DEFT Pascal Language

Pascal

1 Introduction

The **DEFT Pascal** Compiler is a program which reads lines of source code produced with **DEFT Edit** (or any ASCII compatible editor) and produces a listing file and an object file. The object file produced contains actual machine codes which can be directly executed by the 6809 CPU in the CoCo after being linked by the **DEFT Linker.** This differs from a compiler which produces pseudo-code in the following respects:

1. The resulting program does not require an interpreter to execute. It is a self-sufficient program that requires only the Color Computer hardware.
2. The runtime execution environment is closer to assembler than BASIC. However, the **DEFT Debugger** provides some very powerful features which can make debugging the resulting machine language program almost as easy as debugging a BASIC program using the interpreter.
3. The performance of your program will be vastly better since each line of Pascal will result in only a few machine language instructions being executed. With an interpreter, several machine language subroutines within the interpreter will generally be executed per line of source code.
4. The program can be easily linked with **DEFT Macro/6809** assembly language modules and other **DEFT** high level language modules.

A Color Computer with 32K or 64K of RAM memory and 1 or 2 disk drives is a fairly powerful computer capable of most tasks being done on large micros and minicomputers. Using **DEFT Pascal** allows you to exploit that power to its fullest.

This section of the User's Guide describes those portions of **DEFT Pascal** which are ISO Standard. The following section, *Advanced Pascal*, describes the language extensions and assembler interface.

2 The Pascal Program

When programming in BASIC, there is almost no restriction on what order any of the statements must be placed. This is because almost all BASIC statements are *executable* statements. The only exception is the DIM statement, which is a *declaration* statement that defines arrays before they are used. DATA statements are neither *executable* nor *declaration* statements but they do represent a portion of the programs data. One of the primary aspects of the Pascal language is the presence of a very powerful declaration syntax, which requires that all Pascal programs be written in a specific format.

2.1 Block Structure

In Pascal, a program's structure is defined via a number of different types of *declaration* statements. These *declaration* statements allow a programmer to create an environment, or program structure, in which to get his job done with any number of the different types of *executable* statements. This provides the programmer with the ability to create a customized program structure that can match the problem structure of each program that he writes.

Pascal programs require the following elements in this order:

PROGRAM <program heading>;

<declaration statements>

BEGIN

<executable statements>

END.

Throughout this manual, words or phrases enclosed in <> are *non-terminators*. That is, they refer to a class of objects any one or more of which may be substituted at the place where the non-terminator is found. In the example above, *PROGRAM* is a *terminator* which represents exactly itself, whereas <program heading> is a non-terminator and represents some overall program information which will vary from program to program.

The important items in the structure are the <declaration statements> which define elements of your program and the <executable statements> which actually perform work on the defined elements.

Another way of describing the structure of a Pascal program is as follows:

PROGRAM <**program heading**> ; <**block**> **.**

Where <block> is equivalent to:

```
<declaration statements>
BEGIN
<executable statements>
END
```

This concept of *block* is central to the overall philosophy of Pascal. With this structure, <declaration statements> can define sub-blocks which in turn can themselves contain <declaration statements> which can further define sub-sub-blocks, and so forth and so on. It is with this hierarchy of *blocks* that the overall program is broken down into manageable pieces and implemented.

Block execution is initiated when that *block* is invoked or *activated*. Execution within a block starts with the first statement following the *begin* and proceeds sequentially with each of the following statements. (NOTE: in the section on *Compound and Control Statements* you will see how the order of execution can be altered). When the *end* statement is executed, the block is deactivated and control returns to the point at which the block was invoked.

Program execution starts at the last *begin* statement defined in the program. The program's execution will terminate at the last *end* statement defined in the program. Another way of putting it is that the last section of executable statements defined within a program is the first section to be executed. The sub-blocks, sub-sub-blocks, etc., which are the defined *procedures* and *functions* of the *program*, are *activated* by being invoked, or *called*, during the execution of the *program* or one of the previously executing *procedures* or *functions*.

2.2 Scope

The <declaration> statements within a *block* define <identifiers>, or data names, which are used by the <executable statements> within that *block*. When the *block* is activated, these <identifiers> are *activated* and become *known*. When the *block* is deactivated, the <identifiers> are *deactivated* and become *unkown*.

