Advanced Pascal Language Extensions

Adv

1 Introduction

This section describes a number of facilities in the **DEFT Pascal** Compiler which are not found in standard Pascal. These facilities provide the programmer with significant additional capabilities which allow easier text processing, ROM and absolute memory access, and separate compilation with both **DEFT Pascal** and **DEFT Macro/6809** assembly language.

Before deciding whether to use these facilities, the purpose of the program to be written must be considered. If portability is essential then only those facilities described in Pascal should be used. If the program is to run only on the Color Computer and you wish to take maximum advantage of the machine's capabilities, then by all means use the *Advanced Pascal* features.

Note that even when using these advanced features the resulting program may still be moved to other machines since many other Pascals have corresponding features. This is especially true in the areas of string handling, separate compilation and compiler controls.

2 Strings

In standard Pascal, a *string* is little more than an *array of char*. **DEFT Pascal** allows you to treat a *string* in exactly the same way. However, a *string* is not exactly the same as an *array of char* in that you can also treat this type as a true variable length structure. This allows you to access individual elements of the string by including a subscript or access the entire structure by not including a subscript. Note that since *array of string* is allowed, the number of subscripts determine the type of the resulting factor.

A string, in **DEFT Pascal**, contains a string length in element 0. The remaining elements are the string itself. The default maximum length of a string is 80. Other maximums can be declared (up to 255) by including a constant in parentheses following the type identifier *STRING*. See the section on *Type Extensions* for a complete explanation.

Note that this structure is maintained in string constants as well as string variables.

2.1 String Assignments

The assignment statement not only allows you to assign a string variable or constant to a string, but also a general string expression. The syntax of a string assignment statement is as follows:

<string variable> := <string term> + ... + <string term>

Where <string variable> is a simple variable, record member, array element or dereferenced pointer variable with a base type of string. <string term> is any of the following:

- A *string* variable
- A *string* constant
- A *char* type expression

The result of the assignment is to set the <string variable> on the left of the assignment sign to the ordered concatenation of the <string term>s on the right side. Some examples:

```
StringVar := OtherString + ' suffix string';
StringVar := 'First line' + CHR(13) + 'Second line';
StringVar := StringVar + 'A'
```

The last example shows how to append <string term>s to the end of an existing value in a <string variable>.

2.2 String Relations

As mentioned in *Pascal*, strings may be combined with relational operators in boolean expressions. When comparing two strings, **DEFT Pascal** generates code that compares the strings on a character by character basis from left to right. When two characters that are not equal, or the end of a string is encountered the compare stops. If unequal characters are found, the binary value of the corresponding characters determines the result. If the end of a string is encountered, the longer string is considered greater. Only if the current length and all corresponding characters are equal are the strings themselves considered equal.

2.3 STRINGCOPY Procedure

This predefined procedure is used to copy a portion of one string into another. The procedure declaration is:

```
PROCEDURE STRINGCOPY (VAR SOURCE : STRING;
                      INDEX, LENGTH : INTEGER;
                      VAR DESTINATION : STRING);
```

The *string* variable DESTINATION is set to the *string* contained in SOURCE starting with INDEXth character and continuing for LENGTH characters. If the length of SOURCE is less than INDEX then DESTINATION will be null. If the length of SOURCE is less than INDEX+LENGTH-1 then the length of DESTINATION will be the length of SOURCE less INDEX-1.

2.4 STRINGDELETE Procedure

This predefined procedure is used to delete a portion of a *string* variable. The procedure declaration is:

```
PROCEDURE STRINGDELETE (VAR SOURCE : STRING;
                        INDEX, LENGTH : INTEGER);
```

The *string* variable SOURCE has the *string* starting at the INDEXth character and continuing for LENGTH characters removed from it. If the length of SOURCE is less than INDEX then no change is made. If the length of SOURCE is less than INDEX+LENGTH 1 then all the characters in SOURCE following the INDEXth character will be deleted and the new *string* length will be INDEX-1.

2.5 STRINGINSERT Procedure

This predefined procedure is used to insert one *string* into into another at a specified point. The procedure declaration is:

```
PROCEDURE STRINGINSERT (VAR SOURCE : STRING;
                        VAR DESTINATION : STRING;
                        INDEX : INTEGER);
```

The *string* variable SOURCE is inserted into the *string* DESTINATION starting in front of the INDEXth character. If the length of DESTINATION is less than INDEX then SOURCE is appended to DESTINATION.

