by declaring two global variables of type CHAR. COMMAND will be used to hold a single character command read from the keyboard. CH will be used to hold data characters read in from the keyboard. Next we declare a variable of type BYTE called CONTENTS which will be used to hold the contents of a memory location. Then a variable of type DBYTE is declared which will hold the ADDRESS of the memory location. Finally, so we can access absolute memory locations, there is a special BYTE type array called MEMORY. This array is fixed to start at address $0000 and gives us access to the entire 64K.

Following the variable declarations come the procedure declarations. There are two in this example called "HEXDIGIT" and "ASCIITOHEX". These are examples of typed procedures because they return a value. HEXDIGIT will be called later to determine if a given character is a hexadecimal digit. It returns a boolean result, TRUE or FALSE. The body of the procedure is another BLOCK. This time there are no variable declarations. If there were they would be called locals as opposed to globals because they would be local to this procedure. Inside the block is a single statement which is quite a complex assignment statement. It assigns to the variable HEXDIGIT the value of a boolean expression. The assignment operator (:=) is common to many languages and is less confusing than an equals sign as used in BASIC. The equals sign is reserved for comparing one expression with another and returning a TRUE or FALSE result. A similar comparison is done in the expression here four times. The parameter, CH, which is passed to the procedure when it is called, is compared to see if it is greater than or equal to '0', less than or equal to '9', greater than or equal to 'A' and less than or equal to 'F'. Each comparison gives a boolean result, TRUE or FALSE. These boolean results are then combined with the logical operators AND and OR. Since AND has a higher precedence than OR the two AND's are done first and their results are ORed together.

The second procedure called ASCIITOHEX accepts a value of type CHAR and converts it to a hex nibble which is then returned in the lower half of a BYTE. Again there is just one statement within the block and again it is an assignment statement. The expression this time contains a conditional IF clause. This selects one of the two expressions ASC(HEXCH)-$30 or ASC(HEXCH)-$37 according to the result of the initial boolean expression HEXCH<='9'. So essentially if the character is a digit we subtract $30 and if it is a letter we subtract $37. This produces the desired single hex nibble result according to the ASCII codes for the letters and digits.

Notice that all the declarations whether they are variable declarations or procedure declarations are separated by semicolons.

Next there is a comment telling us that the main program begins and the declarations have finished. The first statement is a "DO" statement. It is an example of a structured statement because it contains another statement. That statement is a compound statement which is many statements bracketed by a BEGIN - END pair. The

BEGIN goes after the word DO but the matching END is down at the bottom before the words UNTIL COMMAND='E'; which form the end of the DO statement. In Whimsical wherever you see one statement, you may replace it with many statements bracketed with a BEGIN END pair. Furthermore the nesting of such statements has no limit. The DO statement causes the statement within it to be executed repeatedly. It doesn't stop until the boolean expression after the final UNTIL becomes TRUE.

The first statement inside the DO loop is a WRITE statement. It outputs a carriage return, linefeed and prompt character to the screen. Here we see an alternative method of outputting a carriage return and line feed. The next statement waits for and reads a single character from the keyboard. This character is stored into the variable COMMAND. The CASE statement is then used to select the appropriate response to the command entered. It starts at the line CASE COMMAND OF and finishes at the END 3 lines from the bottom. In between, the two allowable commands, "M" or "E", are listed followed by a colon and the statements to be executed in each case. Notice that if COMMAND contains "E" then no action is taken in the CASE statement because the "E" is followed by nothing but a semicolon. With this case statement it is very easy to add further commands. If an invalid command is entered then the statements following the ELSE: are executed which in this case outputs an appropriate error message.

If an M is typed as the command then the statement, WRITE ' ', outputs one space and then we READ in from the keyboard a DBYTE value. It's a DBYTE value because the type of the variable being read, ADDRESS, is a DBYTE. Up to four hexadecimal digits may be entered followed by a non-hex character such as a carriage return. The 16 bit unsigned value read is stored into ADDRESS. Following the reading of the address we enter the memory examine/change mode. To get out of the mode you enter a carriage return. Here a second DO statement is used which loops UNTIL CH=CHR($0D). Inside the loop we get the contents of memory at the current address and store it into the variable CONTENTS. Next a carriage return, line feed is output followed by the current address, a space, the value of CONTENTS and another space. Both ADDRESS and CONTENTS are output in hexadecimal because they are types DBYTE and BYTE. Next a character is read from the keyboard into the variable CH. To test if the character entered is a hexadecimal digit the procedure HEXDIGIT is called. CH is passed to the procedure and it returns a BOOLEAN value. If the value returned is TRUE, i.e. it is a hexadecimal digit, then we want to get another digit. The IF statement is used here to conditionally execute the contained statement. The contained statement is again a compound statement because it starts with a BEGIN and finishes at the matching END which is directly below the BEGIN. The first statement within the compound statement uses the procedure ASCIITOHEX to change the ASCII hex digit CH into a hex nibble. This nibble is then shifted left four times to put it into the high nibble and the result is assigned to the variable CONTENTS. Then another character is read from the keyboard by the READ CH statement. If this character is also a hex digit, tested by another IF statement, then it too is converted to a hex nibble and added into CONTENTS to form a new BYTE value. This value is put back into memory at the current address. Then the memory is read back and compared with what was stored. If it is the same then a space is stored into CH to cause the current address to go onto the next address later. The two IF

statements end on the two END's one after the other.

Next there is another IF statement which this time contains only a simple statement rather than a compound statement. It causes ADDRESS to be incremented if a space has been entered. The next IF statement causes the current address to be backed up if an up-arrow was entered.

Notice where in the program we have finished up reading it. All the lines following the statement, IF CH='^' THEN, we have already discussed because they belong to statements which began much earlier in the program. It is very important when reading a structured language to match the beginning of each statement with its ending even if it is pages away. Correct indenting is therefore essential.

USING THE COMPILER (UNDER FLEX)

Before using your Whimsical disk make a backup of it and then copy the two files WHIM.CMD and WHIM.ERR onto your system disk. WHIM.CMD is quite large and so if it does not fit onto your system disk you may wish to make a special system disk for use with Whimsical.

Whimsical source programs are prepared by the use of any text editor. They are then compiled to produce an object file and optionally an error file and listing. The compilation is started by typing the name of the compiler followed by the name of the source file.

e.g. WHIM SOURCE<CR>

The compiler (WHIM.CMD) will be loaded by FLEX and begin executing. It will look for a source file called "SOURCE.TXT" and produce an object file called "SOURCE.CMD". Both source and object files default to the working drive.

NOTE: WHIM requires your system to have 48K of RAM from $0000 to $BFFF. It also uses the utility command space.

The name of the object file may be given along with several options.

```
WHIM SOURCEFILE [,OBJECTFILE] [+OPTIONS]
```

The possible options are:

```
L  Generate a full listing.
-L Suppress listing.
E  Produce an error file.
B  Suppress production of a binary file.
N  No stop on error.
R  Runtime checking on.
-R Runtime checking off.
S  Saturation of arithmetic on.
-S Saturation of arithmetic off.
I  Information on stack usage, code addresses, nesting levels,
   unused identifiers, subroutine usage (see section 3-2).
P  Page the output.
T  Trace procedure calls on.
-T Trace procedure calls off.
W  Search for subroutines on disk first (see section 3-6).
X  Make cross reference (see section 3-7).
G  Make a global symbol table (see section 3-7).
```

Most of these options can be specified within the source of your program. A directive "~OPTION" is privided for this. (see section 2-10).

The source filename's extension defaults to ".TXT". The default name for the object file is the source filename with an extension of ".CMD". If the object filename is given, however, then the default extension will be ".BIN". If an error file is to be produced its name will be the object file name with an extension

of ".ERR". This error file is identical to what is listed to the terminal. If a full listing was requested by the L option then the error file will contain a full listing also.

A printout of the listing can be made either by spooling the error file or by using the FLEX "P" command during compilation.
Here are some examples:

WHIM PROG	Creates "PROG.CMD" from "PROG.TXT". Only errors are reported to the terminal.
WHIM PROG +L	Creates a full listing to the terminal.
P WHIM PROG +L	Creates a full listing to the printer.
P WHIM PROG +LP	Creates a full listing to the printer with paging.
WHIM PROG.SRC PROG2 +E	Creates "PROG2.BIN" and an error file containing only errors called "PROG2.ERR"
WHIM PROG3 +ELP	This time the file "PROG3.ERR" contains a full paged listing suitable for spooling.

OUTPUT FORMATTING

The listing generated by the compiler can be formatted in pages by using the "P" option in the command line. The default number of lines per page is 60. The lines per page can be specified immediately following the 'P' option. The page width may be specified following a comma. A margin may also be specified. The width will include the margin. Note that to get a full listing (not just the lines containing errors) you must also use the L option). Here are a few examples:

P WHIM SOURCE +PL	Specifies full paged listing
P WHIM SOURCE +P62L	Specifies 62 lines/page
P WHIM SOURCE +P62,132L	Specifies page width is 132 cols
P WHIM SOURCE +P62,80,4L	Specifies width 80 and margin 4
P WHIM SOURCE +P,80L	Specifies width 80 only
P WHIM SOURCE +P,,4	Specifies margin 4 only

The main function of the width parameter is to allow the compiler to keep the line count correct when lines overflow. So it is still necessary even if your printer automatically does a CR, LF when a line is full.

STOPPING THE COMPILER.

If the compiler is outputting a listing it may be stopped with ESCAPE as with normal FLEX convention. To allow the compiler to be stopped when no listing is occurring, it continually checks the FLEX character input status. If a key is seen to be available the compiler outputs the current line to allow ESCAPE to work. Once the compiler is stopped with ESCAPE, type another ESCAPE to continue or a RETURN to abort. Typing spaces during compiling is a useful way to have lines printed to see how far it has gotten.

TRACE PROCEDURE CALLS

If a T option is specified on the command line, the compiled program will contain code at the beginning of each procedure to output the procedure's name. Alternatively tracing can be turned on and off for selected portions of your program with the ~OPTION directive. (see section 2-10).

```
~OPTION T            turn on tracing
~OPTION /T           revert to previous state
```

If your program contains such options then any command line option of T or -T will override them to either globally turn them on or off respectively.

COMPILING MULTIPLE PROGRAMS

If you are compiling more than one program or a series of modules followed by the main program then it is wasteful to reload WHIM.CMD each time. If WHIM.CMD is already in memory then to execute it again use a JUMP 0 instruction or equivalent followed by the normal command line parameters you would use. e.g.

```
WHIM MODULE1:JUMP 0 MODULE2:JUMP 0 MAINPROG
```

GENERAL SYNTAX RULES

Whimsical has the same general syntax rules as many other high level languages. There are several basic elements. Reserved words are words which have a special meaning in the language. They act as keywords in forming standard structures. For example, in the statement

```
IF DEG>360 THEN DEG:=DEG MOD 360
```

the words "IF", "THEN", and "MOD" are reserved words. The word "DEG" is an identifier, a user defined word. In this case it stands for an integer variable. The reserved words cannot be used as identifiers. Here is a list of all the reserved words.

BEGIN	END	IF	THEN	ELSE
WHILE	DO	UNTIL	CASE	STOP
FOR	DOWNTO	STEP	TRAP	
AND	OR	XOR	NOT	MOD
BYTE	DBYTE	BOOLEAN	BOOL	CHAR
INTEGER	SMALLINT	LARGEINT	POINTER	
FILE	ARRAY	REF	REAL	PROC
PROCEDURE	EXTERNAL	INTERRUPT	FORWARD	CODE
TRUE	FALSE	TO	FROM	AS
READ	WRITE	CREATE	OPEN	CLOSE
CLOSEDELETE				

In addition there are a number of words which are predefined identifiers. These are called intrinsics. The intrinsics may not be redefined either, making the following identifiers unavailable.

ENABLEIRQ	DISABLEIRQ	ENABLEFIRQ	DISABLEFIRQ	
HIBYTE	LOBYTE	COMBINE		
ASL	ASR			
CHR	ASC	HEX	DEC	
HIDBYTE	LODBYTE	LCOMBINE	RCOMBINE	
EXTEND	TRIM	FIX	INT	FLOAT
EOF	LOC	REPORTERROR		

There are a number of special symbols in Whimsical. These are mainly single characters but there are a few double character symbols. The special symbols are:

```
+    -    *    /
=    <    >    <=   >=   <>
,    .    ;    :    :=
"    '    %    ~    |    space
(    )    [    ]    {    }
```

Identifiers must begin with a letter and contain only letters, digits or underbars. There is no limit on the length of an identifier except that it must be completely contained on one input source line. Upper and lower case letters are not distinguished.

Numbers in Whimsical may be in decimal or hexadecimal format. If hexadecimal the number must start with a dollar sign, "$".

Hexadecimal numbers must consist only of the digits 0 to 9 and the letters A to F or a to f.

The braces, "{" and "}", may be used as synonyms for the reserved words BEGIN and END. Also, the special symbol "|" or vertical bar may be used instead of a space. The invisibility of the space can introduce difficulties in indenting text, particularly BEGIN and END markers on a standard terminal. It is suggested that the "|" be placed in a vertical column underneath the B of the BEGIN to maintain the alignment to its matching END.

Where there are two or more consecutive word and number symbols they must be separated by at least one space or a carriage return. Otherwise spaces and carriage returns hold no significance and can be used as required for the program's readability. The maximum length of an input line is however, 255 characters. Comments begin with a percent sign (%) and occupy the rest of the line they are on. Compiler directives are exceptions to the above rules. A directive must begin on a new line and takes up the whole line. All directives start with a tilde (~), not necessarily in column one, followed by a directive keyword and any parameters. The rest of the line is then ignored e.g.

```
      ~STACK=$C000    DIRECTIVE TO SET UP STACK POINTER
```

WHIMSICAL TYPES

Whimsical is a heavily typed language because this helps the programmer to write correct programs without causing any inefficiency in the code. There is a large selection of primitive types which may at first seem bewildering. The programmer will soon find that there is usually one correct type to use in a situation which is both logical and the most efficient for the job. Furthermore, the strict type mixing rules in Whimsical allow a lot more compile time checking to be done.

There are four distinct types which require one byte of storage. They are BYTE, SMALLINT, CHAR and BOOLEAN. There are two distinct types which require two bytes of storage. They are DBYTE and INTEGER. The types LARGEINT and REAL each require four bytes of storage. BYTES and DBYTES are general purpose data types. They are used to represent any kind of data. BYTES and DBYTES always use hexadecimal format for input, output and literal constants. All the other types are special purpose. Integers are used for representing decimal numbers (in 2's complement form). They are fully supported with arithmetic operations and have features such as overflow checking and saturation. Their input, output and literal constants are always in decimal format. CHARs are used for representing ASCII characters. They are relatively restrictive in the operations which can be done on them. BOOLEANS are logical data types which can have the values TRUE or FALSE. They are supported with all the logical operators: AND, OR etc. The following table summarizes the ranges and uses of the various types.

```
BYTE      unsigned 8 bits ($00..$FF)
DBYTE     unsigned 16 bits ($0000..$FFFF)
SMALLINT  2's complement 8 bits (-128..127)
INTEGER   2's complement 16 bits (-32,768..32,767)
LARGEINT  2's complement 32 bits (-2,147,483,648..2,147,483,647).
CHAR      ascii character 8 bits
BOOLEAN   TRUE or FALSE
REAL      32 bit binary (23 bit exponent and 8 bit mantissa)
          -6.80565E38...-5.8775E-39
          0.0
          5.8775E-39 ... 6.80565E38
```

The three types of integers may have full overflow checking and will generate a runtime overflow error if runtime error checking is on (see section 2-2). In addition any overflow can also be made to saturate the respective result. For example if an arithmetic expression of type SMALLINT evalutes to +130, then the value returned will be the maximum, 127. Negative overflow would saturate to the minimum number possible, -128. Sometimes it is necessary to have both runtime checking and saturation on, e.g. you may wish to saturate overflows and call an error trap to record the event. It is usually advisable to have runtime overflow checking on, at least in the testing stages of a new program. However, more code is produced to handle runtime checking.

The BYTE and DBYTE types are supported with limited arithmetic operations. They may be added, subtracted and compared, but there is no overflow checking or saturation. Rather any overflow simply

causes a wrap-around. In addition two BYTES may be multiplied to give a DBYTE result. Again the operation is performed on unsigned values. BYTEs are supported by the logical operations AND, OR, XOR and NOT. They operate on the 8 bits in parallel.

The type CHAR can hold 8 bits. The interpretation on the eighth bit is up to the programmer. Usually he will want to keep it a zero.