Identifiers used by <executable statements> may be either those defined by <declaration statements> within the same <block> or those defined in an enclosing <block>. All <identifiers> defined within all other *blocks* of the program become *unknown.*

Note also, that the same identifier may be redefined in different levels of blocks. At any point in the program the innermost definition known at that point will be used. The following is an example.

```
PROGRAM Example;
VAR I,J : Integer;

   PROCEDURE Proc1;
   VAR I : Integer;

      PROCEDURE Proc2;
      VAR J : Integer;
      BEGIN       (* Proc2 BEGIN *)
        I := J
      END;

   BEGIN       (* Proc1 BEGIN *)
     Proc2;
     I := J
   END;

BEGIN       (* PROGRAM Example BEGIN *)
  Proc1;
  I := J
END.
```

The above statements are discussed in detail later in the manual, but for purposes of this example, short definitions are provided here. The *var* statements declare integer variables named *I* and/or *J*. The *I := J* means that the value in *J* is assigned to *I*. The *Proc1;* and *Proc2;* statements invoke the corresponding procedures.

When the program first begins execution (just after the last *begin*) only the *I* and *J* and the procedure *Proc1* declared within the program *Example* are known. When *Proc1* is invoked and begins executing, *Proc2* becomes known, the *I* declared within *Proc1* becomes known, the *I* within the program *Example* becomes unknown (because of the temporary redefinition of *I*) and the *J* defined in the program *Example* remains known. When *Proc2* is invoked and begins executing, the *J* definition in *Proc2* temporarily replaces the *J* defined within program *Example*.

When *Proc2* returns and is deactivated, the previous *J* definition is restored. When *Proc1* is deactivated the previous *I* definition is restored and *Proc2* becomes unknown. Note that the above definitions and redefinitions apply to any type of <declaration statement> described below.

In *Advanced Pascal* you will find extensions to this fundamental structure.

2.3 Declaration Statements

As shown above, declaration statements come before the executable statements and are separated from them with the *begin* reserved word. The following are the <declaration statements>:

```
LABEL        <identifier>, ... ,<identifier>;
CONST        <identifier> = <constant>;
                               .
                               .
                               .
TYPE         <identifier> = <type definition>;
                               .
                               .
                               .
VAR          <identifier> : <type definition>;
                               .
                               .
                               .
PROCEDURE    <identifier> <parameter definition> ;
             <block> ;
FUNCTION     <identifier> <parameter definition> :
             <type definition> ; <block> ;
```

As in most Pascals, the above declarations may occur in any order; although according to standard Pascal, the above order must be followed with the exception of the *PROCEDURE* and *FUNCTION* declarations, which can be mixed with each other. Note that the above definition is *recursive* in that <declaration statements> are part of both *procedures* and *functions*, both of which are themselves types of <declaration statements>.

More detailed information about each of the above declaration statements can be found in the chapter on each statement.

2.4 Executable Statements

Executable statements are placed after the *begin*. The first statement following the *begin* is the first statement actually executed by the resulting program. Following is a list of the executable statements:

```
<identifier> := <expression>

BEGIN <executable statements> END

CASE <expression> OF
    <constant list> : <executable statement>;
                   ...
    <constant list> : <executable statement>
    ELSE <executable statement>
    END

FOR <identifier> := <expression> TO <expression> DO
                        <executable statement>

FOR <identifier> := <expression> DOWNTO <expression> DO
                        <executable statement>

GOTO <label>

IF <boolean expression> THEN <executable statement>
                             ELSE <executable statement>

READ (<file specifier> <input list>)

READLN (<file specifier> <input list>)

REPEAT <executable statements> UNTIL <boolean expression>

WHILE <boolean expression> DO <executable statement>

WITH <record variable> DO <executable statement>

WRITE (<file specifier> <output list>)

WRITELN (<file specifier> <output list>)

<procedure identifier> <parameter specification>
```

Pascal

Anywhere that you see <executable statements> (plural) you can use the following:

```
    <executable statement>;
              .
              .
              .
    <executable statement>
```

Note that the semicolon (;) is used to *separate* rather than *terminate* individual statements. Multiple statements separated by semicolons are allowed in both *begin* and *repeat* statements. The *else* clauses in both the *if* and *case* statements are optional and may be omitted.

Complete details on each of the above statements can be found in *Expressions and Assignments*, *Control*, *Procedures and Functions*, and *Input/Output*.