2.6 STRINGPOS Function

This predefined function is used to find the location of one *string* within another. The function declaration is:

```
FUNCTION STRINGPOS (VAR IMAGE, TARGET : STRING) : INTEGER;
```

A search of *string* TARGET is made to try to find *string* IMAGE. If IMAGE is found in TARGET then STRINGPOS returns the character position in TARGET where IMAGE was found. If IMAGE is not found in TARGET, STRINGPOS returns a zero.

2.7 ENCODE Function

This predefined function is used to convert a *string* containing an integer constant to an integer. The function declaration is:

```
FUNCTION ENCODE (VAR ASCII : STRING) : INTEGER;
```

The *string* ASCII is scanned and the binary representation of the ASCII characters is returned. The following rules are used during the scan:

1. Leading blanks are ignored
2. A leading + or - sign is allowed
3. The scan stops when the end of the *string* or a non-numeric character is encountered

If no numeric characters are encountered before the scan stops, ENCODE returns zero.

2.8 ENCODEREAL Function

This predefined function is used to convert a *string* containing a real constant to a real. The function declaration is:

```
FUNCTION ENCODEREAL (VAR ASCII : STRING) : REAL;
```

The *string* ASCII is scanned and the binary representation of the ASCII characters is returned. The following rules are used during the scan:

1. Leading blanks are ignored
2. A leading + or - sign is allowed
3. The first set of digits are the mantissa and may contain an imbedded decimal point.
4. The letter *E* may follow the mantissa to indicate that an exponent follows.
5. The exponent may have a leading sign but cannot have an imbedded decimal point.
6. The scan stops when the end of the *string* or a non-numeric character is encountered

If no numeric characters are encountered before the scan stops, ENCODEREAL returns zero.

2.9 DECODE Procedure

This predefined procedure is used to construct a *string* containing the external representation (base 10) of an integer. The procedure declaration is:

```
PROCEDURE DECODE (NUMBER, SIZE : INTEGER;
                  VAR ASCII : STRING);
```

The *string* ASCII is constructed. NUMBER is the binary value to use during the conversion and SIZE is the resulting *string* length of ASCII. The external (base 10) representation of NUMBER is right justified in ASCII. If SIZE is larger than required, leading blanks are appended on the left. If SIZE is too small, the leftmost characters are truncated.

2.10 DECODEREAL Procedure

This predefined procedure is used to construct a *string* containing the external ASCII representation (decimal or scientific) of a real The procedure declaration is:

```
PROCEDURE DECODEREAL (NUMBER : REAL;
                      SIZE, FRACTION : INTEGER;
                      VAR ASCII : STRING);
```

The *string* ASCII is constructed. NUMBER is the binary value to use during the conversion, SIZE is the resulting total *string* length of ASCII and FRACTION is the number of fractional digits to the right of the decimal point. The external (base 10) representation of NUMBER is right justified in ASCII. If SIZE is larger than required, leading blanks are appended on the left. If SIZE is too small, the string is filled with asterisks. If FRACTION is negative, then scientific notation is used, otherwise a decimal display is used.

2.11 HEX Procedure

This predefined procedure is used to construct a *string* containing the ASCII hex representation of a specified area of memory. The procedure declaration is:

```
PROCEDURE HEX    (ADDRESS : INTEGER;
                  BYTECOUNT : INTEGER;
                  VAR ASCII : STRING);
```

The memory area beginning at ADDRESS and continuing for BYTECOUNT bytes is converted to a hex *string* which is placed in ASCII. The hex representation is a pair of hex digits followed by a blank for each byte except the last. The resulting length of ASCII is (BYTECOUNT*3)-1.

3 Type Extensions

A strongly typed language like Pascal can help a programmer gain and maintain control of his program. He can ensure that variables of different types are not inadvertantly combined in an expression or the wrong type expression is passed as a parameter to a *procedure* or *function*.

However, there are occasions when a programmer wants to treat some datum as *usually* of one type and *sometimes* to treat it as another type. The extensions pertaining to type found in **DEFT Pascal** provide a sorely needed type breaking function that is only partially found in standard Pascal.

3.1 Type Conversions

Provided in standard Pascal are the type conversion functions *chr*, *odd* and *ord*. **DEFT Pascal** supports these functions, but also provides a more regular *type* breaking capability. This capability is implemented with implicit builtin function definitions based on ordinal *type* definitions.