Generally no mixing of types is allowed in Whimsical. There are one or two exceptions. Constant arrays allow type mixing of their elements. For example a constant array of type CHAR may contain BYTE constants in the constant list. Most other type mixing is rarely required but where the programmer needs it he can declare his intentions by using the type converting intrinsics.

The structured types in Whimsical are arrays and files. There are no arbitrary restrictions on their size. The elements of arrays may be any simple type (arrays of arrays and arrays of files are not implemented, nor are multi-dimensional arrays).

CONSTANTS

All constants in Whimsical must have a type as do variables. Even literal constants have a type. The best way to illustrate this is with examples.

```
CHAR          'A', "B"
BOOLEAN       TRUE, FALSE
BYTE          $1    $02
DBYTE         $001  $0002
SMALLINT      1     127
INTEGER       1     32767
LARGEINT      1     100000000
REAL          0.1   1E17
```

Note that small integer literal constants such as "1" can be either SMALLINT or INTEGER or LARGEINT. In these cases the type is determined from the immediate context of their use. A literal BYTE constant is one that has one or two hexadecimal digits. A literal DBYTE constant must have three or four digits even if its value is small. A literal REAL constant must begin with a digit (optionally prefixed by a minus sign) and it must contain either a decimal point or the letter "E".

REAL NUMBERS

The type REAL is a 4 byte floating point number. This gives about 6 digits of accuracy, suitable for most control applications. It is based on the IEEE proposed standard format for 32 bit binary floating point numbers. The first (most significant) bit is the sign of the number. The next 8 bits are the exponent in excess 127 notation. The remaining 23 bits make up the mantissa with an assumed leading one. The actual value of a REAL number may be written as:

```
Value = SIGN(-1) * 2**(EXPONENT-127) * (1.MANTISSA)
```

Input of real numbers will allow much more freedom than literal constants allow. For example the number may start with a decimal

point or it may simply be an integer. e.g.

1.0, 0.1, .1, 1E-1, 3.14159, 100, -3.7E24 etc

Output of real numbers is in scientific notation. However a number of versatile formatting modules is provided to output real numbers in fixed point notation.

REAL numbers have full runtime overflow and saturation facilities. The REAL type should not be used for representing monetary values as they are subject to rounding errors. The LARGEINT type should be used for this. (It is easy to write a procedure to output a largeint with a decimal point before the last two digits.)

```
PROCEDURE WRITEMONEY(LARGEINT AMOUNT)=
BEGIN
  BOOL MINUS;
  LARGEINT WIDTH:=10000000;

  IF AMOUNT<0 THEN {MINUS:=TRUE; AMOUNT:=-AMOUNT};

  DO BEGIN % output padding spaces
    IF AMOUNT<WIDTH THEN WRITE " ";
    WIDTH:=WIDTH/10;
  END UNTIL WIDTH=100;

  WRITE IF MINUS THEN "-" ELSE " ","$",
        AMOUNT/100,".";  % write dollar amount
  IF AMOUNT MOD 100<10 THEN WRITE "0";
  WRITE AMOUNT MOD 100;
END;
```

WHIMSICAL EXPRESSIONS

An expression describes how a value is to be calculated using a sequence of simple operands and operations. The value calculated has a type depending on the type of the operands and operators in the expression. Certain operations take precedence over others as outlined in the table below. Operators at the same level are evaluated from left to right.

Highest Precedence	( ), If clause
	unary -, NOT
	*,/,MOD
	+,-
	<,>,=,<=,>=,<>
	AND
Lowest Precedence	OR,XOR

e.g. A*2+1>B OR FOUND AND NOT LOST

is a boolean expression. A*2+1 is an arithmetic expression. Any operand may itself be an expression contained in brackets. e.g. (A+2)*3.

The logical operators, NOT, AND, OR, XOR act differently on BYTES and BOOLEANS. On BYTES their action is simply on all eight bits in parallel. BOOLEANS however are interpreted as being TRUE if they are non zero. The logical operators are designed to reflect a TRUE or FALSE result depending on whether or not the operands are TRUE or FALSE. This is sometimes more complicated than simply operating on the eight bits all at once. Therefore it is assumed during some of the BOOLEAN operators that a TRUE value has at least bit 0 set. The comparisons operate to produce a TRUE or a FALSE result.

Don't forget all integers are interpreted as two's complement numbers while all other types except reals are considered as unsigned values. The operator "/" is used for integer division. The fractional part of the result is truncated. The MOD operator can be used to find the remainder of a division. The "*" operator is used for integer multiplication. In addition a BYTE can be multiplied by a BYTE to give a DBYTE result. In this case the values are treated as unsigned. The operation is therefore very fast.

When dividing a real number by a constant consider multiplying by its reciprocal instead because this will be much faster. Further, when multiplying or dividing an integer or real number by 2, it is much faster to use the ASL or ASR intrinsics (described in the next section).

Note that although an expression is always evaluted in accordance with the above precedences, the operands themselves are always fetched strictly from left to right. So if one of the operands is a typed procedure call, you know exactly when that procedure will be executed. Because the operands can't necessarily be evaluted in left to right order, intermediate results are often stored temporarily. The various operators can operate on the following types:

OPERATOR	TYPES
unary -	integers and reals
NOT	BOOLEAN or BYTE
*	integers and reals or BYTE * BYTE gives DBYTE,
/,MOD	integers and reals
+,-	BYTE, DBYTE, integers and reals
<,>,=,<=,>=,<>	any like types except BOOLEAN
AND,XOR,OR	BOOLEAN, BYTE

Note that when BYTES are operated on by the logical operators, their priority is still low. Therefore brackets will often be required. For example consider the follow expression to return TRUE if the low nibble of a byte is 4.

```
DATABYTE AND $0F = $04
```

The expression should be written

```
(DATABYTE AND $0F) = $04
```

THE IF CLAUSE

The IF clause is an expression subpart which selects between two expressions.

```
IF boolean expr THEN expression ELSE expression
```

First the boolean expression is evaluated. If it is TRUE then the first expression is used otherwise the second expression is used. Both these expressions must have the same type. The IF clause has the same priority as brackets in an expression. Here are some examples.

```
A:=IF A>0 THEN A-1 ELSE A

$0001+IF FOUND THEN ADDR ELSE $0000
```

The following example is not the same because the final $0001 would be part of the IF clause.

```
IF FOUND THEN ADDR ELSE $0000 +$0001
```

CONSTANT EXPRESSIONS

Constant expressions are those that can be evaluated at compile time to a single value. Constant expressions follow exactly the same syntax, hierachy and type mixing rules as normal expressions. However, the comparison operators will not produce a constant expression. Also most of the intrinsics will not produce constant expressions (except ASC, CHR, HEX and DEC). All other functions and types are supported. Of course variables and procedure calls are not allowed but named constants are allowed.

If any subparts of an ordinary expression meet the requirements for a constant expression then that subpart is simplified at compile time. The expression, A+3*20 is simplified to A+60.

INTRINSICS

Intrinsics are like procedures which have been inherently declared in every program. They perform a wide variety of functions which would otherwise be difficult to code. Each of the intrinsics is described below. Any that return a value can only be used in expressions. The ones which do not return a value can only be used as statements. A guide to their apparent formal declarations may be found in part 4.

ENABLEIRQ
DISABLEIRQ
ENABLEFIRQ
DISABLEFIRQ

All these intrinsics perform the suggested primitive operation. That is they set or clear the appropriate interrupt mask bits. Multiprocesses can be supported in Whimsical at a low level by using interrupts. While fast interrupt procedures are not supported directly, they may still be used in machine language routines (see section four on advanced programming). It is therefore necessary to provide the appropriate intrinsics for FIRQ's as well as IRQ's. These intrinsics do not have parameters and do not return a value. They can therefore only be used as statements.

LOBYTE
HIBYTE
COMBINE

These intrinsics operate on BYTE and DBYTE data. LOBYTE and HIBYTE are used for obtaining from a DBYTE, its single byte parts. Each must have one dbyte parameter and returns a byte value. COMBINE will take two byte parameters and produce a dbyte result. The first parameter becomes the most significant byte (HIBYTE). The example below multiplies the DBYTE "A" by 10 and adds in the byte value "DIGIT".

```
A:=LOBYTE(A)*$0A+COMBINE(LOBYTE(HIBYTE(A)*$0A),DIGIT);
```

ASC
CHR
HEX
DEC

These are type conversion intrinsics. They perform no function at runtime but greatly help in finding errors at compile time by forcing the programmer to explicitly declare that he wishes to mix types. In type changing, the BYTE and DBYTE types are central and all conversions are either from these types or to these types. CHR converts from BYTE to CHAR while ASC converts from CHAR to BYTE. HEX will change from SMALLINT to BYTE or from INTEGER to DBYTE. Similarly DEC will change from BYTE to SMALLINT or DBYTE to INTEGER. These two intrinsics are unusual in that the type of the result is determined from the type of the parameter.

EXTEND
TRIM

SMALLINT, INTEGER and LARGEINT types can be changed using these. EXTEND causes a small integer to be sign extended to a 16 bit integer or it makes a 16 bit integer into a LARGEINT. TRIM chops the most significant 8 bits off an INTEGER to make a SMALLINT or it chops 16 bits off a LARGEINT to make an INTEGER. If runtime checking is ON then the operation will be checked to see if the integer was too big or too small to be trimmed. If it has overflowed a runtime error will be generated. If saturation is ON then the resulting integer will be given the maximum or minimum value possible if the integer was too big or small.
If A is an integer then the following example multiplies it by 10 and adds in the value of the CHAR type DIGIT.

```
A:=A*10+EXTEND(DEC(ASC(DIGIT)+$30));
```

ASR
ASL
These are intrinsics incorporated with the policy of supporting functions which are themselves supported by the hardware. They execute a shift either right or left on any of the data types BYTE, DBYTE, SMALLINT, INTEGER ,LARGEINT or REAL. The result has the same type as the parameter. Very efficient multiplying or dividing by two can be achieved using these functions. The operations are identical for signed and unsigned operands for the left shift only. A zero is shifted into the least significant bit and the most significant bit is lost. Runtime error checking and saturation can be performed for the left shifts. The right shifts differ in their operation for signed and unsigned operands. If a BYTE or DBYTE is being shifted then a zero is moved into the most significant bit. If a signed integer is being shifted then the most significant bit remains unchanged in order to preserve the sign of the data. In both cases the least significant bit is lost. The REAL type is "shifted" by incrementing or decrementing the exponent. This is very much faster than a multiply or divide by 2. Again runtime checking may be used on the ASL for REALS.

FLOAT
FIX
INT

FLOAT will convert a LARGEINT into a REAL. Some of the least significant bits will be lost by rounding. FIX will convert a REAL type back to a LARGEINT. The number is rounded towards the nearest integer value. If it is too large then a runtime error will occur if this is switched on. Saturation may also be switched on. INT is similar to FIX but truncates instead of rounding.

HIDBYTE
LODBYTE
LCOMBINE
RCOMBINE

These intrinsics give the programmer raw access to the REAL and LARGEINT types. HIDBYTE and LODBYTE will extract the high or low DBYTE from either a REAL or a LARGEINT number. LCOMBINE will create a LARGEINT from two DBYTES and RCOMBINE will return a REAL from two DBYTES.

LOC

This intrinsic returns the absolute location of a variable or procedure. The returned value is of type DBYTE. The parameter may be any variable (including array or file) or procedure name. This function is very useful if used carefully. One application is in attaching an interrupt procedure to an interrupt vector in RAM. e.g.

```
DBYTE VECTOR($DFC8);

INTERRUPT PROCEDURE IRQROUTINE=
BEGIN
  .
  .
END;


% in main program

VECTOR:=LOC(IRQROUTINE);
ENABLEIRQ;
```

WHIMSICAL STATEMENTS

Statements allow a programmer to instruct the computer on what action to take. They are essentially the most fundamental elements of a computer language. Statements in Whimsical are much the same as executable statements in other structured languages. They are either simple or structured. The structured ones may themselves contain statements. Such a statement may be very complex even though the rules of its structure are very simple.
The statements covered in this section are:

ASSIGNMENT statement
IF statement
WHILE statement
DO statement
FOR statement
CASE statement
Compound statement
Procedure invocation statement
READ statement
WRITE statement
STOP statement

Some further statements can be found in the sections on disk files, and error trapping.

The simple statements are the ones which may not contain other statements. These include the assignment statement, the procedure invocation, the STOP statement and the Input/Output statements (READ and WRITE).

ASSIGNMENT STATEMENT

This is one of the simplest statements. It instructs the machine to compute an expression and assign the result to the specified variable. The general syntax is:

```
variable:=expression
```

The ":=" is the assignment operator and is pronounced "is assigned". For example X:=2 is read "X is assigned two". After execution of that statement the value of X would be 2. Some other examples are:

```
A:=A+$01
FOUND:=PTR>$01FF OR TOP
TABLE[FRED-$01]:=(A+1)*(B-A*3)
```

The first example has the effect of incrementing the variable "A". The second example assigns a boolean expression (TRUE or FALSE) to the boolean variable "FOUND". The third example assigns the value of the expression to an ARRAY element. The variable on the left hand side of the ":=" and the expression on the right must have the same type. To assign an expression of a different type requires the use of one of the type converting intrinsics. In this way any type mixing becomes self-documenting. An example follows:

```
CHARACTER:=CHR($0D)
```

In this example the CHAR variable "CHARACTER" is assigned the carriage return character, which has the ascii code $0D (in hexadecimal).

INPUT/OUTPUT STATEMENTS

Input/Output is the means by which programs communicate with the outside world. Without it a program can serve no useful purpose. Most Input/Output (I/O) can be handled by the READ and WRITE statements. A more complete discussion of Whimsical I/O can be found in the Disk File and the advanced programming sections. The discussion which follows is merely an introduction to the I/O statements.

The READ statement is used to input data and assign it to variables. The source of the data may be the terminal, a disk file or user defined (see advanced programming section).

Once the data has been obtained it is assigned to a variable in much the same way as the assignment statement. The syntax is as follows:

```
READ [FROM file_identifier[,]] variable [,variable . . .]
```

The comma before the first variable is optional. The three dots indicate that as many variables as desired can be listed. One data item is read in and assigned to each variable in the order they appear. The variables must have the same type as the file unless the file is of type CHAR (a text file) or type BYTE. If no file is specified (no FROM file identifier) then the default file identifier "INPUT" is used. This file is predeclared as being the console of the system on which this compiler runs. It may be redeclared, however, to be any other type or kind of file. Some examples follow:

```
READ A,B,C
READ FROM MODEM,CH
READ FROM FILE1 X,Y
```

The first example reads the CHAR variables A, B and C from the terminal. If the identifier "INPUT" has been redeclared then the variables will be read from there instead of the terminal. The second example could read data from a MODEM provided the programmer has declared the file "MODEM" suitably for the purpose. The third example could read data from the file "FILE1" into variables X and Y provided "FILE1" has been declared as a disk file of the same type as X and Y or a text file or a byte file.

When reading a non-BYTE variable from a BYTE file, as many bytes as necessary are read, the first being the most significant.

When reading a non-CHAR variable from a CHAR file then the number is read in text form and converted. If the type being read is a BYTE or a DBYTE then the number must be in hexadecimal ASCII form. A hexadecimal number may have a dollar sign. If the type being read is SMALLINT, INTEGER or LARGEINT then the number must be in decimal ASCII form. A decimal number may have a minus sign and/or leading spaces. The number read can be any length but must not be too big for the variable being read. Runtime

checking and saturation may be switched on to handle numbers which are too big. Any non numeric character will terminate the number being read.

The read statement can read a whole CHAR array as a special case. When numbers are being input from the keyboard it is often desirable to buffer the input to allow back spacing etc. To do this the text must first be read into an array and then the number must be read from the array. Whimsical allows an array to be read as follows:

```
CHAR ARRAY BUFFER[4];

READ BUFFER;
```

The array and the file must be of type CHAR. Only as many characters as will fit in the array may be read. After that characters are ignored until a carriage return is received. The above example would accept 4 characters which would be returned elements 0 to 3. The carriage return is always put into the array after the last character accepted. The backspace and control X functions can be used for correcting errors and starting again respectively. (Do not attempt to read an array which is a parameter. The compiler does not know the size of such an array). Once the text is read into an array, it is best to have a procedure which will read them out consecutively. Using this procedure as its source, a further READ statement can read the number.

```
CHAR ARRAY BUFFER[$4];
DBYTE ADDRESS; % VARIABLE TO BE READ
BYTE I;        % INDEX INTO BUFFER

CHAR PROCEDURE GET=
BEGIN
  GET:=BUFFER[I]; % GET CHARACTER FROM BUFFER
  % CHANGE TO UPPER CASE
  IF GET>='a' AND GET<='z' THEN GET:=CHR(ASC(GET) AND $5F);
  % DO ANY LOCAL CHARACTER PROCESSING
  IF GET=CHR($09) THEN GET:=' ';
  % STOP READING BEYOND END
  IF GET<>CHR($0D) THEN I:=I+$1;
END;

READ BUFFER;
READ FROM GET ADDRESS;
```

This scheme allows full flexibility in parsing input lines etc.

The WRITE statement outputs data to disk files or the terminal or a user defined destination (see advanced programming section for user defined I/O). The general form of the WRITE statement is:

```
WRITE [TO file identifier[,]] expression [,expression . . .]
```

One WRITE statement can send as many data items as required. They must be separated with commas. The types of these expressions must match the type of the file being written to unless it is a file of type CHAR or type BYTE. The "TO file identifier" part is optional and if not present assumes the file identifier "OUTPUT". This file

is predeclared as the terminal of the system on which this compiler runs. It may be redeclared by the programmer so that the simple form of the write statement is still useful even if the program is running on a different target machine. Some examples of WRITE statements follow:

```
WRITE "A"
WRITE TO DATAFILE X,Y+$01,Z-$0E
WRITE TO PRINTER CH
```

The first example will output the character "A" to the terminal unless the programmer has redeclared the identifier "OUTPUT". The second example writes to the disk file "DATAFILE", the results of the three expressions, assuming "DATAFILE" has been declared as a BYTE disk file. The third example could write the value of CH to a printer assuming "PRINTER" has been declared appropriately.

As you can see the READ and WRITE statements are used in exactly the same way for different kinds of files. They only refer to a file identifier. It is how that file identifier is declared that determines where the data actually goes (see advanced programming section).

If an expression not of type CHAR is sent to a file of type CHAR the number is automatically converted to text form. A BYTE value is written as two hexadecimal digits. A DBYTE value is written as four hexadecimal digits. An integer is written with leading zeros suppressed. In no case is a trailing space or any other character output. The formatting is left completely versatile.

If an expression other than type BYTE is output to a file of type BYTE then as many bytes as necessary are output to the file with the most significant byte first.

Literal strings can be output in the write statement as a special case. e.g.

```
WRITE '^M^JANSWER IS ',TEMP,' DEGREES'
```

In such a literal string it is illegal to embed a null ('^@') because the output routine uses this character as the terminator character. You will have to use the equivalent method of outputting the null as a CHAR. e.g.

```
WRITE "This string cannot contain a null^M^J",CHR($00),CHR($00);
```

The statements OPEN, CLOSE and CREATE are used only for disk files and are discussed in the section on (I/O).

PROCEDURE INVOCATION

A procedure may be thought of as a user defined extension to the language. It may take the form of either a statement or a function for use in an expression (see procedures section). A procedure invocation statement consists of the procedure name followed by any actual parameters that are passed to the procedure. The general form is:

```
procedure identifier[(expression or variable ...)]
```

The procedure identifier must be the name of a previously declared procedure. The actual parameters must match the corresponding type of the formal parameters in the declaration. Also the correct number of parameters must be passed and reference parameters can only have variables passed to them. A typed procedure may be called as a procedure invocation statement but the return value is lost. Some examples are:

```
JACK
FRED($C1)
DICK(A+1,RESULT)
```

The first example calls the procedure "JACK" with no parameters. The second example has one value parameter and the third could have one value parameter and one reference parameter. A more complete discussion on procedures and parameter passing can be found in the procedures section and advanced programming section.

THE STOP STATEMENT

This statement causes an immediate halt to program execution and a return to the operating system. The statement is unstructured because it causes a jump out of the program. All other unstructured statements such as exit or goto statements have been omitted from Whimsical. While they are powerful statements they are considered to be much too general for normal programming. As such they destroy any readability or self documenting features the program may have had.

STRUCTURED STATEMENTS

The structured statements are so called because they may contain other statements. The resulting structure is made up of relatively simple building blocks.

The first of these is the compound statement. It is used to group a whole list of statements together as one statement. The general form is:

```
BEGIN
  statement;
  statement;
     .
     .
     .
END
```

The statements are bracketed by the BEGIN END pair and separated by semicolons. They are executed in the order they appear. The number of statements can be anything from zero upwards. A semicolon after the last statement (before the END) is optional. The compound statement is very useful because it allows multiple statements to be used in any of the structured statements which follow where normally only one statement is allowed. It is so common in fact that it will very largely determine how your programs look (aesthetically) and also how easy they are to read. Therefore the indentation in the following example is recommended although the syntax of the language does not force this.

```
BEGIN
  A:=A+$01;
  X:=X+$0001;
  TABLE1[X]:=TABLE2[A];
END
```

The rest of the structured statements are used for sequence control. In other words they control if and when the statements they contain will be executed. There are two types of sequence control statements, conditional and iterative. The conditional ones are the IF statement and the CASE statement. They select one of several possible statements to execute.
The IF statement has the following form:

```
IF boolean expression THEN statement [ELSE statement]
```

The statements within the IF statement may be compound statements, other IF statements, or any other statement. The boolean expression will have a result TRUE or FALSE. If it is true the first statement is executed otherwise the second statement is executed (if it is there). The square brackets enclose an optional part of the IF statement. They are not part of it. An example follows:

```
IF A=0 OR FOUND THEN
BEGIN
  IF X>0 THEN X:=X+1 ELSE X:=1-X;
  FOUND:=FALSE;
END
```

This example is an IF statement without an ELSE. The statement which is conditionally executed is a compound statement. It contains two statements, one of which is an IF statement with an ELSE. Consider the following example:

```
 IF BOOL1 THEN
 IF BOOL2 THEN A:=1 ELSE B:=0
```

Where it is not clear to which statement the ELSE belongs, it is always associated with the most recent IF. If we had wanted the ELSE to belong to the first IF statement we would have to write

```
 IF BOOL1 THEN
 BEGIN
   IF BOOL2 THEN A:=1
 END ELSE B:=0
```

Consider the statement

```
IF PTR=TOP THEN FOUND:=TRUE ELSE FOUND:=FALSE
```

A much simpler statement to do the same job is

```
FOUND:=PTR=TOP
```

Finally consider the statement

```
 IF BIG THEN X:=0 ELSE X:=1
```

A simpler and more efficient way is to use the IF clause

```
X:=IF BIG THEN 0 ELSE 1
```

The other conditional statement is the CASE statement. Instead of selecting between two possible alternatives as does the IF statement, it selects between several possible alternatives. It has the following form:

```
CASE expression OF
BEGIN
  constant list:statement list
  constant list:statement list
              .
              .
              .
END
```

The expression is computed and the result is compared with all the constants in all the constant lists. When a match is found the statements following the constant list in which it is found are executed. The type of the constants must match the type of the expression (except that a BYTE constant may be used in a CHAR case). The only types allowed for case selection are CHAR, BYTE, or SMALLINT. The constant list consists of constants separated by colons or the symbol "ELSE". If no match is found in the constant lists then the statements following the "ELSE" are executed. An example follows:

```
CASE I+$01 OF
BEGIN
  $02,$05:$07: A:=1;B:=2;
  $01: A:=3;
  CONST1:CONST2:$1F: ;
  ELSE: A:=2;B:=1
END
```

The types of the expression and all the constants in this case are BYTE. Note the named constants, CONST1 and CONST2, which must have been declared as such. Also note that on the same line there are no statements in the statement list. If the match is made on this line then no action is taken in the case statement. Any lines with multiple statements must have semicolons separating them. A semicolon is not necessary after the last statement. The constants in the constant list are searched in the order they appear in the statement. It is best therefore to put the statements with the highest probability of use first.

The constants in a constant list must be separated by colons. If you wish to specify a whole range of contiguous constants then just the first and last may be specified separated by a comma.

The iterative statements are the WHILE, DO and FOR statements. They control looping so that a statement can be executed a number of times according to some condition.

The WHILE and DO statements have the following forms:

```
WHILE boolean expression DO statement
```

DO statement UNTIL boolean expression

The two forms are similar in many respects. Both evaluate a boolean expression each time around the loop, to test if looping should continue. They differ in where that test is done. The WHILE statement does the test first and the DO statement does it last. This means that the DO statement has to execute its contained statement at least once. The WHILE statement on the other hand would not execute its statement at all if the value of the expression is FALSE at the beginning. Some examples follow:

```
WHILE A<=$7F DO
BEGIN
  TABLE[A]:=TABLE[A+$01];
  A:=A+$01;
END

DO
  PRINTOUT(TABLE[A]);
  A:=A+$01;
UNTIL A>$7F
```

The FOR statement has the following form:

```
FOR variable := expr TO expr [STEP expr] DO statement
```

The variable is initialized to the value calculated in the first expression, and then the statement is executed repeatedly. Each time the statement is executed the variable is incremented. It continues until the variable equals or exceeds the value which was calculated from the second expression. The size of the increment can be set using the optional STEP and the third expression. If it is desired to count down instead of up, the word TO can be replaced with the word DOWNTO. In either case the increment expression must evaluate to a positive value. If on the very first iteration the variable already equals or exceeds the final value, then the statement is still executed once. The types of all three expressions and of the variable must be the same.

```
e.g.  FOR I:=1 TO 20 DO WRITE I
      FOR COUNT:=$1 TO $FE STEP $3 DO
      FOR X:=30 DOWNTO -30 STEP 10 DO
```

The control variable used in a FOR statement must be a simple type and it must be a SMALLINT, INTEGER, LARGEINT, BYTE or DBYTE.

The exit statement may be used within any of the three looping statements to cause an unstructured exit of the loop. Use of EXIT should be restricted to abnormal or error conditions detected inside the loop. If loops are nested then only the most internal loop is exited by EXIT statement.

DECLARATIONS

The declarations are where the programmer declares the names and types of all the variables and procedures he intends to use in a program. As well as being a necessary element for the functioning of the compiler, declarations also serve to help the programmer. He must not only design the actions a program is to take, but also the data it is to act upon. This design is written down in the variable declarations. Once a clear understanding of the data structures is attained, the task of writing the actual statements to use them is very much easier. This is especially true in a multi-tasking environment. The declarations also help to make a program self-documenting.

VARIABLE AND CONSTANT DECLARATIONS

These are the first items in a BLOCK. The types as well as the names of the variables must be declared. A declaration consists of a type symbol followed by a list of identifiers.

```
Type identifier [,identifier ...]
```

The type may be any of the words: BOOLEAN, CHAR, BYTE, DBYTE, SMALLINT, INTEGER, LARGEINT or REAL. The identifiers can not be previously declared in the block. Declarations are separated by semi-colons, and the identifiers in each declaration are separated by commas. Some examples are:

```
BOOLEAN FOUND,ERROR;
BYTE COUNT;
DBYTE PTR,PTR1;
CHAR CH;
SMALLINT I;
INTEGER J1;
LARGEINT J2;
REAL F;
```

Ideally run time checks should be performed to detect any variables being used before they are assigned. However, this adds an overhead at runtime which cannot be tolerated. If the checks are not implemented it is possible that undefined variables could be used undetected which could lead to unpredictable results. Furthermore it is possible that a bug may surface at any time after testing. Whimsical solves the problem by initializing variables to zero (booleans are initialised to FALSE, chars to null). The automatic initialization to zero is very useful and in Whimsical it is correct to make use of this feature by omitting initialization statements which set a variable or array to zero. Remember that global variables are initialized once but local variables (those inside procedures) are initialized every time the procedure is called.

Variables may be declared at an absolute location in memory. The address in memory can be specified either directly or indirectly.

Two examples follow:

```
DBYTE VECTOR($DFC8);   % IRQVECTOR
BYTE MEMTOP ([$CC2B]); % BYTE AT MEMEND
```

The square brackets indicate indirection. In other words an address is obtained from the DBYTE at $CC2B and this becomes the address of the variable MEMTOP. Variables declared at an absolute location are not automatically initialized to zero.

A variable only exists in the block in which it is declared. If the same identifier has also been used in an outer block then that identifier will be inaccessible inside the current block.

Constants are declared in a similar way to variables, except that an equals (=) sign must follow the identifier with the value of the constant.
The value given must be a constant expression; that is one which can be fully evaluated at compile time.

e.g.
```
BYTE CONST1=$F0;
DBYTE CONST2=$3FF,ZERO=$0000;
BOOLEAN RHS=TRUE;
INTEGER MAXINT=32767,TABLESIZE=100;
SMALLINT LINESIZE=64;
REAL PI=3.141593;
```

The constant declarations can be intermingled with variable declarations. All constant identifiers have a type and the type of the constant expression equated to it must be compatible. Type converting intrinsics are therefore very useful. Here are some more examples:

```
CHAR CR=CHR($0D);
BYTE PLUS=ASC('+');
CHAR SPACE=CHR(HEX(32));
DBYTE PIA_DATA_B=IOBASE+$0002;
```

Arbitrary initialization can be achieved in the declarations by the use of declaration time assignments. The syntax is exactly the same as for an assignment statement except that it is written within the declaration. For example:

```
BYTE X:=$20,Y:=Z+$40;
```

These assignments may contain any arbitrary expressions except that all variables used must have been declared previously. The same rules about type mixing apply as for ordinary assignments. The assignments, as with constant equates can be intermixed freely with the other declarations.

ARRAY DECLARATIONS

Arrays are declared by using the reserved word ARRAY straight after the type. Each array must normally be given a size in square brackets. For example:

```
CHAR ARRAY NAME[15];
BYTE ARRAY FRED[$10],DICK[$100];
INTEGER ARRAY TABLE[100];
```

The size of the array is actually one more than is given in the declaration. This is because the elements include both element zero and the end element equal to the number given in square brackets. The size may be a named constant previously declared. The type of the size may be BYTE, DBYTE, SMALLINT or INTEGER. It makes no difference to the way the array may be indexed. Indexing of arrays may be done with an expression of any of the types BYTE, DBYTE, SMALLINT or INTEGER. However be very careful to use a type which can access the whole array. For example if you declare an array of size $2000 there is little point in indexing it with a BYTE expression. At most it will only access 256 of its elements. Also beware of using a negative index value. Since runtime error checking on array bounds is very inefficient it is only done if you turn on runtime checking around the code which accesses the array. Multi-dimensional arrays are not allowed.

Arrays may be declared with an origin at an absolute location in memory in a similar fashion to ordinary variables. Direct and indirect address specification is possible, as can be seen in the following examples:

```
BYTE ARRAY MEMORY ($0000) [$140];
BYTE ARRAY FCB ([$CC24]);  %FILE OUTPUT ADDRESS
```

The size of the array is optional and is only used for array bounds checking.

CONSTANT ARRAYS

Constant arrays are declared along with the ordinary arrays by using an equals sign followed by a list of constants in round brackets. Since the size is known from the number of elements given, there is no need for the square brackets of normal array declarations.

```
CHAR ARRAY MESSAGE=($0D,$0A,"HELLO");

BYTE ARRAY DATA=(
$1024,
$37,
$AB,$34FC
);
```

Notice that some BYTE constants have been included in a CHAR array and some DBYTE constants have been included in a BYTE array. Type mixing is allowed under the following rules:

Type of array	Type of constants
CHAR	CHAR, BYTE, SMALLINT, LITERAL STRING
BYTE	" " " " or DBYTE
SMALLINT	" " "
DBYTE	DBYTE, INTEGER
INTEGER	" "
BOOLEAN	BOOLEAN
LARGEINT	LARGEINT
REAL	REAL

Where a 2-byte constant is used in a 1-byte type array, two consecutive elements are used. Also note that a string literal may be used instead of a list of CHARs or BYTEs.

Constant arrays should not normally be assigned to. If you wish to set the initial value of a variable array you should copy from a constant array using a FOR statement. However if you know that your program will only ever be run in RAM and you really want to assign to a constant array, then the compiler will not stop you. It is a more efficient way of getting an array initialized. Note that variable arrays are initialized to zeros. Also if you assign to a constant array then the program cannot be re-run without loading it off disk again.

PROCEDURES

A procedure is a certain section of code which can later be called many times without having to repeat the code. The resultant programs are therefore smaller. If written well, a procedure can be like a self-contained subprogram with well defined input and output to the rest of the program. For this reason procedures are quite often used even if they will be called only once. They help to logically separate the program and document it (also see Modules). They also help the programmer to write in a top down, or a bottom up fashion (N.B. both methods of programming are better than a straight beginning to end method). Also note that since procedures must be declared before they are used, the bottom up method is Whimsical's natural style. If you want to write top down you will have to reverse the order of all your procedures as you type them in!).

Procedures have the ability to pass parameters and to have local variables. There is no limit on the number of each. Here is an example procedure which outputs a character a specified number of times.

```
PROCEDURE REPEAT(CHAR CH; SMALLINT N )=
BEGIN
  SMALLINT COUNT;
  FOR COUNT:=1 TO N DO WRITE CH;
END;
```

The reserved word "PROCEDURE" introduces the procedure declaration. The first line of this procedure is called the procedure head. It declares the formal parameters which can be used inside the procedure. In this example there are two parameters called "CH" AND "N" of two different types. When the procedure is called, actual parameters are passed which will set the initial values of CH and N.

```
REPEAT ( " ",X+10 );
```

The statement above calls the procedure. In effect when this happens CH is assigned the value " " and N is assigned the value X+10.

Following the procedure head always comes some form of information about what the procedure will do. Usually, as is the case in our example, this will take the form of a BLOCK. This block may be as large and complicated as you would like. Any variables declared are called locals because they do not exist outside the block. Like any other variables, the locals are given initial values. Whereas the globals of a program are only initialized once, locals are initialized (and in fact created) every time the procedure is called. The local in the example is "COUNT". The parameters "CH" and "N" can also be considered local. They can be changed and used like any other local variable. There is no effect on any data outside of the procedure. The only difference between say "N" and "COUNT" is that "N" is initialized not to zero but to whatever value was passed to it. The rest of the block consists of a few statements. Whenever the procedure is called, these statements would be executed and then control would return back to the

statement following the procedure call.

A procedure may contain a procedure declaration but only to a nesting level of two. Therefore there can only be at most three lex levels. The global variables are said to be at lex level zero. A procedure's parameters and variables are at lex level one and a procedure within a procedure is at lex level two. There is no lex level three. Generally three lex levels are sufficient for normal programming. Procedures have a certain runtime overhead which is a factor against nesting them deeply anyway. It is better to use modules which may be nested to any depth with no runtime overhead.

It is quite possible to manipulate and use variables of a lower lex level, but a procedure which does so should be documented. Using globals can be an efficient way to pass information to and from procedures.

The parameter passing we have seen so far is known as passing by value. Passing by value is no good if you want to return information to the calling statement. In other words you can only pass information to the procedure with passing by value. There are three ways of passing information back from a procedure. The first is to directly manipulate a global variable. Where this is done it should be well commented since it won't be apparent from looking at the procedure's head. The second method is to use a typed procedure and the third method is to use passing by reference.

TYPED PROCEDURES

Typed procedures, in addition to doing everything an untyped procedure can do, may return a value. They are typically used in an expression. For example:

```
BYTE PROCEDURE SHIFT4(BYTE DATA)=
BEGIN
  SHIFT4:=ASR(ASR(ASR(ASR(DATA))));
END;
```

This procedure is designed to look like a function which shifts a byte right by four bits.

```
HINIBBLE:=SHIFT4(NO);
```

Typed procedures don't have to be used in an expression. Sometimes the returned value is not needed and the procedure may be called in an invocation statement like an ordinary procedure. In this case the returned value is ignored. Typed procedures without parameters can sometimes mislead someone reading your programs because they look like an ordinary variable but may actually be a complex routine. On occassion the programmer will wish to make it appear to be a variable! This may be because the procedure behind it causes no side effects and it is best to hide the complexity of it.

Inside a procedure the procedure's name is used to access the return value. The return value is like any other variable. It may be used anywhere in the procedure body. Its value when the procedure exits is what is returned.

PASSING BY REFERENCE

A form of parameter passing is available which tells a procedure the location of a variable rather than passing the value of an expression. The procedure may then change that variable knowing its location. In that way information is passed out of the procedure. For example, say you have a procedure for incrementing a variable in modulo 60. You will need to pass it the variable that it is to increment.

```
BOOLEAN PROCEDURE INCMOD60(SMALLINT REF CLOCKVAR)=
BEGIN
  CLOCKVAR:=CLOCKVAR+1;
  IF CLOCKVAR>=60 THEN
  BEGIN
    CLOCKVAR:=0;
    INCMOD60:=TRUE;
  END;
END;
```

The parameter is a reference parameter because it has the reserved word REF following the type. Only SMALLINT variables may be passed to this procedure. SMALLINT expressions may not be passed because the procedure cannot modify an expression. Notice that the procedure is also a typed procedure returning true if the variable was reset to zero. Using this procedure we could now implement a clock.

```
SMALLINT SECONDS,MINUTES;
.
IF INCMOD60(SECONDS) THEN INCMOD60(MINUTES);
```

Notice how in the second instance the procedure is called as a statement, the return value being ignored.

PASSING ARRAYS

Arrays are automatically passed by reference. The reason is that in passing by value the expression is assigned to the parameter variable. For whole arrays this would involve copying the entire array, taking time and stack space. The programmer must be aware that if he manipulates the array, he is changing the original.

In the formal declaration of the procedure head, array parameters are declared without specifying the size in square brackets.
e.g.

```
PROCEDURE SORT(SMALLINT SIZE; INTEGER ARRAY NUMBERS)=
```

RECURSION AND FORWARD DECLARATIONS

Whimsical supports full recursion. In other words a procedure can call itself or call other procedures which in turn call the first procedure. Only untyped procedures can call themselves from within their own body. This is because in the body of a typed procedure the procedure's name refers to the return value, not the procedure as a call. Sometimes it is desired for a procedure to call another which then calls the first procedure back. Since a procedure must be declared before it is used, which procedure should you put first? The solution is to declare one of them as forward then declare the other in full and finally declare the first one in full. To declare a procedure as forward, only the head is

declared. e.g.

```
PROCEDURE FRED(BYTE DATA)=FORWARD;
```

Any type of procedure can be forward declared in this manner. When the procedure is declared in full, its head must be written out in full again, in the normal way, and it must exactly match that in the forward declaration.

The procedure is a fine way to segment a program into parts with well defined interfaces between the parts. The locals of a procedure are useful but they cannot be made permanent. For example if you wish to create a procedure to return a random number, then the seed variable would need to be made a global and would therefore complicate the whole program instead of just the procedure which used it. What is needed is a way to make the seed variable a permanent variable while still keeping it hidden from the rest of the program. For futher versatility you may wish to define not one but several procedures which can access such local permanent variables. The answer is the module structure.

A module is a logical packaging of data and procedures. The compiler can check that the variables in the module are not used outside it. The variables are said to exist outside the module (because they are still permanent) but are not visible outside it. Extensive use of modules therefore leads to very good programs which are easily proved and maintained. An example module is given below. It implements output buffering.

```
MODULE BUFFEREDOUTPUT=
BEGIN

PUBLIC
  PROCEDURE OUTBYTE(BYTE DATA);

PRIVATE
  BYTE BUFPTR;
  BYTE ARRAY BUFFER[$7F];

  PROC FLUSH=
  BEGIN
    BYTE FLUSHPTR;
    WHILE FLUSHPTR<BUFPTR DO
    BEGIN
      WRITE BUFFER[FLUSHPTR];
      FLUSHPTR:=FLUSHPTR+$01;
    END;
    BUFPTR:=$00;
  END; % OF FLUSH

  PROC OUTBYTE(BYTE DATA)=
  BEGIN
    BUFFER[BUFPTR]:=DATA;
    BUFPTR:=BUFPTR+$01;
    IF DATA=$0D OR BUFPTR=$80 THEN FLUSH;
  END;

END;
```

Note the module name "BUFFEREDOUTPUT" which is entered into the symbol table and therefore cannot be used as an identifier for anything else. The name is used in the global symbol table dump or cross-reference dump.

The module contains two data items, the "BUFFER" and a pointer to

it, "BUFPTR". It also contains two procedures only one of which can be called from outside the module. That one is declared as a public procedure in the module's head. More than one procedure could have been declared there, separated by semi-colons. The full procedure head of each procedure which is to be called from outside the module must appear in the module's head within the publics section.

Statements at the end of the module are allowed. They are executed when the block containing the module is activated. If the module is not inside a procedure the statements are executed just once when the program first starts. The most common use of statements here is to initialize the data structure of the module. Another use is when the main routine of a program is made into a module.

Modules can only be used at Lex level 0, that is you cannot put a module inside a procedure. Modules are declared in a program in the same place as procedures. They can be intermixed. A module may contain a module and they may be nested to any depth.

The distinction between the LOCALS of a PROCEDURE and the PRIVATES of MODULE can be thought of in this way. Each has a scope in which it may be used. The scope of a local begins and ends in time, during the running of the program. The scope of a private begins and ends in code. It is accessable by only a proportion of the program. That is why a private exists throughout the running of a program but a local exists only while the procedure in which it belongs is activated. This is also the reason why modules have no runtime overhead and so there should be no worry about using them. In fact they speed up compilation time because once compiled, their private variables are removed from the symbol table.

Sometimes it may be necessary to declare a procedure within a module as being forward. If that procedure is also a public procedure then its head would have to be declared a total of three times.

Here is another example of a module. It is a random number generator in which the SEED is a permanent variable within the module.

```
MODULE RANDOMNUMBERS=
BEGIN

PUBLIC
  REAL PROCEDURE RANDOM;

PRIVATE
  LARGEINT
    SEED,
    MULTIPLIER=2187,
    MODULUSL=524288; % 2**19
  REAL
    MODULUSR=524288.;

  REAL PROCEDURE RANDOM=
  BEGIN
    SEED:=(SEED*MULTIPLIER+1) MOD MODULUSL
    RANDOM:=FLOAT(SEED)/MODULUSR; %REAL BETWEEN 0 AND 1
  END;
```

```
        WRITE CHR($0D),CHR($0A),"ENTER SEED ";
        READ SEED;
      END;
```

The example demonstrates the use of statements after the procedures in the module to initialize the variables.

Normally large programs have a long list of variable declarations at the beginning. Using modules, it is possible to reduce that list to just a few truely global variables, that is, variables used throughout the program. All other variables should fit into one of the modules. Sometimes variables are strongly associated with a module but are occasionally used outside it. In this case the variables should be declared in the module's public section.

PRE-COMPILED MODULES

A module may be compiled separately and later linked into your main program. This facility has several advantages. Firstly it is a lot faster to link in a module when compiling your main program than it is to compile it (by using ~INCLUDE). Secondly it provides a means of distributing useful routines without giving away your source. Pre-compiling a module produces a file with a default extension of "MOD". This file consists of the compiled code together with enough information to fill in references to external variables or procedures and other miscellaneous linkages. In order to compile such modules, it is necessary to declare all the external references in a special section in the head of the module. This section is headed EXTERNAL, and should not be confused with EXTERNAL procedures which are discussed elsewhere. The externals section immediately follows the public section. The EXTERNAL declarations are for variables or procedures which exist outside your module but which you want to access. The PRIVATE declarations are the normal variables and procedures that form the body of a module.

```
MODULE DRIVER=
BEGIN
  PUBLIC    % PUBLIC DECLARATIONS
    BYTE A;
    PROC FRED;
  EXTERNAL  % EXTERNAL DECLARATIONS
    BYTE B;
    PROC JACK;
  PRIVATE   % PRIVATE DECLARATIONS
    BYTE C;
    PROC MIKE={ STATEMENTS };
END;
```

To link in a pre-compiled module use the following syntax:

```
MODULE [name]=CODE FROM "file specification";
```

The file name has a default extension of "MOD". The file is read in, the code passing straight through to the output file. Any external declarations within the modules are checked to see if they actually exist and are of the correct type etc. The compiler has been designed to accept modules in either source or

pre-compiled form essentially without modification. The external declarations for example will not cause problems if the module source is included into a program instead of linking in the MOD file.

Modules may only be linked into the main program at lex level 0. It is not possible to link a module inside a procedure declaration. Also linking a module into another module which is itself being pre-compiled is not supported at this stage.

EXTERNAL declarations of constants or variables at an absolute location require further explanation. A simple constant must be declared with its value exactly as it is in the actual declaration. This is because the compiler must know the value of constants when pre-compiling so that it can evaluate complex constant expressions etc. At link time the compiler merely checks that the external and actual declarations have the same values.

A constant array is different. When a constant array is declared the values are emitted as part of the program's code. An external declaration of a constant array does not require re-enumeration of all the constant values. Instead this abbreviated syntax is used.

```
EXTERNAL
  CHAR ARRAY MESSAGE=();
```

Variables at absolute addresses, like simple constants, must be declared in the external declaration exactly the same as the actual declaration. The address specified for the variable will be checked at link time that it is the same.

Whimsical has the ability to nest ~INCLUDEs primarily to aid with external declarations in modules. It is suggested that the publics for each module of a program be put into their own little include files. Then each module which needs to make use of another module simply includes the publics file for that module within its own externals section.

A program can be entirely made up of pre-compiled modules. The initialization code at the end of the last module can be thought of as the main program.

When modules are nested it sometimes desirable that a public variable or procedure of the inner module, also be public to the outer module. This is permissible as long as the declarations match. Variables declared publically in both inner and outer modules do not constitute two variables, just two declarations of the same variable.

```
MODULE DATABASE=
BEGIN

PUBLIC
  CHAR ARRAY RECORD[80]; % This is declared in module
                         % diskreadwrite
  PROCEDURE PUT;
  PROCEDURE GET;

PRIVATE

  MODULE INDEXHANDLER=
  BEGIN
  PUBLIC
    INTEGER PROCEDURE LOCATE;
    PROCEDURE UPDATEINDEX(INTEGER RECORDNUMBER);
  EXTERNAL
    CHAR ARRAY RECORD[80];
  PRIVATE
    CHAR ARRAY INDEX[100*10];
    CHAR FILE INDEXFILE;
    .
    .
    .
  END; % OF MODULE INDEX

  MODULE DISKREADWRITE=
  BEGIN

  PUBLIC
    CHAR ARRAY RECORD[80];
    PROCEDURE PUT;
    PROCEDURE GET;

  PRIVATE
   .
   .
   .
  END; % OF MODULE DISKREADWRITE

END; % OF MODULE DATABASE
```

DISK FILES

Whimsical supports all standard sequential disk operations. The fundamental operations are create, open, write, read and close. It is also necessary to have a means of detecting the end of a file when reading from it.

The general Input/Output statements, READ and WRITE have already been discussed in the section on statements. They are used in exactly the same way with respect to disk files. All disk files must be declared so that the compiler can allocate storage for the file control block and so that a logical file name can be given to the files for use within the program. At declaration time the file is also required to be given a type. For FLEX files different types are treated in different ways by the system. Most of the time this will not affect the user, but special cases will be mentioned later.

The general form of a file declaration is:

```
type FILE identifier [,identifier ...]
```

Some examples of file declarations:

```
BYTE FILE CODEFILE;
CHAR FILE ADDRESSES, DATABASE;
INTEGER FILE NUMBERS;
```

In order to create a file the user is provided with the CREATE statement. In a create statement it is necessary to specify the external filename and a file of the type declared in the file declaration is created. If the file is already open, an error condition is reported, but if the file is closed and already exists, the existing file will be deleted and a new one created. Once a file has been created it is said to be open for write. In other words it will be ready for any WRITE statements which occur. The general form is:

```
 CREATE file AS external-filename
```

Filename is an identifier previously declared in a FILE declaration and external-file-name is a valid system file specification, which is either a CHAR typed array or a literal string. Some examples of the use of the CREATE statement:

```
CREATE ADDRESSES AS "1.FRIENDS";
CREATE CODEFILE AS "1.POKEUTE.CMD";
CREATE NUMBERS AS FILENAME;          %FILENAME IS CHAR ARRAY
```

Before a file can be read from, it must be opened using the OPEN statement. The external filename is specified in the OPEN statement and if this file does not exist or the file is already open then an error is reported. The general form is:

```
OPEN file AS external-filename
```

The file must be a file identifier which has been declared previously and the external-filename is either a literal string or

a CHAR array which conforms to system filename requirements. Some examples of the use of the OPEN statement are:

```
OPEN ADDRESSES AS "1.FRIENDS";
OPEN CODEFILE AS "1.POKEUTE.CMD"
OPEN NUMBERS AS FILENAME;       % FILENAME IS CHAR ARRAY
```

If a file has been opened for reading by the OPEN statement or opened for writing by the CREATE statement it should be closed using the CLOSE statement once all required reading or writing has been completed. A file must be closed by using the CLOSE statement before leaving the block containing the file declaration. The general form of the CLOSE statement is:

```
CLOSE file
CLOSEDELETE file
```

The file must be a previously declared file identifier. The file may be closed then deleted using the CLOSEDELETE statement. This is useful for temporary data files. Examples:

```
CLOSE ADDRESSES;
CLOSEDELETE TEMPORARY_SCRATCH_FILE
```

As mentioned earlier, it is useful to have a means of detecting the end of a file when reading from it. This is accomplished with an intrinsic returning a boolean value which is true if there are no more data elements in the file. The general form of this intrinsic is:

```
EOF( file )
```

The file must be a previously declared file identifier. An example of the use of the EOF intrinsic:

```
WHILE NOT EOF(ADDRESSES) DO
BEGIN
  READ FROM ADDRESSES CH;
  % statements using CH
END;
```

Assuming the file ADDRESSES is of type CHAR this loop will terminate as soon as the last character has been read from the file. However, files declared of any type other than CHAR are treated differently in the external system. For these files, once the end of the data has been reached, nulls will be returned until the end of the physical sector. This means that EOF only detects the physical end of the file and not the logical end of file. The user must, therefore, keep track of how much data a non text file contains. For CHAR files EOF detects the logical end of file. A CHAR type file will also perform space compression when storing the data physically, but this is not obvious to the user. FLEX binary format files are not processed directly by Whimsical. The user is responsible for coding and decoding these types of files should he wish to use them.

Files may be of any type. For example an integer file may be used to store integer data. Sometimes however, it is necessary to store different types of data in a file. This is allowed for files of type BYTE or CHAR. The packing and unpacking of data is handled

differently for BYTE and CHAR files. An integer written to a BYTE file is chopped into two bytes, but to a CHAR file it is converted to a text string (of between 1 and 6 characters in length). When data is written to a CHAR file it is up to the programmer to write separating spaces, commas or whatever, so that it may be read back correctly.

COMPILER DIRECTIVES

Compiler directives are commands intended for the compiler and do not form part of the normal syntax of the program. For this reason they are treated in a different way to normal statements. A compiler directive requires a complete line of input and is indicated by a '~' (tilde). The directives are:

~STACK=hexlocation

The initial stack pointer is specified by this directive. If used it must be the first item in a program. Usually it would contain an indirect address which causes the stack pointer to be loaded from the specified address. e.g.

```
~STACK=[$CC2B]
```

This causes the stack to start from the FLEX memory end. If a direct address is specified then the stack pointer is set to that address. e.g.

```
~STACK=$BE00
```

~ORIGIN=hexlocation

This directive must be specified if the desired load address is other than $0000. More than one origin may be used in a program but they should only be put between procedure declarations. i.e. to reposition whole procedures. They may not be used between statements. e.g.

```
~ORIGIN=$C100
```

The compiler always remembers the previous code address as well as the current one. You can switch back to the previous code segment using the ~USE CODE directive.

~NEWPAGE

Causes a new page to be started in the listing if paging is turned on.

~USE BASE

Swaps to the base data segment for subsequent declarations. Variables may be declared in this segment independently of those declared on the stack. The first time this directive is used, the ~BASE directive should be used to set the initial starting address for the segment. To change back to normal variable declarations see the ~USE STACK directive. Section 3-2 has more information.
The base segment is useful if you have a separate RAM chip you wish to use such as non-volatile RAM.

~BASE=hexlocation

Sets an initial location for the base data segment. This directive, if used, must immediately follow a ~USE BASE directive. An absolute hex address must be specified. All following variables declared will be allocated space starting from this address.

~USE STACK

Changes back to the normal stack data segment for subsequent declarations. This directive should be used after you have declared the variables in the base data segment.

~USE TEMP

Changes to a temporary segment for subsequent declarations. This directive should be followed by a ~TEMP directive to set the origin for the temporary segment. To return to declaring variables normally use ~USE STACK.

The temporary segment is different from the base segment in that it is mean't for declaring only a few variables here and there. For example to declare some ports:

```
~USE TEMP
~TEMP=$E000
  BYTE ADATA,ACONTROL,BDATA,BCONTROL; % PIA PORTS
~USE STACK
```

~TEMP=hexloaction

Sets the origin for the temporary segment.

~USE CODE

Swaps to previous code segment. That is the one prior to the last ~USE CODE. The compiler will keep two separate code segments. The first time ~USE CODE is used it should be followed by an ~ORIGIN directive.

~INCLUDE external-filename

A text file may be included in the source program by using this directive. The external filename must be a valid system file specification. Example:

```
~INCLUDE "MODULE1"
~INCLUDE "0.INTRFACE.ASM"
```

These includes may be nested to two levels.

~VERSION literal const,'string',literal const,'string', ... etc

Gives a FLEX compatible version number to the object file for later identification. e.g.

```
~VERSION 1
~VERSION 2,".",3,":",32
~VERSION 1,".2 by Fred Dagg"
```

The version directive must be after any initial ORIGIN but before the first BEGIN.

~TITLE="This is a title for the top of each page of the listing"

The title, if included, is put at the top of each subsequent page of the listing, provided paging is on. The title may be redefined many times in a program.

~IF constant-boolean-expression THEN

The following code is only compiled if the expression evaluates to TRUE. ~IFs may be nested. N.B. The comparison operators do not produce constant expressions.

~ENDIF

End conditional ~IF directive.

~TRAP TO procedure_name

This is a global error trap directive. It affects all the following code up until another trap directive unless parts of the code are overridden by a TRAP statement. This directive does not actually turn runtime checking on. This must be done with the R option for the required parts of the program. An error handling procedure must have been previously defined. The trap directive is useful for programs intended for an environment other than FLEX should you still wish to perform arithmetic overflow or array bounds checking. Example:

```
~TRAP TO OH_NO
```

See section 3-4 on the TRAP statement.

~OPTION

The ~OPTION directive allows most of the command line options to be put into the program itself.

The L, R, S and T options are all treated the same way.

L	listing on
R	runtime checking on
S	saturation on
T	tracing on

If prefixed by a minus then the corresponding feature is turned off. If prefixed by a slash then the function is to revert back to the status in effect before the last time it was turned on or off. This may be nested to a level of 7.

If one of these is specified on the command line then the function

is to completely override all occurances in option directives throughout the program.

The initial default for all four conditions is to OFF.

~OPTION L

Turns on the listing.

~OPTION R

Turns on all runtime arithmetic overflow and array bounds checking for the following source lines. This will cause the compiler to generate more code. An overflow error will cause program execution to stop with an error message unless the TRAP statement is used. The TRAP statement automatically turns on runtime checking in the contained statement so that use of CHECKON is unnecessary there.

~OPTION S

Turns on saturation for signed arithmetic operations in the following code. (The compiler will produce more code.)

~OPTION T

This option turns on tracing for the following code. This means that during the running of the program the name of each procedure called will be printed on the terminal.
The information from the trace is sent to the currently defined OUTPUT destination unless that identifier is declared as a disk file, when it is sent to the terminal.

~OPTION N

Forces the compiler to continue even if there is an error.

~OPTION P

Causes the output to be paged. New lines/page, columns/line and margin may be specifies as with the command line version (see section 1-3).

~OPTION W

Search working disk for subroutine files rather than using internal ones.

~OPTION I

Give futher information on stack usage, code addresses, nesting level. Two I's will give the unused idenifiers. Three I's will give information on the inclusion of subroutine files.

~OPTION G

Make a global symbol table dump. (see section 3-7).

USER DEFINED I/O

In this section we start to describe those features which set Whimsical apart from other languages.

The user defined I/O has been mentioned several times in the previous sections. Basically it allows you to use the power of the READ and WRITE statements while inputting and outputting to any arbitrary destination. The destination is still considered as a file, but it is declared as an ordinary procedure. If the procedure is being used for output then it must have one and only one parameter. The type of that parameter must be BYTE or CHAR. Types other than BYTE or CHAR may be written as disk files. If the procedure is to be used for input then it must be a BYTE or CHAR typed procedure with no parameters. These procedures determine the actual destination or source of data for the READ and WRITE statements. The names of these procedures may be "INPUT" or "OUTPUT" in which case the READ and WRITE statements will use them by default (e.g. if there is no FROM or TO clause).

```
PROCEDURE PRINTER(CHAR DATA)=
BEGIN
  BYTE STATUSPORT($E000);
  CHAR DATAPORT($E001);
  BYTE READYBIT=$40;      % READY STATUS INDICATOR
  IF PRINTEREXISTS THEN   % GLOBAL BOOLEAN
  BEGIN
    WHILE (STATUSPORT AND READYBIT)=$00 DO;   % WAIT
    DATAPORT:=DATA;  % OUTPUT THE CHARACTER;
  END;
END;

WRITE TO PRINTER "THIS GOES TO PRINTER";
```

The above example is a typical use of user defined I/O. Here is an example which allows you to read characters from the FLEX routine "NXTCH". NXTCH gets a character from FLEX's command input buffer and returns it in the A accumulator.

```
CHAR PROCEDURE CMDLINE=
BEGIN
  DBYTE PROCEDURE NXTCH=EXTERNAL($CD27);
  CMDLINE:=CHR(HIBYTE(NXTCH));
END;

FOR I:=0 TO 14 DO
  READ FROM CMDLINE, FILENAME[I];
```

Note that since NXTCH returns its characters in the A accumulator but Whimsical needs them in the B, we must return D from NXTCH and then extract the A accumulator by taking the HIBYTE.

The next example is used to buffer the input characters so that BACKSPACE may be used while inputting numbers.

```
% INPUT BUFFERING PROGRAM

BEGIN
  DBYTE NUMBER;
  PROCEDURE PCRLF=EXTERNAL($CD24);

  MODULE INPUTBUFFER=
  BEGIN
  PUBLIC
    CHAR PROCEDURE BUFFER;

  PRIVATE
    BYTE BUFSIZE=$7F,BUFPTR;
    BOOL EMPTY:=TRUE;
    CHAR ARRAY BUF[BUFSIZE];

    CHAR PROCEDURE BUFFER=
    BEGIN
      CHAR CH,CR="^M",BS="^H",CAN="^X";
      IF EMPTY THEN
      BEGIN
        BUFPTR:=$00;
        DO BEGIN
          READ CH;
          IF CH=BS THEN
            IF BUFPTR>$00 THEN BUFPTR:=BUFPTR-$01 ELSE
          ELSE IF CH=CAN THEN BUFPTR:=$00 ELSE
          IF BUFPTR<BUFSIZE OR CH=CR THEN
          BEGIN
            BUF[BUFPTR]:=CH;
            BUFPTR:=BUFPTR+$01;
          END;
        END UNTIL CH=CR;
        WRITE CHR($0A);
        EMPTY:=FALSE;
        BUFPTR:=$00;
      END;
      BUFFER:=BUF[BUFPTR];
      BUFPTR:=BUFPTR+$01;
      IF BUFFER=CR THEN EMPTY:=TRUE;
    END;
  END; % of MODULE

  PCRLF;
  WRITE "TYPE IN SOME HEX NUMBERS";
  DO BEGIN
    PCRLF;
    READ FROM BUFFER, NUMBER;
    WRITE DEC(NUMBER);
  END UNTIL NUMBER=$0000;
END.
```

The example is written as an entire program so that it may be typed in and run if desired.

The greatest asset that Whimsical has, is probably the ability to produce programs configured for any environment. The versatile I/O handling has already been discussed. Whimsical produces a contiguous block of code which has no separate runtime package. The code file can easily be put into ROM. There are several other aspects to a program which need attention before running it on a foreign target machine.

STACK LOCATION

Whimsical can use the stack in three different modes. Mode 1 is for use under a single user operating system such as FLEX. The stack pointer is loaded at the beginning of the program to be at the top of useable memory, which is determined from a memory end location. Mode 2 is for use in stand alone programs that need to initialize the stack pointer. It is set up at a defined absolute memory location. Mode 3 is for use where the stack pointer has already been set up before the program is run. Examples of directives for the three modes are:

```
Mode 1      ~STACK=[$CC2B]    load stack pointer from a
                              memend location.
Mode 2      ~STACK=$C000      load stack pointer with $C000
Mode 3                        default, existing SP used.
```

These directives must appear before the first begin if they are used. Mode 2 produces programs which are not re-entrant.

The direct page register is used for all globals. Because the DP register must be aligned on a 256 byte page boundary, it is necessary for Whimsical to move the stack pointer down to a boundary before any variable storage is allocated. In addition, Whimsical stores the old stack pointer so that it can be restored at the end of the program. The following runtime map should clarify the situation.

```
          ADDRESS  CONTENTS


    SP --->                 SP before allocation of variables
             xxFF  xx       } Old Stack Pointer stored here
             xxFE  xx       }
               ..
               ..
               ..
    DP --->  xx00  xx       DP points 256 below intial SP
             xxFF  xx       Array allocation continues
             xxFE  xx       downwards below DP
               ..
    SP --->    ..           SP during running of program
```

This picture of the stack shows its origin before any variable storage has been allocated. As storage space is allocated the stack pointer moves down.

When setting up the stack to an immediate location it is best to

set the SP to an address ending in 00. This will cause no memory wastage when the SP is set down to the nearest page boundary. Stack usage in Whimsical can be determined from the amount of declared storage and the runtime nesting of procedure calls. Note that each procedure call uses 4 bytes of stack space in addition to its parameters and locals.

CODE ORIGIN

The code origin can be set using the ORIGIN directive. If you wish to set the origin at the beginning of a program, the directive must precede the first BEGIN. If there is also a ~STACK directive the order does not matter. This will also set the transfer address. After that the origin directive may be used only between two procedure declarations (at lex level 0). If no origin is specified the code starts at $0000 as does the transfer address. Since the code is relocatable, it is often not necessary to use an origin, especially if the code is going into ROM.

e.g. ~ORIGIN=$C100

The code produced by the compiler is in a FLEX format binary file. The origin directive simply causes the load address for subsequent code to be different. Note that if you compile two non-contiguous blocks of code, those blocks are not relocatable relative to each other.

END OF PROGRAM PROCESSING

At the end of a program the code contained in the file WHIMSB09.BIN is appended. This file may be modified to do anything the programmer wishes. For example it could be reassembled as a jump back to the particular systems's monitor on which the program will be run.

If the program would never exit, the file may be left empty. The standard file supplied has the following code:

```
LDA #0       Reset Direct Page register
TFR A,DP
JSR $D403    Close all files
JMP $CD03    Return to FLEX
```

See section 3-6 for information on how to set up WHIMSB files.
If you wish to suppress the use of WHIMSB09 altogether then use an exclamation mark instead of the final full stop.
It is possible to put CODE statements after the final end and before the final fullstop or exclamation mark. e.g.

```
END;                % final end
CODE(JMP,$0000)!    % loop back to beginning
```

ROM INTERRUPT VECTORS

To incorporate ROM vectors it is necessary to change to the alternate code segment and set a new origin, and then use CODE statements for the vectors. Finally change back to the original CODE segment. You must change back to the original code segment even if you use a final ! because subroutines can still be appended to the end of the program. e.g.

```
END;
~USE CODE                 % switch to alternate seg
~ORIGIN=$FFF8             % origin for vectors
  CODE(LOC(IRQHANDLER)); % IRQ
  CODE(LOC(SWIHANDLER)); % SWI
  CODE(LOC(NMIHANDLER)); % NMI
  CODE($0000);           % RESET
~USE CODE                 % switch back to original
!
```

RUNTIME REGISTER USE

At runtime the 6809 registers have specific uses assigned to them. SP is used as the stack pointer in the conventional way. DP points to the programs global (permanent) variables as already discussed. U and Y are used for pointing to the variables allocated at lex levels 1 and 2 respectively. So, for example, if a procedure is invoked, U is set pointing to its parameters and locals. If a procedure which is declared inside that procedure is invoked, then Y is set pointing to its parameters and locals. X is used for general address calculations, such as those involved in passing parameters by reference. D and B are used for general arithmetic and data manipulation. X, D and B are all used temporarily at any time and so are normally considered free. DP, U and Y are in general not free. They must be saved if used in any external machine code.

ALTERNATE DATA ORIGIN

Sometimes it is useful to declare a group of variables to exist in a special part of memory rather than to be allocated space in the stack. For example you may have some non-volatile ram which you wish to use. Whimsical allows you to swap to an alternate data segment called the BASE segment. There you can set up a new data origin. Following variables are then allocated upwards in memory from this origin. Then you can swap back and forth between the stack and base segments to continue declarations where you left off. For example each module you write may have some ordinary variables and some base segment variables. The USE directive is used to change the current segment. You may say ~USE BASE to change to the base segment or ~USE STACK to get back to the stack segment. After the very first ~USE BASE you should have a ~BASE=$xxxx directive to set the initial base origin. Once set the base origin cannot be set again. It is not possible to use a ~BASE=$xxxx directive within a pre-compiled module.

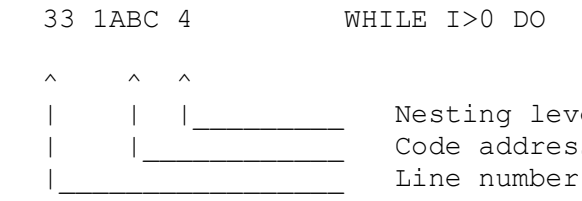
If you wish to allocate just a few variables here and there, the TEMP segment is provided for that purpose. The TEMP segment differs from the base segment in that the compiler does not continue allocation from the end of the last lot of TEMP variables. I.E. every ~USE TEMP directive must be followed by a ~TEMP directive to set its origin (see section 2-10).

Variables in the base and temp segments are not initialized to zero automatically. Arrays may be declared in these segments.

INFORMATION OPTION

If the I option is used either on the command line or in an

~OPTION directive then the listing contains the following information.



The first column contains consecutive line numbers. The second contains the hexadecimal code address of the code emitted so far. This column is the only means by which any runtime errors can be traced back to the source program. A runtime error message has the following form:

```
RUN TIME ERROR AT $xxxx yy
```

xxxx is the code address where the error occurred. It can be used in conjunction with a full listing to find the location of the error in the source code. yy is a decimal error number. If this number is less than 32 then it refers to a FLEX disk error number.

The addresses given in column two can be a few bytes out because of optimization which may have occurred subsequent to the line being printed.

The third column gives the nesting level for statements, modules and procedures. Refer to it if you are having trouble with matching BEGINs and ENDs etc.

The information option also causes the following information to be output at the end of a compilation.

SIMPLE GLOBALS

This is the total number of bytes required for the simple global variables in the direct page. It does not include arrays, file control blocks, nor variables allocated in the BASE or TEMP segments. It does not include variables local to a procedure as these are created only on entry to the procedure. It does include public and private simple variables within modules. This figure cannot exceed $100. The compiler reserves 4 bytes for its own use which is included in this figure for the main program.

TOTAL GLOBALS
This figure gives the number of bytes required for global variables including arrays and files. The figure for the simple globals is included in this figure.

TOTAL STATIC
This figure is similar to the previous figure but also includes the total bytes required for every procedure to be invoked simultaneously. Therefore if recursion is not to occur it gives the absolute maximum stack usage (except for temporary variables used in evaluating complex expressions).

TOTAL CODE
This is the total number of bytes produced. Note that the hex address given in a full listing may be a few bytes out because of subsequent optimizations which may have occurred.

BASE NEXT
This is the next address to be allocated in the base segment.

If two or more I's are specified either on the command line or in an option directive then any unused variables are listed at the end of each procedure and module and at the end of the program.

If three I's are specified then the inclusion of WHIMSBxx subroutine files will be reported.

CODE STATEMENT

The code statement allows inclusion of inline code at any point in a program. e.g.

```
CODE($13);    % SYNC INSTRUCTION
```

The brackets may contain any BYTE or DBYTE type data. It may not contain LOC intrinsics to get the address of procedures or data since these are not constants.

An externally assembled binary file can be included:

```
CODE FROM "filename";
```

PROCEDURES

External procedures in Whimsical are any routines which are already in machine language form. They may be existing routines or they may be written specifically to be included in a Whimsical program. Sometimes an existing routine, due to its incompatibility with Whimsical, may not be capable of being called directly. In such a case an assembly language routine must be written to interface to it.

The syntax for an external routine may have one of the following forms:

```
procedure head = EXTERNAL( hexaddress-specification);

procedure head = CODE( code list );

procedure head = CODE FROM external file specification;
```

The hex address specification is simply the address of the existing routine. e.g. $D406. A named DBYTE constant may be used instead of a DBYTE literal. If the address is put into square brackets then the external routine will be called using an indirect jump. For example if $D3F9 contains a vector to an output routine then the declaration,

```
PROCEDURE(DBYTE AB)=EXTERNAL([$D3F9]);
```

would enable it to be called.

The other two methods involve including a routine into a Whimsical program. The first simply lists the code bytes of the routine. For very simple routines this could be done by hand assembly. The best way however, is to write the routine in assembly language, then create a file of the listing as it is assembled. This file can then be processed by a program called WHIMCVT which will make it suitable for inclusion into a Whimsical program as a code list. An example is given at the end of this section. The process used to get that program is as follows:

Prepare assembly language source file "INTRFACE.ASM"

```
O,1.INTRFACE,ASM09,INTRFACE.ASM
WHIMCVT INTRFACE.OUT INTRFACE.TXT
Use this file as the basis for the procedure.
```

This method has the advantage that the resulting program is well documented. All the original assembly language source would have been included as comments.

The second method is easier to use. The file specification must refer to a binary code file in the FLEX binary format. All you have to do is assemble your routine in the normal way to get a .BIN file.

```
e.g.   PROCEDURE FMS(BYTE A; BYTE ARRAY FCB)=
         CODE FROM "0.FMS.BIN";
```

The file specification defaults to the extension .BIN and the working drive.

Some rules must be adhered to in external routines. Since the compiler cannot check that these rules are obeyed, great care must be taken.

Firstly, the registers DP, U and Y must not be changed. The registers A, B and X, on the other hand, are free for use. Secondly, if there is parameter passing, there are some rules on how they are to be passed. Basically, Whimsical can set up B or D or X before calling an external routine. This is done by passing a single parameter. It should be of type CHAR, SMALLINT, BOOLEAN or BYTE in order for it to be passed in register B. It should be INTEGER or DBYTE in order for it to be passed in register D, and it should be a LARGEINT to pass it in D and X as a concatenated 32 bit register. X may be set up to point to one of Whimsical's own variables. To do this you must pass that variable by reference. A useful example is in interfacing to the FLEX routine, FMS, which needs the X register pointing to an FCB. An array may be passed (arrays are always passed by reference) which would become the FCB outside Whimsical, but which would be totally accessible inside Whimsical. So a DIR utility may be written.

Values may be returned to Whimsical simply by declaring the external procedure as a typed procedure. Any of the simple types may be returned using the registers described above.

PARAMETER PASSING

The mechanism of parameter passing is discussed in this section along with a discussion of how a procedure works. It is strongly suggested that user written assembly language procedures use these same techniques because it is very important that these procedures work correctly. The compiler has no way of checking them.

In Whimsical object code, all variables and parameters are allocated storage on the stack. Those of lex level 0 (globals) are referenced by the DP register, those of lex level 1 by U and those of lex level 2 by Y.

When a procedure is invoked the parameters are passed either on the stack or in a register. In general all but the last parameter

are passed on the stack. The last parameter is passed in the B, D or X register. Passing in a register is more byte efficient because the parameter doesn't need to be pushed onto the stack before the procedure is called. However, there are not enough registers to pass all parameters in this way. The last parameter is passed in the accumulator (B or D or D & X depending on its size) if it is passed by value, or in X if it is passed by reference. This parameter is usually pushed onto the stack once inside the procedure in order to provide local storage for it. However, assembly language routines do not necessarily have to do this. Before a procedure can return it must remove any parameters which may be on the stack. If there were more than one then all except the last will be underneath the procedure's return address. They must still be removed and so the return address is first pulled into the X register, and then an LEAS instruction removes them followed by a JMP ,X to actually return. Another activity associated with a procedure invocation is the changing of the local block. This means that the old block pointer must be saved and a new one set to point to the new block. If the block is at lex level 1 the U register is involved and if at lex level 2 the Y register is involved. The old value of the register is pushed onto the stack inside the procedure and then the new register is set to point to the current local variables and parameters. Again assembly language procedures don't have to do this. For most small routines it is quite reasonable to address the locals relative to S (the stack pointer itself) and not to worry about setting up a separate pointer. An example of stack usage during a procedure call for the following procedure will be given. The setup shown is exactly as it would be in a Whimsical procedure:

```
BYTE PROCEDURE FN(DBYTE A; BYTE REF B; BYTE C)=
BEGIN
  BYTE D,E;  % LOCALS OF THE PROCEDURE
  DBYTE F;
  .
  .
  .
END;
```

Assume the procedure is at lex level 1, i.e. not nested in another procedure. The runtime stack usage during the running of the procedure would be as follows:

```
                item on stack              size      address
                     .
                     .
                     .
                     parameter A        (two bytes)  8,U
                     parameter B        (two bytes)  6,U
                     return address     (two bytes)  4,U
                     old U register     (two bytes)  2,U
                     parameter C        (one byte)   1,U
    new U     --->   return value FN    (one byte)   0,U
                     local variable D   (one byte)   -1,U
                     local variable E   (one byte)   -2,U
                     local variable F   (two bytes)  -4,U
                     local stack usage
                     .
                     .
```

This example also gives the number of bytes taken by each item in

the stack and their offsets that they would have from U when accessed in the body of the procedure at runtime. Not all the items need be used in any given procedure; in fact the only one always used is the return address. All the work in setting up and breaking down this stack is done inside the procedure except for the first two parameters and the return address. They are stacked before the procedure is called.

The following example is an assembly language routine designed as a general interface between Whimsical and an existing machine language routine. It illustrates the method of referring to parameters which have been passed by reference.

```
BYTE PROCEDURE INTRFACE(BYTE REF A,B; DBYTE REF X,Y,U)=
CODE(   % CODE LIST BEGINS

                  %          ORG       $0000
                  % ROUTINE  EQU       $A000
$34,$70,          %          PSHS      X,Y,U     SAVE WHIM'S REGS
                  %                               .AND LAST PARAM
$EE,$F4,          %          LDU       [,S]      LOAD U
$10,$AE,$F8,$08,  %          LDY       [8,S]     LOAD Y
$AE,$F8,$0A,      %          LDX       [10,S]    LOAD X
$E6,$F8,$0C,      %          LDB       [12,S]    LOAD B
$A6,$F8,$0E,      %          LDA       [14,S]    LOAD A
$BD,$A0,$00,      %          JSR       ROUTINE   GO TO ROUTINE
$34,$01,          %          PSHS      CC        SAVE CC
$A7,$F8,$0F,      %          STA       [15,S]    SAVE A
$E7,$F8,$0D,      %          STB       [13,S]    SAVE B
$AF,$F8,$0B,      %          STX       [11,S]    SAVE X
$10,$AF,$F8,$09,  %          STY       [9,S]     SAVE Y
$EF,$F8,$01,      %          STU       [1,S]     SAVE U
$35,$04,          %          PULS      B         GET CC AS RET VAL
$10,$AE,$62,      %          LDY       2,S       GET WHIM'S REGS
$EE,$64,          %          LDU       4,S
$AE,$66,          %          LDX       6,S       GET RET ADDR IN X
$32,$E8,$10,      %          LEAS      16,S      CLEAN ENTIRE STK
$6E,$84           %          JMP       ,X        TO WHIMSICAL
                  %
                  %          END

 );  %END OF CODE LIST
```

The routine allows passing of parameters to or from any of A,B,X,Y,U. In addition it returns to Whimsical the condition code register returned from the existing routine. Only the DP register may not be altered by the existing routine. If all the registers are not used or altered, then the interface routine could be simplified somewhat. Remember that if only the B or D registers need to be passed then it is not necessary to use an interface routine at all.

RUNTIME ERROR HANDLING

Since Whimsical must cater for any runtime environment, error handling, like I/O, must be user defineable. If a program is being written, for example, as a controller, then any runtime errors cannot simply cause an error message and stop. Even in conventional programs it is sometimes very useful to be able to detect errors and take some appropriate recovery action.

Runtime errors come under two main catagories. There are the ones to do with disk operations and there are the ones to do with arithmetic overflow or array index out of bounds. Disk error checking cannot be switched off whereas the other types of error checking can be. Array bounds checking is never done for arrays which are parameters of a procedure since the size of such arrays is not known at compile time.

In the command line options a +R can be used to switch on runtime checking in the program. Alternatively the directive ~OPTION R can be used to turn runtime checking on and ~OPTION -R can be used to turn it off. ~OPTION /R will set it to the same as it was before the last R or -R.

If the command line option of R or -R is used it will act globally and all embedded options will be ignored.

When runtime errors are found they are sent by default to a subroutine which outputs a message to the system console and stops. This subroutine can be found in the file WHIMSB03.BIN. (The source is in WHIMSB03.ASM.) As supplied this subroutine uses the FLEX RPTERR function to report a disk error or outputs a simple runtime error message of the form:

```
RUN ERR AT $xxxx yy
```

This file may be modified to suit the system on which the program will be running. See section 3-6 for information on how to set up WHIMSBxx files.

When integer variables overflow as a result of an arithmetic operation it is sometimes useful to saturate that variable instead of (or as well as) generating a runtime error. For example if "I" is a SMALLINT whose value is currently 125, and 4 is added to it, its value will become 127, the maximum possible. Similarly the variable could saturate in the negative direction giving a value of -128. The corresponding maximum and minimum values for INTEGER types are 32767 and -32768 and for LARGEINT types is 2147483647 and -2147483648. If saturation is not used then operations will give unpredictable results when they overflow. It is important to decide exactly what you want in various parts of your program - saturation, runtime error checking, neither or both. Saturation causes more code to be produced by the compiler so facilites are provided for switching it on and off. These are the directives: ~OPTION S and ~OPTION -S. These options are used in the same way as the R and -R options and can be overridden on the command line.

THE TRAP STATEMENT

This statement has the following form:

```
TRAP [TO error-procedure-name] [FROM] statement
```

Any runtime errors which occur within the contained statement will be handled by the specified procedure. In that procedure the programmer can try to take appropriate action to recover from the error. The error procedure must have one BYTE parameter to which will be passed the runtime error number. Example:

```
PROCEDURE ERRORHANDLER(BYTE ERRNO)=
BEGIN
  BYTE ARRAY FCB=[$01];
  PROC RPTERR(BYTE ARRAY FCB)=EXTERNAL($CD3F);
  IF ERRNO<=$20 THEN
  BEGIN
    FCB[$01]:=NO;
    RPTERR(FCB);
    STOP;
  END ELSE
  CASE ERRNO OF
  BEGIN
    $20: WRITE "^M^JArithmetic overflow or divide by zero";
    $21: WRITE "^M^JToo large for TRIM, FIX or INT";
    $22: WRITE "^M^JNegative step in FOR statement"; STOP;
    $23: WRITE "^M^JArray subscript out of bounds"; STOP;
    ELSE: REPORTERROR(NO); STOP;
  END;
END;
```

This example performs essentially the same function as the WHIMSB03 subroutine. Errors such as arithmetic overflow cause a message to be written to the console and then the program continues running. Any disk file error will be reported in written form from the ERROR.SYS file of FLEX and program execution will halt. Undefined errors or user calls to this procedure with a new error number will invoke the intrinsic procedure REPORTERROR. This is the routine programs use when errors are not trapped.

Error traps are useful to detect the presence or otherwise of a file on disk by opening it and trapping for the error.

The TRAP statement automatically causes runtime error checking to be turned on in the contained statement. After the trap statement the runtime error trap status is set back to its previous state.

If the "TO procedure-name" part of the trap statement is ommited then the default error handler will be used. This essentially becomes another way to turn on runtime error checking. This time though, it cannot be turned off by -R options, even on the command line. The trap statement is therefore used when it is a vital part of the program. For example to test if a file exists try to open it with the open statement trapped to a procedure to set a flag. The other runtime error handling options are more for making sure of correct operation of the program.

THE TRAP DIRECTIVE

The trap directive is similar to a TRAP statement except that it doesn't just affect one statement but rather affects all following code. Example:

```
~TRAP TO ERRORHANDER
```

If the procedure ERRORHANDLER is the first one declared in the program, and it is immediately followed by this directive then the entire program will have error trapping. Note that the TRAP directive does not automatically turn on runtime checking as the TRAP statement does. This must be done with ~OPTION R.

INTERRUPT ROUTINES

Special procedures can be written in Whimsical to service interrupts. These procedures differ from normal procedures only in the fact that they return using an RTI instruction instead of an RTS. They are declared by putting the reserved word "INTERRUPT" before the procedure head.

Generally an interrupt procedure would access global variables in order to do its job. The programmer must be aware that interrupt procedures can run at any time and use suitable locks on data structures where necessary.

The following MODULE will give you an idea of how interrupts can be handled with ease by a WHIMSICAL program. The module will provide a type-ahead buffer for a keyboard (or other serial input device) under interrupt control. A sample program is included on the disk.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MODULE TO DEMONSTRATE USE OF INTERRUPTS  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

MODULE=
BEGIN
PUBLIC
  CHAR PROCEDURE INPUT;
  PROCEDURE STARTKEY;
  PROCEDURE ENDKEY;

PRIVATE
  BYTE CONTROLPORT($E010); % ACIA PORTS FOR SWTPc SYSTEM
  CHAR DATAPORT($E011);
  DBYTE IRQVECTOR($DFC8),
        OLDIRQROUTINE;
  BOOL EMPTY:=TRUE;
  SMALLINT HEAD, TAIL;
  CHAR ARRAY BUFFER[79];

  INTERRUPT PROCEDURE KEY=
% --------- --------- ---
  BEGIN
    CHAR DUMMY;
    IF (CONTROLPORT AND $01)<>$00 THEN % FROM ACIA
    BEGIN
      IF TAIL=HEAD AND NOT EMPTY THEN
      BEGIN
        WRITE "^G"; % BELL IF FULL
        DUMMY:=DATAPORT; % CLEAR INTERRUPT
      END ELSE
      BEGIN
        BUFFER[TAIL]:=DATAPORT;
        TAIL:=TAIL+1;
        IF TAIL=80 THEN TAIL:=0;
        EMPTY:=FALSE;
      END;
   END;
  END; % OF KEY
```

```
  CHAR PROCEDURE INPUT=
% ---- ---- -----
  BEGIN
    WHILE EMPTY DO ; % WAIT FOR KEY
    DISABLEIRQ; % WHILE WE ACCESS BUFFER VARIABLES
    INPUT:=BUFFER[HEAD];
    HEAD:=HEAD+1;
    IF HEAD=80 THEN HEAD:=0;
    IF HEAD=TAIL THEN EMPTY:=TRUE;
    ENABLEIRQ;
  END;

  PROCEDURE STARTKEY=
  BEGIN
    OLDIRQROUTINE:=IRQVECTOR;
    IRQVECTOR:=LOC(KEY);
    CONTROLPORT:=$91;   %ENABLE INTERRUPTS FROM ACIA
    ENABLEIRQ;
  END;

  PROCEDURE ENDKEY=
  BEGIN
    DISABLEIRQ;
    CONTROLPORT:=$11;   %DISABLE INTERRUPTS FROM ACIA
    IRQVECTOR:=OLDIRQROUTINE;
  END;

END;
```

SUBROUTINE LIBRARY FILES

The Whimsical compiler consists of the command file "WHIM.CMD" and its error messages "WHIM.ERR". These are the only two files which you must have on your system disk when compiling. During compilation of your program the compiler may need to put subroutines into your object code to perform various complex tasks such as real arithmetic. Since all these subroutines are also contained within WHIM.CMD the compiler reads them from within itself and puts them into your program. This avoids having to read them from disk.

Sometimes you may wish to modify one or more of these subroutines or even rewrite them yourself. To allow you to have your versions of the routines put into your program and not the compilers, there is the "W" option. When this option is used the compiler will first search the working drive for each required subroutine file. If the file is found it is read in and inserted into your program. If the file is not found the compiler will use its own subroutine which is in memory.

There are about 60 subroutines which are contained in approximately 36 packages. Obviously some of the packages contain more than one subroutine but most of them are a single subroutine. The packages are in files named WHIMSB00, WHIMSB01 etc. The filenames are numbered with two hex digits. The source code is provided for some of these files so that you can modify them if you wish. A complete list of the subroutines may be found at the end of the section.

Since some of the packages have more than one subroutine there must be a set of vectors at the beginning of each file to point to the entry points of the various routines. The compiler reads these vectors and remembers them. They are not put into your program. The first subroutine always starts from the beginning of the code. The others are specified by the list of FDB's terminated with a $0000. After the FDB's the code must be re-origined to $0000 so that all the FDB's will represent pure offsets into the code. The code must of course be relocatable. The following example comes from the file WHIMSB00.ASM.

```
.WHIMSICAL COMPILER SUBROUTINES
.FLEX CONSOLE INTERFACE HANDLERS
.WHIMSB00.ASM
.ASSEMBLE USING ASM09

*FLEX EQUATES

PUTCHR   EQU       $CD18     FLEX PUT CHARACTER TO CONSOLE
GETCHR   EQU       $CD15     FLEX GET CHARACTER FROM CONSOLE

*SUBROUTINE LOCATION VECTORS

         FDB       OUTPUT    .OUTPUT CHARACTER TO CONSOLE
         FDB       $0000     END OF VECTORS

         ORG       $0000

*INPUT CHARACTER FROM CONSOLE ROUTINE


INPUT    JSR       GETCHR    GET CHARACTER FROM CONSOLE
         TFR       A,B       RETURN IN B
         RTS

*OUTPUT CHARACTER TO CONSOLE ROUTINE

OUTPUT   TFR       B,A       GET OUTPUT CHARACTER TO A
         JMP       PUTCHR    OUTPUT TO CONSOLE

         END
```

Please refer to the actual source files supplied for further examples.

The first of the two hex digits in the subrouitne numbers indicate its broad grouping as follows:

0 Console and Disk I/O
1 Miscellaneous
2 I/O conversions
3 Smallint and Integer
4 Largeint
5 Real

Following is a complete list of the subroutine numbers and the file numbers which contain them:

```
00 GETCHR    01
01 PUTCHR
02 STATIN
03 STATOUT
04 INIT
05
06 CREATE    02
07 OPEN
08 READ
09 WRITE
0A EOF
0B CLOSE
0C CLOSED
```

```
0D
0E RUNTIMERR 03
0F
10 CLRLOOP1  04
11
12 CLRLOOP2  05
13
14 CHECK1    06
15 SATCH1
16
17 CHECK2    07
18 SATCH2
19
1A CHECK4    08
1B SATCH4
1C
1D PROGEND   09
1E
1F
20 STRNGOUT  0A
21
22 DBYTEOUT  0B
23 BYTEOUT
24
25 INTOUT1   0C
26 INTOUT2
27
28 GETBUF    0D
29
2A HEXIN     0E
2B
2C SEX2      0F
2D
2E INTIN2    10
2F
30 UNPACK2   11
31 PACK2
32
33 UNPACK4   12
34 PACK4
35
36 MULT1     13
37
38 MULT2     14
39
3A DIV1      15
3B
3C DIV2      16
3D
3E TRIM2     17
3F
40 SUB4      18
41 ADD4
42
43 NEG4      19
44
45 MUL4      1A
46
47 DIV4      1B
48 MOD4
```

```
49
4A OUT4      1C
4B
4C IN4       1D
4D
4E TRIM4     1E
4F
50 SUBTRACT  1F
51 ADDITION
52 MULTIPLY
53 DIVIDE
54 FLOAT
55
56 INPUT     20
57
58 OUTPUT    21
59
5A FIX       22
5B INT
5C
5D COMPARE   23
5E
5F ASL REAL  24
60
61 ASR REAL  25
62
```

MAKING A CROSS REFERENCE OR A GLOBAL SYMBOL TABLE

Making a cross reference or a global symbol table requires the use of the TSC sort/merge package. In the case of a cross reference the Whimsical compiler will create a file containing an entry for every occurrence of every global identifier. In the case of a symbol table the compiler will create a file containing an entry for every public global. This is then sorted and formatted by the sort package.

A +G option on the command line forces Whimsical to create a symbol file with an extension of .SYM. The symbol file has the following variable length fields separated by commas:

IDENTIFIER,ADDRESS,SIZE,MODULE NAME,TYPE

A +X option on the command line forces the compiler to create a cross reference file with an extension of .XRF. The cross reference file has one of the two following variable length fields:

IDENTIFIER,LINE NUMBER,*,MODULE NAME,TYPE
IDENTIFIER,LINE NUMBER,- or + or /,MODULE NAME,PROCEDURE NAME

The asterisk indicates the entry is a declaration. If that field is a minus sign then the entry represents a reference to the identifier. If it is a plus it represents an assignment and if it is a slash it represents a pass by reference. In these cases the type field is used, instead, as the name of the procedure which contains the identifier.

The sort package is used with the following input keys:

(1)1-10,R(2)1-5

The file is therefore sorted by identifier but the entries for any given identifier are kept in original order. That reflects the order they appear in the original source file. When sorting the global symbol table file, it is only necessary to have one sorting key.

The following output keys are suggested to get a reasonably formatted cross reference or symbol table:

(1)1-E,@20,(2)1-E,@26,(3)1-E,@32,(4)1-E,@42,(5)1-E

The module and procedure within that module is given to identify where the variable is accessed as well as the line number. The line numbers correspond with those of the source file as a whole with any ~INCLUDE files in. Therefore it is best to make a full listing of the program at the same time as the cross reference is made.

Any pre-compiled modules in the program will have their publics and external references in the cross reference. But no information can be given about the location within the module. For this reason it is best to change to ~INCLUDES for all modules to make the

cross reference complete. Conditional compiling directives can be used to quickly change to includes as follows:

```
BOOLEAN INCLUDES=FALSE;

~IF INCLUDES THEN
~INCLUDE "MODULE1"
~INCLUDE "MODULE2"
~INCLUDE "MODULE3"
~ENDIF

~IF NOT INCLUDES THEN
MODULE MODULE1=CODE FROM "MODULE1";
MODULE MODULE2=CODE FROM "MODULE2";
MODULE MODULE3=CODE FROM "MODULE3";
~ENDIF
```

If any modules are not named then the module name field will be blank.

The addresses generated in the symbol table are accurate because the symbol table is not output until compiling is complete. The addresses for variables and arrays and files refer to the direct page. The leading FF should be ignored. In the case of arrays and files it is the address of a two byte pointer to its base. The size refers to the size of arrays. For simple constants the address field is used for the value. For largeint or real constants, the size field contains the high word of the value.

USING THE SORT PACKAGE

It is suggested that the "I" utility be used to specify an input text file which can be easily modified to suit individual requirements. Two text files are provided called SORTXREF and SORTST. They are used as follows:

```
I,0.SORTST SORT PROGRAM.SYM
I,0.SORTXREF SORT PROGRAM.XRF
```

The contents of SORTST is given below.

```
YSYMBOLS
1V
',
I(1)1-10

(1)1-E,@20,(2)1-E,@26,(3)1-E,@32,(4)1-E,@42,(5)1-E

NNS
```

The "Y" means "Yes output to disk". The output filename is specified next. The "1" is the drive to use as the intermediate work drive. The "V" means variable length records. The carriage return is the record terminator character. The "'," specifies the field separator character. The "I" specifies that the output is to come from the input file. Next come the input keys terminated by two carriage returns. Then come the output keys. Finally there is an "N" which means no further options, another "N" which means don't save the parameter file and an "S" which means proceed with the sort. The SORTXREF input file is very similar. By changing the

final "N" to a "Y" the sort program will create a parameter file for PSORT. This will allow the output filename to be specified on the command line.

RESERVED WORDS

BEGIN	END	IF	THEN	ELSE
WHILE	DO	UNTIL	CASE	STOP
FOR	DOWNTO	STEP	TRAP	
AND	OR	XOR	NOT	MOD
BYTE	DBYTE	BOOLEAN	CHAR	BOOL
INTEGER	SMALLINT	LARGEINT	POINTER	
FILE	ARRAY	REF	REAL	PROC
PROCEDURE	EXTERNAL	INTERRUPT	FORWARD	CODE
TRUE	FALSE	TO	FROM	AS
READ	WRITE	CREATE	OPEN	CLOSE
CLOSEDELETE				

ENABLEIRQ	DISABLEIRQ	ENABLEFIRQ	DISABLEFIRQ	
HIBYTE	LOBYTE	COMBINE		
ASL	ASR			
CHR	ASC	HEX	DEC	
HIDBYTE	LODBYTE	LCOMBINE	RCOMBINE	
EXTEND	TRIM	FIX	INT	FLOAT
EOF	LOC	REPORTERROR		

WHIMSICAL STATEMENTS SUMMARY

```
ASSIGNMENT STATEMENT
   variable:=expression

READ STATEMENT
   READ [FROM file] variable [,variable . . .]

WRITE STATEMENT
   WRITE [TO file] expression [,expression . . .]

COMPOUND STATEMENT
   BEGIN statement [;statement . . .] END

IF STATEMENT
   IF boolean expr THEN statement [ELSE statement]

CASE STATEMENT
   CASE expression [OF]
   BEGIN
     constant list: statement; [statement; . . .]
     constant list: statement; [statement; . . .]
     constant list: statement; [statement; . . .]
       .
       .
     [ELSE: statement; [statement; . . .]]
   END;


WHILE STATEMENT
   WHILE boolean expression DO statement

DO STATEMENT
   DO statement [;statement. . .] UNTIL boolean expression

FOR STATEMENT
   FOR variable:=expr TO expr [STEP expr] DO
       statement
   FOR variable:=expr DOWNTO expr [STEP expr] DO
       statement

STOP STATEMENT
   STOP

CREATE STATEMENT
   CREATE file AS external filename

OPEN STATEMENT
   OPEN file AS external filename

CLOSE STATEMENT
   CLOSE file

TRAP STATEMENT
   TRAP [TO procedure] [FROM] statement

CODE STATEMENT
   CODE(code list)
   CODE FROM "filename"
```

Interrupt mask group.

```
PROCEDURE ENABLEFIRQ;
PROCEDURE DISABLEFIRQ;
PROCEDURE ENABLEIRQ;
PROCEDURE DISABLEIRQ;
```

Byte and Dbyte manipulation group.

```
BYTE PROCEDURE LOBYTE(DBYTE parameter);
BYTE PROCEDURE HIBYTE(DBYTE parameter);
DBYTE PROCEDURE COMBINE(BYTE parameter1,parameter2);
```

Integer extend and trim group.

```
INTEGER PROCEDURE EXTEND(SMALLINT parameter);
LARGEINT PROCEDURE EXTEND(INTEGER parameter);
SMALLINT PROCEDURE TRIM(INTEGER parameter);
INTEGER PROCEDURE TRIM(LARGEINT parameter);
```

Type converting intrinsics.

```
BYTE PROCEDURE ASC(CHAR parameter);
CHAR PROCEDURE CHR(BYTE parameter);
BYTE PROCEDURE HEX(SMALLINT parameter);
DBYTE PROCEDURE HEX(INTEGER parameter);
SMALLINT PROCEDURE DEC(BYTE parameter);
INTEGER PROCEDURE DEC(DBYTE parameter);
LARGEINT PROCEDURE FIX(REAL parameter);
LARGEINT PROCEDURE INT(REAL parameter);
REAL PROCEDURE FLOAT(LARGEINT parameter);
DBYTE PROCEDURE HIDBYTE(LARGEINT parameter);
DBYTE PROCEDURE LODBYTE(LARGEINT parameter);
DBYTE PROCEDURE HIDBYTE(REAL parameter);
DBYTE PROCEDURE LODBYTE(REAL parameter);
LARGEINT PROCEDURE LCOMBINE(DBYTE parameter1, parameter2);
REAL PROCEDURE RCOMBINE(DBYTE parameter1, parameter2);
```

Shifting intrinsics.

```
BYTE PROCEDURE ASR(BYTE parameter);
DBYTE PROCEDURE ASR(DBYTE parameter);
SMALLINT PROCEDURE ASR(SMALLINT parameter);
INTEGER PROCEDURE ASR(INTEGER parameter);
LARGEINT PROCEDURE ASR(LARGEINT parameter);
BYTE PROCEDURE ASL(BYTE parameter);
DBYTE PROCEDURE ASL(DBYTE parameter);
SMALLINT PROCEDURE ASL(SMALLINT parameter);
INTEGER PROCEDURE ASL(INTEGER parameter);
LARGEINT PROCEDURE ASL(LARGEINT parameter);
```

Miscellaneous intrinsics.

```
DBYTE PROCEDURE LOC(anytype REF parameter);
BOOL PROCEDURE EOF(any type FILE parameter);
```

Sometimes, since Whimsical is a recursive descent compiler, it will give more than one error message as a result of a single error. In this case fixing the first error usually gets rid of the remaining error messages. An arrow is printed below where the compiler first picks up an error. This may be somewhat later than where the actual error is. For example, a common error is a missing semicolon. Often this error is not detected until the beginning of the next line.

It is possible for some other error numbers to occur which are flagged as compiler errors. You should refer the error, along with all relevant details and a source listing of your program to the supplier.

0=UNDECLARED IDENTIFIER
: Could be a mis-spelt identifier or you may be trying to use one which is declared locally or privately inside a procedure or module from outside.

1=SYMBOL TABLE FULL
: The symbol table can hold 640 symbols. Most declarations require one entry but procedure declarations require one entry plus one for the return value if there is one and one for each parameter. If this error occurs then you should try to put more variables inside procedures as locals or inside modules as privates. Then they will be removed from the symbol table when the procedure or module is compiled.

2=IDENTIFIER TABLE FULL
: The identifier table holds all the names for symbols in the symbol table. It has room for 3072 characters. The identifiers for formal parameters of a procedure are removed from the identifier table once the procedure is compiled even though the entries remain in the symbol table. Follow the suggestions under the symbol table full error or reduce the size of your global identifiers.

3=VACUOUS CONTRUCTION

4=INVALID STATEMENT START
: This error is most often caused when the compiler gets itself out of step due to a previous error. Commonly an error in the declarations will make the compiler think it has come to the end of them.

5=VITIATED MODULE OR BINARY FILE
: While linking in a pre-compiled module or reading in some binary code for a procedure the compiler found an invalid record type. This probably means that the file is corrupt.

6=IDENTIFIER DECLARED BEFORE
: The identifier has been previously declared within this block.

7=NOT IMPLEMENTED

The feature you are trying to use is not implemented. REALs and LARGEINT cannot be specified at an absolute indirect address, and neither can files. Arrays cannot be assigned initial values.

8=CAN'T ASSIGN TO CONSTANT

The variable you are assigning to is a constant.

9=ELSE MUST BE AT END OF CASE

A case statement must have just one else and it must be the last case.

10=EXPRESSION IS OF WRONG TYPE

11=OPERAND EXPECTED

A constant, variable, number, procedure, intrinsic, IF clause or "(" is expected as an operand of an expression.

12=ASININE CONSTRUCTION

A construction doesn't make any sense such as taking the LOC of a scalar constant.

13=NUMBER CONSTANT EXPECTED

A BYTE, DBYTE, SMALLINT or INTEGER is required to specify an array size.

14=ASSIGNMENT EXPECTED

The compiler has begun compiling a statement which begins with a variable. The ":=" operator is expected for the assignment statement.

15=IDENTIFIER EXPECTED

16=CONSTANT EXPECTED

17=PUBLIC MUST BE REDECLARED FIRST

A public procedure or constant array must not be used until after it is redeclared in full. A procedure can be used if it is declared forward. A constant array can be declared in full in the publics section in which case it is available for use immediately but cannot subsequently be redeclared.

18=USE AN EXTEND

Very rare but can happen if you use a small constant at the beginning of an expression or sub-expression and the compiler does not know its type yet. The compiler may want you to extend it first to make it correctly match an INTEGER or LARGEINT with which it is being added, subtracted or divided. e.g. 1-L where L is a large integer will need to be rewritten EXTEND(EXTEND(1))-L. The compiler can handle multiply okay since it is commutative. Note that a small constant is automatically extended if the compiler already knows the type from the context. e.g. L-1 compiles okay.

20=STRING EXPECTED

23=STACK OVERFLOW HAS OCCURRED

The compiler checks that the stack has not "grown" down and abliterated the subroutines which reside on the end of itself. This check is done at the end of compilation.

24=INVALID TYPE

25=TYPE MISMATCH
Two operands or expression subparts are incompatible or an expression does not match the variable to which it is being assigned or passed to as a parameter. The arrow will point to the end of the expression which does not match.

26=CONSTANT EXPRESSION OVERFLOW
In evaluating a constant expression at compile time, there was an arithmetic overflow.

27=BRANCH TABLE FULL
The branch table contains entries for all the unstructured branches in a program. i.e procedure calls, subroutine calls, the STOP or EXIT branches and PC relative references to constant arrays. As code is written to disk (and therefore cannot be optimised) the corresponding entries are deleted from the branch table. The table holds 256 entries for the last 1 & 1/2 Kbytes of code which is kept in a code buffer. It is rare that this table becomes full but if it does there must be a very high concentration of these jumps in the code buffer. For example there might be many procedure calls or many calls to arithmetic subroutines. Try putting two consecutive ~USE CODE directives after a procedure declaration. This will force the code buffer and the branch table to be flushed.

28=IDENTIFIER IS OF WRONG TYPE
The identifier used as the loop control in a FOR statement must be a BYTE, DBYTE, SMALLINT, INTEGER or LARGEINT. An identifier being read in a READ statement cannot be of type boolean.

29=THIS DIRECTIVE CANNOT BE USED HERE
A ~STACK directive must be used before any code is produced by the compiler. An ~ORIGIN directive can only be used directly before or after a lex level 0 procedure. An ~ORIGIN or ~BASE directive cannot be used in a pre-compiled module. A ~USE directive cannot be used at other than lex level zero (not inside procedures).

30=FORWARD OR PUBLIC PROC NOT REDECLARED
A forward procedure or a public procedure has been declared but the actual declaration for the procedure has not been found within the block.

31=INVALID CHARACTER IN INPUT FILE
The input file contains an invalid control character or a delete character.

32=INVALID CONTROL CHARACTER
The carat operator in a string may only be followed by the characters "A" through "_". Other characters will not make valid control characters. The ^@ is not allowed in strings

in a WRITE statement because a NULL is used to terminate the string.

33=CAN'T LINK MODULE INTO MODULE
A module may only be linked into the main routine of a program. It may not be linked inside another module or inside a procedure. This only applies to pre-compiled modules.

34=PROCEDURE DECLARATION NESTED TOO DEEP
Procedures can only be nested to lex level 2.

35=DOES NOT MATCH FORWD OR ENTRY DECLARATION
An procedure is being declared whose identifier already exists in the symbol table. Either the identifier has already been reused for a different purpose or the declaration head does not match.

36=PARAMETER MISMATCH
A procedure is being invoked with the wrong number or type of parameters according to its formal declaration.

37=THIS PROCEDURE MUST BE TYPED
An untyped procedure is being used in an expression or on the right hand side of an assignment.

38=INCLUDE NESTED TOO DEEP
Includes may only be nested two deep. This is because a separate FCB must be allocated for each level.

40=LEFT PARENTHESIS EXPECTED '('

41=RIGHT PARENTHESIS EXPECTED ')'

42=INVALID ARRAY INDEX
An array index must be of type BYTE, DBYTE, SMALLINT or INTEGER. A constant array index may be out of bounds.

43=HOLE BINARY FILE
A binary file being used for a procedure cannot contain "holes" in its memory map. That is a record's load address must follow on from the end of the previous record.

44=COMMA ',' EXPECTED

46=NO FINAL FULLSTOP
The compiler thinks it has gotten to the end of the program. This is usually caused by too many ENDs in the program.

47=COMPILER DIRECTIVE EXPECTED
The "~" character must be followed by a valid compiler directive. See section 2-10.

48=MISSING ~ENDIF
An ~IF directive has been used for which there is no match ing ~ENDIF

49=INVALID OPTION
An invalid option has been specified on the command line

or in an ~OPTION directive. The E, and B options can only be used on the command line.

50=ILLEGAL DECLARATION
A module may not be declared within the public or external declarations of another module. An external constant array must be declared as a null array.

51=ERROR PROCEDURE IS NOT PROPER FORMAT
Error handling procedures must be untyped and they must have just one parameter of type byte to which the error number is passed.

52=FILENAME EXPECTED

53=CAN'T PASS A CONST BY REFERENCE

54=NUMBER OR STRING TOO BIG
The integer constant is bigger than a LARGEINT can handle or the string is too long for use as a filename.

55=INPUT LINE TOO BIG
Input lines must be less than 256 characters.

56=INVALID TYPE FOR CASE INDEX
Case types must be single byte sized. i.e. BYTE, SMALLINT or CHAR.

57=CASE INDEX DOESN'T MATCH
The constant type is not compatible with the case's expression's type.

58=COLON ':' EXPECTED

59=SEMICOLON ';' EXPECTED

60=TOO MANY GLOBALS FOR DIRECT PAGE
All global variables are allocated in the direct page. This includes publics and privates of modules. Global arrays require two bytes to be allocated in the direct page. If the direct page becomes full, then either try to make some variables local inside procedures or try to use variables for more than one purpose or move some variables to a separate allocation base using the ~USE BASE directive. (See section 2-10).

61=EQUALS '=' EXPECTED

62=INVALID COMMAND LINE
An invalid filename or option is on the command line. The arrow does not necessarily point to the error depending on the position of the command on the command line.

63=UNEXPECTED END OF SOURCE
The compiler has reached the end of the source file when it hasn't finished. This is usually caused by having too many BEGINs.

65=EXTERNAL DECLARATION DOESN'T MATCH
An external declaration doesn't match the actual

declaration in type, value, address or parameters.

66=EXTERNAL DECLARATION NOT FOUND
An external declaration for which there is no actual declaration previously declared. Usually one or other of the declarations is mis-spelled.

67=ADJUSTMENT TABLE FULL
The adjustments table is used for marking parts of the code which must later be adjusted. In a program this only applies to calls to subroutines which aren't yet included in the program. The subroutines specified in the adjustments table are included at the end of each procedure or pre-compiled module so the adjustments table could only be filled up if there is a very large procedure or module with many calls to subroutines not already included. Try rearranging the order of the procedures or modules so that the commonly called subroutines get included earlier. Alternatively try splitting the procedure or module up into smaller ones.

During pre-compiling of a module the adjustment table is also used for references to public, external and private variables, calls to external procedures and other miscellaneous functions. Its entries are removed when the code to which they apply is written to disk. The code buffer is 1 & 1/2K. The adjustments table can handle 240 entries. It should not become full during pre-compiling because if it does code is flushed to disk in order to get rid of some entries.

68=MODULE NOT ALLOWED INSIDE PROCEDURE
A pre-compiled module cannot be linked inside a procedure. It can only be linked at lex level 0.

69=INCOMPATIBLE MODULE VERSION
The module being linked was compiled by an older version of the compiler. The module must be recompiled with the new compiler.

91=LEFT SQUARE BRACKET EXPECTED

93=RIGHT SQUARE BRACKET EXPECTED

97='BEGIN' EXPECTED

100='THEN' EXPECTED

101='ELSE' EXPECTED

103='DO' EXPECTED

104='UNTIL' EXPECTED

113='PROC' EXPECTED

120='TO' EXPECTED

129=MUST BE A DISK FILE
The OPEN, CREATE, CLOSE and CLOSEDELETE statements and the

EOF intrinsic must have a file identifier as a parameter.

131=I/O PROCEDURE IS NOT CORRECT FORMAT
An I/O procedure used in a WRITE statement must be untyped and have one parameter. The type of the parameter must be either CHAR or BYTE. A procedure used in a read statement must be typed and it must have no parameters. The type must be CHAR or BYTE.

134='FROM' EXPECTED

135='AS' EXPECTED

RUNTIME ERRORS

The runtime error numbers are the same as those in the FLEX manual with the following additions.

$20 ARITHMETIC OVERFLOW
    DIVISION BY ZERO
$21 INTEGER TOO LARGE FOR TRIM
$22 NEGATIVE STEP VALUE IN FOR NOT ALLOWED
$23 ARRAY SUBSCRIPT OUT OF BOUNDS

THE 6801 & 6301 VERSIONS OF WHIMSICAL

The 6801 version of the compiler differs from the 6809 version on a few minor points.

FILES INVOLVED

The name of the 6801 version is WHIM1.CMD and its associated error message file is WHIM1.ERR. There is an extra file which must be kept available called WHIM1SUB.BIN which is a random file containing all the 6801 runtime subroutines. All three of these files should be put on the system drive.

SUBROUTINES FILE

The 6809 version of the compiler contained all the runtime subroutines within the compiler itself. The 6801 version, being a cross compiler, gets the 6801 subroutines from the file WHIM1SUB.BIN. This is a random file so that the compiler can quickly extract the subroutines it wants for inclusion in the target program. It is opened only once per compilation. The working drive is searched first. If not found the system drive is then searched.

A program called WHIM1SUB.CMD is used to combine all the WHIMSBxx.BIN files for the 6801 and create the random file, WHIM1SUB.BIN. The W option of the 6809 version also still works in the 6801 version to cause reading of individual WHIMSBxx files from the working drive.

Several of the original 6809 WHIMSBxx files are of little or no use in a 6801 environment and are not re-implemented in 6801 code yet:

02	Flex file handlers
03	Runtime error handler
0D	Buffered input
0E	Hex input
1C	LARGEINT output
1D	LARGEINT input
20	REAL input
21	REAL output

The corresponding operations in the language therefore perform no function. e.g.

File operations: CREATE, OPEN, CLOSE, EOF, CLOSEDELETE.
Writing and reading of LARGEINTS and REALS.
Reading of BYTES and DBYTES and CHAR arrays.

LIMITATIONS

The following features of the 6809 version are unimplemented or unavailable at this time.

1) For loop statement.
2) ENABLEFIRQ and DISABLEFIRQ.
3) Declaration time assignment to a local in a procedure.

4) Procedures within procedures.
5) Global sysmbol table dump and cross reference.
6) Procedure trace.

DIFFERENCES

The type REAL is a 3 byte not a 4 byte value. It has a 17 bit mantissa (not including an assummed one), a 6 bit exponent and 1 sign bit. This change is to help with the limited ram problem of the 6801. The range of numbers is from 2**-31 to 2**32, negative and positive which is twice the range of LARGEINTS. The precision is 17 bits or about 5 decimal digits.

Pressing a key while compiling activates the listing. Pressing it again deactivates the listing.

READ and WRITE statements default to the serial port of the micro-computer. This results from the fact that WHIMSB01 is rewritten to do this. If used, the port is automatically initialized first. This is done by a call to an initialization routine in WHIMSB01 put at the beginning of the target program.

The default extension for the object file is .BIN, not .CMD as in the 6809 version of Whimsical.

EXTRA FEATURES

An extra directive and option is available to optimise code for the 6301 or 6303 (otherwise code is for 6801). This option is "3" and is best put at the front of the program. e.g.

```
~OPTION 3
```

Another option "M" may be used to specify the drive on which to create or read .MOD files. This is useful when all the source for a given program won't fit on one disk. You can precompile some modules on other disks, having the .MOD files placed on another drive. Then when the main program is compiled it can fetch the precompiled modules from that drive. The drive number is placed immediately after the M. This option can be specified either in the command line or in option directive. e.g.

```
~OPTION M2
```

CONTROL OF RAM ALLOCATION

Since the 6801 has 128 (or 192) bytes of RAM, its use is very critical. All the runtime subroutines have been written to use as little temporary stack space as possible. Also the stack frame set up when a procedure is activated does not use any more RAM than is required for parameters, locals and the return address. Boolean variables can optionally be "packed" so as to use just one bit each (see ~BITBOOL and ~BYTEBOOL below).

In some applications more RAM will be installed. In this case the compiler can make use of this RAM for arrays and for stack. The scalar globals however, are still allocated in the direct page where they can be accessed with 2 byte instructions (as with the 6809 version). The only way to put scalar globals elsewhere is

with the ~BASE or ~TEMP directives.

If extra RAM is installed it is used if a ~STACK directive is used. The top of the RAM should be specified. The arrays are allocated downwards from this point and the stack pointer begins just below them. The scalar globals which still go in the direct page are allocated downwards from $0100 unless a ~GLOBALS directive is given. The ~GLOBALS directive is useful to reserve a few bytes for use by assembler routines etc.

```
e.g.     ~STACK=$4800        % 2K of RAM installed at $4000
         ~GLOBALS=$00F0      % reserve 16 bytes in direct page
```

Note that the value given in a ~GLOBALS directive must be a DBYTE even though it must never be set to a value above $0100.

If no stack directive is given then everything must fit in the direct page. Allocation begins at $0100 unless the ~GLOBALS directive is used as before.

The bottom limit of the RAM in the direct page may be $40 or $80 depending on the microcomputer. The compiler needs to know this limit so that it can check that not too many globals are declared. The ~GLIM directive is provided for this, standing for Global limit. e.g.

```
         ~GLIM=$0080
```

In the case where no extra ram is installed (no ~STACK directive) the stack will be in the direct page. The compiler cannot take account of the RAM required for the stack, only that for global variables and arrays. Therefore, in this case, it is prudent to set ~GLIM above the actual bottom of memory (by the estimated maximum runtime stack requirements). For this reason the default value for this limit is $00A0 which provides at least 32 bytes of stack in a microcomputer with 128 bytes of RAM.

BIT BOOLEANS

The 6809 version of Whimsical allocates a byte for each boolean. The 6801 version will do the same but there is a directive to force it to use only one bit for each boolean. This only applies to scalar booleans (which must be in the direct page). It does not apply to booleans in the BASE or TEMP segments or to arrays. There is a cost in terms of code and speed of using bit booleans. To start allocation of booleans a bit at a time, use the ~BITBOOL directive. To switch back to normal booleans, use the ~BYTEBOOL directive.

The information option prints out how many unused bits there are in the last byte allocated for bit booleans. Each precompiled module begins allocation of bit booleans with a new byte.