2.5 Program Statement

As shown above, the *program* statement is the first statement of your Pascal program. It has the following format:

PROGRAM <identifier> [(<identifier> , ... , <identifier>)] ;

The first <identifier> is the *program name* and serves no other purpose within the program. Following this is an optional parameter list enclosed in parentheses. In standard Pascal, this list identifies those file variables declared within the program which represent *external* files. The pre-defined file variables *input* and *output* must be present in this list if used (explicitly or implicitly) within the program.

In **DEFT Pascal**, the optional parameter list is allowed but ignored. This is because *all* files within a **DEFT Pascal** program are assumed to be external.

3 Language Elements

Before describing a Pascal program, it is necessary to describe the fundamental elements which make up one. Like BASIC, the Pascal language is constructed from the ASCII character set used on the Color Computer. These are as follows:

ABCDEFGHIJKLMNOPQRSTUVWXYZ	**<upper case characters>**
abcdefghijklmnopqrstuvwxyz	**<lower case characters>**
0123456789	**<numbers>**
!"#$% ^ &'()*+,-./:;<=>?@[]	**<special characters>**

All the following definitions will be in terms of these characters. Note that except in character and string constants (defined below), there is no distinction between upper and lower case characters for those language elements using letters.

3.1 Reserved Words

Reserved words are groups of upper or lower case characters whose meaning has been predefined in the language. The following is a list of all the reserved words used in **DEFT Pascal**:

ABS	**AND**	**ARRAY**
BEGIN	**BYTE***	**CALL***
CASE	**CHAR**	**CHR**
CONST	**DIV**	**DO**
DOWNTO	**ELSE**	**END**
EXIT*	**EXTERNAL***	**FILE**
FOR	**FORWARD**	**FUNCTION**
GOTO	**IF**	**IN**
INTERFACE*	**LABEL**	**LSL***
LSR*	**MOD**	**MODULE***
NEW	**NOT**	**ODD**
OF	**OR**	**ORD**
PACKED	**PRED**	**PROCEDURE**
PROGRAM	**PUBLIC***	**READ**
READLN	**RECORD**	**REPEAT**
RESET	**REWRITE**	**SET**
SIZEOF*	**STATIC***	**SUCC**
THEN	**TO**	**TYPE**
UNTIL	**VAR**	**WHILE**
WITH	**WORD***	**WRITE**
WRITELN	**XOR***	

Those reserved words which are suffixed with an asterisk are part of the language extensions of **DEFT Pascal**.

3.2 Identifiers

Identifiers are groups of letters and numbers which begin with a letter (either upper or lower case) and contain up to 12 upper or lower case letters and numbers which are not the same as any of the above listed reserved words. As in BASIC, these identifiers are used to represent variables. However, in Pascal they can also be used to represent constants, types, procedures and functions as well.

3.3 Labels

Labels are used to uniquely identify executable statements so that an executable statement may be referenced with the GOTO statement. A Pascal label functions much in the same way as line numbers do in BASIC. A label is a number which can be up to four digits long, which prefixes an executable statement with a colon (:) in between. The following is an example:

```
100: I := J
```

All labels within a block of executable statements must be declared with the LABEL declaration statement prior to the block of executable statements. The following is an example:

```
LABEL 100;
```

3.4 Constants

There are five types of constants supported by the **DEFT Pascal** Compiler. They are individually described below:

Decimal Integer Constant - A decimal integer constant is a group of numbers which may be optionally preceded with either a + or -. The allowable range for decimal integer constants is -32768 to 32767. The following are some examples:

```
-45
 45
+10234
+32768 (illegal, too large)
```

Hexadecimal Integer Constant - A hexadecimal integer constant is a group of up to 4 hexadecimal digits that is preceded with a $. A hexadecimal digit may be any of the following: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Note that only upper case characters can be used. The range of hexadecimal integer constants is $0000 to $FFFF. The following are some examples:

```
$ABC
$12A5
$5
```

Hexadecimal integer constants are not part of standard Pascal but a form of it can be found in many Pascal implementations.

Character Constant - A character constant is a single ASCII character (other than carriage return) contained between single quotes ('). Following are some examples:

```
'A'
'a'
'&'
''''
```

The last example is a character constant that represents a single quote. The single quote is doubled.

String Constant - A string constant is similar to a character constant except that more than one character is contained between the quotes. The following are some examples:

```
'PAGE HEADING TITLE'
'Sam and Joe''s Sub Shop'
```

Note that in the last example, the two single quotes in Joe''s actually is interpreted as one single quote in the string. In addition, a character constant can be used anywhere a string constant is required but the reverse is not true.