When any ordinal *type* is defined, **DEFT Pascal** also implicitly defines a conversion function with the same name as the *type*. This function has a value parameter which is of any ordinal *type*. It returns (in the same way that *chr* and *ord* do) the equivalent value with a *type* equal to the named *type* identifier. For example:

```
.
.
.
TYPE Color = (Red, Green, yellow);
     Fruit = (Apple, Lime, Lemon);
VAR ColorVar : Color;
    FruitVar : Fruit;
.
.
.
FruitVar := Fruit (ColorVar);
.
.
.
```

In the above example, *Colorvar* produces an expression of type *Color*. This expression is used as a parameter to the function *Fruit* (implicitly declared in the *type* definition) which converts it to a

Fruit type expression. Operation of the assignment statement is to set *FruitVar* equal to the *fruit* whose corresponding *color* is in *ColorVar*.

Note that as a result of this extension, the builtin function *integer* is equivalent to *ord* and *char* is equivalent to *chr*.

3.2 Pointer/Integer Conversions

In order to allow full use of the addressing capability of the 6809, **DEFT Pascal** provides the ability to convert between *integer* and *pointer types*. The builtin function *ptr* will convert an *integer type* to a *pointer type*. In addition, a *pointer* can be converted to an integer via the *ord* and *integer* builtin functions. These facilities make it possible to manipulate *pointers* arithmetically. For example:

```
TYPE BigRecord = RECORD ... END;
VAR BigPtr : ^BigRecord;
  ...
BEGIN
  ...
  BigPtr := PTR (ORD (BigPtr) + SIZEOF (BigRecord));
  ...
```

In the above example, *BigPtr* is incremented to point to the next *BigRecord* in memory.

3.3 Arrays

In standard Pascal an *array type* definition includes both the upper and lower bounds of the *array* as well as the element *type*. This of course is also true with **DEFT Pascal**. However, when using a previously defined array *type* identifier, you may specify a different upper bound than the default contained in the original *type* declaration. Example:

```
TYPE MyArray = ARRAY[1..200] OF Integer;
VAR Array1 : MyArray;
    Array2 : MyArray(150);
```

In the above example, *Array1* and *Array2* are equivalent *types*. However, *Array1* has 200 elements and *Array2* has 150 elements. This variable size capability is useful when creating *procedures* and *functions* which process *arrays* of a given *type* but with varying sizes. However, for all arrays except *strings*, the new upper bound

must be less than or equal to the upper bound of the original array. Standard Pascal has a *conformant array* facility which provides an equivalent capability when used in *procedures* and *functions*.

Note that since the type *string* can be used as an *array of char* type, you can also specify an upper bound (up to 255) when declaring *strings*. This upper bound will determine the amount of memory reserved for the *string* variable and the maximum length *string* value that can be stored.

4 Absolute Memory Access

This section describes the **DEFT Pascal** Compiler's facilities for accessing specific areas of the 6809 address space. In addition to the facilities shown here, specific areas of memory can be accessed in **DEFT Macro/6809** assembly language via the *Separate Compilation* facilities and the *Assembler Interface*. However, the facilities described in this section can be used entirely within Pascal and results in *position independent code (PIC)*.

4.1 BYTE and WORD Arrays

Absolute memory can be accessed as BYTEs or WORDs by using the corresponding pre-defined *array*. BYTE is ARRAY[$0000..$FFFF] OF 0..255 and WORD is ARRAY[$0000..$FFFF] OF INTEGER. The subscript used represents the actual memory address that is used. Example:

```
IF BYTE[1024] <> $41 THEN BYTE[1024] := $42;
WORD[$7FFE] := $FFFF
```

4.2 Absolute Address Operator (@)

The absolute integer address of any variable can obtained with the unary operator @. Example:

```
WORD[@I] := 5;
I := 5
```

The above two statements are equivalent. This facility can be combined with the *ptr* builtin function to put the address of any variable into a *pointer* type variable.

4.3 CALL Function

The predefined function CALL provides the ability to invoke the machine language functions and subroutines typically found in the Color Computer's ROM. The Function definition is:

```
TYPE ROMAddress = Integer;
     ARegister = 0..255;

FUNCTION CALL (RtnAddress : ROMAddress;
                    Parm : ARegister) : Aregister
```

When using the CALL function, the first parameter is the absolute memory address of the subroutine to be invoked. The second parameter is the value to be passed in the *A* register. The value

returned by the function is the value that the subroutine returned in the *A* register. Example:

```
REPEAT Key := CALL (WORD[$A000],0) UNTIL Key <> 0
```

The above example invokes the ROM subroutine whose address is located at absolute memory WORD $A000 (POLCAT). A zero is passed to this routine in the *A* register and the value returned by the subroutine is stored in the variable *Key*. The effect of the *repeat* statement is to wait until a keystroke is entered at the keyboard and to store the keystroke in *Key*. NOTE: In order to access ROM routines, you will have to run your program in 32K mode.

5 Static Variable Allocation

In the section *Variables* in the Pascal Language Summary, the standard automatic allocation scheme of Pascal is described. This is the default variable allocation incorporated into **DEFT Pascal**. However, it is also possible to *statically* allocate a variable.

When a variable is statically allocated, memory is reserved at compile time. This means that every time the variable is accessed, the *same* memory area is accessed *even if the block that the variable is defined in has been deactivated and then reactivated.*

This allows you to store a value into a statically allocated variable that is local to a procedure, before exiting from the procedure. Then when the procedure is subsequently invoked, be able to access that variable and retrieve the previously stored value. This can't be done with automatically allocated variables since the specific memory location occupied by the variable may change on each allocation.

5.1 STATIC Attribute

Variables are statically allocated when one of several *attributes* are added to the *var* statement in which they are defined. An attribute is a keyword which immediately follows the *var* keyword. The simplest of these attributes is the keyword *static*. The only result of this attribute is to cause all variables defined in the current *var* statement to be statically allocated. Example:

```
VAR          A : Char;
VAR STATIC   B, C : Integer;
             D : Char;
VAR          E, F : Integer;
             G : Char;
```

In the above example, variables *B*, *C* and *D* are all statically allocated. Variables *A*, *E*, *F* and *G* are all dynamically allocated. The scope of all the variables is the same.

5.2 PUBLIC Attribute

The *public* attribute, like the *static* attribute, causes all the variables defined in the corresponding *var* statement to be statically allocated. However, the *public* attribute can only be used in *var* statements at the *PROGRAM* or *MODULE* (see *Separate Compilation*) level and may not be used in *var* statements in *procedures* or *functions*.

In addition to causing a variable to be statically allocated, the *public* attribute *extends the scope* of the affected variables to other *separately compiled modules*. These other modules reference these *public* variables by declaring the same variables using the *external* attribute (see below). Example:

```
VAR PUBLIC     A, B : Char;
               C : Integer;
```

In the above example all three variables are statically allocated and made public. See The section on *Separate Compilation* for more information.

5.3 EXTERNAL Attribute

The *external* attribute is the complementary attribute to the *public* attribute. All variables defined in a *var* statement with the *external* attribute are *not actually allocated* by that *var* statement. This statement causes the static allocation performed by the *var public* statement to be used. Example:

```
VAR EXTERNAL   A, B : Char;
               C : Integer;
```

In the above example the variables *A*, *B* and *C* have been declared *public* in another module where memory for them has been allocated. All references to *A*, *B* and *C* in the module with the *external* attribute will access the publicly defined variables. See the section on *Separate Compilation* for more information.

6 Separate Compilation

This section details a facility in the **DEFT Pascal** Compiler that allows a programmer to break up a large program into a number of smaller programs. These smaller programs (known generically as *modules)* can then be compiled and (usually) tested independently. One of the primary advantages of separate compilation is the additional level of identifier scoping that is provided.

6.1 Rationale

In general, identifiers (constants, types, variables, procedures and functions) defined within a module are known only within that module. These identifiers are thought of as *private* and are not known to other modules. Of course if *all* the identifiers are private then there is no way for the module to be used. For this reason some identifiers are always made *public* so that *controlled* access to the module is assured.

For example, a complete set of routines to handle high-resolution graphics could be a module. Some of these routines would be called from outside the module and would consitute the *interface* to your graphics package. These routines would be declared *public.*

Other routines would be utilities whose express purpose is to perform functions common to several of the *public* routines. These utility routines would remain private so that they would not be inadvertantly invoked by other *modules.* This also ensures that their names would not conflict with other names used in other *modules.*

The variables used by this graphics module are also divided into public and private. The public variables may provide a means to pass data to or from several of the procedures in the module or may be used to specify operational modes. The private variables would be used to store temporary or intermediate results.

A special **DEFT Pascal** language construct, called an *interface module,* could be used to provide the compile time linkage between the graphics module and those other modules that use it. This *interface module* would be included at the beginning of the other modules and would provide all the *external* declarations for the *public* procedures, functions and variables. In addition, it would include const and type statements in order to define any special constants or types required by the graphics module.

6.2 MODULE Block

In standard Pascal, a complete program is a self-contained unit. For many smaller programs this is quite adequate and provides a simple environment in which to develop them. However, when you wish to divide your program into several relatively independent pieces; you have a problem if these pieces do not map, one-to-one, into procedures or functions. It is this problem that **DEFT Pascal's** *module* solve.

A *module* is a **DEFT Pascal** construct that allows you to group a set of procedures, functions and variables into a sort of a self-contained subprogram which is compiled by itself. This Pascal *module* can then be combined with other Pascal *modules*, **DEFT Macro/6809** Assembler *modules* and to only *one* Pascal *program*, via **DEFT Linker**, to create a complete program.

The syntax of a MODULE is as follows:

```
MODULE <module name>;
  CONST <identifier> = <constant>;
                .
                .
                .
  TYPE <identifier> = <type definition>;
                .
                .
                .
  VAR <identifier> : <type definition>;
                .
                .
                .
  PROCEDURE <identifier> <parameter definition>;
            <block>;
                .
                .
                .
  FUNCTION <identifier> <parameter definition>;
           <block>;
                .
                .
                .
END.
```

As you can see, this is *almost* the same as a *program*. In fact, with **DEFT Pascal**, a *program* is merely a special type of *module*. A *program* is the only *module* which contains its own BEGIN <executable statements> END. It is with these <executable statements> in the final binary program that execution begins.

One other difference between a *program* and a *module* is *variable allocation*. In a *program*, the default allocation is *automatic*. In a *module*, the default type of allocation is *static*. Since there is no way of *explicitly* specifying automatic allocation, a *module's* variable types are all static. The primary reason for this is that there is no *frame structure*, (see *Assembler Interface*), for a *module* in which to automatically allocate a variable.

Linkage between *modules* and the *program* is provided via the *public* and *external* attributes described below.

6.3 PUBLIC Procedures and Functions

Public procedures and functions are declared at the outer most block level of a *program* or *module*, and contain a *public* attribute immediately following the procedure or function statement. Procedures and functions which are nested within other procedures or functions may not have the *public* attribute. The syntax of a *public* procedure is as follows:

```
PROCEDURE <identifier> <formal parameter definition>;

  PUBLIC;

  <declaration statements>

  BEGIN
  <executable statements>
  END
```

The only difference between this and a standard procedure (or function) is the *public* attribute immediately following the *procedure* or *function* statement.

Once a procedure or function has been declared *public*, it may be invoked from other *modules* which have declared the *same* procedure or function as *external* (see *EXTERNAL Procedures and Functions*). Note: you may not use the same identifier to declare a procedure, function or variable as *public* in more than one *module*. However, once it is declared as *public*, you may declare it as *external* in as many *modules* (or the *program*) as you wish. An identifier cannot be declared as both *public* and *external* in the same *module* or *program*.

6.4 EXTERNAL Procedures and Functions

An *external* declaration allows a *public* procedure or function to be known and invoked in any *module* or *program* in which it is declared as *external*. A procedure or function is declared as *external* by following the procedure or function statement with only the *external* statement. The syntax is as follows:

```
PROCEDURE <identifier> <formal parameter definition>;
EXTERNAL
```

This type of procedure or function does not have a <block> associated with it. However, it *must* have a corresponding *public* procedure or function declared in another *module* whose procedure or function statement is *identical* to the one used with the *external* statement. Note that like the *public* statement, the *external* statement can be used only with procedures and functions which are declared at the outer most block level of a *module* or *program*.

6.5 PUBLIC and EXTERNAL Variables

The *public* and *external* attributes, in the VAR statement, cause static memory allocations to be made, as described in the section on *Static Variable Allocation*. *Public* variables (like *public* procedures) are those variables whose scope has been explicitly extended beyond the enclosing *module* or *program*. *External* variables are those variables which actually exist in other *modules* (OR the *program*) as *public* variables, but whose scope has been extended into this *module* or *program*.

As mentioned in the section on *public* procedures, you may not use the same identifier to declare a procedure, function or variable as *public* in more than one *module*. However, once it is declared as *public*, you may declare it as *external* in as many *modules* (or the *program*) as you wish. Any identifier cannot be declared as both *public* and *external* in the same *module* or *program*.

6.6 INTERFACE Block

An *interface* Block is a special **DEFT Pascal** Compiler construct which is used in conjunction with a *program* or *module*. Its purpose is to simplify the compile time *module* linkage (which would normally occur via *external* attributes and statements).

The *interface* block is an optional construct which may be included 1 or more times before the *module* or *program* statement. The syntax is as follows:

INTERFACE <interface name>;

<special declaration statements>

END

The <special declaration statements> are generally the same as <declaration statements> with the exception that all procedure, function and VAR statements are *assumed to be external*. That is,