Real Constant - A real constant is a signed, decimal, fractional number, optionally raised to a signed decimal power. The general form of a real constant is:

```
<sign><number>.<number>E<sign><number>
```

The allowable range of real constants is 1E-64 to 9E63 both positive and negative. Following are some examples:

```
1.
-6.74
56.3E6
1.2E-3
```

The only required elements in a real constant are the first <number> and the decimal point (.). NOTE: Standard Pascal requires at least one decimal digit after the decimal point.

Constant Identifiers - Through the use of the CONST statement described later, identifiers can be defined as constants of some *type*. Three constant identifiers are predefined: *true*, *false* and *nil*. Later sections on *Constants*, *Types* and *Expressions and Assignments* provide more information on these constants.

3.5 Special Operators

As in BASIC the characters +, -, * and / are used as operators. However, Pascal also has several two character operators. These are as follows:

<>	**not equal**
>=	**greater or equal**
<=	**less or equal**
..	**range**
:=	**assignment**

3.6 Comments

Comments may be interspersed between (but not in the middle of) any of the above language elements. A comment starts with the characters (* and ends with the characters *). Unlike BASIC, Pascal comments can extend through more than one line. All the characters following the (* are considered comments until the *) is found later on the current or subsequent line.

4 CONST Statement

Constants as language elements are a part of practically every programming language. BASIC contains both real number and string constants. As described in the section on *Language Elements* Pascal contains decimal integer, hexadecimal integer, character, string and real constants as well as constant Identifiers.

There are two ways to create *constant identifiers*. One way is through the definition of *enumerated types* described in the section on *Types*. The other is through the use of the *const* statement. The general form of the *const* statement is as follows:

```
    CONST <identifier> = <constant>;
                         .
                         .
                         .
```

Following are some examples:

```
 CONST  MinSize = -3;
        MaxSize = 3451;
        CharLit = 'G';
        StringLit = 'This is a STRING constant';
        ExtraSize = MaxSize;
        Yes = True;
```

The purpose of the CONST statement is to allow the programmer to symbolically define a particular constant value for use later in the program. Note that any type of constant including a previously defined constant identifier may be used on the right hand side of a constant statement.

5 Types

The concept of *type* is not entirely unique to Pascal. However, the existence of a *TYPE* statement is a new concept for those programmers used to BASIC. When using BASIC, you have four kinds (types) of data: numbers, strings, number arrays and string arrays. You have different operations that can be performed with each and their internal representations are different.

A *type* refers to a data structure rather than any particular allocation of that structure. It has both a size and a set of operations that can be used on it. See the section on *Variables* for the actual allocation of memory for a given *type*.

In **DEFT Pascal**, *real numbers* and *strings* are both available along with a number of other *types, including some types that you can define yourself.* There are three classes of *types: simple, structured* and *pointer*. Those *types* which refer to indivisible entities are referred to as simple. An example is the set of whole numbers. Those which are made up of groups of simple *types* are referred to as structured. An array is an example of a structured *type*. A *pointer type* refers to those entities (such as *memory addresses)* which identify an occurrance of a *type*.

As shown in the chapter on *Program Structure* the general form of the TYPE statement is as follows:

```
        TYPE <identifier> = <type definition>;
                              .
                              .
                              .
```

This statement causes the <identifier> to be associated with the <type definition>. Following are descriptions of all the possible *type* definitions.

5.1 Type Identifier

A previously defined *type* identifier can be used as a *type* definition. These identifiers include all those defined in previous *TYPE* statements as well as a number of *pre-defined types* that are available. These predefined *types* are as follows:

- *Integer* - This is a 16 bit (2 bytes) Ordinal *type* which can range in value from -32768 to 32767.
- *Real* - This is a 6 byte floating point number. The high-order bit of the first byte is the sign of the number. The low-order 7 bits of the

first byte is the signed exponent. The last 5 bytes contain the mantissa in the form of 10, BCD digits. The range of the exponent is 63 to -64 and reflects powers of 10.

- *Char* - This is an 8 bit (1 byte) ordinal *type* which can range in value from NUL to DEL. These are the ASCII characters with binary values from 0 to 127. In addition, the characters that correspond to the binary values from 128 to 255 are also included.
- *Boolean* - This is an 8 bit (1 byte) Ordinal *type* which can have only two possible values: 0 (false) or 1 (true).
- *String* - This is an 81 byte structured *type* which can contain a variable number of *Char types*. A minimum of 0 and a maximum of 80 Chars can be contained in a String *type*. See *Advanced Pascal* for more information on strings.
- *Text* - This is a structured *type* which defines a *FILE OF Char*. This *type* occupies 286 bytes. See the section on *Input/Output* for more information.

One additional term is that of *ordinal type*. All simple *types* except *real* are also *ordinal* types. Ordinal *types* are simple types that have explicit, discrete values.

See the section on *Expressions and Assignments* for a discussion of the kinds of operations that can be performed on these various *types*. An example of a TYPE statement using a *type* identifier:

```
TYPE Number - Integer;
```

Number is a new *type* that is fully compatible in expressions with Integer.

Pascal

5.2 Enumerated

One way you can define your own *type* is by listing a set of values that are to be associated with a *type*. This defines a new ordinal *type*. The general form of an enumerated *type* definition is as follows:

```
(<identifier>, ... , <identifier>)
```

An example of a *TYPE* statement using an enumerated *type* definition is the following:

```
TYPE Color = (Red, Green, Yellow, Blue, Orange, Brown);
```

Color becomes a new independent *type* and any variables of this *type* will be protected from variables of other *types* in an expression. All enumerated *types* are 8 bit values where the identifiers contained in the list are implicitly defined as *constants* of that *type*. The order of the identifiers in the list is important. The internal representation of the first value is always 0, the second is 1 and so forth. See the section on *Expressions and Assignments* for a description of the operations that can be performed on an Enumerated *type*.

5.3 Subrange

A Subrange is a subset of values of an *Ordinal type*. The general form of a Subrange definition is as follows:

```
    <constant>..<constant>
```

Where the first <constant> must be less than or equal to the second <constant>. Some examples of subrange *TYPE* statements are as follows:

```
TYPE     SmallColor = Green..Blue;
         SmallInt = -128..127;
```

Note that in the case of a subrange of integers, a subrange of -128..127 or less will result in an 8 bit *type* which is fully compatible with the full 16 bit integer *types*.

5.4 Sets

A set is a collection of specific occurrances of objects of the same type. The general form of a set definition is as follows:

```
    SET OF <type identifier>
```

Where the <type identifier> specifies the types of objects comprising the set. The following is an example of use:

```
TYPE     SmallColor = (Green,Yellow,Red,Blue);
         SomeColors = SET OF SmallColor
```

SmallColor is an enumeration, and *SomeColors* is a set type. Variables of the type *SomeColors* are sets with 0 to 4 members which were listed in the declaration for the type *SmallColor*.

All Sets are 32 byte structured *types*. Each bit position within those 32 bytes represents each member of the set. Where bit 0 of byte 0 represents member 0. Bit 1 of byte 0 represents member 1, and so on up to 255. All Sets may have up to 256 members. Sets are given values by specifying a set constant as a list of constants enclosed by []s. If a set has no values assigned, it is called an empty set, which is denoted by two empty brackets [].

```
BriteColors := [Yellow, Red];
DarkColors := [Green, Blue];
NoColors := [];
```

5.5 Arrays

An array is a familiar concept to most programmers. In Pascal, it is a list of *types* (which themselves can be arrays). The general form of an Array definition is as follows:

```
ARRAY[<ordinal type definition>] OF <type definition>;
```

where the <ordinal type definition> defines not only the *quantity* of <type definitions> in the ARRAY but also *how each element is identified by type.* The following examples should make this clear.

```
TYPE    ColorList    = ARRAY[1..6] OF Color;
        Numbers      = ARRAY[Green..Orange] OF Integer;
        Flags        = ARRAY[Color] OF Boolean;
        ColorPlane   = ARRAY[0..200] OF ColorList;
```

In the first example, a list of colors is being defined. Elements of the list are identified by the integers 1 through 6 for a total of 6 elements. Note that one of the most frequent uses of *subrange* types are in *array* definitions.

The second example shows one of the unique properties of Pascal. In this case we are defining a 4 element list of numbers where elements of the list are identified, in order, by the colors Green through Orange. The third example is similar where the number of Boolean elements is equal to the total number of colors and each element of the list is identified by a different color.

The final example shows a definition of a two-dimensional *array*. In this example there are 201 lists defined. Variables of this type would have memory organized as follows: