Personal Computer
Computer Language
Series

Pascal Compiler

First Edition (August 1981)

Changes are periodically made to the information herein; these changes will be incorporated in new editions of this publication.

Products are not stocked at the address below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM Personal Computer Dealer.

A Product Comment Form is provided at the back of this publication. If this form has been removed, address comment to: IBM Corp., Personal Computer, P.O. Box 1328, Boca Raton, Florida 33432. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligations whatever.

CONTENTS

CHAPTER 1. INTRODUCTION

Contents

IBM Personal Computer Pascal

This document describes IBM Personal Computer Pascal. We have assumed that you have a general knowledge of Pascal from such publications as "Pascal User Manual and Report" by Jensen and Wirth, "Introduction to Pascal" by Welsh and Elder, or from other sources.

The Pascal Language

Pascal was originally designed by Niklaus Wirth with two primary goals: to teach programming as a systematic discipline, and to implement programs in a reliable and efficient way. But there are other reasons for the widespread interest in Pascal as a general purpose programming language and as a system program implementation language.

- Pascal, a relatively modern language, has benefited from many earlier languages (such as ALGOL) and is forming the basis of several new ones.

- The key emphasis of Pascal is its role as a higher level language; that is, a language providing useful abstract tools for specifying data structures and algorithms, independent of the underlying implementation.

- The user need not be concerned with the representation of data (the number of bytes per variable, organization of arrays, address size, and so on).

- The programmer can more accurately specify the characteristics of variables, such as the range of values allowed (VAR I: 0..99) or decide whether

to trade space for time in accessing variable components. (PACKED keyword)

IBM has added several goals to this list:

1. IBM Personal Computer Pascal is designed to be a systems implementation language, especially suitable for writing compilers, interpreters, operating systems, and so on.

2. Generating efficient code is paramount. The various language extensions, and the global optimizer phase (pass two of the compiler) all work toward minimizing the time and space needed by compiled programs.

3. Operations that are done easily in assembly language should be easy to do in IBM Personal Computer Pascal.

IBM Personal Computer Pascal generally conforms to the ISO draft (ISO/TC 97/SC 5 N595).

IBM's intent is that correct standard programs compile and run correctly on the IBM Personal Computer Pascal compiler without changes. Since IBM features introduce new reserved words and other elements, both the IBM features and extensions are summarized in Appendix E.

IBM Personal Computer Pascal Extensions

The extensions include:

- Compiler directives
- Attributes
- Super array type
- String processing with CONCAT
- Constant values
- Systems implementation extensions

Compiler Directives

We provide control over generating error checking code, listing format controls, a construct for conditional compilation, and a mechanism for inserting other source files into a compilation. There are 30 of these "metalanguage" commands available.

Unit

Pascal is an excellent language for very large programs that must be reliable (such as system software applications), and IBM Personal Computer Pascal supports separtely compiled modules using the concept of a unit.

A *unit* is a set of related data types, variables, constants, procedures, and functions, plus an initialization procedure. It has two parts: the interface and the implementation.

The *interface* contains a list of identifiers to export and their declarations (including procedure and function headers).

The *implementation* contains any local declarations, procedure and function bodies, and the initialization code. Some unit routines may be implemented in assembly code instead of Pascal.

A program (or implementation or other interface) can use a unit, giving the interface but not the implementation. This method provides a more structured way of breaking a program into modules than external procedures and variables.

Attributes

Attributes for variables, procedures, and functions give you control at the linker text level.

The attributes include:

- PUBLIC and EXTERN for global identifier linking
- STATIC and READONLY for variables and PURE for procedures and functions.

Super Array

IBM Personal Computer Pascal provides a *super array* type, to let array lengths vary. With the super array type, the lower bound is given but the upper bound is undefined.

Although the super array type cannot be used directly for variables (that is, VAR V = SUPER ARRAY [0..*] of REAL;), it can be used in two important ways:

- As the type of a formal reference parameter
- As the referent type of a pointer.

In addition, any variable can be given a type derived from the super array type using a "designator". For example:

```
TYPE
  VECT = SUPER ARRAY [0..*] OF REAL;
VAR
  PVEC: ^VECT; V10: VECT (10);
PROCEDURE SORT (VAR V: VECT);
BEGIN
  NEW (PVEC, UPPER (V));
    •
    •
    •
END;
    •
    •
    •
SORT (V10);
```

Here VECT is the super array type. PVEC is a pointer to an instance of the type.

The NEW allocates a new VECT with the upper bound given in the second parameter, in this case the same upper bound as the parameter V.

V10 is a variable, an instance of a VECT with an upper bound of 10. The parameter V can take a variable of any type derived from VECT; for example, V10 or PVEC^.

The super array concept handles both dynamic and conformant arrays in a clean and efficient way.

Strings

Strings in standard Pascal are fixed length. However, the ability to declare a variable of a string type, where the length of the string can vary, is a key feature of BASIC and PL/I, and an often-used abstract data type. The string type is:

```
SUPER PACKED ARRAY [0..*] OF CHAR
```

where element zero contains the length. We also have a set of string procedures and functions.

IBM Personal Computer Pascal has a CONCAT procedure, which takes two string parameters and concatenates the second to the first.

This makes string processing portable, since all the procedures and functions can be written in any Pascal by:

- Declaring the string type to be a fixed length character array
- Using a variable length string type in a Pascal with a super array or other conformant array extension

String assignment, comparison, and READ/WRITE are done automatically in IBM Personal Computer Pascal.

Constant Values

Another set of features in IBM Personal Computer Pascal applies to constant values. Most expressions with constant values are evaluated at compile time, so if WORDCOUNT and SYMBOLCOUNT are constants, another constant TOTALCOUNT = WORDCOUNT + SYMBOLCOUNT can be defined.

Numbers can be in binary, octal, decimal, or hexadecimal. This is also supported in READ and WRITE.

There is a string constant concatenation operator, as well as array and record constants.

A program can contain one or more VALUE sections, in which variables are given an initial constant value.

Finally, a formal parameter can be preceded by the keyword CONST; the effect is the same as VAR, except that the actual parameter can be a constant and cannot be modified in the body of a routine.

Systems Implementation

Several extensions are specific to systems implementation work:

- The WORD type
- ADDRESS types
- Input/output capabilities
- Interactive READ
- Random files and file modes
- Intrinsic procedures and functions

The most important is the WORD type; it is really just a subrange from 0 to 65535 (MAXWORD), just as the INTEGER type is a subrange from -32767 to 32767 (MAXINT).

Sixteen bit quantities can be treated as either signed or unsigned, so the range of numbers really goes from -32768 to 65535.

Having both WORD and INTEGER types allows this range (except -32768) to be used in a Pascal program, although not with one type.

The unsigned word is often used in system implementation work, as an address, or an integer with a large maximum, or a set of bits, or just as a memory value with unknown semantics.

Trying to use the INTEGER type for these purposes runs into the problems of comparisons (whether #FFFF is greater than #FFFE).

Additional operators, such as AND, OR, XOR, are also allowed with the WORD type.

We also added an address type, similar to a pointer but allowing for segmented addressing. This is very useful for accessing system data areas.

Other features we have added include various input/output capabilities (I/O error trapping, interactive READ using "lazy evaluation", random files, file modes, string READ, etc.) and additional intrinsic procedures and functions (such as ENCODE and DECODE, RETYPE, and RESULT).

Summary

IBM Personal Computer Pascal can be used for system software implementation. It includes many features useful for creating and maintaining Pascal programs in a structured way, and/or necessary for common system programming tasks.

Features of particular note include separate compilation units, variable length strings, the super array type, and machine oriented constructs.

IBM Personal Computer Pascal is more than a tool; it is a tool maker, designed for writing smart programs that make our computer easier to use and more accessible to people.

> **Note:** For the remainder of this manual, the term *IBM Pascal* is intended to mean IBM Personal Computer Pascal.

CHAPTER 2. COMPILING A PASCAL PROGRAM

Contents

What You Need

To successfully compile Pascal programs on your IBM Personal Computer, you need:

- Your Pascal package:
  - Three 5-1/4 inch master diskettes, one marked PAS1, one marked PAS2, and one marked PASCAL.LIB

    - PAS1 contains the files:
      - PAS1.EXE
      - PASKEY
      - FILKQQ.INC
      - FILUQQ.INC
      - ENT6XS.ASM

    - PAS2 contains the file:
      - PAS2.EXE

    - PASCAL.LIB contains the files:
      - PASCAL.LIB
      - PASCAL

  - This manual: the *IBM Personal Computer Pascal* reference manual

- A minimum of 128K bytes of machine-resident memory

- Two diskette drives

- A printer

- A display (an IBM Personal Computer Monochrome Display, a monitor, or a TV with an RF modulator)
- The *IBM Personal Computer Disk Operating System (DOS)* reference manual and diskette
- One 5-1/4 inch diskette which we will call the *scratch* diskette

The First Time Through

The first time through, you will need:

- Three 5-1/4 inch diskettes to make backup copies of the master diskettes

Backing Up the Master Diskettes

We recommend that you back up your Pascal master diskettes as soon as possible by making copies of PAS1, PAS2, and PASCAL.LIB. We also recommend that you use these copies for your day-to-day operations, and put the master diskettes in a safe place.

Setting Up the Diskettes: PAS1 and PAS2

Now that you have made copies of PAS1 and PAS2, you will need to copy COMMAND.COM from the DOS diskette onto PAS1 and PAS2. This is because when PAS1 and PAS2 are loaded they overwrite COMMAND.COM in memory (COMMAND.COM is loaded when the system is started).

COMMAND.COM will then be automatically reloaded when either PAS1 or PAS2 is finished executing and is in the diskette drive.

Setting up the Diskettes: PASCAL.LIB

Copy the linker, LINK.EXE, from the DOS diskette to your copy of the PASCAL.LIB diskette.

You are now ready to compile a Pascal program.

Starting the Compilation

We recommend the following sequence of steps as a general rule:

1. Format your scratch diskette. See the *IBM Personal Computer Disk Operating System (DOS)* reference manual for information about formatting.

2. Put your program onto the scratch diskette either by copying it from the diskette it is already on, or by creating a new program using the line editor (see "EDLIN" in the *IBM Personal Computer Disk Operating System (DOS)* reference manual).

3. Give your program the filename extension ".PAS", for Pascal.

You are now ready to compile your Pascal program.

> **Note:** You may enter compiler commands using all uppercase letters, all lowercase letters, or a combination of uppercase and lowercase letters.

Starting the Compiler: PAS1

PAS1 is the first pass of the compiler. PAS1 reads your source file and checks it for syntactic correctness. It generates two intermediate files which are stored on the scratch diskette in the space not occupied by your source program. These files are called:

PASIBF.SYM – the symbol table
PASIBF.BIN – the intermediate binary code

PAS1 also creates your source listing file. If you have a printer with your system, we recommend that you print a copy of the source listing to aid you in debugging.

Use these steps for the PAS1 portion of the compilation of your program:

1. Change the default drive to B by entering:

   B:

2. Put the scratch diskette containing your program into drive B

3. Put the PAS1 diskette into drive A

4. Enter:

   A:PAS1

PAS1 will be loaded into the computer. After a short time, the compiler will display a heading and the following prompt:

Source filename [.PAS]:_

Source filename is the name of the file in which you have stored your program. For example:

Source filename [.PAS]:myfile

It is not necessary to enter the .PAS filename extension because the compiler will look for .PAS automatically. If you gave your filename a different extension, use that extension. After you enter your source filename, you will see this prompt:

```
Object filename [MYFILE.OBJ]:_
```

Object filename is the name you want the object (machine-readable) file to have. If you wish to have your object file stored under the name MYFILE.OBJ (or whatever name appears in the brackets), you can simply press the Enter key, or you may give the file another name, taking care to add the filename extension **.OBJ**. For our example, assume we have simply pressed the Enter key:

```
Object filename [MYFILE.OBJ]:
```

The next prompt will look like this:

```
Source listing [NUL.LST]:_
```

Source listing is the name you wish to give to the file that will contain the compiled source listing. If you do *not* want a listing, press the Enter key. This will give you the default filename **NUL.LST**, which tells the compiler not to create a source listing file.

For our example, assume we do want a listing and enter:

```
Source listing [NUL.LST]:myfile
```

> **Note:** The compiler will add the .LST extension.

The last prompt is:

```
Object listing [NUL.COD]:_
```

Object listing is the name for the file that will contain the disassembled object file listing. For our example, assume we have responded as follows:

```
Object listing [NUL.COD]:myfile
```

Note: The compiler will add the .COD extension.

Here is what the completed screen would look like if you had used our example filenames:

```
Source filename [.PAS]: myfile
Object filename [MYFILE.OBJ]:
Source listing [NUL.LST]: myfile
Object listing [NUL.COD]: myfile
```

As soon as you have entered the last of these names, the compiler will begin its first pass through your program. If the program contains any syntax errors, the compiler displays the errors on the screen as well as in the listing file (see "Compiler Listing" at the end of this chapter).

When it has completed its first pass, the compiler displays a message with the number of errors and warnings it has found. The message will look like this if you sent the source listing to a file:

```
Pass One      No Errors Detected
```

If there were errors or warnings, they are displayed on the screen along with one or both of these messages:

```
Errors Detected.
Pass One Had Warnings.
```

If the compiler has indeed found errors, you must locate and fix those problems in your source program

and rerun PAS1 **before** you continue the compilation with PAS2.

If you have not copied COMMAND.COM onto your PAS1 diskette you will now be asked to insert the DOS diskette.

Continuing the Compilation: PAS2

PAS2 is the name of the second pass of the Pascal compiler. During this second pass, the compiler reads the **.SYM** and **.BIN** files made by PAS1 and creates the two object files, **.COD** and **.OBJ**. This is the optimization pass.

PAS2 creates, writes, reads, and deletes a file called **PASIBF.TMP** (the intermediate link text) as well as reading and deleting **PASIBF.SYM** and **PASIBF.BIN**.

Some programs may compile correctly during PAS1, but may produce errors during PAS2. These errors may include:

- Out of memory
- Out of range
- Overflow

When your corrections (if any were necessary) are completed, and you have run the corrected program through PAS1 again to be sure that no further syntax errors exist, you are ready to complete the compilation.

Your scratch diskette with the PAS1 files on it should be in diskette drive B.

We recommend the following steps for the PAS2 portion of the compilation of your program:

1. Take the PAS1 diskette out of drive A

2. Put the PAS2 diskette into drive A

3. Enter:

A:PAS2

PAS2 requires no input from you. It will generate the object file and listing. When the compilation is completed, PAS2 will give you a message similar to this:

```
Code Area Size = #0116 (278)
Cons Area Size = #005E (94)
Data Area Size = #000E (14)

Pass Two      No Errors Detected.
```

The Code Area Size is the total number of bytes taken up by your program (in our example, 278 bytes). Cons Area Size is the number of bytes taken up by the constants in your program (arrays, strings, structures, REALs, etc.). The Data Area Size refers to the STATIC allocated data. This area always starts at offset #2. All three sizes are given in both hexadecimal and decimal.

If errors are detected during the second pass, see Appendix A, "Messages", in this book, and correct the errors. Then rerun PAS1 and PAS2, if necessary.

Linking

We recommend that you read the *IBM Personal Computer Disk Operating System (DOS)* reference manual for an explanation of linking.

Linking should be done using the following steps:

1. Take PAS2 out of drive A

2. Put your copy of PASCAL.LIB (the one with the Linker program copied onto it) into drive A

3. Enter:

   A:link

   This will lead the Linker and give you this prompt:

   Object Modules:

   Next to the **Object Modules** prompt, enter the name of your *object file* (*not* your object *listing* file). The **.obj** extension is *not* needed here. If you used our example names, you would enter:

   Object Modules: myfile

   The next prompt will be:

   Run File:

   Next to the **Run File** prompt, enter the name you want to give to the file containing the executable code for your program. This filename will be given the extension **.exe** and put onto the default diskette drive (B). For example:

   Run File: myfile

   The next prompt is:

   List File [MYFILE.MAP]:

Pressing the Enter key will cause the Linker to choose the default filename, MYFILE.MAP. This file contains the Linker map and goes to the default drive (B).

Next is:

Libraries [ ]:_

Libraries refers to the runtime routines needed by Pascal to run your program. All these routines are included in PASCAL.LIB. In response to this prompt, you may press the Enter key:

Libraries [ ] :

When you link a Pascal program, the Pascal library is brought in automatically. You may, although it is not required, enter:

a:pascal.lib

The next five prompts are:

Publics [No]:
Line Numbers [No]:
Stack size [Object file stack]:
Load Low [Yes]:
DSAllocation [No]:

If you enter "y" (for yes) for "Publics" and "Line Numbers", these listings will appear in the file called MYFILE.MAP.

We suggest that for now you choose the default values for "Publics" and "Line Numbers" by pressing the Enter key in response to each of these two prompts.

The compiler will ignore any choice except the default responses for "Stack size" and "DSAllocation" (they are set by Pascal).

The linker will set the "Y" response to DSAllocation automatically, thus allowing Enter (the default response) as a valid reply.

The reply to "Load Low" must be "Y" which is the default response, so a reply of Enter is valid here too.

Here is what this screen of prompts would look like if you used our example filenames:

```
B>a:link
IBM Personal Computer Linker
Version 1.00 (C) Copyright IBM Corp 1981
Object Modules: myfile
Run File: myfile
List File [MYFILE.MAP]:
Libraries[ ]:
Publics[No]:
Line Numbers [No]:
Stack size [Object file stack]:
Load Low [Yes]:
DSAllocation [No]:
```

The linker will now begin to link the program. When linking has been completed, you should have the **Run File** stored on your scratch diskette in drive B. We recommend that you display the diskette directory for the scratch diskette to confirm that the run filename is there (it will have the **.exe** filename extension). Using our example filename, you would see **myfile.exe** listed in the directory.

Running Your Pascal Program

To run your program simply type your run filename, without the .exe filename extension. For example:

myfile

You may want to copy this file to another diskette once you are sure that it does what you intended it to do.

Optional PAS1 Command Lines

PAS1 can also be started using the following command line (substituting, of course, your filenames for the four files shown):

PAS1 Source File,Object File,Source List,Object List;

If you use this command line, you will not be shown the PAS1 prompts described in the PAS1 section.

> **Note:** You may add any filename extension you choose to the listing files. In our examples we use the extension .**any** to indicate this option.

Certain other variations of this command line are permitted. For instance, you may specify only:

PAS1 Source File;

Given this command line, the compiler will use the default names for the remaining files. They are:

Object File.OBJ	
NUL.LST	(no source listing)
NUL.COD	(no object listing)

Another variation is:

```
PAS1 Source.any,,;
```

This will produce an object file named **Source.OBJ**, a source listing names **Source.LST**, but no object listing.

> **Note:** Insertion of a comma will override the default names for the **.LST** and **.COD** files, and give you a listing under the SOURCE filename with the appropriate extension.

Or you may use the construction:

```
PAS1 Source,,,SCODE.ANY;
```

This will produce a **Source.OBJ**, a **Source.LST**, and a **SCODE.ANY** file.

Finally, you could specify:

```
PAS1 Source,,,;
```

This will produce the three files: **Source.OBJ**, **Source.LST**, and **Source.COD**. If you do not give the **Source File**name, then PAS1 will ask for one. If the ";" is not used to end the command line, the compiler will ask for the remaining files.

Optional PAS2 Command Lines

PAS2 can be made to pause after being started by using a parameter after the word "PAS2." For example:

```
a: PAS2 /p
```

After PAS2 had been read in, you will be prompted with:

Hit "enter" to begin pass two.

You must now press the Enter key for pass two to begin.

Optional Link Command Lines

The Linker may optionally use an "Automatic Response File." To specify this option on the command line, enter:

Link ARFILE

You must include the drive name for the Automatic Response File if it is located on a drive other than the default drive.

For example:

a:link a:ARFILE

Creating an Automatic Response File for the Linker is described in the *IBM Personal Computer Disk Operating System (DOS)* reference manual.

IBM Pascal provides an automatic response file for you to use. It is on the PASCAL.LIB diskette and is called PASCAL (no extension). You may use this file, or modify it as you wish. To use this file, enter:

a:link a:PASCAL

This file will ask you for your object modules, create a runfile called RESULT.EXE, and use the default values for the rest of the prompts.

Compiling Using a Batch File

The automatic response capability allows the Linker to run automatically using a batch file (see the *IBM Personal Computer Disk Operating System (DOS)* reference manual for an explanation of batch files (.BAT)).

The following is an example of a batch file you could use:

```
Pause.....Insert PAS1 in drive A
a:PAS1 myfile,,,;
Pause.....Insert PAS2 in drive A
a:PAS2
Pause.....Insert Linker in drive A
a:LINK    a:PASCAL
```

If you store this on your scratch diskette in a file called RUN.BAT, then all you need to do is type:

```
RUN
```

You will be prompted for PAS1, PAS2, and the Linker.

Compiling Large Programs

You may find that there is not enough space on the scratch diskette to hold all the files produced by the compiler (an "Out Of Space" error message will be displayed).

In this event, we recommend that you erase the compiled source listing from the scratch diskette before continuing on to PAS2.

It may also, on occasion, be necessary to send the **Source.LST** and **Object.COD** files to the screen (**CON** for console), to the printer (**LPT1**), to an RS232 port (**AUX** or **COM1**), or to suppress them entirely (**NUL**) to

save space on the scratch diskette for the compiler's intermediate files.

The special filenames just described can also be used with other prompts (see "Procedure ASSIGN" in Chapter 12 for an explanation of these and other alternate filenames).

In some instances, it may even be necessary to erase the source itself (first copying it to another diskette, if necessary), leaving only **PAS1BF.SYM** and **PAS1BF.BIN** on the scratch diskette. This is to provide PAS2 with sufficient room on the scratch diskette to create PASIBF.TMP and store it temporarily, and then create and save the .OBJ and .COD files.

Very large programs can be broken down into smaller **units** or **modules** and compiled separately with PAS1 and PAS2 (see Chapter 13, "Compilands", in this book, for information about how **units** and **modules** are constructed and compiled separately). They can then be joined together by the Linker to create a single run file.

This is done by specifying the object modules for each file after the Object Modules prompt given by the Linker:

Object Modules: file1, file2, file3, file4

```
LISTING FORMAT EXAMPLE                                              Page   1
CHAPTER 2                                                           07-27-81
                                                                    11:09:55
 JG IC  Line#   Source Line       IBM Personal Computer Pascal Compiler V1.00
    00      1   {$title:'LISTING FORMAT EXAMPLE',$subtitle:'CHAPTER 2'}
            2
    00      3   Program Nonsense;
    10      4   Label 10;
    10      5   Var   int,k:integer;  finished:boolean;
            6
    20      7   Procedure Print(var stop:boolean);
    20      8   label 12;
    20      9   var i,j:integer;
    30     10     Function Addit(var j:integer):integer;
    30     11     begin  (*addit*)
*   31     12       if j = 10 then return;     {return jump}
 %  31     13       int := j + addit(int)      {combination of globals}
    20     14     end;  (*addit*)
           14   -----^306 Function Assignment Not Found

Symtab     14   Offset Length   Variable - ADDIT
                -    4     10   Return offset, Frame length
                -    8      2   (function return)  :            Integer
                -    2      2   J                              :Integer VarP
```

Compiler Listing

There are references throughout the following explanation to the IBM Pascal compiler metacommands. Refer to Chapter 4 in this book for a complete description of the metacommands.

Every page of the Pascal source listing has a header at the top. In the upper left hand portion of the page, the first two lines contain the user's choice of program title and subtitle, set with the $TITLE and $SUBTITLE metacommands, respectively.

The first three lines in the upper right hand portion of the page contain the page number, the date, and the time respectively. The page number can be set with $PAGE:n, and a new page started with $PAGE+. The compiler name and version number appear on the first line below the header, along with the column labels.

The JG Column Labels

The JG columns are for flag characters generated for the user's information.

The "J" column contains jump flags which include the following:

J Flag	Meaning
+	Forward jump: GOTO label not encountered yet, or BREAK.
-	Backward jump: GOTO a label already encountered, or CYCLE.
*	Other jumps: RETURN, etc.

The "G" column flags global variables (not local to the current procedure or function). This column can have one of the three following flags:

G Flag	Meaning
=	An assignment to a non-local variable.
/	Passing a non-local variable as a reference parameter.
%	Other global references.

```
LISTING FORMAT EXAMPLE                                                  Page   1
CHAPTER 2                                                               07-27-81
                                                                        11:09:55
 JG IC  Line#   Source Line        IBM Personal Computer Pascal Compiler V1.00
    00      1   {$title:'LISTING FORMAT EXAMPLE',$subtitle:'CHAPTER 2'}
            2
    00      3   Program Nonsense;
    10      4   Label 10;
    10      5   Var   int,k:integer;  finished:boolean;
            6
    20      7   Procedure Print(var stop:boolean);
    20      8   label 12;
    20      9   var i,j:integer;
    30     10     Function Addit(var j:integer):integer;
    30     11     begin  (*addit*)
*   31     12       if j = 10 then return;     {return jump}
 %  31     13       int := j + addit(int)      {combination of globals}
    20     14     end;  (*addit*)
           14   -----^306 Function Assignment Not Found

Symtab     14   Offset Length   Variable - ADDIT
                -    4     10   Return offset, Frame length
                -    8      2   (function return)  :            Integer
                -    2      2   J                              :Integer VarP
```

The IC Column Labels

The IC columns contain the current nesting levels. The "I" column contains the (scope) level of identifiers. It changes with procedure and function declarations, as well as record declarations and the WITH statement.

The "C" column shows the control statement level. It changes with BEGIN. . . END pairs, CASE. . .END pairs, and REPEAT. . .UNTIL pairs. The numbers in this column can be used to find missing END keywords.

All these columns are blank if the line is not actively used by the compiler, so a portion of the source program that has been accidently commented out or skipped due to an $IF. . .$END pair can be found.

The Line# Column

This column contains the listing line numbers, which are internally generated. The line number and the source file name identify runtime errors if $LINE is on.

Additional Listing Metacommands

Several other metacommands affect the listing. $LINESIZE:n and $PAGESIZE:n set the width and height; $SKIP:n skips n lines; $PAGEIF:n skips to the next page if less than n lines remain; and, $LIST+ and $LIST- can be used to turn the listing on or off (errors are always listed).

The metacommands themselves appear in the listing, except for $LIST-. $SYMTAB+ controls the listing of data about variables. The symbol table listing is completely independent of $LIST.

```
    20      15   begin  (*print*)
+   21      16     if k>10 then [stop:=true; goto 12]; {forward jump}
    21      17     for i := k to 10
    21      18     begin
      ^     17   ------------------^Warning 172 Insert DO
  / 22      19        j := addit(int);          {passing a global}
    22      20        writlen(j)
            20   ----------^303 Unknown Identifier Skip Statement
            20   -187 End Skip^
    21      21     end;
  = 21      22     12: k := k + 1;              {assign to global}
    21      23     writeln('Exit Procedure Print for the ',k:1,'th time')
            23   ---------^Warning 238 Assumed OUTPUT
    10      24   end;  (*print*)

 Symtab     24   Offset Length   Variable - PRINT
                 -    2     12   Return offset, Frame length
                 -    0      2   STOP                              :Boolean VarP
                 -    6      2   I                                 :Integer
                 -    8      2   J                                 :Integer


 PRINT
```

Compiler Messages

Two kinds of compiler messages appear in the listing: **errors** and **warnings**. A compilation with any errors cannot be used to generate code; one with only warnings can be used to generate code, but the result may not execute correctly.

Warnings start with the word "Warning" followed by a number. **Errors** start with a number. Errors and warnings are listed by number in Appendix A of this book. Warnings can be suppressed with $WARN-, but we do not recommend it.

Errors and warnings are displayed at the user's console with $BRAVE+. The location of the error is indicated by a caret. The message itself may appear either to the left or to the right of the caret, and it is preceded by a dashed line.

Sometimes an error in a line is not detected until the following line has been listed. In this case, the error message line number will not be in sequence. Tabs are allowed in the source and are passed on to the listing unchanged. If the tab spacing is every eight columns, the error pointer is generally correct, except for an occasional error near the end of a line if the following line has tabs.

Unrecoverable Errors

The compiler may find an error from which it cannot recover. In this case, it gives the message: "Compiler Cannot Continue!"

```
JG IC  Line#   Source Line        IBM Personal Computer Pascal Compiler V1.00
           25  {$page+}
   10      26  begin  (*main*)
   11      27    k := 1;
   11      28    finished := false,
           28  --------------------^Warning 156 , Assumed ;
   11      29    10: print(finished);
-  11      30    if not finished then goto 10; {backwards jump}
   00      31  end.  (*main*)

Symtab     31  Offset Length   Variable
                    0      8   Return offset, Frame length
                    4      2   K                          :Integer Static
                    2      2   INT                        :Integer Static
                    6      1   FINISHED                   :Boolean Static


                Errors  Warns  In Pass One
                    3      3
```

This can occur at the beginning if the keyword PROGRAM (or IMPLEMENTATION, or INTERFACE, or MODULE) is not found, or the program or unit identifier is missing. It can occur at the end of the listing if an unexpected end of file is encountered.

It may also be triggered by the detection of too many errors. The maximum number of errors per page is set with $ERRORS:n (default is 25). In all these cases, the compiler lists the rest of the program with very little error checking.

The Symbol Table

For a complete explanation of the symbol table, see "SYMTAB" in Chapter 4 of this book, and "Internal Calling Conventions" in Chapter 10.

Start	Stop	Length	Name	Class
00000H	000F9H	00FAH	EXAMPLE	CODE
00100H	00425H	0326H	MISGQQ	CODE
00426H	00D63H	093EH	ERREQQ_CODE	CODE
00D70H	00D70H	0000H	INIXQQ	CODE
00D70H	00E36H	00C7H	ENTXQQ	CODE
00E38H	01EDAH	10A3H	FILUQQ_CODE	CODE
01EDCH	024B6H	05DBH	ORDFQQ_CODE	CODE
024B8H	03504H	104DH	FILFQQ_CODE	CODE
03506H	03533H	002EH	MISOQQ_CODE	CODE
03534H	03DA1H	086EH	CODCQQ_CODE	CODE
03DA2H	03EC2H	0121H	UTLXQQ_CODE	CODE
03EC4H	04254H	0391H	STRFQQ_CODE	CODE
04256H	043BCH	0167H	PASUQQ_CODE	CODE
043BEH	0456EH	01B1H	HEAHQQ_CODE	CODE
04570H	04615H	00A6H	MISHQQ_CODE	CODE
04616H	04705H	00F0H	MISYQQ_CODE	CODE
04710H	04710H	0000H	HEAP	MEMOR'
04710H	04710H	0000H	MEMORY	MEMOR'
04710H	0490FH	0200H	STACK	STACK
04910H	04FE7H	06D8H	DATA	DATA
04FF0H	056F1H	0702H	CONST	CONST
05700H	05700H	0000H	??SEG	

This listing is a map of the executable file. It shows the relative displacement from the beginning of the file of th various elements of the program. The publics map and tl line number map show where the different elements of these are relative to the beginning of the file.

The partial map given is for the following program:

```
 1   Program Example(output);
 2
 3   Var first,second,answer:integer;
 4
 5   Function Addit(i,j:integer):integer;
 6   begin
 7      addit:= i+j;
 8   end;
 9
10   begin
```

```
11      first:=2; second:=2;
12      answer:=addit(first,second);
13      writeln(first:1,' + ',second:1,' = ',answer:1);
14  end.
```

If the linker prompt "Publics [No] ?" is responded to with a "Y", the following two listings are produced: "Publics by Name" and "Publics by Value."

The address has the form:

segment:offset.

For example:

Address	Publics by Name
043B:000FH	ALLHQQ
0010:0247H	ASMGQQ
0042:0470H	ASNEQQ
0010:0265H	ASNGQQ
024B:0624H	ASSFQQ
0010:018EH	AVAGQQ
F570:F242H	BEGHQQ
0350:0007H	BEGOQQ
00D7:0000H	BEGXQQ
0010:0000H	BRTEQQ
024B:06DAH	BUFFQQ
0425:0007H	BUFUQQ
0010:01F8H	CESGQQ

-
-
-
-

Address	Publics by Value
0000:0031H	EXAMPLE
0000:0031H	ENTGQQ
0010:0000H	BRTEQQ
0010:002BH	ERTEQQ
0010:003BH	MOVEL
0010:0057H	MOVER
0010:0074H	FILLC
0010:0092H	MOVESL
0010:00B6H	MOVESR
0010:00DBH	FILLSC
0010:00FAH	UADDOK
0010:0113H	UMULOK
0010:012CH	SADDOK
0010:014CH	SMULOK
0010:016CH	LOCKED
0010:0180H	UNLOCK
0010:018EH	AVAGQQ
0010:01AEH	SOVGQQ

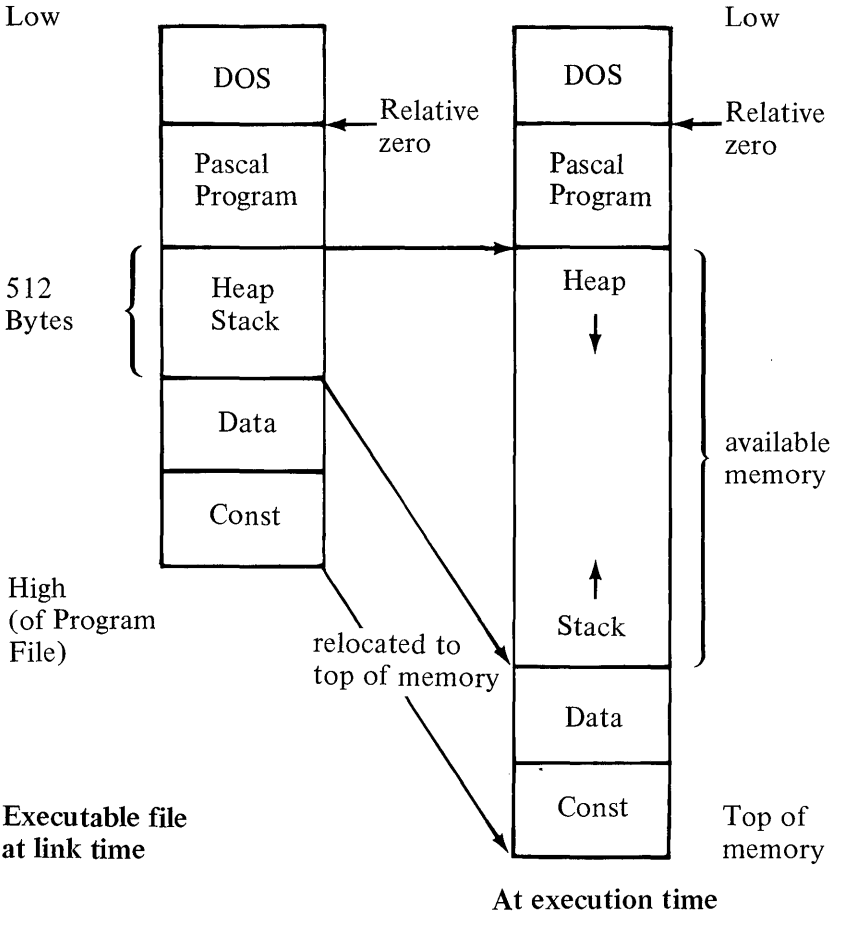
•
•
•
•

If the linker prompt "Line No [No]:" is responded to with a "Y", the following map is produced, broken down by line number. The line numbers listed in the runtime modules are the original line numbers from when the modules themselves were compiled.

Here is a sample, partial listing of the line number map:

```
Line numbers for EXAMPLE

   7 0000:0019H       8 0000:0027H
  11 0000:004EH      12 0000:005AH
  13 0000:0068H      14 0000:00DCH

Line numbers for ERREQQ_CODE

  18 0042:0011H      24 0042:0036H
  25 0042:0044H      27 0042:004FH
  28 0042:0060H      29 0042:006AH
       •
       •
       •
 213 0042:090EH     214 0042:0918H

 Program entry point at 00D7:0000

Line numbers for FILUQQ_CODE

 185 00E3:0013H     186 00E3:0023H
       •
       •
       •
```

The following diagram illustrates how the link map is translated during execution time:



Note at execution time:

1. The code segment:offsets are correct relative to zero (load point of program just above DOS).

2. The data and constant references use the map offsets relative to the DS register, *not* the segment value displayed in the maps.

CHAPTER 3. NOTATION AND TERMINOLOGY

Contents

In this chapter, IBM Pascal notation and terminology will be discussed in the context of the different "levels" of Pascal represented in this book. Other topics covered will be syntax and vocabulary, Pascal reserved words, attributes, directives, and predeclared identifiers.

Pascal Levels

Pascal on your IBM Personal Computer is divided into four "levels": metalanguage, standard Pascal, extended Pascal, and systems Pascal.

Metalanguage

The metalanguage is used for compiler option setting and conditional compilation.

Standard Pascal

All standard ISO Pascal programs are intended to compile and run correctly using IBM Pascal. All of the extensions to the language are documented in Appendix E of this manual.

Extended Pascal

IBM Pascal enhances ISO Pascal by providing relatively "safe" extensions so that programs can be better engineered. These extensions include such constructions as the BREAK statement, UNITs, and structured constants.

Systems Pascal

IBM Pascal also provides other more sophisticated extensions, including those either useful or necessary for systems programming. These additional extensions include the address types and the RETYPE function.

Syntax and Vocabulary

This manual is an informal description of Pascal syntax. The syntactic examples are intended to aid understanding only and do not form the complete grammar. The syntax for IBM Personal Computer Pascal is described in Appendix F.

The basic Pascal vocabulary consists of letters, digits, reserved words, and special characters. Lowercase and uppercase letters are interchangeable, except in string literals.

The special characters are grouped by category as follows:

- **Metalanguage**

  $ is used to prefix metacommands only.

- **Standard Pascal**

  + - * / = < > ( ) [ ] . , : ; ' { } <> <= >= := .. ∧. The "_" (underscore) is allowed in identifiers.

- **Substitutes**

  (*...*) for {...} (. for [.) for].
  The ? or @ may be substituted for the ∧ to denote a pointer.

- **Higher level substitutes**

  "#" in integer constants (non-decimal number feature). "!" comment to the end of the line. [...] for BEGIN...END.

- **Unused characters**

  % & " | ~ ` \

Spaces (blanks), tabs, and form feeds are also part of the standard character set. A tab, CHR(9), is treated as a space and passed on to the listing file. A form feed, CHR(12), is converted to a new-page metacommand ($PAGE+).

All other characters including CHR(0) to CHR(31), the unused characters just listed, and characters from CHR(127) to CHR(255) generate an error when used in a source file, unless they appear in a comment or a string literal.

Pairing (. with] or [with .) is allowed (but not recommended). { must be paired with } and (* with *).

Pascal Reserved Words

These words are reserved in Pascal:

AND	END	NIL	SET
ARRAY	FILE	NOT	THEN
BEGIN	FOR	OF	TO
CASE	FUNCTION	OR	TYPE
CONST	GOTO	PACKED	UNTIL
DIV	IF	PROCEDURE	VAR
DO	IN	PROGRAM	WHILE
DOWNTO	LABEL	RECORD	WITH
ELSE	MOD	REPEAT	

IBM Pascal features add the following reserved words:

FEATURE	RESERVED WORDS
Unit Interface	IMPLEMENTATION
	INTERFACE
	UNIT
	USES
Modules	MODULE
Extended CASE	OTHERWISE
Super Array Type	SUPER
Control Flow	BREAK
	CYCLE
	RETURN

FEATURE	RESERVED WORDS
Extended Operator	XOR
Address Type	ADR
	ADS
	VARS
Value Section	VALUE

Attributes

An "attribute" is a keyword used to give a special characteristic to a variable, procedure, or function. The following attributes are reserved words:

EXTERN	PUBLIC	READONLY
EXTERNAL	PURE	STATIC

Directives

A "directive" is a word used in place of a procedure or function block.

EXTERN EXTERNAL FORWARD

> **Note:** EXTERN is both an attribute and a directive. EXTERNAL is a synonym for EXTERN, providing compatibility with a number of other Pascals.

Predeclared Identifiers

The following are predeclared identifiers. They can be re-defined by the programmer, but doing this is not recommended. They are:

- **Standard**

ARCTAN	FALSE	OUTPUT	SIN
ABS	FLOAT	PACK	SQR
BOOLEAN	GET	PRED	SQRT
CHAR	INPUT	PUT	SUCC
CHR	INTEGER	READ	TEXT
COS	LN	READLN	TRUE
DISPOSE	MAXINT	REAL	TRUNC
EOL	NEW	RESET	UNPACK
EOLN	ODD	REWRITE	WRITE
EXP	ORD	ROUND	WRITELN

- **Extended Intrinsics Feature**

ABORT	EVAL	RESULT
BYWORD	HIBYTE	SIZEOF
DECODE	LOBYTE	UPPER
ENCODE	LOWER	

- **String Intrinsics Feature**

CONCAT	POSITN	COPYLST
DELETE	SCANEQ	COPYSTR
INSERT	SCANNE	

- **System Intrinsics Feature**

FILLC	MOVER	FILLSC	MOVESR
MOVEL	RETYPE	MOVESL	

- **Extended I/O Feature**

ASSIGN	DISCARD	READFN	SEQUENTIAL
CLOSE	FILEMODES	READSET	TERMINAL

- **System I/O Feature**

FCBFQQ

- **WORD Type Feature**

MAXWORD	WORD	WRD

- **Super Array Type Feature**

LSTRING	NULL	STRING

Comments

Comments are enclosed in { } or (* *) and can span multiple lines. The use of a "!" will start a comment which can run to the end of that line. Comments with the same delimiter cannot be nested.

Separators

A comment, blank, or line boundary must occur between two adjacent numbers or adjacent reserved words and identifiers, and cannot be embedded within them.

IBM Pascal will, in some cases, accept these without a separator and give no error. For example:

100mod#50	is accepted as 100 mod #50
100mod127	is assumed to be 100, followed by the identifier mod127

CHAPTER 4. COMPILER COMMANDS (METALANGUAGE)

Contents

This chapter introduces a series of IBM Pascal compiler commands, called *metacommands*, because they function conceptually "between" the compiler and the user.

There are 30 of these special *metacommands* in the *metalanguage* which allow you to control compiler debugging, listing file format, and source file execution.

The remainder of the chapter is devoted to a detailed look at each of these metacommands and its use.

Metacommands

You can give instructions to the compiler by using any of a group of specially formatted commands which we call *metacommands*. The metacommands are used for compiler option setting and conditional compilation.

They include all the compiler directives and provide a limited amount of string processing on the source document.

One or more *metacommands*, optionally separated by commas, can be given only at the *start* of a comment. A metacommand imbedded within a comment is *ignored*. Blanks, tabs, and line ends between the elements of a metacommand are also ignored, so {$PAGE:12} is the same as { $ PAGE : 12 }.

The metacommands can be grouped in three general categories as follows:

- Debugging and Error Handling:

  $BRAVE
  $DEBUG
  $ENTRY
  $ERRORS
  $GOTO
  $INDEXCK
  $INITCK
  $LINE
  $MATHCK
  $NILCK
  $RANGECK
  $RUNTIME
  $STACKCK
  $WARN

- Listing File Format:

  $LINESIZE
  $LIST
  $OCODE
  $PAGE
  $PAGEIF
  $PAGESIZE
  $SKIP
  $SUBTITLE
  $SYMTAB
  $TITLE

- Source File Control:

  $IF . . $THEN . . $ELSE . . $END
  $INCLUDE
  $INCONST
  $MESSAGE
  $POP
  $PUSH

Most compiler commands can be changed within a program. For example, a large program being recompiled might use $LIST-, with a few sections using $LIST+ as needed to show those lines where changes occurred.

The compiler actually processes the metacommands that follow a symbol before the symbol itself, because the compiler keeps one lookahead symbol. For example:

```
CONST Q=1; {$IF Q $THEN}
   {Q is undefined in the $IF}

X : = P^; {$NILCK+}
   {NILCK applies to P^ here}
```

Metacommands invoke or set the value of a *metavariable.* Metavariables are either:

- TYPELESS: those that are invoked when used, as in $PUSH.

- INTEGER: those that can be set to a numeric value, as in $PAGE:101.

- ON/OFF: those that can also have a numeric value such that a value > 0 turns on the switch and a valu <= 0 turns it off, as in $MATHCK:1 (+ and - may also be used).

- STRING: those that take a string literal or string constant identifer value, as in $TITLE: 'My name'.

Metavariables are set as follows:

COMMAND	RESULT
$METAVAR+	Sets the value to 1 (on)
$METAVAR-	Sets the value to 0 (off)
$METAVAR:number	Sets value to literal integer constant
$METAVAR:identifier	Sets value equal to constant identifier

In metacommands only, Boolean and enumerated constants are changed to their ORD (ordinal) values. For Boolean, false becomes 0 and true becomes 1.

Although constant expressions are not allowed, the effect of a constant expression can be obtained by declaring a constant identifier equal to the expression and using the identifier in the metacommand.

In the following descriptions, a metacommand followed by + or - is an on/off switch, one followed by an :n is an integer, one followed by :'text' is a string, and one without a following item is typeless.

Error Conditions

There are five kinds of error conditions:

1. Compiletime *warnings*

2. Compiletime *caught* errors

   Note: An error is *caught* if the compiler or runtime system discovers the error and gives the user a message.

3. ISO Pascal compiletime and runtime *errors not caught*

4. Runtime *switchable-caught* errors

5. Runtime *always caught* errors

A "warning" is a caught error that the compiler has fixed so that a source program might run correctly. These warnings include common substitution errors and syntax errors.

Compiletime caught errors include all the errors described in this manual and not otherwise classified; for example, something which is "invalid," "illegal," or "not permitted."

ISO Pascal errors not caught are always identified as such. They are generally infrequent or very hard to detect conditions, given as an error in the ISO Standard.

For some runtime errors, checking can be switched on or off, but this does not guarantee that these errors are never caught. Due to optimization, an invalid operation with error checking off may not have the desired effect. Runtime errors always caught do not generate extra code, but the switchable ones may.

These runtime errors are *always caught*:

- Heap overflow during NEW variable allocation

- Incorrect variable size in the long form of DISPOSE
- Version number mismatch, using an interface
- REAL number overflow
- Transcendental function errors
- READ an out-of-range value to a subrange variable

The following examples illustrate runtime error messages as they might appear (see Appendix A, "Messages", in this book for a complete list of the error messages):

```
? Error: No Room In Heap
  Error Code 2001
PC = 1F0C:199, FP = EAFE, SP = EAF4
```

The error code number and message appear in Appendix A. PC is the program counter, FP is the frame pointer, and SP is the stack pointer. Their values are given in hexadecimal notation.

> **Note:** Another possible error (but one which should not occur) is an INTERNAL ERROR. An internal error means a malfunction has occurred in the compiler. Report this problem to your compiler dealer along with a description of the conditions under which it occurred.

The following is an example of an Internal Error message:

```
*** Internal Error #510 ***
   Near Line 220
   Contact Technical Support
Error: Compiler Internal Error
```

The following is an alphabetical list of the IBM Pascal metacommands:

$BRAVE

Purpose: Causes errors and warnings to be given on the display screen. If $BRAVE–, errors will not appear at the display screen, but will still appear in the listing file.

Format: $BRAVE+ (Default is +)

$BRAVE–

Purpose: Turns on or off all the runtime debug checking the compiler can do.

Format: $DEBUG+ (Default is +)

$DEBUG–

Remarks: $DEBUG checking is always on unless turned off. This includes:

$ENTRY

$INDEXCK

$INITCK

$MATHCK

$NILCK

$RANGECK

$STACKCK

These may also be turned off individually.

$ENTRY

Purpose: Generates procedure entry/exit calls for the debugger.

Format: $ENTRY- (Default is -)

$ENTRY+

Remarks: Gives the name of the procedure or function in which the error occurred, and displays it on the screen as part of the runtime error message.

If $ENTRY- is used after $LINE, then $LINE+ is also turned off (see $LINE).

Purpose: Sets the number of errors that will be allowed per page of the listing.

Format: $ERRORS:n (Default is 25)

$GOTO

Purpose: Causes each GOTO statement in the listing to be flagged with a "considered harmful" warning.

Format: $GOTO- (Default is -)

$GOTO+

$IF . . . $THEN . . . $ELSE . . . $END

Format:

$IF constant $THEN . . . text1 . . .
$END

$IF constant $ELSE . . . text2 . . .
$END

$IF constant $THEN . . . text1 . . .
$ELSE . . . text2 . . . $END

Remarks: If constant is true (>0), process text1, skip text 2.
If constant is false (<=0), skip text1, process text2.

Constant can be a literal number or a constant identifier. It cannot be an expression.

Text is arbitrary, and can include line breaks, comments, and other metacommands (including nested $IFs).

Metacommands within skipped text are ignored, except corresponding $ELSE and $END commands.

Example:

```
{$IF CHIP $THEN}
     CODEGEN (FADCALL,T1){$END}
{$IF DOSYS $ELSE}
     IF MYSYS THEN DOITTOIT{$END}
```

$INCLUDE

Purpose: Switches to source file 'name' after current source line. Switches back when EOF encountered.

Format: $INCLUDE:'name'

Remarks: Any source file can be $INCLUDEd. It is especially useful when using UNITs.

'name' is a standard DOS filename.

Purpose: Prompts user for constant declaration.

Format: $INCONST:identifier

Remarks: The constants in metaconditional statements can be changed at compile time if the user wishes to compile different versions of source code.

The source code, then, does not have to be edited for each change. When the compiler encounters $INCONST:identifier, it prompts the user to type a WORD value.

The effect is the same as if a CONST section with the WORD value had been processed. That value is available for use in expressions, type clauses, etc.

An $INCONST metacommand should appear somewhere in or after the first program section (that is, LABEL, CONST).

$INDEXCK

Purpose: Checks for array index values in range.

Format: $INDEXCK+ (Default is +)

$INDEXCK-

Remarks: Includes super array indices. Since array indexing occurs so often, bounds checking is separate from other subrange checking.

Purpose: Generates code to set the value of all uninitialized integers to -32768 and uninitialized pointers to 1 (if $NILCK is on).

Format: $INITCK- (Default is -)

$INITCK+

Remarks: The following classes are *not* initialized with $INITCK:

- Variables mentioned in a VALUE section
- Variant fields in a record
- Components of a SUPER ARRAY allocated with NEW

$LINE

Purpose: Generates line number calls for the debugger so the run-time system can report the line number where the error occurred.

Format: $LINE- (Default is -)

$LINE+

Remarks: If $LINE+, then $ENTRY+ is automatically generated.

Example:

```
? Error: No Room In Heap
   Error Code 2001
   Line  20 In (proc. name) Of (prog. name)
PC = 1F0C:199, FP = EAFE, SP = EAF4
```

Purpose: Sets the width of the program listing.

Format: $LINESIZE:n (Default is 79)

Remarks: Useful when changing the printer line length from 80 to 132 characters.

$LIST

Purpose: Generates the source code listing.

Format: $LIST+ (Default is +)

$LIST-

Remarks: Errors are always listed. $LIST is useful, for instance, if you want to make a change to a large program, and you want a listing of only the change. You can cause the partial listing by placing $LIST+ at the start of the new code and $LIST- after the end of the new code.

Purpose: Detects INTEGER and WORD overflow, and division by zero.

Format: $MATHCK+ (Default is +)

$MATHCK-

Remarks: Checking for an INTEGER result of exactly -MAXINT-1 (that is, #8000) is not included.

Turning off $MATHCK does not always disable overflow checking. Library routines described in Chapter 11 provide addition and multiplication functions which always permit overflow.

$MESSAGE

Purpose: Displays text on the display screen *during* compilation.

Format: $MESSAGE:'text'

Remarks: Useful when metaconditionals ($IF $THEN $ELSE $END) are extensively used to indicate, at the terminal, which version of a program is being compiled.

$MESSAGE is also useful when using $INCONST to prompt the user for necessary input.

Purpose: Checks for dereferencing a pointer.

Format: $NILCK+ (Default is +)

$NILCK-

Remarks: Checks for dereferencing a pointer whose value is:

- NIL (a value of 0)
- Uninitialized (value of 1; only with $INITCK)
- Out of range
- Pointing to a free block
- Odd (valid pointers are even)

Occurs when a pointer is dereferenced or passed to DISPOSE. No address checking is done.

> **Note:** "Dereferencing" occurs when the value at the memory address contained in a pointer is accessed.

$OCODE

Purpose: Turns on or off the disassembled object code listing.

Format: $OCODE+ (Default is +)

$OCODE-

Remarks: The $OCODE command controls listing the generated code the way $LIST controls the source code listing.

The format is basically like an assembly listing, with code addresses and operation mnemonics. Procedure, function, and static variable identifiers may be truncated in the object code listing file.

Due to optimization during PAS2, the $OCODE metacommand may not always take effect in the desired spot.

Purpose: Sets page number for the next page.

Format: $PAGE:n

Remarks: Does not skip to next page.

$PAGE

Purpose: Skips to the next page.

Format: $PAGE

Remarks: The line numbers are not reset, as in $PAGE:n.

Purpose: Skips to the next page if less than n lines are left.

Format: $PAGEIF:n (Default is no action)

$PAGESIZE

Purpose: Sets the page length of the listing in lines.

Format: $PAGESIZE:n (Default is 53)

Remarks: This command must be on the first line of the text, or it will not take effect until the second page.

Purpose: Saves/restores the value of the current metacommands.

Format: $PUSH

$POP

Remarks: Not all metacommands can be recovered with $POP. For example, if the default value for $LIST is used (that is, $LIST+ is not explicitly included in a comment), and then the $PUSH command is used followed by a $LIST- (for an $INCLUDEd file which you do not want listed, for example), and $POP is then used, the $LIST- will still be in effect.

$RANGECK

Purpose: Checks for subrange validity.

Format: $RANGECK+ (Default is +)

$RANGECK-

Remarks: Includes:

- Assignment to subrange variables (including FOR control variables and structured constant values)
- The CHR and BYWORD functions
- Super array upper bounds passed to a NEW procedure
- The SUCC and PRED functions
- Indices in PACK and UNPACK
- LSTRING assignments and value parameters
- Set type assignments and value parameters
- Checking the CASE statement control value when there is no OTHERWISE clause

Purpose: Special switch for Pascal runtime use.

Format: $RUNTIME- (Default is -)

$RUNTIME+

Remarks: If the $RUNTIME switch is on when a procedure or function is compiled, the "location of the error" becomes the point where the procedure or function was called, not where it occurs within the procedure or function itself.

This command is the object code counterpart to $LINE and $ENTRY.

The $RUNTIME switch effects the Stack Pointer (SP), the Frame Pointer (FP), and the Program Counter (PC).

$SKIP

Purpose: Skips n lines, or to the end of the page, whichever occurs first.

Format: $SKIP:n

Purpose: Checks for stack overflow at procedure and function entry, and when pushing parameters larger than four bytes onto the stack.

Format: $STACKCK+ (Default is +)

$STACKCK-

$SUBTITLE

Purpose: Sets listing page subtitle to 'text'.

Format: $SUBTITLE:'text'

Remarks: 'text' will appear in the upper left hand corner of every page of the listing on the second printed line.

The command must appear on the first line for the subtitle to appear on the first page.

Maximum length should be ten less than $LINESIZE for proper alignment to occur.

Purpose: Lists the variables for a program, procedure, or function at the end of the listing.

Format: $SYMTAB+ (Default is +)

$SYMTAB-

Remarks: If $SYMTAB is on at the end of a procedure, function, or program, the listing contains information about its variables. The left columns contain the offset to the variable from the frame pointer (procedures and functions), or in fixed memory (main program and STATIC attribute variables).

A leading minus sign indicates a frame offset. This offset is to the lowest address used by the variable, since the frame grows from higher to lower addresses.

The first line of the $SYMTAB listing contains the offset to the return address (zero for the main program) and the total length of the frame, including front end temporaries but excluding code generator temporaries.

For functions, the second line contains the offset, length, and the type of the returned value.

The remaining lines list the variables including their

type and keywords indicating various attributes. These are as follows:

KEYWORD	ATTRIBUTE
Public	Has the PUBLIC attribute
Extern	Has the EXTERN attribute
Origin	Has the ORIGIN attribute
Static	Has the STATIC attribute
Const	Has the READONLY attribute
Value	Occurs in a VALUE section
ValueP	Is a value parameter
VarP	Is a VAR or CONST parameter
VarsP	Is a VARS or CONS parameter
ProcP	Is a procedural parameter

Note: The entries shown in the symbol table on your listing are arranged alphabetically within type and not as they exist in memory. See the section on "Internal Representation", in Chapter 6, for more information on the offsets.

Purpose: Sets listing page title to 'text'.

Format: $TITLE:'text' (Default is no title)

Remarks: 'text' will appear in the upper left hand corner of every page of the listing on the first printed line.

The command must appear on the first line for the title to appear on the first page.

Maximum length should be ten less than $LINESIZE for proper alignment to occur.

> **Note:** The logical line can be longer than one physical line, allowing several metacommands to be put on the "first" line.

$WARN

Purpose: Gives warning messages in the listing file.

Format: $WARN+ (Default is +)

$WARN-

CHAPTER 5. IDENTIFIERS AND CONSTANTS

Contents

Identifiers

Identifiers denote constants, types, variables, procedures, functions, programs, and fields and tagfields in records. Some features also use identifiers; as super array types, modules, units, and statement labels.

Statement labels are unsigned integers or identifiers, and have the same scope rules as identifiers; leading zeros are not significant. An identifier consists of a letter followed by letters, digits, or _s; the _ (underscore) is significant. Identifiers with _ in them are allowed, a minor extension to ISO Pascal. Two _s in a row, or an identifier ending with _, is permitted.

Length Restrictions

Identifiers can be of any length, but must fit on one line. Only the first 31 characters of an identifier are significant. An identifier longer than 31 characters causes a warning.

The identifiers passed to the linker may also be 31 characters long. These include those identifiers used for a program, module, or unit which are passed to the linker, as well as those with the PUBLIC or EXTERN attitude.

In the disassembled object code, variable and procedure identifiers may be truncated to six characters if these identifiers are not used by the linker.

Using identifiers of seven or fewer characters saves space during compilation.

All external identifiers used by the runtime system are four alphabetic characters followed by "QQ", therefore users should avoid this form when making external names.

Scope

Identifier

The scope of an identifier is the same as in ISO Pascal. An identifier's association must be unique within its scope.

Basically, the scope of an identifier is the procedure, function, program, module, implementation, or interface in which it is defined, including any nested procedures or functions.

A nested procedure or function can redefine an identifier *only* if the identifier has not already been used (so if M is an integer constant in a program, a nested procedure cannot do CONST Q = M + 1; M = 4).

However, this error is not caught except when T is declared as a type identifier, and a nested procedure or function declares a reference type ^T and in the same TYPE section declares T again.

The meaning of a type ^T would be ambiguous. In this case a warning is given and the T that occurs later in the TYPE section is used as the referent type of ^T.

Procedure or Function

The scope of a procedure or function begins immediately after the procedure or function identifier, so it includes the formal parameter list. This means the procedure or function identifier itself is one level "above" its parameters, variables, and nested procedure and function identifiers.

For example, declaring a variable in a function with the same identifier as the function is permitted, even though this hides the function identifier so a value cannot be assigned to it.

A program, module, or unit identifier becomes part of the level "above" other user identifiers. However, this level contains the predeclared identifiers, so using a program identifier like INTEGER or READ produces a duplicate identifier error.

Field or Tagfield

The scope of a field (or tagfield) of a record is any field designator or WITH statement using a variable of the record's type. With $LIST+, the current identifier level can be seen in the I column of the listing as one of the 16 identifier levels provided (see Chapter 2 for an explanation of the compiler listing symbols).

An identifier must be defined before it is used, except for program parameters and interface constituents, and referent types.

When a referent type ^T (or ADR OF T or ADS OF T) is defined, the type T it refers to can occur later in the same TYPE section, allowing indirect recursion.

FORWARD Directive

The FORWARD directive, described later, allows indirect recursion for procedures and functions. Identifiers which are parameters in a procedure or function heading with the FORWARD directive do not take effect until the actual procedure or function's block is encountered.

Identifiers used when declaring the parameters of a procedure or function used as a parameter (for example, a procedural type parameter) are ignored.

With the unit interface feature, a USES clause defines a group of identifiers listed in the interface or listed in the USES clause itself. These identifiers are "constituents" of a "unit" which is defined in an "interface" and actually brought about by an "implementation". Units are discussed in Chapter 13.

For example, USES (IVAR, COLOR, COLORPROC) defines the three identifiers in the list, which can be constants, variables, types, super array types, procedures, or functions.

If COLOR is the enumerated type (RED, BLUE, GREEN), then these enumerated type constant identifiers are not automatically defined; they must be listed in the interface as well. However, the field identifiers of a constituent record variable or type are available in a field designator ("."), or WITH statement as usual.

Constants

Numeric Constants

The usual decimal notation is used for numbers, which are constants of type REAL or INTEGER or WORD. The type of a number is REAL if it includes a decimal point or exponent, and INTEGER or WORD if it does not.

INTEGER constants can range from -MAXINT to MAXINT (MAXINT is a predeclared constant equal to 32767).

WORD constants can range from 0 to MAXWORD (MAXWORD is a predeclared constant equal to 65535).

A number from -MAXINT to MAXINT becomes type INTEGER, or else one from MAXINT+1 to MAXWORD becomes type WORD, or else an error occurs.

However, any INTEGER type constant (including constant expressions and values from -32767 to -1) automatically changes to type WORD if necessary; if an INTEGER value is negative, 65536 is added to it (that is, the 16 bit underlying value is not changed).

As an example, a constant subrange of type WORD can be declared as WRD(0)..127; the upper bound is automatically given the type WORD. *The reverse is not true*; constants of type WORD do *not* change to type INTEGER automatically. The ORD and WRD functions can also be used to change the type of an ordinal constant to INTEGER or WORD. Examples:

0	type INTEGER, could become WORD
32768	type WORD
0..20000	type INTEGER subrange
0..50000	type WORD subrange
0..80000	invalid
-1..50000	invalid, becomes 65535..50000

The REAL constant range provides about seven digits of precision, with a maximum value of about 1.70141200E+38 (for more details see "Internal Representation" in Chapter 7 of this book).

At least one digit is expected on each side of a decimal point, or a warning is given.

A real number starting or ending with a decimal point may be misleading; for example, "(.1+2.)" is interpreted as "[1+2]" since "(." substitutes for "[" and ".)" for "]".

Scientific notation in real numbers (as in 1.23E-6 or 4e7) is supported. When an exponent is given, the decimal point and exponent sign are optional. An uppercase"E" or lowercase "e" in real numbers is allowed.

Numbers can have a leading + or -, except within an expression (so "ALPHA :=+10" is valid but "ALPHA +-10" is invalid). Embedded blanks (after the sign, in the exponent part, etc.) are not permitted.

The compiler truncates any number with more than 31 characters (the same as the identifier length) with a warning.

The syntax for numbers applies to textfiles as well as programs; all numbers are converted by the compiler using the DECODE function. Examples of numeric constants:

123	+12.345	0.17	007
-26.0	26.0e12	-1.7E-10	
17e+3	-17E3	1e1	

The nondecimal number feature supports numbers in hexadecimal, octal, or binary radix, as well as decimal,

as in 16#FF02, 10#987, 8#776, or 2#111100. Leading zeros are permitted in the radix, as in 008#147.

In hexadecimal notation, uppercase or lowercase A-F is permitted. A nondecimal constant without the radix (such as #44) is assumed to be hexadecimal.

Whether the usual decimal or nondecimal notation is used, the type of a number is determined as described above; nondecimal notation does not imply the WORD type, for example. Nondecimal notation cannot be used for REAL constants, or numeric statement labels.

Any numeric constant (literal or identifier) can be given a + or – sign. The constant expression feature allows ordinal constants to be expressions involving nonreal numbers, constant identifiers, and some operators and predeclared functions. With this feature, a constant expression is allowed anywhere a literal constant can appear, except in metacommands.

Constants can appear in the CONST section, expressions, type clauses, set constants and other structured constants, and in CASE constants in CASE statements and variant record tag values.

Constant Operators and Functions

Constant operators with REAL or INTEGER operand:

unary + unary –

Constant operators and functions with INTEGER or WORD operands:

+	DIV	OR	HIBYTE( )
–	MOD	NOT	LOBYTE( )
*	AND	XOR	BYWORD( )

Constant operators and functions with ordinal type operands:

<	>=	ORD( )	LOWER( )
<=	=	CHR( )	UPPER( )
>	<>	WRD( )	

Constant operators with Boolean operands:

AND
OR
NOT

Constant functions with an array operand:

LOWER( )
UPPER( )

Constant functions with any type of operand:

SIZEOF( )
RETYPE( )

For example:

```
(100 + ORD('X')) * 8#100 + ORD('Y')
(MAXSIZE > 80) OR (INTYPE = PAPERTAPE)
```

Strings

Sequences of characters enclosed in single quotes are called strings in most Pascal manuals. We call them *string literals*, as opposed to *string constants* (which may be an expression) or the STRING type.

A string constant can have from 1 to 255 characters. A string constant longer than one character is of type PACKED ARRAY [1..n] OF CHAR, also known as the type STRING (n). A string constant containing just one character is of type CHAR.

However, the type changes from CHAR to PACKED ARRAY [1..1] OF CHAR (for example, STRING (1)) if necessary for assignment, parameter passing, etc. A literal single quote is represented by two adjacent quotes

The null string '' is not permitted (but see NULL below) A string literal must fit on a line. String literals using " instead of ' are recognized, but a warning message is given.

The constant expression feature permits string constants made up of concatenations of other string constants, including string constant identifiers and the CHR( ) function. These can span multiple lines, but are still limited to the 255 character maximum.

The * is the concatenation operator. String constant expressions are allowed wherever a string literal is allowed, except in metacommands.

> **Note:** The concatenation operator only works with constants, not with string variables.

Example of a string expression:

```
HEADER = 'Now"s the time'
POLITC = HEADER * CHR(13) * 'For all good men'
```

LSTRING

The LSTRING feature adds the super array type LSTRING, or length-string. It is similar to PACKED ARRAY [0..n] OF CHAR. Element 0 contains the length of the string, which can vary from 0 to a maximum of n (n <= 255). LSTRINGs are discussed further in Chapter 5.

Note that a constant of type STRING (n) or CHAR changes automatically to type LSTRING if necessary for assignment, etc.

NULL is a predeclared constant for the null string, with element 0 (the only element) equal to CHAR(0). NULL cannot be concatenated, since it is not a STRING. It is the only constant of type LSTRING; the user cannot declare a constant of an LSTRING type, even as a structured constant.

Constant Definition

A constant definition introduces an identifier as a synonym to a constant. These definitions are in the CONST section of a program, procedure, function, module, interface, or implementation. The general form is:

constant-identifier = constant

The constant-identifier is not defined until after the definition is processed, so a constant definition such as N = (N+1) is invalid. The "constant" can be a constant expression, as defined above. The $INCONST metacommand also defines a constant identifier.

Structured Constants

The structured constant feature permits constant arrays and records of a particular type, and constant sets of a particular type or an unnamed implied type (unnamed type set constants are the same as ISO Pascal set values with all elements constant).

Structured constants can be used anywhere a structured value is allowed, as well as in CONST and VALUE sections.

An array or record constant consists of a type identifier followed by a list of constant values in parentheses separated by commas.

A set constant consists of an optional set type identifier followed by set constant elements in square brackets separated by commas; a set constant element is either an ordinal constant, or two ordinal constants separated by two dots to indicate a range of

constant values. Here are some simple examples:

```
TYPE TVECT = ARRAY [-2..2] OF INTEGER

CONST VECT = TVECT (5, 4, 3, 2, 1);


TYPE TRECR = RECORD
               A, B: BYTE;
               C, D: CHAR;
             END;
CONST RECR = TRECR (#20, 0, 'A',
             CHR (20));

TYPE TSETC = SET OF (RED, BLUE, WHITE,
             GREY, GOLD);
CONST SETC = TSETC [RED, WHITE..GOLD];
```

A constant in an array or record structured constant must have a type assignable to the corresponding component type. The value of a constant element corresponding to a tag field selects a variant, even if the tag field is empty.

The number of constant elements must equal the number of components in the structure, except for super array type structured constants. Nested structured constants are permitted.

An array or record constant nested within another structured constant still must have the preceding type identifier. For this reason and others, a super array constant can have only one dimension.

The size of the representation of a structured constant must be from 1 to 255 bytes.

Example of a structured constant:

```
TYPE R3 = ARRAY [1..3] OF REAL;

TYPE SAMPLE = RECORD
                  I: INTEGER;
                  A: R3;
                  CASE BOOLEAN OF
                  TRUE: (S: SET OF 'A'..'Z';
                         P: ^SAMPLE);
                  FALSE: (X: INTEGER);
              END;

CONST S1 = SAMPLE (27, R3 (1.4, 1.4, 1.4);
TRUE, ['A','E','I'],NIL);
```

Constant elements can be repeated with the "DO n OF constant" phrase, so the above example could have included "DO 3 OF 1.4" instead of "1.4, 1.4, 1.4". Set constant expressions (like ['_'] + LETTERS) are not supported, and neither are file constant expressions.

The STRING (3) constant 'ABC' is equivalent to the structured constant STRING ('A', 'B', 'C'). LSTRING structured constants are not permitted (nor are they very useful).

Structured constants (and other structured values, like variables and values returned from functions) can be passed by reference using CONST parameters (see Chapter 10).

There are two kinds of set constants, one with an explicit type (that is, having a type identifier before the [) as in CONST SETC = TSETC[RED, WHITE.. GOLD] ; and one with an unknown type (that is, having no type identifier before the [) as in CONST RANGE = [20..40]. Either can be used in an expression or to define the value of a constant identifier.

The kind with an explicit type can also be passed as a reference (CONST) parameter.

Passing sets by reference is generally more efficient than passing them as value parameters.

Notes on Constants

The identifiers defined in an enumerated type are constants of that type and cannot be used with numeric (or string) constant expressions directly. They can be used with the ORD, WRD, or CHR functions, however, as in 12 + ORD (BLUE).

TRUE and FALSE are predefined constants of type BOOLEAN; they can be redefined. As mentioned, MAXINT, MAXWORD, and NULL are also predefined. NIL is a constant of any pointer type; since it is a reserved word in ISO Pascal, it cannot be redefined by the programmer.

Numeric statement labels have nothing to do with numeric constants; one cannot use a constant identifier or expression as a label. All constants (including string expressions and structured constants) are limited to an internal representation of 1 to 255 bytes.

CHAPTER 6. DATA TYPES

Contents

Data Types

A data type determines the set of values which variables of that type can assume and associates an identifier with the type. Data type definitions are found in the TYPE section of a program, procedure, function, module, interface, or implementation.

A type definition has the general form:

type-identifier = type

The type-identifier is not defined until after the definition is processed, so a type definition such as T = ARRAY [0..9] OF T is invalid, with an exception for reference types.

A super-type determines the set of types which designators of that super-type can assume and associates an identifier with the super-type. Super-type definitions are also in the TYPE section and have the general form:

super-type-identifier = super-type

Only super array types are available.

Data Types in IBM Pascal

The data types in IBM Pascal can be outlined as follows:

SIMPLE TYPE
- **Ordinal**
  - **Elementary**
    - INTEGER — -MAXINT..MAXINT
    - WORD — 0..MAXWORD
    - CHAR — CHR(0)..CHR(255)
    - BOOLEAN — (FALSE,TRUE)
    - Enumerated — (RED,BLUE,GREEN)
    - Subrange — 100..5000

REAL (max value of 1.7041200E+38)

STRUCTURED TYPE
- **Non-file**
  - **ARRAY OF type**
    - General (OF any type)
      - N-dimensional notation
    - Strings (PACKED ARRAY [...] OF CHAR)
      - STRING (n) ([1..n])
      - LSTRING (n) ([0..n])
    - Super array types
  - **RECORD**
    - Variant records
  - **SET OF type**
- **FILE OF**
  - **General (binary) files**
  - **TEXT (like FILE OF CHAR)**

REFERENCE TYPE
- **Pointer type** — for example, ^type
- **ADR OF type** — relative address
- **ADS OF type** — segmented address

PROCEDURAL TYPE (only as parameter type)

Simple Data Types

The term "ordinal type" is any simple, "finite" type, that is, INTEGER, WORD, CHAR, BOOLEAN, subrange, or enumerated (any simple type except REAL).

Elementary Types

There are four elementary data types: INTEGER, WORD, CHAR, and BOOLEAN.

INTEGER

INTEGER values are a subset of the whole numbers, ranging from -MAXINT through 0 to MAXINT. MAXINT is 32767 (2^{15} - 1). Note that -32768 is *not* a valid INTEGER.

INTEGER type constants change to WORD type if necessary, but *not* INTEGER variables. INTEGER values (variables, constants, or function returns) change to REAL in an expression if necessary.

The ORD function changes any ordinal type value to INTEGER type.

WORD

WORD values are used in two ways: as a subset of the whole numbers from 0 to MAXWORD (65535 or (2^{16}) - 1) and as a group of 16 bits. The motivation for the WORD type rises from:

- The need for values between 32768 and 65535
- The need to interface with operating systems and machine architecture

- The need to operate on addresses

- And the need to easily get at machine primitives (like AND) without using the INTEGER type (and running into the -32768 value).

Assembly language programmers are accustomed to using numbers from -32768 to 65535 and mapping a 16 bit quantity to a number in two ways, signed and unsigned. Having both an INTEGER and a WORD type permits this in a structured way.

WORD and INTEGER values cannot be mixed in an expression (unless one is a constant INTEGER), and are not assignment compatible.

Mixing INTEGER and WORD values results in a warning instead of an error; the compiler will arbitrarily use signed or unsigned arithmetic in the expression.

The WRD function changes any ordinal type value to WORD type.

CHAR

CHAR values are 8 bit ASCII. SET OF CHAR is supported. All 256 byte values are in the type CHAR. Relational comparisons use the ASCII collating sequence.

The "line-marker" character used in TEXT files is not part of the CHAR type in ISO-Pascal, but in IBM Pascal CHR(13), carriage return, is used as the "line-marker". CHR(26) is used to mark the end of a text file.

The CHR function changes any ordinal type value to CHAR type as long as ORD of the value is in the range 0..255.

BOOLEAN

BOOLEAN values are (FALSE, TRUE). The BOOLEAN type is a special case of an enumerated type; ORD (FALSE) is 0, and ORD (TRUE) is 1.

In Pascal, FALSE < TRUE. BOOLEAN, FALSE, and TRUE can be re-defined by the programmer, but the old type is implicitly used by the compiler for things like the IF statement and Boolean expressions.

There is not a function that changes ordinal type values to BOOLEAN type, but the ODD function for INTEGER and WORD values and the expression "ORD (value) < > 0" may be useful in some cases.

Enumerated Types

An enumerated type defines an ordered set of values by enumerating the identifiers for these values, and which are constants of that type. For example:

```
TYPE
COLOR = (RED, BLUE, GREEN, YELLOW,
         PURPLE, ORANGE, GOLD);
SUITS = (CLUB, DIAMOND, HEART, SPADE);
```

Identifiers for all enumerated type constants must be unique within their declaration level.

The ORD value always starts at zero (ORD (RED) = 0, ORD (BLUE) = 1, and so on).

Conceptually, all ordinal types can be considered enumerated types. There is not a function to change other ordinal values to an enumerated type, except for RETYPE.

Enumerated types are particularly useful for representing an abstract collection of names, such as names for operations, kinds of things, commands, and so on. Modifying a program by adding a new value to an enumerated type is much safer than using raw numbers; any arrays indexed with the type or sets based on the type automatically change. Since enumerated types are ordered, comparisons like RED < GREEN can be useful.

Sometimes (in FOR statements, for example) access to the lowest and highest values of the enumerated type can be useful; the LOWER and UPPER functions permit this, as in:

```
VAR TINT: COLOR;
FOR TINT := LOWER (TINT) TO UPPER (TINT)
            DO PAINT (TINT);
```

Subrange Types

A subrange of an ordinal type, called the "host type", can be defined by giving the lower and upper bound (in that order) of the subrange. The lower bound must not be greater than the upper bound but they can be equal.

The subrange type is frequently used as the index type of an array bound, or the type for variables used to index an array, or as a set base type. Examples:

100..200
'A'..'Z'
RED..YELLOW

In set constants and set constructors, a subrange clause substitutes for a list of values, and the extended CASE constant feature allows CASE statement constants and variant labels to also use a subrange clause.

The subrange bounds can be constant expressions; however, the first subrange expression cannot start with a left parenthesis, because the compiler assumes the left parenthesis is starting an enumerated type declaration. For example:

```
TYPE FEE = (A, B, C);
     FIE = M+2*N. .(P-2)*N; {permitted}
     FOO = (M+2)*N. .P-2*N; {invalid}
```

Besides array and set declarations, the subrange type is particularly useful to guarantee the value of a variable (especially if $RANGECK is on).

If the logic of a program implies a variable will always have a value from 100 to 999, declaring the variable with that subrange insures that the compiler will detect any value out of that range.

In addition, the compiler may also be able to allocate less room and use simpler operations with a subrange type. For example, declaring a type 1..100 (an INTEGER subrange) means the type can be allocated in 8 bits instead of 16. This is especially true when an array is PACKED (see "Internal Representation" later in this chapter).

Two subrange types are predeclared:

- BYTE = WRD(0)..255; {8 bit WORD subrange}
- SINT = -127 .. 127; {8 bit INTEGER subrange}

REAL Type

The Real format has a 24 bit mantissa and 8 bit exponent, giving about 7 digits of precision and a maximum value of about 1.70141200E+38 (see "Internal Representation" in this chapter).

The standard REAL runtime unit provides these 21 functions:

SQR
SQRT
SIN
COS
ARCTAN
EXP
LN
TRUNC
ROUND
FLOAT
ENCODE (REAL-to-STRING)
DECODE (STRING-to-REAL)
add
subtract
multiply
divide
negate
absolute
compare-equal
compare-less-than
compare-less-or-equal

The standard library provides additional REAL functions, but these are not predeclared (see "Arithmetic Functions" in Chapter 11).

Structured Types

A structured type is characterized by the type(s) of its components and by its structuring method (array, record, set, or file). Components of structures can be structured to "any" level.

Structured types can also be PACKED, but the PACKED command is used only for type checking and does not cause bit packing. The PACKED prefix can only be used preceding one of the structure names ARRAY, RECORD, SET, or FILE, and cannot be used preceding a type identifier; for example, if COLORMAP is an unpacked array type identifier, PACKED COLORMAP is not accepted.

A component of a PACKED structure cannot be passed as a reference parameter (with an exception for string types). Getting the address of a PACKED component with ADR or ADS is not well defined.

A PACKED prefix only applies to the structure being defined; any components of that structure which are also structures are not classified as packed unless explicitly given the PACKED reserved word in their definition, except for n-dimensional array notation (see "Arrays" in this chapter).

The maximum length of any structured type is 32766 (MAXINT-1) bytes.

Arrays

An array is a structure consisting of a fixed number of components which are all of the "component type". The elements of the array are designated by indices, values of the "index type" (an ordinal type).

Arrays in Pascal are, strictly speaking, one-dimensional, but since a component type can also be an array, n-dimensional arrays are effectively supported by using a shorthand notation. Examples:

```
ARRAY [1..10] OF INTEGER
ARRAY [..9] OF ARRAY [0..99] OF 0..999
ARRAY [0..9, 0..99] OF 0..999 {same as above}
ARRAY [COLOR] OF (PRETTY, TACKY)
```

All of an n-dimensional PACKED array is packed:

```
PACKED ARRAY [1..2, 3..4] OF C;
```

is equivalent to:

```
PACKED ARRAY [1..2]
       OF PACKED ARRAY [3..4] OF C;
```

Super Arrays

A "super-type" can be thought of as a set of types or a parametric type. For example:

```
TYPE
  VECTOR = SUPER ARRAY [1..*] OF REAL;
  MATRIX = SUPER ARRAY [1..*, 1..*] OF REAL;

  VECT10 = VECTOR (10); MATDEC = MATRIX (100,10(

VAR
  ROW: VECT10; COL: VECTOR (10);
  ROWP: ^VECTOR;
```

A super array type identifier itself can only be used after an identifier for a reference parameter, or following the ^ in a type clause, or in a type designator, or in a structured constant.

A super array type designator can be used wherever a type clause can be used, and is considered a unique

type. In this example, VECTOR and MATRIX are *super array type identifiers*; VECTOR (10) and MATRIX (100,100) are called *type designators* (giving the upper bound of the super array type identifiers), and VECT10 and MATDEC are the type identifiers.

ROW[3] or COL[4] would be components in the normal sense.

The general super-type concept would allow other "types of types", such as super subranges and super sets, but the only super types allowed are arrays with parametric upper bounds.

A super array type clause is an array type clause prefixed with the keyword SUPER and with every upper bound replaced with a "*".

Parameters: A formal reference parameter can be given a super array type:

```
PROCEDURE A (VAR S: STRING);
```

An actual parameter can be:

- A variable of the super array type itself (this can occur only in a procedure or function (see the following example) or as a pointer referent)
- A variable or a constant of a type derived from the super array type

This type of parameter usage is called a "conformant array" in the Pascal literature.

For example:

```
TYPE VECTOR = SUPER ARRAY [1..*]
                OF REAL;
VAR X:  VECTOR (12);
    Y:  VECTOR (24);
    Z:  VECTOR (36);
    (*X, Y, and Z are derived from
    VECTOR, the super array *)
    •
    •
    •
FUNCTION SUM (VAR V: VECTOR): REAL;
    (*V, the formal reference parameter,
    is the conformant array *)
VAR S: REAL; I: INTEGER;
  FUNCTION SUB (VAR V: VECTOR): REAL;
  VAR S: REAL; I: INTEGER;
  BEGIN (* SUB *)
    S := 0;
    FOR I := 1 TO UPPER (V) DO
    S := S - V [I];
    SUB := S;
  END; (* SUB *)

BEGIN (* SUM *)
  S := 0;
  FOR I := 1 TO UPPER (V) DO
  S := S + V [I];
  SUM := S + SUB (V);
  (* V is a super array type being
  passed to the nested function SUB *)
END; (* SUM *)

BEGIN (* MAIN *)
    •
    •
    ZERO := SUM (X) + SUM (Y) + SUM (Z);
    •
    •
END. (* MAIN *)
```

Note: The statement SUB (V) in the FUNCTION SUM is allowed because V is assigned an actual value by the time it is passed.

A pointer to a super array type allows an array of a particular size allocated on the heap (the actual upper bound of the array wanted is passed to the NEW procedure). This is especially useful for STRINGs and LSTRINGs.

Super arrays allocated with NEW are not initialized and so give a warning if they contain any files. This kind of heap variable is sometimes called a "dynamic array" in the Pascal literature. For example:

```
VAR PST: ^LSTRING;
  (* PST is a pointer referent type *)
  •
  •
  NEW (PST, 10);
  (* Upper bound of LSTRING will be 10 *)
```

Variables: Variables with super array types can only exist as formal reference parameters or pointer reference types. Do not declare a variable with (or containing) a super array type in the VAR section or as a value parameter. However, variables or constants of a type derived from a super array type are permitted.

For example, the string literal 'abc' is a constant of type STRING (3), a derived super array type.

Also, variables with a super array type (that is, formal reference parameters) are not compatible or assignment compatible with any other variables, even other variables with the same super array type.

Components of a super array type (for example, X[3], Y[2], Z[6]) and type designators (for example, VECTOR (12), VECTOR (24), VECTOR (36) above), follow the normal type rules; components can be assigned, compared, passed as parameters, and so on.

The UPPER function is used to get the actual upper bound of a super array parameter or referent (see Chapter 11, "Extended Intrinsics Feature").

The maximum upper bound of a type derived from a super array type is limited to the maximum value of the index type implied by the lower bound (MAXINT, MAXWORD, etc.), and by the limit of 32766 bytes in any structure. An exception is the super array type LSTRING which is limited to a maximum upper bound of 255. The predeclared super array types are:

```
STRING = SUPER PACKED ARRAY [1..*] OF CHAR;
LSTRING = SUPER PACKED ARRAY [0..*] OF CHAR;
```

STRINGs: A constant like 'Pascal' has the type PACKED ARRAY [1..n] OF CHAR; "n" in this case is 6. The super array feature predeclares the super array type STRING, allowing the use of STRING (n) as identical to PACKED ARRAY [1..n] OF CHAR.

There is no default for n. The range of n is limited to 1..MAXINT-1. The super array type STRING can also be used as a formal reference parameter type or pointer referent type.

The usual super array type restrictions apply; such a parameter or dereferenced pointer cannot be compared or assigned as a whole.

The following may be passed to a formal reference parameter of super array type STRING:

- Any variable or constant of the super array type STRING (only possible within a function or procedure)
- A variable declared as type CHAR
- A variable declared as type STRING(n)
- A variable declared as type PACKED ARRAY [1..n] OF CHAR
- A variable declared as LSTRING(n)
- A variable declared as LSTRING (only possible within a function or procedure)

Reading a STRING(n) or the super array type STRING (when used as a formal reference parameter or pointer referent) inputs characters until the end of a line or the end of the string is reached; if the end of the line is reached first, the rest of the string is filled with blanks. Writing a STRING(n) or STRING writes all of its characters.

The normal Pascal type compatibility rules are relaxed for STRINGs; any two variables (or constants) with type PACKED ARRAY [1..n] OF CHAR or STRING (n) can be compared or assigned if the lengths (n) are equal.

Since the length of a STRING super array type may vary, comparisons and assignments are *not* allowed.

In ISO Pascal, a problem with strings being PACKED ARRAY [1..n] OF CHAR is the PACKED prefix, which normally implies a component cannot be passed to a reference parameter.

In IBM Pascal, this restriction does not apply (officially passing a CHAR component of a STRING to a reference parameter is an "error not caught").

Also, the index type of a string is officially INTEGER, but WORD type values can also be used to index a STRING.

Many string processing applications are expected to use the LSTRING type, below. Some of the string intrinsic procedures and functions in Chapter 11 can be used for STRINGs as well as LSTRINGs.

LSTRINGs: The LSTRING feature allows variable-length strings. LSTRING (n) is a PACKED ARRAY [0..n] OF CHAR, but (unlike STRINGs) a variable with the explicit type PACKED ARRAY [0..n] OF CHAR is not "identical" to the type LSTRING (n) even though they are structurally the same. There is no default for n. The range of n is limited to 0..255.

Characters in an LSTRING can be accessed with the usual array notation. The length is in the first element and can vary from 0 to n.

The length of an LSTRING variable T can be accessed as T[0] if type CHAR, or as T.LEN if type BYTE.

```
VAR Y: INTEGER;
    X: BYTE;
    CH: CHAR

CH := T [0]
 Y := ORD (CH);
 (* Y will now contain the length *)
 X := T.LEN
 (* X will also contain the length *)
```

The word LSTRING was chosen because it contains an L (length) in front of a STRING.

String constants (of type CHAR or STRING (n)) change automatically to type LSTRING. The predeclared constant NULL is the empty string, LSTRING (0).

NULL is the only constant with type LSTRING; there is no way to define other constants with type LSTRING.

As with STRINGs, a component of an LSTRING can be passed as a reference parameter, and WORD as well as INTEGER values can be used to index an LSTRING.

Several operations on LSTRINGs work differently than on STRINGs. Any LSTRING can be assigned to any other LSTRING as long as the current length of the source string is not greater than the maximum length of the destination string; this is checked if $RANGECK is on.

However, neither side can be the super array type LSTRING itself; both must be types derived from it. Any two LSTRINGs can be compared, including a super array of type LSTRING (*the only super array type comparison allowed*).

A READ into an LSTRING inputs characters until the end of the current line or the end of the LSTRING, and sets the length to the number of characters read.

WRITE from an LSTRING writes the current length string. Various predeclared procedures and functions support LSTRINGs; Chapter 11 describes them. LSTRING can also be used as a super array type for formal reference parameters and pointer referents.

A special transformation permits an actual LSTRING parameter to be passed to a formal reference parameter of type STRING. The length of the formal STRING is the actual length of the LSTRING; that is, if LSTR is of type LSTRING (n) or LSTRING, then it can be passed to a parameter VAR STR: STRING and UPPER (STR) is equal to LSTR [0].

This means procedures and functions can operate equally well on STRINGs and LSTRINGs. The only reason to declare a parameter of type LSTRING is when the length must be changed.

Records

A record type is a structure consisting of a fixed number of components, possibly of different types. The record type definition specifies for each component, called a "field", its type and an identifier which denotes it.

The scope of these "field identifiers" is the record definition itself (so they must be unique within the definition); they are also accessible within a field-designator ("."), or a WITH statement.

A record can also have several "variants", in which case a certain field called the "tag field" indicates which variant to use. The tag field may or may not have an identifier and storage in the record.

Some operations (structured constants, the NEW and DISPOSE procedures, and the SIZEOF function) can specify a tag value, even if the tag is not part of the record. Examples:

```
RECORD
  NAME: LSTRING (30);
  PHONE: RECORD
           AREA, PREFIX, EXTEN: INTEGER
         END;
  RIPE: BOOLEAN
END;
```

```
RECORD
  X, Y: REAL;
  CASE S: SHAPE OF
    SQUARE: (SIZE, ANGLE: REAL);
    CIRCLE: (DIAMETER: REAL)
END;

RECORD
  CASE BOOLEAN OF
    TRUE: (I, J: INTEGER);
    FALSE: (CASE COLOR OF
              BLUE: (X: REAL);
              RED: (Y: LONGINT));
END;
```

Note that only one variant part of a record is allowed, and that it *must* come at the end. However, this variant part can also have a variant, and so on. All field identifiers in a given record type must be unique (even if in different variants).

ISO Pascal requires every possible tag field value to select some variant, so it is invalid to include "CASE INTEGER OF" and not include a variant for every possible INTEGER value; however, this error is not caught in IBM Pascal.

Full CASE constant options are supported in the variant clause; that is, a list of constants (and with the extended CASE constant feature and subranges) can define a case. An empty variant, like POINT:( ) is permitted and frequently useful.

An entirely empty record type (RECORD ; END), though not useful, can be declared; however, a warning is given whenever it is used (and in fact the compiler does allocate a byte for it).

The ISO standard defines a number of errors having to do with variant records which IBM Pascal does not catch.

However, in the last example, uses of I cannot be checked because IBM Pascal does not allocate the BOOLEAN tagfield. The ISO standard declares that when a "change of variant" occurs (such as when a new tag value is assigned) all the variant fields become undefined.

IBM Pascal does not set the fields uninitialized when a new tag is assigned and so does not catch the use of a variant field with an undefined value.

A record variable allocated on the heap with the long form of NEW has various special restrictions which our compiler does not enforce (the long form of NEW only allocates a large enough variable for a particular variant).

However, the compiler does ensure that an assignment to such a short record (that is, a record allocated using the long form of NEW) only modifies the short record itself.

A record allocated with the long form of NEW can be released using the short form of DISPOSE with no ill effects (except that the ISO standard defines this as an error and it is not caught).

A DISPOSE of a record passed as a reference parameter or used by an active WITH statement is also an error in the ISO standard not caught by IBM Pascal.

Variant records do interact in two ways with IBM Pascal features:

- Declaring a variant containing a file is not safe, since any change to the file's data using a field in another variant, even if the file is closed, may lead to I/O errors.

- Giving initial data to several overlapping variants in a variable in a VALUE section could have unpredictable results.

The compiler permits both of these cases, but generates a warning message.

The explicit field offset feature permits assigning an explicit byte offset to fields in a record.

It also permits unsafe operations like overlapping fields, word values at odd byte boundaries, and so on. It is not recommended unless necessary for interfacing considerations.

Examples:

```
TYPE ABC = RECORD
       NDRIVE [00]: BYTE;
       FILENM [01]: STRING (8);
       FILEXT [09]: STRING (3);
       EXTENT [12]: BYTE;
       ABCRES [13]: STRING (20);
       RECNUM [33]: WORD;
       RECOVF [35]: BYTE;
       END;

       FOO = RECORD
       BYTEAR [00]: ARRAY [0..7] OF BYTE;
       WORDAR [00]: ARRAY [0..3] OF WORD;
       BITSAR [00]: PACKED ARRAY [0..63]
                                    OF 0..1;
       END;
```

Note that anything larger than one byte is rounded up to an even length.

For example:

```
EXM = RECORD
        A[1]  : STRING(3)
        B[4]  : WORD;
        END;
```

An assignment of 'abc' to field A will overwrite the first byte of B.

If any field is given an offset, all fields should be given offsets, although the compiler will pick an offset if none is given. If offsets are used variant fields should not be used (and may cause errors), but again the compiler will process such a declaration.

With explicit offsets field overlap can be completely controlled anyway, but variants provide the long forms of NEW, DISPOSE, and SIZEOF (the ALLHQQ function could be used instead to allocate a given number of bytes directly). Structured constants are supported for record types with explicit offsets.

Sets

A set type defines the range of values which is the powerset of its "base type", which must be an ordinal type. The null set, [], is a member of every set. Examples:

```
TYPE
  COLOR = (RED, YELLOW, GREEN, BLUE);
  PALETTE = SET OF COLOR;
VAR
  LETTER : SET OF 'A'..'Z';
  TINT   : PALETTE;
```

Operations on sets are defined in Chapter 8. The ORD value of the base type can range from 0 to 255, allowing SET OF CHAR but not SET OF 1942..1984.

Set operations are implemented either by routines in the set unit or directly by generated code. Sets whose

maximum ORD value is 15 (that is, sets that fit into a word) are usually more efficient than larger ones (that is, are done in-line).

The number of bytes allocated for a set is always even. Set temporaries (intermediate values in set expressions) are put on the stack.

Also, passing a set as a value parameter will invoke a routine to check compatibility at runtime, unless the formal and actual sets have the same type.

The SET OF 0..15 can also be used to test and set the bits in a word. The set operators are IN, +, and –. IN tests a bit, + sets a bit, and – clears a bit.

To use the word both as a set of bits and as, for example, the WORD type requires giving two types to the word, using a variant record, the RETYPE function, or an address type.

Files

A file type is a structure consisting of a sequence of components which are all of the same type. The number of components, called the length of the file, is not fixed by the file type definition.

A file with no components is called "empty". Examples:

```
TYPE F1 = FILE OF COLOR;
TYPE F2 = FILE OF INTEGER;
```

In Pascal, a "file" is conceptually another data type, like an array, but with no bounds and with only one component accessible at a time.

However, in IBM Pascal, a file usually corresponds to a binary or textual operating system diskette file, or a device such as a display screen or printer.

This implies a restriction: a FILE OF FILE is invalid, directly or indirectly (but other structures, like ARRAY OF FILE or FILE OF ARRAY, are permitted).

Files in record variants or super array types are not recommended and give a warning. A file variable cannot be assigned, compared, or passed by value; it can only be declared and passed as a reference parameter.

File Control Block

A file variable is implemented as a file control block (FC record. Fields of this record can be accessed using the usual record notation; for example, given a file FILEVAI the error trapping flag is FILEVAR.TRAP, the error status is FILEVAR.ERRS, and the file's mode as FILEVAR.MODE.

The record type, FCBFQQ, can be used directly; also, any FILE type can also be passed to a formal reference parameter of type FCBFQQ and vice versa, allowing procedures and functions operating on a file of any type (see Chapter 12 and Appendix B).

Every file F has an associated buffer variable, F^. This buffer variable can be referenced (the value fetched and stored) like any other Pascal variable; in some cases, accessing the value of a file buffer variable also causes physical input from the file (see Chapter 12, "File System").

A file buffer variable can be passed as a reference parameter or used as a record in a WITH statement;

however, these uses are not valid if the file position is changed (with GET or PUT) within the procedure, function, or WITH statement using the buffer variable. The compiler issues a warning in these cases.

IBM Pascal supports files both as local variables (allocated on the stack) and as pointer referents (allocated on the heap). The compiler generates code to initialize a file when it is allocated and to close a file when it is deallocated, except for super arrays containing files.

Files in the program header are assigned a DOS filename by the user when the program starts (except for INPUT and OUTPUT). Files can also be given an explicit filename with the ASSIGN or READFN procedures.

INPUT and OUTPUT

The files INPUT and OUTPUT are predeclared as TEXT files, and are initially connected to the user's keyboard and display screen and opened automatically. Except for this, they can be treated like other files. If they are redeclared explicitly in a program, the original versions are still used as the default in I/O procedures.

It is not necessary to declare them as program parameters to use them, but a warning is generated each time an I/O operation is performed.

Files can also be given a mode, to indicate the access method or other characteristics.

The mode is a value of the predeclared enumerated type FILEMODES; the modes are SEQUENTIAL, TERMINAL, and DIRECT.

All files are given SEQUENTIAL mode by default, except for INPUT and OUTPUT which are given TERMINAL mode.

ISO Pascal defines a standard type TEXT, similar (but not identical) to FILE OF CHAR. Files of this type are called "textfiles", each component of which is of type CHAR, but the sequence of characters is substructured into lines with a special "line marker". Various special functions and procedures are provided that use this line-division feature.

In IBM Pascal, textfiles have ASCII structure and all other files have BINARY structure. No additional formatting is imposed on DOS files, thus allowing these files to be generated and used by other system software.

Reference Types

A reference to a variable or constant provides an indirect way to access it. ISO Pascal provides the pointer type, an abstract type used to create, use, and destroy variables allocated from an area called the "heap".

IBM Pascal also provides two machine-oriented address types, one for a single 16 bit address and one for a pair of 16 bit addresses.

Use of pointers is portable, structured, and relatively "safe"; they are intended for things like list processing, trees, graphs, and so on.

Use of the address types is machine specific, low level, and not "safe"; they are intended for hardware and operating system interfacing.

Pointers

A pointer type is an "unbounded" set of values pointing to variables of a given type called the "reference type". These variables are all dynamically allocated from an area called the "heap" with the NEW procedure, as opposed to normal Pascal variables which are allocated on the stack or at fixed locations.

The only operations defined on pointers are assignment, testing for equality and inequality with = and <>, and passing them as value or reference parameters (dereferencing is not considered a Pascal operator).

The pointer value NIL belongs to every pointer type. Pointers are frequently used to create list structures of records.

Examples:

```
TYPE
  TREETIP = ^TREE;
  TREE = RECORD
           VALYU: INTEGER;
           LBRANCH, RBRANCH: ^TREE;
         END;
```

Unlike most type declarations, a pointer type can refer to the type it is a component of or a type declared later in the same TYPE section, as in TREE and TREETIP above. This is called a forward pointer declaration, and permits recursive and mutually recursive structures.

Since pointers are so often used in list structures, forward pointer declarations are often used. The compiler checks for one ambiguous pointer declaration.

Suppose the example above was in a procedure nested in another procedure that also declared a type TREE. Then the reference type of TREETIP could be the outer definition or the one following in the same TYPE section.

In this case the compiler assumes the TREE type intended is the one later in the same TYPE section, and gives a warning ("Pointer Type Assumed Forward")

A pointer can have a super array type as a referent type. The actual upper bounds of the array are passed to the NEW procedure to create a heap variable of the correct size. Forward super array type pointer declarations are **not** allowed (or needed).

ISO Pascal requires strict compatibility between pointers; for example, two pointers declared with different types cannot be assigned or compared even if they happen to point to the same type.

In the example above, a variable of type TREETIP cannot be assigned to the field LBRANCH. Usually programs contain only one type declaration for a pointer to a given type; in the example above, the type of LBRANCH and RBRANCH would be TREETIP instead of ^TREE.

However, sometimes it is useful to make sure two classes of pointer, even if to the same type, are not used together. For example, given a type RESOURCE kept in a list, two types OWNER and USER could both be declared as ^RESOURCE. The compiler will catch any assignment of an OWNER value to a USER variable, and vice versa.

Note that pointers have nothing to do with actual machine addresses.

If $NILCK is on, pointer values can be tested for various invalid values, like NIL, uninitialized, refering to a heap item that has been DISPOSEd, or not valid as a heap reference at all.

Addresses

A system implementation language needs a method of creating, manipulating, and dereferencing actual machine addresses (the pointer type, in theory, is only applicable to variables in the heap). Two kinds of addresses are necessary, called relative and segmented.

A *relative machine address* is a 16 bit quantity, the offset in the default data segment in the 8088 segmented memory environment.

A *segmented machine address* is a 32 bit quantity, consisting of a 16 bit relative offset value and a 16 bit 8088 segment register value.

The keywords ADR and ADS refer to the address-of types; they are both typeclause prefixes and prefix operators. ADR is for the relative address type, and ADS the segmented address type.

As usual, @ (and substitutes ? and ∧) are used as a type clause prefix for pointers and to dereference pointers; they also dereference addresses.

A variable of type ADR OF some type can also be used as a record type with one component, R (relative address) of type WORD.

A variable of type ADS OF some type can be used with two components, R (relative address) and S (segment address), both of type WORD.

A variable of type ADS OF can also be used with one component, usually with either .R or .S notation. Some examples are shown on the next page:

```
VAR
  P:  ADS OF FOO;
(* P is segmented address of type FOO *)

  Q:  ADR OF FOO;
(* Q is relative address of type FOO *)

  X:  FOO;
(* X is some variable of type FOO *)

BEGIN
  P := ADS X;
(* assign the address of X to P *)

  X := P^;
(* assign value whose address
is in P to X *)

  P := ADS P^;
(* assign address of value
whose address is in P to P;
P is unchanged by this *)

  Q := ADR X;
(* assign the relative address of X to Q *)

  P := ADS Q^;
(* assign address of variable at Q to P *)

  Q := ADR P^;
(* ILLEGAL; cannot apply ADR to <ADS>^ *)

  P.R := 16#8000;
(* assign 32768 to P's offset field *)

  P.S. := 16;
(* assign 16 to P's segment field *)

  Q.R := P.R + 4;
(* assign P's offset plus 4
to be value of Q *)

END;
```

In Pascal, the ^ for component selection is done before the ADR or ADS unary operators. The ^ selector can appear after any address variable to produce a new variable, so it can be used in the target of an assignment, a reference parameter, as well as in expressions.

Since ADS and ADR are prefix operators, they are only used in expressions. They can only be applied to a variable or constant, not an arbitrary expression or a procedure identifier.

Two address types are considered the same type if they are both ADR or both ADS types. This permits assigning an ADR OF WORD to an ADR OF STRING(2 for example; in this case it would be easy to wipe out part of memory by assigning a variable of type STRING (200) to the 200 bytes starting at the address of a WORD variable.

If P1 is type ADR OF STRING (200) and P2 is any ADR OF type, the assignment P1^ := P2^ generates fast code with no range checking.

Although not "safe", operating systems and other systems software sometimes need this capability. ADR and ADS are not compatible with each other, but if the .R notation is used this should not be a problem.

The keyword VARS is available as a parameter prefix, like VAR and CONST, to pass the segmented address of a variable.

In the 8088 environment, a "default data segment" is assumed; in this case, a VAR parameter is passed as the default data segment offset of a variable, and a VARS parameter is passed as both the segment value and offset value. The DS and SS registers contain the default segment.

Both VARS parameters and ADS variables have the offset value in the word with the lower address, and the segment value in the address plus two.

Note that the address type and pointer type are entirely distinct. The pointer type, in theory, is just an undefined mapping from a variable to another variable.

The address type is machine-oriented; it is always implemented as a physical machine address. In summary the pointer type is an abstract data type which works the same way in all implementations, but the address type is not at all portable.

Restrictions

Some special facilities using pointer variables are *not* allowed with address variables:

- NEW and DISPOSE are only permitted with pointers.
- NIL does not apply to the address type
- There are no special address values for empty, uninitialized, or invalid address.
- The type "address of super array type" is not supported, because an instance of a super array can only be created dynamically with NEW, and because a pointer to a supertype contains the bounds, which conflicts with the segment value contained in an address.

Getting the address of a super array variable is still permitted with ADR and ADS (so if a parameter is VAR S: STRING the expression ADS S is fine) but unlike a pointer the address does not contain any upper bounds.

There are two predeclared address types:

```
ADRMEM = ADR OF ARRAY [0..32765] OF BYTE;

ADSMEM = ADS OF ARRAY [0..32765] OF BYTE;
```

Because they are address of array of byte types, array notation can be used to get the byte offsets.

For example:

```
VAR BX : ADRMEM; FOO : SOMETYPE;
BEGIN
  BX := ADR FOO;
  BX^[0] := BX^[2];
END;
```

Procedural Types

Procedural types are not like other Pascal types; one cannot declare an identifier for a procedural type in a TYPE section or declare a variable of a procedural type.

However, they are used to declare the type of a procedural parameter, and so are part of the Pascal idea of a type.

A procedural type defines a procedure or function heading, giving any parameters and (for a function) the result type.

The syntax of a procedural type is the same as a procedure or function heading without the identifier (but including any attributes). There are *no* procedural variables in IBM Pascal.

```
PROCEDURE ZEROPOINT (FUNCTION FUN (X:
REAL):REAL);
```

The identifiers (such as X above) used for parameters in a procedural type are ignored; only their type is important.

Type Compatibility

IBM Pascal uses ISO Pascal type compatibility, with some additional rules added for super array types, LSTRINGs, and constant coercions. Type transfer functions (to defeat the typing rules in some cases) are available, like ORD and RETYPE.

Two types can be identical, compatible, or incompatible. An expression may or may not be "assignment compatible" with a variable, value parameter, or array index.

Type Identity and Reference Parameters

Two types are identical if they have the same identifier, or if the identifiers are declared equivalent with a type definition of the form TYPE T1 = T2. "Identical" types are really identical; there is no difference between types T1 and T2.

The type of some constants will change if necessary. A constant of type INTEGER will change to type WORD, a constant of type CHAR to type STRING (1), and a constant of type STRING (n) to type LSTRING (n).

Actual and formal reference parameters must be of identical type, or if a formal reference parameter is of a super array type, the actual parameter must be of the same super array type or a type derived from the super array type.

Two special exceptions apply to reference parameters:

- An actual parameter of type LSTRING or LSTRING (n) can be passed to a formal parameter of super array type STRING.

- An actual parameter of any FILE type or the type FCBFQQ can be passed to a formal parameter of any FILE type or the identical type FCBFQQ.

Two record or array types must be identical for assignment, except as noted for the strings.

STRING (n) is just a shorthand notation for PACKED ARRAY [1..n] OF CHAR; the two types are identical.

However, LSTRING (n) is not a shorthand notation for PACKED ARRAY [0..n] OF CHAR, because variables with the type LSTRING are treated specially

in assignments, comparisons, and READ and WRITE; the two types are not identical or compatible or assignment compatible.

Type Compatibility and Expressions

Two simple or reference types are compatible if:

- They are identical
- One is a subrange of the other
- Both are subranges of compatible types
- Both are ADR types
- Both are ADS types

Two structured types are compatible if:

- Neither is a FILE or contains a FILE, and neither is a super array type, and they are identical
- Both are SET types with compatible base types and both are PACKED or neither is PACKED
- Both are STRING derived types with equal upper bounds
- Both are LSTRING derived types

Two values must be of compatible types when combined with an operator in an expression (most operators have additional limitations on the type of their operands, described later).

A CASE index expression type must be compatible with all CASE constant values (this is a form of the equality operator).

If T and TE are **simple types**, then:

```
VAR X: T;
    XE: TE;

    X := XE;
```

XE is assignment-compatible with X if:

- T and TE are identical types
- T and TE are compatible and XE is in the range of T
- T is real and TE is compatible with INTEGER

If T and TE are **structured types**, then:

```
VAR  X: T;
     XE: TE;

     X := XE
```

XE is assignment-compatible with X if:

- T and TE are compatible
  - for sets, if every member of XE is in the base type of T
  - if T is LSTRING(m) and TE is LSTRING(n) and m >= TE.LEN

Besides the assignment statement itself, assignment compatibility is required for implicit assignments; that is, value parameters, the READ and READLN procedures, the control variable and limits in a FOR statement, super type array bounds, and array indices.

Usually assignment compatibility is known at compile time, and an assignment generates a simple move instruction.

However, some set and LSTRING assignments depend on the value of the expression to be assigned, and compatibility is checked, with a runtime procedure, if $RANGECK is on.

Internal Representation

For simple variables and unpacked structures, data values of simple types have the following internal forms:

INTEGER values are 16 bit two's complement numbers, but a subrange requiring 8 bits or less (that is, in the range -127..127) is allocated an 8 bit byte.

WORD values are 16 bit unsigned numbers, but a WORD subrange in the range 0..255 is allocated an 8 bit byte.

REAL values always take 4 bytes. REAL numbers have an 8 bit excess 128 binary exponent, sign, and 24 bit mantissa; since the high order bit of the mantissa is always 1, it is not stored in the number (the sign bit replaces it). This gives an exponent range of about 10^{38} and about 7 digits of precision. The maximum real number is about 1.70141200E+38.

CHAR values and *BOOLEAN* values take 8 bits. For BOOLEANs, false is zero and true is one. Enumerated values take 8 bits if 256 or fewer values are declared, 16 bits otherwise. Values are assigned starting at zero. Subrange values, as mentioned, take 8 or 16 bits.

POINTER values take 16 bits. A pointer is a DS segment offset value. A pointer to a super array type includes the bounds, increasing the length (see below).

An *address* is either a 16 bit data segment offset, or a 16 bit segment register value and a 16 bit offset value. In memory, the offset value is stored at the lower address, and the segment value at the higher address. Addresses are used for address type values, reference parameters, and temporary WITH record pointers.

Procedural parameters contain a reference to the procedure or function's location and a reference to the "upper frame pointer" (a list of stack frames of statically enclosing routines). The parameter always contains two words, in one of two formats.

Procedural Parameters

In the first format, the first word contains the actual routine's address (a code segment offset), and the second word contains the upper frame pointer or zero if the actual routine is not nested in a procedure or function and therefore has no upper frame pointer.

In the second format, the first word is zero and the second word contains a data segment offset address to two words in the constant area which contain the segmented address of the actual routine (there is never an upper frame pointer in this case).

A 16 bit (or larger) variable or component never crosses a word boundary; that is, it always has an even byte address (with one exception: the user can give explicit field offsets at any boundary).

The 8088 stores a 16 bit value as (low byte, high byte), that is, the most significant byte is at the higher (or

odd) address. This is generally transparent to the programmer, unless a 16 bit item is also accessed as two 8 bit items (for example, using variant records).

The LOBYTE, HIBYTE, and BYWORD functions access the least significant and most significant bytes in a word, respectively; not the even and odd addressed bytes.

An 8 bit variable is also aligned on a word boundary, but an 8 bit component of a structure is aligned on a byte boundary; it can be at an even or odd address. An array of 8 bit values starts on a word boundary if it is within an unpacked structure.

For unpacked arrays and records, the internal form is simply the internal forms of the components, in the same order as in the declaration.

Arrays, records, variants, sets, and files always start on a word boundary in this case. In any case variables cannot be allocated more than MAXWORD (64K) bytes; generally components of a structure cannot be accessed unless the structure has no more than MAXINT-1 (32K) bytes.

A super array type's representation is similar whether it is a reference parameter or the referent of a pointer.

First comes the address (reference parameter) or pointer value, either 2 or 4 bytes long.

Following either of these are the upper bounds, which are signed or unsigned 16 bit quantities. The bounds occur in the same order in which they are declared.

Note that a pointer value to a super array type is longer than other pointers.

The number of bytes allocated for a SET variable or component is 2*((ORD (upperbound) DIV 16) + 1); the maximum required is 32 bytes. For example, SET OF 'A' . . 'Z' requires 12 bytes. Note that the size is always even, and that there are no one-byte sets.

Internally, a set consists of an array of bits, with one bit for every possible ORD value from 0 to the upper bound. Bits in a byte are used starting with the most significant bit.

The occurrence of a given ORD value as an element of a set implies the bit is 1, and the byte and bit position of a given ORD value of any set is the same. For example, the ORD value of 'A' is 65, and the second bit of the eleventh byte in a set is 1 if 'A' is in the set.

The internal form of a file is complex; see Appendix B for more information. A FILE type in a program is a record called a file control block (of type FCBFQQ) in the file unit.

The FCBFQQ record contains the IBM DOS File Control Block (FCB) as well as other data. FCBs for textfiles contain a fixed-length line buffer. FCBs for other (binary) files contain the current buffer variable.

Some variables are initialized automatically, whether they reside in fixed memory, the stack, or the heap. Files (FCBFQQ records) are initialized by calling NEWFQQ, passing the size of a textfile line buffer or binary file component and a Boolean true if the file is a textfile.

Some variables are never initialized: variables found in a VALUE section, variant fields in a record, and super arrays allocated in the heap. Note that the compiler does generate the extra code necessary to initialize stack and heap variables.

CHAPTER 7. VARIABLE DECLARATION AND USE

Contents

Variable Declarations

Variable declarations consist of a list of identifiers denoting the new variables, followed by their type. They are found in the VAR section of a program, procedure, function, module, interface, or implementation, or a formal parameter list for a procedure, function, or procedural type. Those in a VAR section can have any type clause; those in a parameter list can only have a type identifier. Examples:

```
TYPE VECTOR=SUPER ARRAY[1..*] OF REAL;
     S8=STRING(8);
VAR  XT, YT: REAL;
     PAINT: ARRAY [1..10] OF COLOR;
     VECTXX: VECTOR (10);
PROCEDURE NAME_IT(VAR N:S8);
```

Every declaration of a file variable F of type FILE OF T implies the declaration of a "buffer variable" of type T, denoted by F^. A file declaration also implies the declaration of a record variable of type FCBFQQ, whose fields are denoted as F.TRAP, F.ERRS, F.MODE, and so on. Buffer variables and FCBFQQ fields are discussed in "File Buffers and Fields," in this chapter.

Attributes

An attribute gives special information about a variable or routine to the compiler. A variable declaration or procedure/function header can have one or more "attributes".

Attributes can be given in a VAR section or after a procedure or function heading, but not in a TYPE section or a parameter list (except for procedural types).

Procedural as well as variable attributes are discussed in this section because some attributes apply to both; procedural attributes are discussed further in Chapter 10.

Attributes are given in brackets separated by commas. The brackets can occur in one of four places:

- After the variable identifier in a VAR section (to apply to that variable only)
- After the VAR reserved word of a VAR section (to apply to all variables in the section)
- After a procedure or function heading (to apply to the procedure or function, including a procedural type declaration in a parameter list)

STATIC

The STATIC attribute gives a variable a fixed location in memory, as opposed to a normal Pascal variable which is usually allocated on the stack or the heap. However, all variables at the program (or module or unit) level are also assigned a fixed memory location and given the STATIC attribute.

Every STATIC variable has a unique absolute address (or data segment offset address). Their use can save space and time.

Functions and procedures with STATIC variables will execute recursively, but the programmer must only use these variables for data common to all invocations. Files declared in a procedure or function with the STATIC attribute are initialized when the routine is entered and closed when the routine terminates like other files.

Note that some other Pascals call variables allocated on the stack "static".

Examples:

```
VAR BASEVECTOR [STATIC]:
        ARRAY [0..MAXVECT] OF INTEGER
VAR [STATIC] I, J, K: 0..MAXVECT
```

The STATIC attribute does not apply to procedures and functions.

PUBLIC and EXTERN

The PUBLIC and EXTERN attributes indicate a variable accessed by or residing in other loaded modules. PUBLIC and EXTERN variables are implicitly STATIC.

PUBLIC variables are allocated by the compiler. EXTERN variables are not allocated by the compiler.

The keyword EXTERNAL is a synonym for EXTERN (this increases portability since various other Pascals use one or the other). The identifier is passed to the linker in the generated code object file.

Examples:

```
VAR [EXTERN] GLOBIT, GLOBLIT: INTEGER;
VAR BASEPAGE [PUBLIC]: BYTE;
```

The PUBLIC attribute and EXTERN directive indicate a procedure or function to be accessed by, or residing in, other loaded modules. An EXTERN procedure or function must not be followed by a block (also true of the FORWARD directive). As with variables, the keyword EXTERNAL is a synonym for EXTERN.

> **Note:** EXTERN is an attribute when used with a variable, but is a directive when used with a

procedure or function. Attributes are in brackets and are separated by commas; a directive takes the place of a procedure or function block, and so is separated by a semicolon.

EXTERN is also permitted in an implementation of a unit for a constituent procedure or function, although attributes are not permitted. As with variables, the identifier is passed to the linker.

Any procedure or function with the PUBLIC attribute or EXTERN directive must be directly nested within a program or implementation. For example:

```
FUNCTION HPOWER (X, Y: REAL): REAL [PUBLIC];
PROCEDURE ACCESS (KEY: KTYP); EXTERN;
```

Linkage between Pascal routines can be done in a safer way with units, described in Chapter 13.

READONLY

The READONLY attribute prevents assignments to a variable (including READ and FOR-loop control variables), and also prevents passing the variable as a VAR (or VARS) parameter. It can be used with any of the other attributes.

The CONST parameter automatically gets the READONLY attribute; so does the control variable of a FOR loop within the loop. Giving a variable the READONLY attribute along with PUBLIC or EXTERN in one source file does not mean it is READONLY when used in another source file (a way of making assignments). Examples:

```
VAR INCAME [READONLY]: BYTE;
VAR [READONLY] I, J [PUBLIC]
    K [EXTERN]: INTEGER;
```

The READONLY attribute does not apply to procedures and functions.

PURE

The PURE attribute only applies to functions (not procedures or variables). It indicates to the optimizer that no global variables are modified by the procedure or function, either directly or indirectly by calling some other procedure or function which changes a global variable.

For example:

```
A := VEC [I*10+7]
B := FOO;
C := VEC [I*10+9];
```

If FOO is PURE, the optimizer will only generate code to compute I*10 once. But if FOO is not PURE, it might modify I so I*10 is recomputed after the call to FOO.

Functions are not considered PURE unless the attribute is given explicitly. A PURE function cannot assign to, or examine a nonlocal variable, have any VAR (or VARS) parameters (the CONST parameter is allowed), or call a non-PURE procedure or function; these are not checked at compile time.

A PURE procedure or function should not modify the referents of references passed by value (for example, pointer or address type referents) or do input and output, although these are not checked. Since the result of a PURE function with the same parameters must always be the same, the entire function call may be optimized away; for example:

```
HX := A * DSNN (P[I,J]*2);
HY := B * DSNN (P[I,J]*2);
```

If DSNN is PURE then, depending on the optimization it may only be called once.

Example of PURE declaration:

```
FUNCTION AVERAGE (CONST TABLE: RVECTOR):
                  REAL [PURE];
```

Rules for Combining Attributes

When declaring variables, the following rules apply when combining attributes:

- STATIC and READONLY can always be used.
- Either EXTERN or PUBLIC may be used but not both.

In procedure and function declarations, the following rules apply when combining attributes and the EXTERN directive:

- Any function can be given the PURE attribute.
- Procedures and functions must be directly nested within a program, module, or implementation to use any attribute or the EXTERN directive (except PURE).
- Either EXTERN or PUBLIC may be used but not both.

In a unit's interface, EXTERN or FORWARD is given automatically to all constituents, and in a unit's implementation PUBLIC is given automatically to all constituents that are not EXTERN.

Since constituents of a unit are only declared in the interface (not in the implementation), the attributes are only given in the interface, but the EXTERN directive can be used in an implementation if all EXTERN procedures and functions come first.

The VALUE Section

Variables can be given initial values in the VALUE section of a program, module, implementation, procedure, or function. Only statically allocated variables can be initialized in this way; that is, any variable declared at the program, module, or implementation level, or a variable with either of the attributes STATIC or PUBLIC.

Variables with the EXTERN attribute cannot occur in a VALUE section since they are not allocated by the compiler.

The VALUE section contains assignments of constants (including constant expressions and structured constants) to entire variables or components of variables. Only one variant in a record should be given an initial value. Example:

```
TYPE
  VECTOR = SUPER ARRAY [1..*] OF REAL;
  VEC10 = VECTOR (10);
  YEAR = (FROSH, SOPH, JNR, SNR);
  PTR = ^ IDREC;

  IDREC = RECORD
          NAME : LSTRING (10);
          CLASS : YEAR;
          NEXT : PTR;
          END;

VAR
  VIVEC : ARRAY [1..10] OF VEC10;
  IDRECORD : IDREC;
  I : INTEGER;

VALUE
  I := 10;
  IDRECORD := IDREC ('ZEUS', JNR, NIL);
  VIVEC[4] := VECT10 (DO 10 OF 0);
```

Value

A denotation of a variable designates either an entire variable, a component of a variable, or a variable referenced by a pointer. In IBM Pascal, several features introduce the idea of a "value" (for example, an expression factor); a value can be:

- A variable
- A constant
- A function designator
- A component of a value
- The variable referenced by a pointer or address value

The extended function feature allows a function to return an array, record, or set, and components of

these returned structures can be denoted with the same syntax used for variables.

This feature also allows dereferencing a reference type returned by a function. However, the function designator cannot be used as a variable, only as a value; for example, the variable referenced by a pointer or address type returned by a function cannot be assigned to directly (that is, cannot be on the left hand side of an assignment statement).

The structured constant feature allows you to declare a constant of a structured type, and the components of a structured constant identifier can be denoted with the same syntax.

Components of a structured constant clause itself can be selected; however, since the other components are never accessed, this is not very useful.

Examples:

```
TYPE REAL3 = ARRAY [1..3] OF REAL;
CONST PIES = REAL3 (3.14, 6.28, 9.42);
     •
     •
     •
X := PIES [1] * PIES [3];
     (*X := 3.14 * 9.42*)

Y := REAL3 (1.1, 2.2, 3.3) [2];
     (*Y := 2.2*)
```

Entire Variables and Values

An entire variable is denoted by its identifier, and an entire value by a literal constant or constant identifier, a variable identifier, a function designator, or a structured constant.

Component Variables and Values

A component of a variable or value is denoted by following it with a selector specifying the component. The form of a selector depends on the type of structuring (array, record, file, or reference). For example:

```
VICTOR [20,I]
VICTOR [20,I].COMPONE^
VICTOR [20,I].COMPONE^.INDY500^
       [12]['Q', RED].PHONE
```

Indexed Variables and Values: A component of an array is denoted by the array variable or value followed by an index expression. The index expression must be assignment compatible with the index type given in the array type definition. For example:

```
ARRICHR ['C']
BETAMAX [12] [-3]
BETAMAX [12,-3] {same as above}
ARYFUNV (A, B) [3, 7];
          {selection made on a function
          returning an array}
```

With the LSTRING feature, the current length of an LSTRING is denoted by the LSTRING variable followed either by [0] for the length of type CHAR or .LEN for the length of type BYTE.

Field Variables and Values: A component of a record is denoted by the record variable or value followed by the field identifier for the component. In a WITH statement, the record variable or value is only given once. Within the statement, its field identifiers can be used directly. Examples:

```
PERSON.NAME
PEOPLE.DRIVERS.NAME
```

```
WITH PEOPLE.DRIVERS DO...
     NAME... {same as above}
RECFUNX ('Alpha').BETA
     {selection (.BETA) on a function
     returning record 'Alpha' is being
     passed as a parameter}
```

Record field notation also applies to files for FCBFQQ fields, address type values for numeric representations, and LSTRINGs for the current length.

File Buffers and Fields: Only one component of a file, determined by the current file position, is accessible at any time. It is represented by the buffer variable.

Depending on the status of the buffer variable, fetching the value of a file buffer variable may just copy the current value, or it may actually input data from an external device (see Chapter 12).

A file buffer variable passed as a reference parameter or used as a record of a WITH statement gives a warning message; after the position of the file is changed with GET or PUT, the buffer variable's value may not be correct in these cases.

Examples:

```
INPUT^
ACCOUNTPAYABLE.FILE^
```

Components of the FCBFQQ record corresponding to a file are denoted using the usual record notation:

```
FILEP^.BINFILE.TRAP
INPUT.ERRS
```

Referenced Variables

If P is a pointer variable or value pointing to a type T (or, with the address type feature, an ADR OF T or ADS OF T variable or value (see Chapter 6)), P denotes the variable or value and P^ denotes the variable of type T referenced by P. Examples:

```
CURRENTPERSON^
NEXT^[J]^
```

A component of a constant structure which is a reference cannot be dereferenced by appending a ^ since the only constant reference value is NIL, and NIL^ is an error.

The designator for a function returning a pointer or address type can also be appended with a ^ to denote the variable referenced by the value returned. However, this variable cannot be assigned to or passed as a reference parameter. Examples:

```
FUNK (I, J)^
FUNK (K, L)^.FOO [2]
```

If P is of type ADR OF some type then P.R denotes the address value of type WORD. If P is of type ADS OF some type, then P.R denotes the offset portion of the address (and P.S the segment portion of the address), both of type WORD. For example:

```
BUFFADR.R
DATAREA.S
```

CHAPTER 8. EXPRESSIONS

Contents

Simple Expressions

Expressions consist of operators and operands (an operand is a value; that is, a constant, variable, or function designator). The rules of composition specify operator precedence in four classes.

The unary operators have the highest precedence, followed by the "multiplying" operators, the "adding" operators, and finally the relational operators. Parentheses are used to change precedence.

The compiler may rearrange expressions and evaluate common subexpressions only once to generate optimized code.

The semantics of the precedence relationships are retained, but normal associative and distributive laws are used. For example: 3 * (6 + (X - 2)) might become X * 3 + 12.

A Pascal expression is either a value or the result of applying an operator to one or two values. Although a value can have almost any type, the Pascal operators only apply to a few types.

The types used with operators are the numeric types INTEGER, WORD, REAL, the BOOLEAN type, and the SET types.

For all operators (except the set operator IN), operands must be compatible. Additional restrictions are given below.

Operators and Operands

The standard and extended operators are:

	STANDARD	EXTENDED
UNARY	NOT	
MULTIPLYING	*, /, DIV, MOD, AND	
ADDING	+, -, OR	XOR
RELATIONAL	=, <>, <=, >=, <, >, IN	

As a rule, the operands and result of an operator all have the same type. However, the type of an operand will sometimes be changed to the type needed for an operator.

This occurs on two levels: one for constant operands only, and one for all operands. INTEGER to WORD only happens for constant operands. INTEGER to REAL happens for all operands.

In constant expressions, only INTEGER values will change to WORD type if necessary. Mixed INTEGER and WORD constants in expressions should be used carefully.

For example, if CBASE is CONSTANT #C000 and DELTA is CONSTANT -1, WRD (CBASE) + DELTA gives an overflow since DELTA is changed to the WORD value #FFFF. WRD (ORD (CBASE) + DELTA) would work, giving INTEGER -16385 which changes to WORD #BFFF.

Any operand of type INTEGER will be converted to type REAL if needed by an operator or for an assignment.

Any operand of type S, where S is a subrange of T, is treated as if it were type T. Any operand of type

Q, where Q = T, is treated as if it were type T. The result type of an expression involving these types can be determined by the following rules:

1. +, –, *

These operate on INTEGERs and REALs (or a mixture) or on WORDs. If either operand is REAL, the result type is REAL or the result type is the type of the operands (both INTEGER or both WORD). Unary + and – are supported as well as binary. Unary – on a WORD type is 2's complement (NOT is 1's complement); since there are no negative WORD values, this always gives a warning. Note that unary – has precedence with the adding operators, so (X * –1) is illegal, and (–256 AND X) is interpreted as –(256 AND X).

2. /

This is a "true" divide, so the result is always REAL. Operands can be INTEGER or REAL (not WORD).

3. **DIV, MOD**

These are the integer divide quotient and remainder. The left operand (dividend) is divided by the right operand (divisor). Operands must be both INTEGER or both WORD (not REAL). The sign of the remainder (MOD) is always the sign of the dividend (as is true of most signed divide instructions), so:

ABS(I)–ABS(J) < ABS((I DIV J)*J) <= ABS (I)

Also,

I MOD J=I(I DIV J)*J
{even if I or J negative}

IBM Pascal does not use the standard semantics for the MOD operator, with either operand negative. Programs intending to be portable should not use DIV and MOD unless both operands are positive.

4. AND, OR, XOR, NOT

These extended operators provide "bit wise" logical functions. Operands must be both INTEGER or both WORD (not REAL); result has the same type. NOT is 1's complement, so if an INTEGER variable V has the value MAXINT, NOT V gives the invalid INTEGER value -32768.

Boolean Expressions

The Boolean operators in ISO Pascal are NOT, AND, and OR. IBM Pascal also includes XOR. The relationals =, <>, <=, >=, <, and > can also be used with Boolean operands. P <> Q is another way of expressing the exclusive-or function. Since FALSE < TRUE, P <= Q denotes the Boolean operation "P implies Q".

No assumptions should be made as to whether an operand to AND or OR is evaluated. Consider the following:

```
WHILE (I <= MAX) AND
      (V [I] < > T) DO
      I := I+1;
```

In the example above, if array V has an upperbound MAX, then the evaluation of V [I] for I > MAX is a runtime error. This evaluation may or may not take place; at times in the optimization process both operands are always evaluated. Sometimes, the evaluation of one may cause the evaluation of the other to be skipped (in the latter case either operand may be evaluated first).

Instead, use a construct like:

```
WHILE I <= MAX DO IF V[I] < > T THEN
          THEN I := I+1
          ELSE BREAK;
```

In the Boolean expressions after IF, WHILE, and UNTIL, the sequential control feature can be used to handle situations like this (see Chapter 10); for example, the above becomes:

```
WHILE I <= MAX AND THEN V[I] < > T DO
      I := I+1;
```

Also note that the Boolean AND and OR operators have nothing to do with the WORD and INTEGER operators of the same name, which are bitwise logical functions.

The relational operators produce a Boolean result. The types of the operands of a relational (except for IN) must be compatible, or one is REAL and the other compatible with INTEGER.

Reference types can only be compared with = and <>. To use one of the other relationals with an address type the .R or .S notation must be included.

Files, arrays, and records cannot be compared as a whole, except for the string types STRING and LSTRING. Two STRING types must have the same upper bound to be compared; two LSTRINGs can have different upper bounds.

In LSTRING comparison, characters past the current length are ignored and if the current length of one LSTRING is less than the other, the compiler assumes the shorter is extended with CHR (0) components.

The six relationals =, <>, <=, >=,<, and > have their normal meaning when applied to numeric, enumerated, or string operands.

Their meaning (along with the relational operator IN) when applied to sets is discussed in this chapter, under "Set Expressions."

Two notes on Pascal Boolean expressions:

- The relationals have a lower precedence than AND and OR. This means the following is incorrect:

  IF I < 10 AND J = K THEN ...

  Instead, one must write:

  IF (I < 10) AND (J = K) THEN ...

- The numeric types cannot be used where a Boolean is called for, as in some other languages. For an integer I, the clause IF I THEN is invalid; one must use IF I <> 0 THEN instead. The metalanguage allows $IF I $THEN, however.

Set Expressions

Pascal uses several operators in a different way when they are applied to sets:

+	is set union
-	is set difference
*	is set intersection
= and <>	test set equality
<= and >=	test set inclusion
< and >	test proper set inclusion
IN	tests set membership

The < and > are extended operators since ISO Pascal does not support them for sets.

Any operand whose type is SET OF S, where S is a subrange of T, is treated as if it were SET OF T (T is restricted to 0..255 or the equivalent ORD values).

When the IN relational is used, the left operand (an ordinal) must be compatible with the base type of the right operand (a set).

The expression X IN B is true if X is a member of the set B, and false otherwise. X can be outside of the range of the base type of B.

For example, if X = 1 and B = SET OF 2..9, then X IN B is always false (1 is compatible, but *not* assignment compatible, with 2..9).

Expressions involving sets can use the "set constructor," which gives the elements in a set enclosed in [ ]s.

Each element can be an expression whose type is in the base type of the set or the lower and upper bounds of a range of elements in the base type. Elements cannot be sets themselves.

Examples of set expressions:

```
SETCOLOR := [RED, BLUE..PURPLE] - [YELLOW]
SETNUMBER := [12, J+K, TRUNC (EXP (X))..TRUNC
(EXP (X+1))]
```

Note that the syntax inside the set constructor is similar to the CASE constant syntax. If X > Y then [X..Y] denotes the empty set. [ ] also denotes the empty set and is compatible with all sets.

Also, if all elements are constant a set constructor is the same as a set constant; like other structured constants, the type identifier for the set can be included, as in:

```
CONST COLOR=COLORSET[RED..BLUE];
```

However, a set constructor with variable elements is not allowed to be given a type in an expression. For example:

NUMBERSET[I. .J] is invalid.

Other Expression Features

EVAL Procedure

The EVAL procedure only evaluates its parameters without actually calling them. EVAL is only useful for functions with side effects.

For example, a function that advances to the next item and also returns the item might be called in EVAL just to advance to the next item.

The same result could be achieved by calling the function, ignoring the value it returned, and just using the known side effect.

Examples:

```
EVAL (NEXTLABEL (TRUE))
EVAL (SIDEFUNC (X, Y), INDEX (4), COUNT)
```

RESULT Function

The RESULT function permits you to access the current value of a function. Using RESULT is more efficient than using a separate local variable for the function's value and assigning this local variable to the function identifier before returning. An example:

```
FUNCTION FACTORIAL (I: INTEGER): INTEGER;
BEGIN
  FACTORIAL := 1;
  WHILE I > 1 DO
  BEGIN
```

```
    FACTORIAL := I * RESULT (FACTORIAL);
    I := I - 1;
  END;
END;

FUNCTION ABSVAL (I: INTEGER): INTEGER;
BEGIN
  ABSVAL := I;
  IF I<0 THEN ABSVAL := -RESULT (ABSVAL);
END;
```


RETYPE

Occasionally, the type of a value needs to be changed. IBM Pascal includes the RETYPE "function" to change the type of a value. If the new type is a structure, RETYPE can be followed by the usual selection syntax.

RETYPE must be used with caution. It works on the memory byte level, and ignores issues like whether the low order byte of a two-byte number comes first or second in memory. Examples:

```
TYPE COLOR=(RED,BLUE,GREEN,ORANGE);
VAR X,Y: INTEGER; C:  COLOR;
X := ORD(ORANGE); (*X will be 3*)
Y := 3;
C := RETYPE(COLOR,Y);
(*C will be ORANGE-acts as inverse of ORD*)

RETYPE(STRING2,I*J+K) [2]
(*effect may vary*)
```

Function Designators

A function designator specifies the activation of a function. It consists of the function identifier, followed by a (possibly empty) list of "actual parameters" in parentheses.

These actual parameters are substituted in place of their corresponding "formal parameters" defined in the function declaration. The correspondence is by position in both lists.

Parameters can be variables, expressions, procedures, and/or functions. The parameter lists must be compatible (as defined under "Data Parameters", in Chapter 10). If the parameter list is empty, the parentheses must be omitted.

The order of evaluation and binding of the actual parameters will vary, depending on optimizations used. For example, when evaluating FUNC (I+J+K, I+J) the compiler will add I and J, save the result, add K, PUSH this as the first parameter, then PUSH the saved value as the second parameter.

Uses

Functions in most languages have two different uses:

1. They are used in the mathematical sense, using one or more values (called the domain) to produce a resulting value (called the range). In this case if the function never does anything else (like assign to a global variable or do input/output) it is called a *pure* function.

2. A function is a convenient way to call a general procedure and also get some value back. In this case the function may do anything and is called *impure*. The optimizer must assume a function is impure, unless it is declared with the PURE attribute.

In some cases, especially those where a function is doing mathematical calculations, using the PURE attribute where appropriate can save time and space.

Parameters

There are three kinds of parameters: value parameters, reference parameters, and procedural parameters. These are discussed under "Data Parameters" in Chapter 10, with procedure and function declarations.

- Value parameters are evaluated and their value copied (assigned) to the corresponding formal parameters.

- Reference parameters are passed by address and the corresponding formal parameters represent them.

- Procedural parameters pass a procedure or a function as a parameter.

A function can return a simple type or a pointer. A pointer returned by a function can only be compared, assigned, or passed as a value parameter.

Functions can also return any assignable type; that is, any type except a file or super array. The usual selection syntax for reference types, arrays, and records is allowed following the function designator. Examples of function designators:

```
SIN (X+Y)
NEXTCHAR
NEXTREC (17)^
NAD.NAME [1]
```

Note that returning a structure and then only using one component of it is less efficient than just returning the component itself.

The compiler treats a function returning a structure sort of like a procedure with an extra VAR parameter for the result of the function.

Every function designator for a function returning a structure may allocate (on the stack) an unseen "variable" to receive the returned value. This adds to the required memory, so the number of places at which a function returning a structure is called should be minimized.

Frequently, the current value of a function is needed within the function (or a procedure or function nested within the function). However, one cannot just use the name of the function as a variable in an expression, because that would invoke the function recursively.

With the extended intrinsics feature, the RESULT "function" can be used to get the current value of a function.

CHAPTER 9. STATEMENTS

Contents

Statement Labels

Statements denote algorithmic actions, and are said to be "executable". They can be prefixed by a label which can be referenced by a GOTO statement.

A *label* can be one or more digits (leading zeros are ignored), or an identifier using the usual rules for identifiers. Constant identifiers, expressions, and nondecimal notation cannot be used for labels. All labels must be declared in a LABEL section.

Example:

```
LABEL 10,20,INNER,OUTER;
```

A *loop label* is defined in this manual as any label immediately preceding a looping statement (WHILE, REPEAT, or FOR statement). Also, a BREAK or CYCLE statement can refer to a loop label.

CASE constants will be described later in this chapter, under "CASE STATEMENT." For now, note that a statement can be preceded by both a CASE constant list and a GOTO label. In this event, they are given in the following order: first the CASE constants, then the GOTO label. Example:

```
321:      123:  <statement>
{321 is a CASE value, 123 is a label}
```

Simple Statements

A simple statement is a statement no part of which constitutes another statement. The Pascal simple statements are the assignment statement, the procedure statement, the GOTO statement, and the empty statement. The control flow feature adds the RETURN statement, BREAK statement, and CYCLE statement.

Assignment Statement

The assignment statement serves to replace the current value of a variable (or function value) with a new value specified as an expression. The expression's value must be assignment compatible with the type of the variable or function.

If the selection of the variable involves the indexing of an array or the dereferencing of a pointer or address, these actions may be intermixed with the evaluation of the expression, depending on the optimizations performed.

An assignment to a nonlocal variable (including a function return) gets a = or % in the G column of the listing (see Chapter 4, "Metacommands").

Within the block of a function, an assignment to the function's identifier serves to set the value returned by the function. The assignment to a function F's identifier may occur either within the actual body of F, or in the body of a procedure or function nested within F.

This value is not available to the programmer after assignment, however, since using the function

identifier in an expression invokes the function recursively (except as a procedural type parameter). The RESULT function can be used to access the current value of a function. It can also be used within the function's body, or in a procedure or function nested within the function.

Examples of assignment statements:

```
A := B;
SUM := X * (Y + Z);
FUNX := 2 * RESULT (FUNX);
```

The optimizer considers each section of code not containing a label or other point which could receive control as eligible for rearrangement and common subexpression elimination. It retains the order of execution when necessary. For example:

```
X := A + C + B;
Y := A + B;
Z := A;
```

Here A + C + B can become C + (A + B), and the generated code will get the value of A and save it, add the value of B and save the result, add the value of C and assign to X, assign the saved A+B to Y, and assign the saved A value to Z.

This optimization can only be done if assignments to X or Y (or getting the value of A, B, or C), are all independent. If C is a function without the PURE attribute and A is a global variable, evaluating C might change A.

In this case, the value of A in the first assignment may be the old value or the new one, since the order of evaluation within an expression is not fixed. However, the value of A in the second and third assignments will

be the new value, since the order of evaluation among statements is fixed.

The following will limit the ability of the optimizer to find common sub-expressions:

- An assignment to a non-local variable, reference parameter, or referent to an address variable
- Call to a function without the PURE attribute
- Call to a procedure

The optimizer does allow a single variable to have two identifiers, perhaps one as a global variable and one as a reference parameter or address value referent.

Procedure Statement

A procedure statement serves to execute the procedure denoted by the procedure identifier. The form is the same as that of a function designator (read Chapter 10 and substitute "function" for "procedure"). Examples:

```
DOIT;
READ (INFILE, I, J, K);
```

GOTO Statement

A GOTO statement serves to indicate that further processing continues at another part of the program text, namely at the place of the label. Examples:

```
GOTO 10;
GOTO HAWAII;
```

A GOTO cannot be used to jump to a "lower level". Basically, this means it cannot jump into an IF, CASE, WHILE, REPEAT, FOR, or WITH statement, or to any statement in another procedure or function at the same level or lower.

It can jump out of these statements, or to a statement in a higher level procedure or function as long as the statement is directly within the body.

GOTOs from one branch of an IF or CASE statement to another are permitted. The following are some examples of legal and illegal GOTOs:

```
PROGRAM LABELEX;
LABEL 1, 2, 3, 4;

  PROCEDURE ONE;
  LABEL 11, 12, 13;
    PROCEDURE INONE;
    LABEL 21;
    BEGIN (*INONE*)
      IF TUESDAY THEN GOTO 1 ELSE GOTO 11;
      {valid}
      21: { no valid GOTO comes here }
    END;(*INONE*)
  BEGIN (*ONE*)
    IF RAINING THEN GOTO 1 ELSE GOTO 11;
    {valid}
    11: GOTO 21;
    {invalid}
  END;(*ONE*)

  PROCEDURE TWO;
  BEGIN
    GOTO 11; {invalid}
  END;

BEGIN {main}
  IF SEATTLE THEN
    BEGIN
      GOTO 2; (*valid*)
      4: WRITE ('here');
    END
  ELSE GOTO 4; {valid}
  2: GOTO 3; {invalid}
  REPEAT
    WHILE MS_BYRON DO 3: GOTO 2; {valid}
  UNTIL DADATE;
  1: GOTO 11; {invalid}
END.{main}
```

A GOTO from one procedure, function, program, or implementation to another (within the same source), or from a nested procedure or function to its host or to the main program, is permitted. However, they do generate extra code both at the location of the GOTO and at the location of the label. The rule of thumb is that the destination label must be "visible" from the procedure and function; that is, it must be at a higher level.

If the $GOTO metavariable is on, every GOTO statement is flagged with a warning reminding users that GOTOs are "considered harmful". This is most useful in an educational environment, or for finding all GOTOs in a program in order to locate a bug.

GOTOs are marked in the listing in the J (jumps) column with a + or * for a GOTO to a label later in the listing and a - or * for a GOTO to a label already encountered in the listing.

Empty Statement

The empty statement consists of no symbols and denotes no action. It is included primarily to permit the ; to be used after the last statement in a BEGIN. . .END pair.

BREAK, CYCLE, and RETURN Statements

These three statements all belong to the control flow feature. All are functionally equivalent to a GOTO statement.

A BREAK statement is a GOTO to the first statement following a repetitive statement.

A CYCLE statement is a GOTO to an implied empty statement ending the body of a repetitive statement. It starts the next iteration of a loop. CYCLE in

either a WHILE or REPEAT statement does the Boolean test in the WHILE or UNTIL clause before executing the statement again; CYCLE in a FOR statement goes to the next value of the control variable.

A RETURN statement is a GOTO to an implied empty statement following the last statement in the current procedure or function, or the body of a program or implementation. In the listing, the J (jump) column contains a + or * for a BREAK statement, a - or * for a CYCLE statement, and a * for a RETURN statement.

BREAK and CYCLE have two forms, one in which a loop label is given, and one without a loop label. If a loop label is given, the label identifies the loop to exit or restart. If no label is given, the innermost loop is assumed. Example:

```
OUTER: FOR I := 1 TO N1 DO
  INNER: FOR J := 1 TO N2 DO
    IF A [I, J] = TARGET THEN BREAK OUTER;
```

In theory, a program using IBM Pascal BREAK and CYCLE statements does not need to use any GOTO statements.

A loop label is a normal GOTO label prefixed to a FOR, WHILE, or REPEAT statement. Since the control flow feature allows integer or identifier labels, it is suggested that programmers use integers for labels referenced by GOTOs and identifiers for loop labels. Examples:

```
SEARCH: WHILE I <= ITOP DO
  IF PILE [I] = TARGET THEN BREAK
    SEARCH ELSE I := I + 1

FOR I := 1 TO N DO
IF NEXT [I] = NIL THEN BREAK

CLIMB: WHILE NOT ITEM^.LEAF DO
  BEGIN
    IF ITEM^.LEFT < > NIL
      THEN [ITEM := ITEM^.LEFT;
        CYCLE CLIMB];
    IF ITEM?.RIGHT < > NIL
      THEN [ITEM := ITEM^.RIGHT;
        CYCLE CLIMB];
    WRITELN ('Very strange node');
    BREAK CLIMB;
END
```

Structured Statements

Structured statements are constructs composed of other statements. They are divided into four groups:

- Sequential (compound)
- Conditional
- Repetitive
- The WITH statement

In the compiled listing, the C (control) column gives the control level, which increases the statements within a compound statement. This feature can be handy when fixing a program with a missing or extra END keyword.

Compound Statement

The compound statement specifies that its component statements are to be executed in the same sequence in which they are written. The symbols BEGIN and END act as statement brackets. The semicolon is a statement separator (*not* a statement terminator, as in some languages). Examples:

```
BEGIN
  TEMP := A [I];
  A [I] := A [J]
  A [J] := TEMP
END;
```

```
BEGIN FOO; ZOO; (*empty statement*) END;
```

The compiler permits the use of [and] instead of BEGIN and END. Note, however, that a] alone is *not* a substitute for END and cannot be used to match

a BEGIN, CASE, or RECORD keyword. Also, BEGIN. . .END (not [. . .]) must be used to enclose the body of a program, implementation, procedure, or function. Examples:

```
IF FLAG THEN X:=1 ELSE [X:=-1,Y:=0];
WHILE P.N < > NIL DO
  [Q:=P; P:=P.N; DISPOSE(Q)];
FUNCTION R2 (R: REAL): REAL;
[R2 := R * 2] {illegal}
```

Conditional Statements

A conditional statement selects for execution a single one of its component statements. The conditional statements are the IF statement and the CASE statement.

IF Statement

If the Boolean expression following the IF is true, the statement following the THEN is executed. If the Boolean expression is false, the statement following the ELSE, if present, is executed. The Boolean expression can use the sequential control operators. Examples:

```
IF I>0 THEN I:=I-1 ELSE I:=I+1;

IF (I<=TOP) AND (ARRI[I]<>TARGET) THEN
I:=I+1;

IF I<=TOP THEN IF ARRI[I]<>TARGET THEN
I:=I+1;

IF C1 THEN IF C2 THEN S1 ELSE S2;
{an ELSE is paired with the closest previous
IF; here S2 is executed if C1 is true and C2
is false}
```

IF C1 THEN [IF C2 THEN S1] ELSE S2 {now S2 is executed if C1 is false} .

A ";" preceding an ELSE is always incorrect. If found, it is skipped and a warning is given.

CASE Statement

The CASE statement consists of an expression (the CASE index) and a list of statements, each statement preceded by a constant list, called the CASE constants.

The statement executed has a CASE constant containing the current value of the CASE index. The CASE index and all constants must be of compatible, ordinal types.

The CASE constant syntax is also used in RECORD variant declarations. In Standard Pascal, a CASE constant is one or more constants separated by commas.

IBM Pascal allows a range of constants to be substituted for a constant, such as 'A'..'Z'. No constant value can apply to more than one statement.

The CASE statement can also be ended with an OTHERWISE clause containing additional statements executed when the CASE index value is not in the set of CASE constant values.

If no OTHERWISE clause is present and the CASE index value is not in the set, a runtime error is generated if $RANGECK is on and the result is undefined if $RANGECK is off. Examples:

```
CASE OPERATOR OF
  PLUS: X := X + Y;
  MINUS: X := X - Y;
  TIMES: X := X * Y
END;

CASE NEXTCH OF
  'A'..'Z','_': IDENTIFIER;
  '+','-','*','/': OPERATOR;
  OTHERWISE
    WRITE ('Unknown Character');
    BREAK SCANNER;
END;
```

A ";" can appear after the final statement in the list. A ":" after an OTHERWISE is skipped and a warning given.

The code generated by a CASE statement may be a jump table or series of comparisons, depending on optimization. If the control variable is out of range and $RANGECK is off, then a jump to an arbitrary location in memory would occur in the jump table case. In the series of comparisons case one of the CASE statement branches will be executed.

Repetitive Statements

Repetitive statements specify that certain statements are to be executed repeatedly.

These include the WHILE, REPEAT, and FOR statements. BREAK and CYCLE can be used to leave or restart the statements being repeated.

WHILE Statement

The WHILE statement repeats a statement zero or more times until a Boolean expression becomes false. The Boolean expression can use the sequential control operators. Examples:

```
WHILE P <> NIL DO P := NEXT (P);

WHILE NOT MICKY DO
  [NEXTMOUSE; MICE:=MICE+1];
```

REPEAT Statement

The REPEAT statement repeats a sequence of statements one or more times until a Boolean expression becomes true. The Boolean expression can use the sequential control operators. Examples:

```
REPEAT
  READ (LINEBUFF);
  COUNT:=COUNT+1
UNTIL EOF;

REPEAT GAME UNTIL TIRED;
```

FOR Statement

The FOR statement indicates that a statement is to be executed repeatedly while a progression of values is assigned to a variable called the "control-variable" of the FOR statement. The general form is:

```
FOR control-variable := initial TO
      final DO statement
```

or

```
FOR control-variable := initial DOWNTO
      final DO statement
```

The ISO standard gives very explicit rules about FOR statements. The control-variable must be of an ordinal type and must be an entire variable (not a component of a structure). Also, in ISO Pascal, the control-variable must be local to the immediately enclosing program, procedure, or function and cannot be a reference parameter of the procedure or function.

In IBM Pascal, the control-variable can also be any STATIC variable, such as a variable declared at the program (module, implementation) level. Using a program-level variable is an ISO Pascal error not caught.

In addition, no assignments to the control variable are allowed, either in the repeated statement (this error is caught), by making the control variable READONLY within the FOR statement) or a procedure or function invoked by the repeated statement (this is an error not caught). The control variable cannot be passed as a VAR (or VARS) parameter to a procedure or function.

The initial and final values must be compatible with the type of the control variable. The initial value is always evaluated first, followed by the final value.

They are evaluated only once, before executing the statement.

The statement following the DO is not executed at all if initial > final (TO case) or initial < final (DOWNTO case). If the statement is executed, both the initial and final values must also be assignment compatible with the control variable. For example, if I is type 1..5:

```
FOR I := 6 TO 5      is permitted.
FOR I := 5 TO 6      is invalid.
```

The sequence of values given the control-variable starts with the initial value, and the sequence is defined with the SUCC function (TO case: increment the ORD value) or the PRED function (DOWNTO case: decrement the ORD value), until the last execution of the statement, when the control-variable has the final value.

The value of the control-variable after a FOR statement terminates "naturally" (whether or not the body executes) is undefined, and may vary due to optimization.

However, the value of the control-variable after leaving a FOR statement with a GOTO or a BREAK is defined as the value it had at the time of exit. Examples:

```
FOR I := 1 TO 10 DO
SUM := SUM + VECTOR [I]

FOR CH := 'Z' DOWNTO 'A' DO
WRITE (CH)
```

To get more efficient code from FOR statements, keep in mind that the body of a FOR statement may

or may not be executed, so generally if the initial or final value is not constant an extra test must be generated.

However, if the control variable has the WORD type (or a subrange), and the initial value is constant zero, the body must be executed no matter what the final value and the test is not generated. (Zero is the lowest possible WORD value.)

WITH Statement

The WITH statement opens the scope of a statement to include the fields of a record (or records), so the fields can be referred to directly. For example:

```
WITH PERSON DO WRITE (NAME, ADDRESS, PHONE);
```

is the same as

```
WRITE (PERSON.NAME, PERSON.ADDRESS,
       PERSON.PHONE);
```

The record given can be a variable, constant identifier, structured constant, or function result.

It cannot be a component of a PACKED structure. If it is a file buffer variable, a warning is generated. A list of records can be given after the WITH, but these should not be of the same type, since the field identifiers would only refer to the last instance of the record with the type. The statement:

WITH record1, record2 DO statement

is the same as

WITH record1 DO WITH record2 DO statement

If any record variable is a component of another variable, the component is selected before the statement is executed. Active WITH variables cannot be passed as VAR (or VARS) parameters, or their pointers passed to the DISPOSE procedure; however, these errors are not caught by the compiler.

Assignments to any of the record variables in the WITH list or components of these variables are allowed, as long as the WITH record is a variable.

Every WITH statement allocates an address variable on the stack holding the address of the record.

If the record variable is on the heap, the pointer to it should not be DISPOSEd within the WITH statement, except perhaps at the end.

If the record variable is a file buffer, no I/O should be done to the file within the WITH statement, except perhaps at the end. Assignments to the WITH record itself should be avoided in programs intended to be portable.

Sequential Control Operators

It is often useful in IF, WHILE, and REPEAT statements to treat the Boolean expression as a series of tests, such that if one test fails, the remaining tests are not executed.

The sequential control feature provides two "operators," AND THEN and OR ELSE. Each operator is two keywords:

- X AND THEN Y is false if X is false and Y is not evaluated
- X OR ELSE Y is true if X is true, and Y is not evaluated.

If several sequential control operators are used, they are evaluated strictly from left to right.

These "operators" can only be used in the Boolean expression of an IF, WHILE, or UNTIL clause, but not in other expressions. They cannot occur in parentheses, and are evaluated after all other operators. Examples:

```
IF SYMBOL<>NIL AND THEN SYMBOL^.VAL<0
  THEN NEXT_SYMBOL;

WHILE I<=MAX AND THEN VECT[I] <> KEY DO
  I:=I+1;

REPEAT GEN(VAL) UNTIL VAL=0
  OR ELSE (QU DIV VAL)=0;

WHILE W AND THEN X OR ELSE Y AND THEN Z DO
  SOMETHING;
```

CHAPTER 10. PROCEDURES AND FUNCTIONS

Contents

Procedure and Function Declarations

Procedure declarations and function declarations associate an identifier with a portion of a program so that the portion can be activated with a procedure statement or function designator, respectively. The general form of a procedure or function declaration is the same as a program, except for the heading. The form includes the heading, declarations, and body. For example:

```
PROCEDURE MODEL (I:INTEGER; R:REAL);
LABEL 123;
CONST ATOP = 199;
TYPE INDEX = 0..ATOP;
VAR ARAY:ARRAY[INDEX] OF REAL; J:INDEX;

  FUNCTION FONE(RX: REAL): REAL;
  BEGIN FONE:=RX*I END;

  PROCEDURE FOUT(RY:REAL);
  BEGIN WRITE ('Output is ', RY) END;

BEGIN (* MODEL *)
  FOR J:=0 TO ATOP DO
    IF GLOBALVAR THEN
    FOUT(FONE (R+ARAY[J]))
    ELSE GOTO 123;
  123: WRITELN ('Done');
END; (* MODEL *)
```

The procedure declared above illustrates the general Pascal form:

1. A heading

2. Declarations for labels, constants, types, and variables

3. Local procedures and functions

4. The body, which is a BEGIN. . .END statement

The procedure returns when the body finishes execution (or a RETURN statement is executed).

In Standard Pascal the order of declarations must be LABEL, CONST, TYPE, VAR, followed by the procedures and functions.

In IBM Pascal, any number of LABEL, CONST, TYPE, VAR, and VALUE sections, and procedure and function declarations, can be included in any order.

In general the initial value of variables is not defined. However program, module, implementation, STATIC, and PUBLIC variables can be explicitly initialized in a VALUE section.

File variables are always initialized by calling NEWFQQ in the file unit (see Appendix B in this book). The use of a procedure identifier in a procedure statement or function identifier in a function designator within its declaration implies recursive execution of the procedure or function, except for passing it as a procedural type parameter.

Procedure and Function Headings

The procedure or function heading specifies the identifier naming the procedure or function and the formal parameters (if any).

To allow the call of a procedure or function before it is defined, permitting indirect recursion where A calls B and B calls A, a "forward declaration" is given before any calls to it.

This forward declaration consists of the heading, followed by the directive FORWARD. Later on, the actual declaration is given but without repeating the formal parameter list (or attributes or function return type). For example:

```
PROCEDURE ALPHA (Q, R: REAL) [PUBLIC]; FORWARD;

PROCEDURE BETA (S, T:REAL);
  BEGIN ALPHA (S, 3.14) END;

PROCEDURE ALPHA;
  BEGIN BETA (Q, 6.28) END;
```

Attributes, if enabled, can be given to procedures and functions, and are placed after the heading in brackets separated by commas (for example, [PUBLIC] in the above example). The EXTERN directive is also available.

These can only be used on a procedure or function directly nested in a program, module, or implementation, or in an interface (that is, the first level of nesting–the PURE attribute, discussed with functions below, can be used at any level). This restriction prevents access to non-local stack variables.

The PUBLIC attribute allows a procedure (or function) to be called by other loaded code, and cannot be used with the EXTERN directive.

The EXTERN directive permits a call to some other loaded code, using the linker. PUBLIC, EXTERN, and ORIGIN provide a low level way to link Pascal routines with other routines in Pascal or other languages; the MODULE and UNIT provide higher level methods of linking Pascal to Pascal.

A procedure or function declaration with the EXTERN or FORWARD directive consists only of the heading,

without an enclosed block. EXTERN routines have an implied block outside of the program.

FORWARD routines will be fully declared later in the same program or implementation. The keyword EXTERNAL is a synonym for EXTERN.

Function Specifics

Function declarations serve to define parts of a program which compute a value. Functions are activated by the evaluation of a function designator which is part of an expression.

A function declaration has the same form as a procedure declaration, except that the header also gives the type of value returned by the function.

Functions are the same as procedures, except that they are invoked in an expression instead of a statement, and they return a value.

Within the function's block, either in the function body itself or a procedure or function nested within the block, at least one assignment to the function identifier must be executed to set the returned value.

This is not checked at runtime. However, if there is no assignment at all to the function identifier, the compiler will give an error message.

Functions can return any simple, structured, or reference type, except a super array type (a derived type is permitted) or a structure containing a file; that is, any type that can be assigned.

A function heading is the same as a procedure heading, with the type returned by the function added. For example:

```
FUNCTION MAXIMUM (I,J:INTEGER):INTEGER [PURE];
```

The occurrence of the function identifier in an expression (except for passing the function as a parameter) invokes the function recursively, rather than giving the current value of the function.

To use the current value, the function RESULT, which takes the function identifier as a parameter, is available.

The PURE attribute applies to functions, not procedures; it is the only attribute that can be used in nested functions.

The optimizer assumes that the result of a PURE function is determined by its parameters, and that it does not modify the global environment in any way (that is, it leaves unchanged any variables not local to the function, and does no input/output).

The compiler does *not* check whether a PURE function modifies global variables, has any VAR or VARS parameters (CONST parameter is permitted), calls any procedures, or calls any functions that are not PURE.

Data Parameters

There are three types of parameters in procedures and functions:

- Value parameter
- Reference parameter
- Procedural parameter

IBM Pascal allows a super array type to be passed as a reference parameter (from within a procedure or function), a reference parameter to be declared as CONST, and explicit segmented reference parameters (ADR, ADS).

Value Parameter

When a *value parameter* is passed, the actual parameter is an expression. The expression is evaluated in the scope of the caller and assigned to the formal parameter, which is a variable local to the called procedure or function.

The actual parameter expression must be assignment-compatible with the type of the formal parameter. A formal parameter is assumed to be a value parameter unless otherwise specified.

Passing structured types by value is permitted but is inefficient since the entire structure must be copied.

A file variable or super array variable cannot be passed as a value parameter, since it cannot be assigned. However, a variable with a type derived from a super array can be passed.

Reference Parameter

When a *reference parameter* is passed, the keyword VAR precedes the formal parameter and the actual parameter must be a variable. The formal parameter denotes this actual variable during the execution of the procedure.

Any operation done on the formal parameter is performed immediately on the actual parameter. This is done by passing the machine address of the actual

variable to the procedure. This address is an offset into the default data segment.

If the selection of the variable involves indexing an array or dereferencing a pointer or address, these actions are executed before the procedure is executed.

The type of the actual parameter must be identical to the type of the formal parameter.

In the listing, passing a non-local variable as a VAR parameter gets a / or % in the G (global) column.

The following cannot be passed as VAR parameters:

- Component of a PACKED structure (except CHAR of a STRING or LSTRING)
- Any READONLY variable (includes CONST parameter, FOR control variable)
- A file buffer variable, or record of a WITH statement (errors in ISO Pascal not caught by IBM Pascal)

With the super array type feature a procedure or function can operate on an array with a particular component type and index type but with any upper bounds.

The formal parameter is a reference parameter of the super array type itself. The actual parameter type must be a type derived from it, or the super array type itself (that is, another reference parameter or dereferenced pointer).

For example:

```
TYPE REALS=ARRAY[0..*] OF REAL;
PROCEDURE SUMRS(VAR X:REALS; CONST X:REALS);
```

Super array type parameters cannot be assigned or compared as a whole, except for a formal parameter with the super array type LSTRING. Components of a super array type can be used normally.

The actual upper bounds (and lower bounds) of the array are available with the UPPER (and LOWER) functions. This permits routines that sort an array of any size, write it out, etc.

Since an actual parameter of an LSTRING type can be passed to a reference parameter of the super array type STRING, string handling routines frequently use STRING reference parameters.

With the constant parameter feature, a formal parameter can be preceded by the reserved word CONST, implying that the actual parameter is a read-only reference parameter. This is especially useful for parameters of structured types which may be constants, since a time-consuming value parameter copy is not needed.

The actual parameter can be a variable, function result, or constant value (including a constant identifier or structured constant). It cannot be an arbitrary expression value with operators and non-constant operands (since there are no operators returning structured types this is not a serious limitation).

No assignments can be made to the CONST parameter or any of its components. CONST super array types are permitted. A CONST parameter in one procedure cannot be passed as a VAR parameter to another procedure, but the reverse is permitted.

Example:

```
PROCEDURE ERROR (CONST ERRMSG: STRING);
```

With the address type feature, a VAR or CONST parameter passes an address that is really an offset into a default data segment.

In some cases, access to objects residing in other segments is required; to pass these objects by reference, the compiler must recognize that a segmented address containing both segment register and offset values is necessary. To ensure this, the parameter prefix VARS is used instead of VAR or CONST. For example:

```
PROCEDURE CONCATS(VARS T:STRING);
```

Procedural Parameter

When a *procedure* or *function parameter* is passed, the actual identifier is a procedure or function, and the formal parameter is a procedure or function heading (including any attributes) preceded by the word PROCEDURE or FUNCTION.

The actual procedure or function parameter list must be compatible with the formal procedure or function parameter list, and a formal function result type must be identical to an actual function result type.

Two parameter lists are compatible if they contain the same number of parameters, and for every parameter both are value parameters of identical (not just compatible) type, reference parameters of identical type and identical kind of reference (VAR, CONST, VARS), or procedure or function parameters with compatible parameter lists (and for functions, with identical result type).

In addition, the set of attributes for both the formal and actual procedural type must be the same, except that the PUBLIC attribute and EXTERN directive are ignored.

A PUBLIC or EXTERN procedure, or any local procedure at any nesting level, can be passed to the same type of formal parameter. However, the PURE attribute and any calling sequence attributes must match

Any actual procedure or function passed to an EXTERN procedure or function must itself be EXTERN or PUBLIC.

Note that a parameterless function identifier used as an actual parameter can be either a value parameter or a formal functional parameter, depending on the formal parameter type.

Examples of procedural parameters:

```
PROCEDURE ALPHA (FUNCTION FUNI(X,Y:REAL): REAL[PURE]

PROCEDURE PASS (PROCEDURE PARAMETER);
```

When a procedure (or function) passed as a parameter is finally activated, any non-local variables accessed are those in effect at the time the procedure (function) is *passed* as a parameter, *not* those in effect when it is activated. Internally, both the address of the routine and the address of the upper frame (in the stack) are passed.

IBM Pascal imposes a restriction that some predefined procedures and functions cannot be passed as parameters. Procedures and functions which are compiled in-line and therefore have no actual separate code cannot be passed.

Also, the READ, WRITE, ENCODE, and DECODE families are translated into other calls by the compiler

based on the argument types and so cannot be passed (corresponding routines in the file unit, or encode/decode unit, can be passed).

Standard level procedures and functions that *cannot* be passed are:

- CHR
- ORD
- SUCC
- PRED
- NEW
- DISPOSE
- ABS
- SQR
- ODD
- PACK
- UNPACK
- READ
- READLN
- WRITE
- WRITELN

Procedures and functions in features that *cannot* be passed are:

- LOWER
- UPPER
- LOBYTE
- HIBYTE
- ENCODE
- DECODE
- WRD
- BYWORD
- SIZEOF
- RESULT
- EVAL
- RETYPE

Example of formal procedure use:

```
PROCEDURE ALPHA;
  VAR I:INTEGER;

  PROCEDURE DELTA;
    BEGIN END;

  PROCEDURE BETA (PROCEDURE XPR);
    VAR GLOB:INTEGER;

    PROCEDURE GAMMA;
      BEGIN GLOB:=GLOB+1 END;

    BEGIN (* BETA *)
      GLOB:=0;
      IF I=0
        THEN [I:=1; BETA(GAMMA)]
        ELSE [GLOB:=GLOB+1; XPR]
    END; (* BETA *)

  BEGIN (* ALPHA *)
    I:=0;
    BETA(DELTA)
  END; (* ALPHA *)

BEGIN (* main *)
  ALPHA;
    •
    •
    •
END. (* main *)
```

In the example, when ALPHA is called, BETA is called passing the procedure DELTA. This call creates an instance of GLOB on the stack, which we'll call GLOB1. BETA first clears GLOB1.

Since I is 0, BETA is called recursively passing GAMMA, and at this time the access path to any non-local variables used by GAMMA (that is, GLOB1) is passed as well.

The second call to BETA also creates another instance of GLOB, which we'll call GLOB2. GLOB2 is cleared, and this time I is 1, so GLOB2 is incremented.

Then XPR is called, which is bound to GAMMA, so GAMMA is executed and increments GLOB1 (GLOB1 was in effect when GAMMA was passed). GAMMA returns, the second BETA call returns, the first BETA call returns, and ALPHA returns.

Procedural parameters have three general uses:

- Numerical analysis
- Calling some library routines
- Special applications

In numerical analysis a function might be passed to a procedure or function that finds the integral between limits, a maximum or minimum value, and so on.

In some libraries, routines take an error handling procedure or next value function, although none of the IBM Pascal library routines uses this method.

Finally, there are some interesting algorithms in such areas as parsing and artificial intelligence which use procedural parameters.

Internal Calling Conventions

Every active procedure or function has a "frame" allocated on the stack. The frame contains the following data, listed with higher addresses at the top:

frame pointer→upper frame pointer, if any
- parameters (values or address), if any
- return address to caller
- saved caller frame pointer
- local data, temporaries, etc.

The frame pointer is in the BP register. The frame pointer generally points at the highest address word in the frame, and is used to access frame data. A procedure or function nested within another procedure or function has an upper frame pointer to get at variables in the statically enclosing frame.

The following process takes place when calling a procedure or function:

- The caller saves any registers it needs (except the frame pointer), pushes parameters in the same order as in the source, and does the call.

- Calls to PUBLIC or EXTERN procedures or functions are 8088 FAR calls; other local calls are NEAR calls.

- The callee pushes the old frame pointer, allocates any other stack locations needed, and sets up its new frame pointer. It can alter any registers except DS, SS, and BP.

- To return, the callee restores the caller's frame pointer, releases the entire frame, and returns.

  Since the callee must remove any parameters, the 8088 "RET n" instruction is usually used, with "n" equal to the number of bytes occupied by the upper frame pointer.

Functions always return their value in registers; for structured types and pointers to super arrays (no matter what length), the caller allocates a frame temporary for the result, passing the address to the function like a parameter and receiving the address back in a register. For simple types and other reference types, a single byte is returned in AL, two bytes in AX, and four bytes in the pair ES.

The $RUNTIME and $ENTRY/$LINE metacommands control special calls at the beginning and end of a routine.

The next example illustrates some of the internal calling conventions and shows how to read and obtain information from the PAS2 listing file. The assembler routine following the example program illustrates how the offsets in the symbol table are arrived at.

Example 1:

```
 1   PROGRAM EXAMPLE;
 2     VAR I:INTEGER;
 3
 4     PROCEDURE PROC1;
 5       VAR K:INTEGER;
 6       PROCEDURE PROC2(VAR J:INTEGER);
 7         PROCEDURE PROC3;
 8           BEGIN
 9             J:=J+1;
10             K:=J
11           END;

Offset Length      Variable - PROC3
-   2      6       Return Offset, Frame Length

12         BEGIN
13           PROC3
14         END;

Offset Length      Variable - PROC2
-   4      8       Return Offset, Frame Length
-   2      2       J

15       BEGIN
16         PROC2(I)
17       END;

Offset Length      Variable - PROC1
-   0      6       Return Offset, Frame Length
-   4      2       K

18     BEGIN
19       I:=1;
20       PROC1
21     END.

Offset Length      Variable
    0      4       Return Offset, Frame Length
    2      2       I
```

```
Procedure/Function : PROC3

0000001   PUSH   BP
;Push the caller's frame pointer.
0000002   MOV    BP,SP
0000004   ADD    BP,04H
;Make the caller's SP(top of frame)
;the callee's frame pointer and add
;4 bytes to allow for the upper frame
;pointer and caller's BP.
0000007   SUB    SP,0004H
;Set up the SP(top of frame) for the
;callee. Leave room for local variables
;and temporaries.

L9:
   •
   •
   •
L10:

00001D    MOV    DI,[BP].00H
;Get the caller's frame pointer
000020    MOV    DI,FEH[DI]
;Get the address of J by adding the
;offset to the frame pointer.
000023    MOV    DX,[DI]
;Get the value of J.
000025    MOV    SI,[BP].00H
;Get the caller's frame pointer.
000028    MOV    SI,[SI]
;Get the caller's caller's
;frame pointer.
00002A    MOV    FCH[SI],DX
;Move the value of J to K
;through K's address.
00002D    LEA    SP,[BP]FCH
;Reset SP(top of frame) to
;point to caller's BP.
000030    POP    BP
;Pop the caller's BP.
000031    RET    0002H
;Throw away two bytes
;(the upper frame pointer)
;and return.
```

Procedure/Function : PROC2

```
000034    PUSH   BP
;Push the caller's frame pointer.
000035    MOV    BP,SP
000037    ADD    BP,06H
;Make the caller's SP
;(top of frame) the callee's
;frame pointer and add 6
;bytes to allow for the
;upper frame pointer,
;the VAR parameter, and the
;saved caller's frame pointer.
00003A    SUB    SP,0004H
;Set up the callee's SP
;(top of frame) and leave
;room for local
;variables and temporaries.

L13:

00003E    PUSH   BP
;Push the caller's BP(frame pointer).
00003F    CALL   PROC3
;Call PROC3 and push the return address.
000042    LEA    SP,[BP].FAH
000045    POP    BP
;Back up the SP to point to BP.
000046    RET    0004H
;Throw away four bytes
;(caller's frame pointer
;(upper frame pointer,
;and VAR parameter)) and
;return.
```

```
000049    PUSH   BP
;Push the caller's frame pointer.
00004A    MOV    BP,SP
00004C    ADD    BP,02H
;Make the caller's SP(top of frame) the callee's
;frame pointer and add 2 bytes to allow for the
;push of BP.
00004F    SUB    SP,0006H
;Set up the SP(top of frame) for the callee.
;Leave room for local variables and temporaries.

L16:

000053    PUSH   BP
;Push the caller's frame pointer (PROC1 is now
;the caller) because this is a nested procedure.
000054    MOV    DX,@@I
000057    PUSH   DX
;Push the address of I which is a global static
;variable.
000058    CALL   PROC2
;Call PROC2 and push the return address.
00005B    LEA    SP,[BP].FEH
;Reset the SP(top of frame) to point to the
;caller's BP.
00005E    POP    BP
00005F    RET
;Pop the BP for the caller and return (pop
;the return address).
```

Procedure/Function : Example

```
    •
    •
    •

L19:

00006F    MOV    I,0001H

L20:

000075    CALL   PROC1
;Call PROC1 and push the return address.
000078    LEA    SP,[BP].FCH
00007B    POP    BP
00007C    LRET
    •
    •
    •
```

Example 2 is an assembler routine callable from Pascal:

```
;Unsigned Addition

;FUNCTION UADDOK (A, B: WORD; VAR C: WORD
;                  BOOLEAN;
;set C:=A+B and return TRUE
;if no overflow
;FRAME:  VALUE A    ;10
;        VALUE B    ;8
;        ADR C      ;6
;        <RET/BP>   ;0

UADDOK  PROC  FAR
        PUSH  BP           ;save frame pntr
        MOV   BP,SP        ;to address parms
        MOV   AX,[BP+10]   ;get A
        ADD   AX,[BP+08]   ;add B
        MOV   BP,[BP+06]   ;adr C
        MOV   [BP],AX      ;result
        MOV   AX,0         ;assume bad
        JB    UADRET       ;yes it was
        INC   AX           ;make good
UADRET: POP   BP           ;get frame pntr
        RET   6            ;return pop 6
UADDOK  ENDP               ;bytes
```

CHAPTER 11. AVAILABLE PROCEDURES AND FUNCTIONS

Contents

Predeclared Procedures and Functions

Standard procedures and functions are "predeclared" in Pascal, so they can be re-defined within a program. IBM Pascal provides additional predeclared procedures and functions. These should be avoided if portability is necessary. All parameters to standard procedures and functions are value parameters unless otherwise stated.

There is a distinction in IBM Pascal among three kinds of procedures and functions provided:

- Those which the compiler translates into other calls or special generated code (for example, ORD, RETYPE).
- Those, like normal procedures and functions, that are simply predeclared (for example, SIN, RESET).
- Those not predeclared but part of the IBM Pascal standard library (for example, TIME, DATE).

File oriented procedures and functions are discussed separately in Chapter 12.

Dynamic Allocation Procedures

PROCEDURE NEW (VAR P: pointer)

Allocates a new variable V and assigns a pointer to V to the pointer variable P (a VAR parameter). If V is a super array type, the long form (below) must be used. If V is a record type with variants, the variants giving the largest possible size are assumed, permitting any variant to be assigned to P^.

PROCEDURE NEW (VAR P: pointer; T1, T2, ...Tn: tags)

Allocates a variable with the variant specified by the tag field values T1, T2, ...Tn. The tag field values are listed in the order of declaration. Any trailing tag fields can be omitted.

If all tag field values are constant, only the amount of space required on the heap is allocated, rounded up to a word boundary. The value of any omitted tag fields is assumed to be such that the maximum possible size is allocated.

If some tag fields are not constant values, the compiler assumes the first non-constant tagfield and all tags following have an unknown value and allocate the maximum size necessary for this case.

The programmer should set all tag fields to their proper values after the call to NEW and never change them. The compiler does not assign the tag values, or check that they are initialized correctly, or check that their value is not changed during execution.

In ISO Pascal, a variable created with the long form of NEW cannot be used as an expression operand, nor can it be passed as a parameter or a value be assigned to it. IBM Pascal does not catch these errors. Fields within the record can be used normally.

Assigning a larger record to a smaller one allocated with the long form of NEW would wipe out part of the heap. This condition is difficult to detect at compile time.

Therefore, any assignment to a record in the heap which has variants uses the actual length of the record in the heap, rather than the maximum length.

However, an assignment to a field in an invalid variant may destroy part of another heap variable or the heap structure itself. This error is not caught.

The super array type feature allows pointers to super arrays. The long form of NEW is used as described above, except that array upper bound values are given instead of tag values. *All upper bounds must be given.*

The entire array referenced by such a pointer cannot be assigned or compared, unless the reference type is LSTRING. The entire array can be passed as a reference parameter if the formal parameter is of the same super array type. Components of the array can be used normally.

PROCEDURE DISPOSE (VAR P: pointer)

Releases the memory used for the variable pointed at by P. P must be a valid pointer; not NIL, uninitialized, or pointing at a heap item that has been DISPOSEd already (these are checked if $NILCK is on).

P should not be a reference parameter, or a WITH statement record pointer, but these errors are not caught.

If the variable is a super array type or a record with variants, this form can be used safely to release it regardless of whether a long or short form of NEW was used to allocate it (using short form DISPOSE on a heap variable allocated with long form NEW is an ISO Pascal error not caught).

PROCEDURE DISPOSE (VAR P: pointer;
T1, T2, ...Tn: tags)

Can also be used to release the memory pointed at by P. The size of the variable is checked against the size implied by the tag field or array upper bound values T1, T2, ...Tn, where these are defined as in the NEW procedure.

See the SIZEOF function, which uses the same array upper bounds or tag value parameters to return the number of bytes in a variable.

FUNCTION TRUNC (X: REAL): INTEGER

X is REAL, result INTEGER; truncates toward zero (for example, TRUNC (1.4) is 1; TRUNC (-1.4) is -1). Error if ABS (X) > MAXINT.

FUNCTION ROUND (X: REAL): INTEGER

X is REAL, result INTEGER; rounds away from zero (for example, ROUND (1.6) is 2; ROUND (-1.6) is -2). Error if ABS (X) > MAXINT.

FUNCTION FLOAT (X: INTEGER): REAL

X is INTEGER, result REAL; converts integer to real. Normally not needed by the programmer, since integer-to-real is usually done automatically. Needed by the run time package, so included in Standard level.

FUNCTION ORD (X: ordinal): INTEGER

X can be any ordinal type; result is INTEGER:
X is INTEGER: return X
X is WORD: if X <= MAXINT, return X else return X - 2 * (MAXINT+1) (that is, return same 16 bit value)
X is CHAR: return ASCII code for X
X is enumerated type: return position of X in the type definition, starting with zero.
X can also be a pointer; returns value as INTEGER.

FUNCTION WRD (X: ordinal): WORD

X can be any ordinal type; result is WORD:
X is INTEGER: if X >= 0, return X, else return X + MAXWORD + 1 (that is, return same 16 bit value)
X is WORD: return X
X is CHAR: return ASCII code for X
X is enumerated type: return position of X in the type definition, starting with zero.
X can also be a pointer; returns value as WORD.

FUNCTION CHR (X: ordinal): CHAR

X can be any ordinal type; result is CHAR: ASCII code of result is ORD (X);
extension to ISO Pascal (in which X must be INTEGER); error if ORD (X) > 255 or ORD (X) < 0 (if $RANGECK on).

FUNCTION ODD (X: ordinal): BOOLEAN

X can be any ordinal type; true if ORD (X) is odd, else false.

FUNCTION SUCC (X: ordinal): ordinal

X is any ordinal type; result is same type;
returns Y such that ORD (Y) is ORD (X) + 1;
error if out of range, caught if $RANGECK on;
error if overflow, caught if $MATHCK on.

FUNCTION PRED (X: ordinal): ordinal

X is any ordinal type; result is same type;
returns Y such that ORD (Y) is ORD (X) - 1;
error if out of range, caught if $RANGECK on;
error if overflow, caught if $MATHCK on.

PROCEDURE PACK(CONST A:unpack-array; I:index; VAR Z: packed-array)

Used to move the elements of an unpacked array to a packed array. If A is an ARRAY [M..N] OF T and Z is a PACKED ARRAY [U..V] OF T then the above call is the same as:

```
FOR J := U TO V DO Z [J] := A [J-U+I]
```

or the equivalent for non-integer indices.

PROCEDURE UNPACK(CONST Z:packed-array; VAR A:unpack-array; I: index)

Moves from a packed array to an unpacked array, in a similar way. It is the same as:

```
FOR J := U TO V DO A [J-U+I] := Z [J]
```

In both PACK and UNPACK the parameter I is the initial index within A. The bounds of the arrays and the value of I must be reasonable; that is, the number of components in the unpacked array A from I to M must be at least as great as the number of components in the packed array Z. The $RANGECK switch controls checking the bounds.

Arithmetic Functions

All arithmetic functions take a value parameter of type REAL or a type compatible with INTEGER. ABS and SQR also take WORD.

All functions on REALs check for an invalid (uninitialized) value, as well as particular error conditions given below, and generate a runtime error if an error condition is found.

For INTEGER and WORD type ABS and SQR functions, if $MATHCK is on, error conditions generate a runtime error. If it is off, the result of an

error is undefined. The arithmetic functions are:

- ABS(X) Absolute value of X (REAL, INTEGER, WORD, INTLONG).
- SQR(X) Square of X [X * X] (REAL, INTEGER, WORD).
- SQRT(X) Square root; result REAL; error if X < 0.
- SIN(X) Sine of X radians; result REAL.
- COS(X) Cosine of X radians; result REAL.
- ARCTAN(X) Arctangent of X returns radians; result REAL.
- EXP(X) Exponential [e to the X] ; result REAL.
- LN(X) Logarithm [base e] of X; result REAL; error if X<=0.

Some mathematical functions found in other languages are not in Pascal, but are relatively simple to do in-line:

- MAX (X, Y) = X + (Y - X) * ORD (X < Y)
- MIN (X, Y) = X + (Y - X) * ORD (X > Y)
- SIGN (X) = ORD (X > 0) - ORD (X < 0)
- POWER (X, Y) = EXP (Y * LN (X)) {X to the Y for X > 0}

These could also be written in Pascal as user functions. This would be a good place to use the PURE attribute and $RUNTIME metacommand. For example:

```
{$RUNTIME+}
FUNCTION POWER (A,B:REAL): REAL [PURE];
```

```
BEGIN
  IF A<=0 THEN ABORT('Nonplus real to power',24,0);
  POWER:=EXP(B*LN(A));
END;
```

REAL Functions

The runtime library provides several additional REAL functions. These must be declared with the EXTERN directive:

RSIRQQ (A:REAL, B:INTEGER): Returns A**B, integer power.

RSRRQQ (A,B:REAL): Returns A**B, real power (A>=0).

MINRQQ (A,B:REAL): Returns minimum of A and B.

MAXRQQ (A,B:REAL): Returns maximum of A and B.

AT2RQQ (A,B:REAL): Returns arctangent of (A / B).

TANRQQ (A:REAL): Returns tangent of A.

ASNRQQ (A:REAL): Returns arcsine of A.

ACSRQQ (A:REAL): Returns arccosine of A.

TNHRQQ (A:REAL): Returns hyperbolic tangent of A.

SNHRQQ (A:REAL): Returns hyperbolic sine of A.

CHSRQQ (A:REAL): Returns hyperbolic cosine of A.

LNDRQQ (A:REAL): Returns logarithm base 10 of A.

ANNRQQ (A:REAL): Returns integral part of A (type REAL).

AINRQQ (A:REAL): Like ANNRQQ but rounds toward zero.

DXPRQQ (A:REAL):INTEGER: Returns decimal exponent of A. For example, if E is DXPRQQ(A) then 10 to the E-1<=ABS(A)<10 to the E.

M10RQQ (A:REAL; I:INTEGER): Returns A times 10 to the I.

MP2RQQ (A:REAL; I:INTEGER): Returns A times 2 to the I.

Extended Intrinsics Feature

The extended intrinsics feature provides the following procedures and functions:

PROCEDURE ABORT (CONST STRING, WORD, WORD)

Procedure that stops program execution in the same way as an internal runtime error. The STRING (or LSTRING) is an error message; the first WORD is an error code (see Appendix B for error code allocations), and the second WORD can be anything, and will appear in a field called STATUS.

The parameters, and available data about the machine state (program counter, frame pointer, stack pointer) and the source position of the ABORT call (if $LINE or $ENTRY are on) are given to the user in a termination message.

If $RUNTIME is on, machine state data is for the location where the first call to a procedure or function compiled with $RUNTIME on occurred.

FUNCTION LOWER (expression): value

Function with one parameter of one of the following types: array, set, enumerated, or subrange; returns respectively the lower bound of an array, lowest allowable element of a set, first value of an enumerated

type, or lower bound of a subrange, of the appropriate type; note that only the type (not the value) of the expression is used; always constant.

FUNCTION UPPER (expression): value

UPPER is similar to LOWER, but returns the upper bound, etc. of the type of the parameter. UPPER is constant unless the expression is of a super array type, in which case the actual upper bound of the super array type is returned.

FUNCTION LOBYTE (integer-word): integer-word
FUNCTION HIBYTE (integer-word): integer-word

Returns the least significant or most significant byte (value 0..255), of the parameter with the same type (INTEGER or WORD) as the parameter; note that the least significant byte may be the first or second byte of the word, depending on the target processor.

FUNCTION BYWORD (one-byte, one-byte): WORD;

Takes two parameters, of any ordinal type that fits in one byte; returns a WORD with the first byte in the most significant part and the second byte in the least significant part; note again that the order of the bytes is by significance and not by address.

PROCEDURE EVAL (expression, expression, ...expression)

Procedure that evaluates its parameters only; used to evaluate an expression as a statement; any number of parameters of any type. Can be used to evaluate a function for its side effects only.

FUNCTION RESULT (function-identifier): value

Function used to access the current value of a function; can only be used within the function itself or in a procedure or function nested within it.

FUNCTION SIZEOF (variable): WORD
FUNCTION SIZEOF (variable, tag1, tag2, ...tagn): WORD

Returns the size of a variable, in bytes of type WORD; tag values or array upper bounds are set as in the NEW and DISPOSE functions; if the variable is a record with variants, and the first form is used, the maximum size possible is returned; if the variable is a super array, the second form giving upper bounds must be used.

ENCODE (VAR LSTRING, X:M:N): BOOLEAN

Boolean function that converts expression X to its external ASCII representation and puts this character string into the LSTRING. Returns true unless the LSTRING is too small to hold the string generated, in which case returns false and the value of the LSTRING is undefined.

Works exactly the same as the WRITE procedure, including the use of M and N parameters (see Chapter 12, "Textfile Input and Output"). X must be type INTEGER, WORD, enumerated, one of their subranges, BOOLEAN, REAL, or a pointer (address types need the .R or .S suffix).

DECODE (CONST STRING, VAR X:M:N): BOOLEAN

Boolean function that converts the character string in the STRING (or LSTRING) to its internal representation and assigns this to X.

Returns true unless the character string is not a valid external ASCII representation of a value whose type is assignment compatible with X, in which case returns false and the value of X is undefined.

Works exactly the same as the READ procedure, including the use of M and N parameters. If X is a subrange, DECODE returns false if the value is out of range (regardless of the setting of $RANGECK).

Leading and trailing spaces and tabs in the STRING are ignored; all other characters in the STRING must be part of the representation. X must be one of the types given above for ENCODE.

System Intrinsics Feature

The system intrinsics feature provides the following procedures and functions:

RETYPE (type-identifier, expression)

Generic type escape; function returning expression as type. The types implied by the type-identifier and the expression should usually have the same length, but this is not checked. RETYPE for a structure can be followed by component selectors (array index, fields, deference, etc.).

RETYPE is a "dangerous" type escape, and may not work as intended. There are two other ways to change type:

- One can declare a record with one variant of each type needed, assign an expression to one variant and get the value back from another variant (this can be done in Standard level and is an error not caught).

- One can declare an address variable of the type wanted and assign to it the address of any other variable (using ADR).

These methods have subtle differences and quirks, and should be avoided whenever possible.

The following procedures take value parameters of type ADRMEM, but since all ADR types are compatible, the ADR of any variable or constant can be used as the actual parameter. They are dangerous, but useful in some cases.

PROCEDURE MOVEL (S, D: ADRMEM; L: WORD)

Moves L characters (bytes) starting at S^ (source) to D^ (destination), starting at the left end of the strings and continuing to the right. *There is no bounds checking*, regardless of the $RANGECK or $INDEXCK setting (that is, it MOVES bytes starting from the left end of th source into the left end of the destination).

PROCEDURE MOVER (S, D: ADRMEM; L: WORD)

Like MOVEL but starts at the right end of the strings.

Example:

```
TYPE S10=STRING(10);
VAR ST:S10;
BEGIN
  ST:='1234567890'
  MOVER(ADR ST[6],ADR ST&1],5);
  (*result: '6789067890'*)
  MOVEL(ADR ST[1],ADR ST&1rb.3],6);
  (*result; '6767676790'*)
END.
```

PROCEDURE FILLC (D: ADRMEM; L: WORD; C: CHAR)

Fills D with L copies of the char C; as with MOVEL and MOVER *there is no bounds checking.* The corresponding segmented address versions of these routines, called MOVESL, MOVESR, and FILLSC, are also available; they are declared with ADSMEM instead of ADRMEM parameters.

String Intrinsics Feature

The string intrinsics feature provides a set of procedures and functions; some operate on STRINGs and LSTRINGs and some on LSTRINGs only.

The compiler supports STRINGs and LSTRINGs directly in the following ways:

LSTRING Assignment

Any LSTRING value can be assigned to any LSTRING variable, as long as the maximum length of the target variable is greater than or equal to the current length of the source value and neither is the super array type LSTRING. If the maximum length of the target is less than the current length of the source, only the target length is assigned and a runtime error occurs if $RANGECK is on.

STRING Assignment

A STRING value can be assigned to a STRING variable, as long as the length of both sides is the same and neither is the super array type STRING. STRING assignments always generate relatively fast code. Passing either kind of string as a value parameter is much like an assignment.

Comparison

For LSTRINGs, the <, <=, >, >=, < >, and = operators use the length byte for string comparisons, and the operands can be of any length. Comparison assumes the shorter operand is extended with CHR (0) components. The operands can be of the super array type LSTRING. For STRINGs, the same relational operators can be used, but the bounds must be the same and super array type STRINGs cannot be used.

READ LSTRING

Reads until the LSTRING is filled or the end of a line, setting the current length to the number of characters read. Write LSTRING uses the current length. See also READSET (Chapter 12) which reads into an LSTRING as long as input characters are in a given SET

OF CHAR. Read STRING pads with blanks if the line is shorter than the STRING. Write STRING writes all the characters in the string.

T.LEN

The current length of an LSTRING variable T can be accessed as T.LEN as well as T[0]. T.LEN has type BYTE. This notation can be used to assign a new length as well as get the current length.

The maximum length of an LSTRING, as well as the length of a STRING, can be found with the UPPER function; this is especially useful to find the upper bound of a super array reference parameter.

Note that mixed STRINGs and LSTRINGs cannot be assigned or compared (unless the STRING is constant). The MOVEL routine can be used to assign STRINGs to LSTRINGs or vice versa (see also the COPYSTR and COPYLST procedures below).

LSTRING constants are normal STRING constants, since constants of type STRING or CHAR change automatically to type LSTRING if necessary. NULL (the zero length LSTRING) is the only explicit LSTRING constant.

In the following descriptions, all STRING parameters (CONST or VAR) can take either a STRING or an LSTRING. All LSTRING parameters are VAR LSTRING and must take an LSTRING variable.

LSTRING Specific Intrinsics

PROCEDURE CONCAT (VAR D: LSTRING; CONST S: STRING)

S is appended to the end of D. D length increases by length of S. Error if upper (D) < length (D) + upper (S)

PROCEDURE DELETE (VAR D: LSTRING; I, L: INTEGER)

Deletes L characters from D starting with D [I]. D length decreases by L. Error if length (D) < I + L - 1.

PROCEDURE INSERT (CONST S: STRING; VAR D: LSTRING; I: INTEGER)

Inserts the string S characters starting just before D [I]. D length increases by the length of S. Error if upper (D) < upper (S) + length (D) or length (D) < I.

PROCEDURE COPYLST (CONST S: STRING; VAR D LSTRING)

Copies S to D. Error if upper (D) < upper (S). D lengtl set to S length.

STRING or LSTRING Intrinsics

FUNCTION POSITN (CONST P: STRING;CONST S: STRING;I: INTEGER): INTEGER

Returns integer position of the pattern P in S starting the search at S [I]. Returns 0 if not found or I > upper (S); returns 1 if P is the null string. No error conditions

FUNCTION SCANEQ (L: INTEGER; P: CHAR; CONST S: STRING; I: INTEGER): INTEGER

Scans, starting at S [I], and returns the number of characters skipped; stops scanning when a character equal to pattern P is found or L characters have been skipped; if L < 0, scans backwards and returns a negative number. Returns L parameter if no characters equal to pattern P found. Returns 0 if I > upper (S). No error conditions.

FUNCTION SCANNE (L: INTEGER; P: CHAR; CONST S: STRING; I: INTEGER): INTEGER

Like SCANEQ, but stops scanning when a character not equal to pattern P is found.

PROCEDURE COPYSTR (CONST S: STRING; VAR D: STRING)

Copies S to D. Error if upper (D) < upper (S). Remaining characters in D set to blanks.

See also the MOVEL, MOVER, and FILLC procedures with the system intrinsics.

Library Procedures and Functions

The following routines are not predeclared; the user must declare them using the EXTERN directive.

FUNCTION UADDOK(A,B:WORD;VAR C:WORD): BOOLEAN;
FUNCTION SADDOK(A,B:INTEGER;VAR C: INTEGER):BOOLEAN;
FUNCTION UMULOK(A,B:WORD;VAR C:WORD): BOOLEAN;
FUNCTION SMULOK(A,B:INTEGER;VAR C: INTEGER):BOOLEAN;

These four functions do 16 bit signed or unsigned arithmetic without causing a runtime error on overflow (normal arithmetic may cause a runtime error even if $MATHCK is off). They all return true if there was no overflow, or false if there was overflow. They can be useful when doing extended precision arithmetic, or modulo 65536 arithmetic, or arithmetic based on user input data.

FUNCTION ALLHQQ (SIZE: WORD): WORD;

This is the heap allocation routine. It returns zero if the heap is full, one if the heap structure is in error, or the pointer value for an allocated variable with the size requested.

PROCEDURE TIME (VAR S: STRING);
PROCEDURE DATE (VAR S: STRING);

These procedures assign the current time or date to their STRING (or LSTRING) variables, as "HH:MM:SS" or "DD:MM:YY". They are set using DOS.

FUNCTION TICS: WORD;

Returns the value of the DOS timing location, from 0 to 99 hundreths of a second. The timing location is updated 18.2 times per second. Thus, the value returned by TICS is in increments of approximately .055.

PROCEDURE BEGXQQ;

Overall initialization routine; resets the stack and the heap, initializes the file system, calls BEGOQQ, and calls the program body. May be useful to restart from a catastrophic error. Does not close files.

PROCEDURE ENDXQQ;

Overall termination routine; calls ENDOQQ, terminates the file system closing any open files, and returns to the target operating system (or whatever called BEGXQQ). May be useful to end program execution without calling ABORT from inside a procedure or function.

BEGOQQ and ENDOQQ are called during initialization and termination, respectively. They could be used to invoke a debugger, or write customized messages like time of execution. They may also be used for other special purposes.

FUNCTION DOSXQQ (COMMAND: BYTE; PARM: WORD): BYTE;

This function can be used to invoke the IBM Personal Computer DOS directly.

CHAPTER 12. FILE SYSTEM

Contents

The File System

The discussion of the file system first describes the file types, file structures, and file modes. Next is a discussion of the primitive operations: GET, PUT, RESET, REWRITE, EOF, EOLN, PAGE, and accessing the buffer variable.

Then the higher level routines READ, READLN, WRITE, and WRITELN are described, followed by the extended I/O feature routines: ASSIGN, CLOSE, DISCARD, READSET, and READFN, including temporary files and default file control block fields for error and mode control.

Finally the system I/O feature is introduced.

Introduction to Files

In the Pascal language a "file" is, in theory, just another abstract data type. Of course, most Pascal implementations connect variables of this type to actual operating system data files.

IBM Pascal files connect to the normal kinds of files available in the IBM Personal Computer Disk Operating System (DOS).

These include video displays, printers, generic stream devices (like an RS232 port), and disk files with textual or internal format data that are accessed with several different methods.

IBM Pascal always uses the DOS to access files and does not impose additional formatting or structure.

Files come in two basic structures and three modes.

File Structures

The file structures are BINARY and ASCII.

BINARY structure files in IBM Pascal (similar to the standard Pascal FILE of some type), correspond to unformatted operating system files.

The primitives GET and PUT transfer a fixed number of bytes per call equal to the length of one component of the BINARY file.

ASCII structure files (like the standard Pascal data type TEXT) correspond to textual operating system files. The Pascal TEXT type is like a FILE OF CHAR, but groups of characters are organized into "lines." Primitives like GET and PUT always operate on a character basis on these files.

In Pascal, textfiles are divided into lines with a "line marker," conceptually a character not in the type CHAR. Although a textfile can in theory contain any value of type CHAR, in IBM DOS writing a CHAR(13), carriage return, terminates the current line (record).

This character value is used as the "line marker" in this case, and when read always looks like a blank. To the user every line is followed by a line marker which reads as a blank.

In a similar way, the character CHR(26) indicates the end of a text file. This means that writing CHR(26) ends the file, and an attempt to read CHR(26) is an error (reading past end of file).

File Modes

The standard file modes are TERMINAL, SEQUENTIAL, and DIRECT:

- TERMINAL mode files always correspond to an interactive display/keyboard or printer
- SEQUENTIAL mode files correspond to a diskette file or other sequential access device
- DIRECT mode corresponds to diskette file or other random access device

Standard Pascal files can have SEQUENTIAL, TERMINAL, or DIRECT mode. The extended I/O feature includes the ability to access the mode of a file.

SEQUENTIAL and TERMINAL mode files are opened for either reading or writing at the beginning of the file, and records are accessed in order until the end of the file.

SEQUENTIAL and TERMINAL mode ASCII structure files have variable length records (lines). All BINARY structure files have fixed length records (lines or components).

The declaration for a file in Pascal tells the structure but not the mode; for example, FILE OF STRING (80) indicates BINARY structure and TEXT indicates ASCII structure.

The assignment F.MODE := SEQUENTIAL sets the mo for SEQUENTIAL files.

Pascal has two predeclared files, INPUT and OUTPUT. In IBM Pascal they start out connected to the user's keyboard and display and have ASCII structure and TERMINAL mode, but can be reassigned and/or given another mode.

TERMINAL mode input operates in different ways depending on whether the file has ASCII or BINARY structure. For ASCII structure (type TEXT), entire lines are read at one time, allowing the usual DOS inter-line editing (backspace, advance cursor, cancel, etc.) while the line is being typed and echoing characters as typed in the usual way.

Since an entire line is read at once, the programmer cannot read characters as they are typed. However, for BINARY structure TERMINAL mode (usually type FILE OF CHAR), the programmer can GET characters

as they are typed; no inter-line editing or echoing is done. This method permits doing screen editing, menu selection, and other interactive programming on a keystroke rather than line basis.

The READ procedure cannot be used to access this facility. In this TERMINAL/BINARY mode, two special characters exist to facilitate this operation. The value CHR(0) as the first byte means that no data is present. This value cannot be read. The value CHR(255) cannot be written to a TERMINAL BINARY file because it means "read a character" to DOS. For example:

```
PROGRAM SAMPLE (INPUT,OUTPUT);
VAR F: FILE OF CHAR;
FUNCTION INKEY: CHAR;
  BEGIN
    REPEAT GET(F) UNTIL F^<>CHR(0);
    INKEY:=F^;
END;

VAR C:CHAR;
  BEGIN
    WRITELN ('Terminal binary(echo)');
    ASSIGN (F,'USER');
    RESET (F);
    ASSIGN (G,'USER');
    REWRITE (G);
    REPEAT C:=INKEY; WRITE (G,C); UNTIL
      C = '.';
END.
```

This example program reads and echoes characters until a "." is encountered. The INKEY function waits in a loop until a character is available. Special characters are also read in, like Ctrl-C, Esc, backspace, etc.

Later descriptions of READ and WRITE procedures are defined in terms of the primitives GET, PUT, and buffer variable access. In all descriptions below, F is a file parameter (files are always reference parameters), and F∧ is the buffer variable.

PROCEDURE GET (VAR F)

If there is a next component in the file F, then the current file position is advanced to the next component, the value of this component is assigned to the buffer variable F∧, and EOF (F) becomes false.

Advancing and assigning may be deferred internally using "lazy evaluation" (see below). If no next component exists, then EOF (F) becomes true and the value of F∧ becomes undefined. EOF (F) must be false before GET (F) since reading past the end of file gives a runtime error.

If F∧ is a record with variants, the variant with the maximum size is read.

PROCEDURE PUT (VAR F)

If the previous operation on F was a REWRITE or PUT (or other write procedure) but not a RESET or GET (or other read procedure), then the value of the buffer variable F∧ is written to file F and the value of F∧ becomes undefined, or else an error occurs.

EOF (F) must be true before PUT (F). EOF (F) will always be true after PUT (F).

If F^ is a record with variants, the variant with the maximum size is written.

PROCEDURE RESET (VAR F)

Reset the current file position to its beginning and do a GET (F). If the file is not empty, the first component of F is assigned to the buffer variable F^, and EOF (F) becomes false. If the file is empty, the value of F^ is undefined and EOF (F) becomes true.

This is a necessary initialization prior to reading the file F. A RESET closes the file and then opens it again.

If the filename has not been set (as a program parameter, with READFN, or with ASSIGN), or the file cannot be found by the operating system, an error occurs.

If an error occurs during RESET the file is closed (even if the file was opened correctly and the error came with the initial GET). RESET (INPUT) is done automatically when a program is initialized, but is also allowed explicitly.

Note that an explicit GET (F) immediately following a RESET (F) assigns the second component of the file to the buffer variable. However, a READ (F, X) following a RESET (F) sets X to the first component of F, since READ (F, X) is "X := F^; GET (F)."

A RESET will clear the error trapping flag (set it false).

PROCEDURE REWRITE (F)

Reset the current file position to its beginning. The value of F∧ is undefined and EOF (F) becomes true. This is a necessary initializing operation prior to writing the file F.

REWRITE closes the file and then opens it again. If the file does not exist in the operating system, it is created; if it does exist, its old value is lost. The filename must have been set (as a program parameter, with READFN, or with ASSIGN).

If an error occurs during REWRITE, the file is closed; an existing file with the same name is not affected.

REWRITE (OUTPUT) is done automatically when a program is initialized, but is also allowed explicitly.

REWRITE does not do an initial PUT the way RESET does an initial GET. A REWRITE will clear the error trapping flag (set it false).

FUNCTION EOF: BOOLEAN
FUNCTION EOF (VAR F): BOOLEAN

EOF tells whether the buffer variable F∧ is positioned at the end of the file F for modes SEQUENTIAL and TERMINAL, so if EOF (F) is true either the file is being written or the last GET reached the end of the file. EOF with no parameter is equivalent to EOF (INPUT).

EOF (INPUT) is generally never true, except when the terminal character CHR(26) generates an end of file status (entered as Ctrl-Z), or if INPUT is reassigned to another file. Calling the EOF (F) function also accesses the buffer variable F∧, causing a GET if no previous GET was done, because "lazy evaluation" defers the initial GET.

FUNCTION EOLN: BOOLEAN
FUNCTION EOLN (VAR F): BOOLEAN

Indicates whether the buffer variable F$\wedge$ is positioned at the end of a line in the textfile F after a GET (F). Calling EOLN (F) when EOF (F) is true is an error in ISO Pascal usually caught in IBM Pascal. The file must have ASCII structure (be of type TEXT) to use EOLN.

If EOLN (F) is true the value of F$\wedge$ is a blank but the file is positioned at a "line marker." EOLN with no parameter is equivalent to EOLN (INPUT). Calling the EOLN (F) function also accesses the buffer variable F$\wedge$.

PROCEDURE PAGE
PROCEDURE PAGE (VAR F)

Causes skipping to the top of a new page when the textfile F is printed. Since PAGE writes to the file, the initial conditions described for PUT must be true. The file must have ASCII structure. PAGE with no parameter is equivalent to PAGE (OUTPUT).

If F is not positioned at the start of a line, PAGE (F) first writes a line marker to F.

If F has mode SEQUENTIAL, then PAGE (F) writes a form feed, CHR (12).

If F has mode TERMINAL, then PAGE (F) writes the following six characters: carriage return, line feed, form feed, backspace, space, carriage return.

Accessing the Buffer Variable

An internal mechanism (generally transparent to the programmer) is used to handle interactive terminal input in a natural way.

- "Lazy Evaluation" applies to all ASCII structure files and is necessary for natural terminal input.

In this case, the compiler generates a runtime call executed before any use of the buffer variable (assignment to it, using it in an expression, checking EOF or EOLN, and so on).

Lazy Evaluation

Lazy evaluation only occurs when reading from a textfile. The file's buffer variable has a special status value which can be "full" or "empty."

Two rules apply to any textfile F:

- When F's buffer variable is accessed, if its status is "empty" the next file component is physically input; after an access the status is always "full."

- When the GET (F) procedure is called, if its status is "empty" the next file component is physically input; after a call to GET, the status is always "empty."

This effectively defers physical input until actually needed by a buffer variable access (including the EOLN and EOF functions). RESET first sets the status "full," and calls GET which just sets it "empty" without doing physical input. Lazy evaluation is permitted in the ISO standard.

Examples:

```
{RESET (INPUT); done automatically}
WRITE (OUTPUT, "Enter number: ");
READLN (INPUT, F00);
```

The RESET does a GET, which just sets the buffer variable status to "empty." The first physical action to the screen/keyboard is the prompt output from the WRITE.

The READLN does a series of "temp := INPUT^; GET (INPUT)" operations; in each of these physical input occurs when INPUT^ is fetched, and the GET just sets the status back to "empty." READLN ends with the sequence:

WHILE NOT EOLN DO GET; GET

This has the effect of skipping trailing characters and the line marker (that is, the carriage return). The EOLN function invokes the physical input; when the carriage return is input, the EOLN status is set.

Both the GET in the WHILE loop and the trailing GET just set the status back to "empty." The last physical input in the sequence above is reading the carriage return.

The Standard Pascal READ procedure is always one component ahead; it calls GET after setting the target variable to the buffer variable's value.

The motivation for this is setting the EOF value before READ to avoid READing the end of file. In Pascal's original batch processing sequential file environment, this method is acceptable.

If RESET and READ were used without lazy evaluation, the user would have to type the first character of F00 to satisfy the GET in RESET, then see the prompt, and type the rest of F00, a carriage return, and the first character of the next response.

IBM Pascal generates an I/O system call when accessing the buffer variable. However, if the buffer variable is an actual reference parameter, the procedure or function using that parameter can do I/O to the same file, and these special calls cannot be executed.

Passing any buffer variable as a reference parameter is an error (although only a warning is given), and the effect of doing GET or PUT on a file in a procedure or function accessing the buffer variable indirectly through a reference parameter is undefined. Assigning the address of a buffer variable to an address type variable has much the same insecurity.

Textfile Input and Output

Legible input and output in Standard Pascal is done with textfiles (that is, of type TEXT, with ASCII structure) that are passed as program parameters to a Pascal program and in its environment represent some input or output device such as a display, a keyboard, a line printer, or a diskette file. The extended I/O feature permits using additional files not given as program parameters.

In order to facilitate the handling of textfiles, the four standard procedures READ, READLN, WRITE, and WRITELN are introduced in addition to the procedures GET and PUT.

The new procedures are used with a more flexible syntax for their parameter lists, allowing for, among other things, a variable number of parameters.

Moreover, the parameters need not necessarily be of type CHAR, but can also be of certain other types, in which case the data transfer is accompanied by an implicit data conversion operation. In some cases parameters can have additional formatting values included which affect the data conversions used.

If the first variable is a file variable, then this is the file to be read or written. Otherwise, the standard files INPUT and OUTPUT are automatically assumed as default values in the cases of reading and writing respectively. These two files have TERMINAL mode and ASCII structure and are predeclared as:

```
VAR INPUT, OUTPUT: TEXT
```

The files INPUT and OUTPUT are treated like other textfiles. They can be used with ASSIGN, CLOSE, RESET, REWRITE, and the other procedures and functions. However, if present as program parameters, they are not initialized with a filename; instead they are assigned to the user's keyboard and display.

RESET or REWRITE, as appropriate, is done automatically whether or not they are present as program parameters.

Textfiles represent a special case among file types insofar as they are substructured into lines by "line markers." If, upon reading a textfile F, the file position is advanced to a line marker (that is, past the last character of a line), then the value of the buffer variable F$\wedge$ becomes a blank, and the standard function EOLN (F) yields the value true.

Advancing the file position once more either causes EOF (F) to become true (if the end of the file is reached), or assigns to F$\wedge$ the first character of the next line and sets EOLN (F) to false (if the next line is not empty), or assigns a blank to F$\wedge$ and sets EOLN (F) true again (if the next line is empty).

Line markers, not being elements of type CHAR in the standard, can in theory only be generated by the procedure WRITELN. However, an actual character, CHR(13), is used for the line marker, and it is possible to write (but not to read) one.

When a textfile being written is CLOSEd, a final line marker is automatically appended to the last line if one is not already present and the file is not empty.

When a textfile being read reaches the end of the file, if the file is not empty a line marker for the last line is returned even if one was not present in the file. Therefore, lines in a textfile file always end with a line marker.

The set of readable and writeable types is: any simple type, pointer type, STRINGs, and LSTRINGs. Also any list of data written by a WRITELN is generally readable with the same list in a READLN unless an LSTRING occurs that is not on the end of the list.

There is no provision for formatted reading, although the effect can be achieved by reading a STRING and DECODEing it. However, provision for M and N parameters (see below) in the READ and READLN procedures has been made for later input format control.

Data parameters to READ, READLN, WRITE, and WRITELN with textfiles can always take one of the following forms:

P P:M P:M:N P::N

The P is a variable for READs and an expression for WRITEs, as described below.

The M and N values can be considered value parameters of type INTEGER and are used for input and output formatting in various ways. The extended I/O feature permits M and N values for both READs and WRITEs and permits giving N without M (as in P::N); using them in a non-standard way is an "error not caught."

In some cases only M or N, or neither, will be actually used; unused M and N values are ignored. Omitting M or N is the same as using the value MAXINT; for example, WRITE (12:MAXINT) will use the default M value (8 in this case). M and N values are not accepted for BINARY files.

Interactive prompt and response is very easy. To have input on the same line as the response, use WRITE for the prompt. READLN must always be used for the response. For example:

```
WRITE ('Enter command: '); READLN (response);
```

As mentioned, most of the textfile procedures and functions will assume the file INPUT or OUTPUT, as appropriate, if no file is given. For example, if I is type INTEGER then READ (I) is the same as READ (INPUT, I). Any variable that is, or contains, a file and occurs as the first parameter is assumed to be the explicit file to use.

However, suppose FINT is declared as a FILE OF INTEGER and one wants to read from INPUT to the buffer variable of FINT. The call READ (FINT^) will not work because FINT is assumed to be the file to READ from; instead the call READ (INPUT, FINT^) must be used.

The Procedure READ

The following rules hold for the procedure READ:

- F denotes a textfile (type TEXT) and P1, P2, ...Pn denote variables (with optional M and N values) of type CHAR (or a subrange), INTEGER (or a subrange), or REAL.

- With the extended I/O feature, variables can also be of type WORD (or a subrange), an enumerated type (including BOOLEAN), a pointer type, STRING, or LSTRING.

- When a variable of a subrange type is read, the value read must be in range or an error occurs, regardless of the setting of $RANGECK.

The READ process for formatted types (everything except CHAR, STRING, and LSTRING) first reads characters into an internal LSTRING and then DECODEs the string to get the value. Two important points apply to formatted reads:

- Leading spaces, tabs, form feeds, and line markers are skipped. For example, when doing READLN (I, J, K) where I, J, and K are INTEGERs the numbers can all be on the same line or spread over several lines.

- Characters are read as long as they are in the set of characters valid for the type wanted, using the READSET procedure. For example, "-1-2-3" is read as the string of characters for a single INTEGER, but gives an error in DECODE. This means items should be separated by spaces, tabs, line markers, or some character not permitted in the format.

The procedure READ has the following characteristics:

1. READ (P1, P2, ...Pn) is equivalent to:
   READ (INPUT, P1, P2, ...Pn)

2. READ (F, P1, P2, ...Pn) is equivalent to:
   BEGIN READ (F, P1); READ (F, P2); ...READ (F, Pn) END

3. M and N values in READ are ignored, except as noted for an N value with enumerated types.

4. If P is of type CHAR, then: READ (F, P) is equivalent to: BEGIN P := F^; GET (F) END

5. If P is type INTEGER (or WORD) or a subrange thereof, then READ (F, P) implies reading a sequence of characters from F which form a

number according to the normal Pascal syntax and assigning the number to P. Non-decimal notation (16#C007, 8#74, 10#19, 2#101, #Face) is accepte for both INTEGER and WORD. If P is an INTEGER type a leading + or – sign is accepted. If P is a WORD type decimal numbers up to MAXWORD are accepted (that is, from 32768 to 65535).

6. If P is type REAL, then READ (F, P) implies reading a sequence of characters from F which form a number of the appropriate type and assignin the number to P. Non-decimal notation is not accepted. When reading a REAL value, a number with a leading or trailing decimal point (not both of course) is accepted, even though this form gives a warning if used as a constant in a program.

7. If P is an enumerated type or BOOLEAN, a number is read as a WORD subrange and a value assigned to P such that the number is the ORD of the enumerated type's value. In addition, if P is type BOOLEAN, reading one of the character sequences 'TRUE' or 'FALSE' cause true and false, respectively, to be assigned to P. The number read must be in the range of the ORD values of the variable identifiers.

8. Should P be a pointer type, a number is read as a WORD and assigned to P, in an implementation defined way such that writing a pointer and later reading it yields the same pointer value. The address types should be read as WORDs using .R or .S notation.

9. If P is a STRING (n), then the next n characters are read sequentially into P. Preceding line marker spaces, tabs, or form feeds are not skipped.

When the line marker is encountered before n characters have been read, the remaining characters in P are set to blanks and the file position remains at the line marker.

If the STRING is filled with n characters before the line marker is encountered, the file position remains at the next character. P can be the super array type STRING (for example, a reference parameter or pointer referent variable).

READ (STRING) is handy when reading numbers packed together without spaces; for example, reading a line in a FORTRAN format as "(20I4)" can be done by repeatedly reading into a STRING (4) and then calling DECODE to get the integer value of the STRING.

10. If P is an LSTRING (n), then next n characters are read sequentially into P, and the length of the LSTRING is set to n. Preceding line markers, spaces, tabs, or form feeds are not skipped.

    When the line marker is encountered before n characters have been read, the length of the LSTRING is set to the number of characters read and the file position remains at the line marker.

    If the LSTRING is filled with n characters before the line marker is encountered, the file position remains at the next character. P can be the super array type LSTRING (for example, a reference parameter or pointer referent variable).

    READ (LSTRING) is handy when reading entire lines from a textfile, especially when the length of the line is needed. For example, the easiest way to copy a textfile is by using READLN and WRITELN with an LSTRING variable.

Note: Points 5 through 8 also apply to the function DECODE.

The procedure READ can also be used to read from a file which is not a textfile (that is, BINARY mode). In this case,

READ (F, X) is equivalent to: BEGIN X := F^; GET (F) END

The form READ (F, P1, P2, ...Pn) can be used, as above. Normally the M and N values are not accepted in BINARY reads.

Procedure READLN

READLN (P1, P2, ...Pn) is equivalent to: READLN (INPUT, P1, P2, ...Pn)

READLN (F, P1, P2, ...Pn) is equivalent to: BEGIN READ (F, P1, P2, Pn); READLN (F) END

READLN (F) is equivalent to: BEGIN WHILE NOT EOLN (F) DO GET (F); GET (F) END

READLN is used to skip to the beginning of the next line. It can only be used with textfiles (ASCII mode).

Procedure WRITE

The following rules hold for the procedure WRITE:

- F denotes a textfile (type TEXT) and P1, P2, ...Pn denote expressions (with optional M and N values) of type CHAR, INTEGER, REAL, BOOLEAN, or STRING.

- With the extended I/O feature, expressions can also be of type WORD, an enumerated type, a pointer type, or LSTRING.

The procedure WRITE has the following characteristics:

1. WRITE (P1, P2, ..Pn) is equivalent to:
   WRITE (OUTPUT, P1, P2, ...Pn)

2. WRITE (F, P1, P2, ...Pn) is equivalent to:
   BEGIN WRITE (F, P1); WRITE (F, P2); ...WRITE (F, Pn) END

3. In WRITE the M value can always be used as the number of characters to write, called the field width. In ISO Pascal, M must be greater than zero, and if the expression being written requires less than M characters it is padded on the left with spaces.

   M can also be negative or zero; if negative, the absolute value of M is used but padding occurs on the right instead (this is officially an error not caught). If the representation of the expression cannot fit in ABS (M) character positions, then extra positions are used as needed for numeric types, or the value is truncated on the right for string types.

   If M is omitted or equal to MAXINT, a default value is used (see below). The N value only applies if P is type REAL (for the number of decimal places); INTEGER, WORD, or pointer (for the output radix); or enumerated (to choose the numeric or identifier value).

4. If P is of type CHAR, default M is 1, and WRITE (F, P) is equivalent to: BEGIN F^ := P; PUT (F) END

5. If P is type INTEGER (or WORD) or a subrange thereof, the decimal representation of P is written on the file.

   If P is positive, first a leading blank is written, unless the M value is less than zero indicating flush left output.

   If P is negative, a leading minus sign is always written; of course WORD values are never negative. The default value of M is 8. If ABS (M) is smaller than the representation of the number, additional character positions are used as needed.

   N is used to write in hexadecimal, decimal, octal, or binary using N equal to 16, 10, 8, or 2; an extension to ISO Pascal. Omitting N or setting N to MAXINT implies decimal. WORD decimal numbers from 32768 to 65535 are written normally (not their negative integer equivalents).

   All values written should be separated by spaces or some other character not valid in numbers, so they will be read as separate numbers.

6. If P is of type REAL, a decimal representation of the number P, rounded to the specified number of decimal places, is written on the file.

   If the N is missing or equal to MAXINT, a floating-point representation of P is written to the file, consisting of a coefficient and a scale factor.

   If N is included, a rounded fixed point representation of P is written to the file, with N digits after the decimal point. If N is zero, P is written as a rounded integer, with a decimal point.

The exact representation used is described in ISO Pascal.

The default value of M for REAL values is 14. If ABS (M) is smaller than the representation of the REAL value, additional character positions are written as needed. Some examples:

```
WRITE (123.456)
' 1.2345600E+02'

WRITE (123.456:20)
' 1.2345600000000E+02'

WRITE (123.456::3)
'          123.456'

WRITE (123.456:2:3)
' 123.456'

WRITE (123.456:-20:3)
'123.456              '
```

7. Should P be an enumerated type and N omitted or equal to MAXINT, then ORD (P) is written on the file, as if it were a WORD.

   If P is of type BOOLEAN, then one of the strings 'TRUE' or 'FALSE' (depending on the value of P) is written on the file, as a STRING. The default value of M is 6 for BOOLEANs. The ORD value is never written for BOOLEANs as it is for enumerated types.

8. If P is a pointer type, it is written as a WORD (above) in an implementation defined way such that writing the value and later reading it yields the same pointer value. The address types should be written as WORDs using .R or .S notation.

9. If P is of type STRING (n), then the value of P is written on the file. The default value of M is the length of the STRING, n.

   Should ABS (M) be less than the length, only the first ABS (M) characters are written; if M is zero nothing is written. The right portion of the STRING is always truncated, even if M is negative.

10. With the extended I/O feature, if P is of type LSTRING (n) then the value of P is written on the file. The default value of M is the current length of the string, P.LEN.

    If ABS (M) is less than the current length only the first ABS (M) characters are written; if M is zero nothing is written. The right portion of the LSTRING is always truncated, even if M is negative.

    If ABS (M) is greater than the current length, spaces fill the remaining positions (not characters past the length in the LSTRING). Note that a string of M blanks can be written with NULL:M.

    Note: Points 5 through 8 also apply to the function ENCODE.

The procedure WRITE can also be used to write onto a file which is not a textfile (that is, BINARY). In this case:

WRITE (F, X) is equivalent to: BEGIN F^ := X; PUT (F) END

The form WRITE (F, P1, P2, ...Pn) can be used, as above. Normally the M and N values are not accepted in BINARY writes, but may be available in customized versions of the compiler.

The Procedure WRITELN

WRITELN (P1, P2, ...Pn) is equivalent to:
WRITELN (OUTPUT, P1, P2, ...Pn)

WRITELN (F, P1, P2, ...Pn) is equivalent to:
BEGIN WRITE (P1, P2, ...Pn); WRITELN (F) END

WRITELN (F) appends a line marker to the file F.

WRITELN is used to write a line marker (end a line).
It can only be used with textfiles (ASCII mode).

PROCEDURE ASSIGN (VAR F; CONST N: STRING)

This procedure assigns a DOS filename in a STRING (or LSTRING) to a file F. ASSIGN always truncates any trailing blanks, and overrides any filename set previously.

A filename must be set (during program parameter initialization, with READFN, or with ASSIGN) before the first RESET or REWRITE on a file. ASSIGN on an open file (after RESET or REWRITE but before CLOSE) produces an error. ASSIGN on the files INPUT and OUTPUT is allowed, but since they are opened automatically they must be CLOSEd first.

The filename takes the following form:

[*drive*:]*filename*[*.ext*]

Filename can be 1-8 characters and the optional extension *ext* can be 1-3 characters. The drive specification (such as A:) can also be given.

Filename can be CHR(0)-tempfile.

Filename can be one of a number of special DOS names for non-buffered I/O:

- **PRN** and **LPT1** for the line printer
- **CON** for the console
- **NUL** for the dummy device
- **COM1** or **AUX** for an RS232 port

Filename can be one of two special IBM Pascal names for non-buffered I/O:

- **USER** for the console
- **LINE** for an RS232 port

  Note: Specifying **CON** for input is not recommended because DOS buffers 512 characters before they become available to IBM Pascal. **USER** should be used for non DOS-buffered console I/O.

PROCEDURE CLOSE (VAR F)

CLOSE does a DOS close on a file, ensuring that the file access is correctly terminated.

This is especially important for file variables allocated on the stack or the heap; these files must be closed before a RETURN or DISPOSE releases the file control block. Therefore, these files are closed automatically when a RETURN or DISPOSE releases stack or heap file variables. Files allocated statically (in fixed memory) are closed automatically when the program terminates.

Note that some runtime errors might cause loss of control by the Pascal runtime system; in these cases files being written may not be closed, and the information in them may be lost.

If necessary, any DOS buffers associated with a file being written are emptied (however, the Pascal buffer variable is not PUT).

If the file is of type TEXT, was being written, and the last non-empty line did not end with a line marker, one is added to the end of the last line.

If the file has the mode SEQUENTIAL and was being written, an end-of-file is written. A CLOSE on a file that is already closed or never opened (no RESET or REWRITE) is permitted. CLOSE is not ignored if error trapping is on and there was a previous error. CLOSE turns off error trapping for the file, and clears the error status if no errors were found.

PROCEDURE DISCARD (VAR F)

Discard closes and deletes an open file. It is much like CLOSE, except that the file is deleted.

Temporary Files

Sometimes a program needs a "scratch" file for intermediate data only. To use the program without needing to give the scratch file a name in the correct format, IBM Pascal provides a feature called a *temporary file*. To create a temporary file, ASSIGN the name CHR (0); for example, ASSIGN (F, CHR (0)). The file system will create a unique name for the file.

Temporary files get deleted when they are CLOSEd, either explicitly by the programmer or implicitly when the file gets de-allocated. RESET and REWRITE do not delete the file.

PROCEDURE READSET (VAR F, VAR L: LSTRING, CONST S: SETOFCHAR)

READSET reads characters and puts them into L as long as the characters are in the set S and there is room in L. If no file parameter is given INPUT is assumed, as in READ and WRITE.

Leading spaces, tabs, form feeds, and line markers are always skipped. Reading ceases when the line marker is encountered, since it is never in the CHAR type.

This procedure is used, along with ENCODE, by the runtime system to do the formatted READ procedures, as well as to read filenames wtih READFN (below). It is handy when reading and parsing input lines for simple command scanners.

PROCEDURE READFN (VAR F, P1, P2, ...Pn)

READFN is the same as READLN with two exceptions: 1) the file parameter F should be present (INPUT is assumed but a warning is given), and 2) if a parameter P is of type FILE, a sequence of characters forming a valid filename is read from F and assigned to P in the same manner as ASSIGN. Parameters of other types are read in the same way as the READ procedure.

Note that READFN (unlike READLN) does not read a line marker. If the first parameter in a READFN call is a file of any type, it is assumed to be the textfile to read characters from (it is not assumed that the file's name should be read using INPUT as the default source).

READFN is used internally to read a program's parameters. It is handy when reading a filename (perhaps from the user) and assigning the filename to some file in one operation.

File Field Values

A file variable is really a record called a file control block of type FCBFQQ. A few standard fields in this record can be used to handle file modes and error trapping. Additional fields and the record type FCBFQQ itself can be used as described below.

The normal record field syntax is used to access a file's FCB fields. For a file F, the fields available are F.MODE, F.TRAP, and F.ERRS. These fields can be examined or changed at any time.

F.MODE This field contains the mode of the file. These values are constants of the predeclared enumerated type FILEMODES. The MODE field is only used by the file system during RESET and REWRITE, so changing the MODE field of an open file has no effect (but it is not recommended).

A file's mode is SEQUENTIAL by default, except for INPUT and OUTPUT which have mode TERMINAL.

F.TRAP: BOOLEAN This field is initially false; if set true, it turns on error trapping for file F so that if an input/output error occurs the program does not end and the error code can be examined. If the field is false and an I/O error occurs, the program ends. Closing the file will set the trap to false.

Note: RESET and REWRITE close the file.

F.ERRS: 0..15 This field contains the error code for file F; zero means no error, and values from 1 to 15 (see the list below) imply an error condition. If a file operation (except CLOSE and DISCARD) is attempted for file F and F.ERRS is not zero, and if F.TRAP is true, the operation is ignored (the program does not end). If F.TRAP is false, the program immediately ends.

CLOSE and DISCARD do not examine the initial value of F.ERRS, so they are never ignored and do not cause an immediate termination. However if CLOSE or DISCARD themselves generate an error condition, F.TRAP is used as above to determine whether to trap the error or end.

An operation that is ignored because of an error condition does not change the file itself, but it may change the buffer variable or READ procedure input variables.

File F Error Codes:

CODE	ERROR
0	No error
1	Reserved
2	Reserved
3	OPERATION–invalid operation: GET if EOF, RESET a printer, etc.
4	Reserved
5	Reserved
6	Reserved
7	FILE NAME–invalid syntax, name too long, no temp names, etc.
8	DEVICE FULL–disk full, directory full, all channels allocated
9	Reserved
10	FILE NOT FOUND–file itself not found
11	Reserved
12	Reserved
13	FILE NOT OPEN–file closed, I/O to unopen FCB
14	DATA FORMAT–data format error, decode error, range error
15	LINE TOO LONG–buffer overflow, line too long

File Variables in Headings

A file variable may be placed in the heading of a Pascal program as a program parameter, and declared as a file in a VAR section. At runtime, the runtime system will obtain the filename from the user at the keyboard.

The filename here has the same format as in the ASSIGN function.

Example:

```
PROGRAM FILES(FILE1,FILE2);

VAR FILE1: FILE OF REAL;
    FILE2: TEXT;
BEGIN
  •
  •
  •
END.
```

When FILES is started, the program will attempt to read the program parameters from the command line. If the program cannot find them there, it will prompt for them. Thus, starting the program FILES with the files NUMBERS.DAT and LINES.DAT might look something like this:

```
A>files numbers.dat lines.dat
    or
A>files numbers.dat
FILE2: lines.dat
    or
A>files
FILE1: numbers.dat
FILE2: lines.dat
```

(Remember, each line ends with Enter.)

System I/O Feature

The system I/O feature allows procedures and functions with a formal reference parameter of any file type or the type FCBFQQ to be called with an actual parameter of the identical FILE type or the identical FCBFQQ type.

FCBFQQ is the underlying record type used to implement the file type.

Usually the interface for the FCBFQQ type (and any other types needed) is brought in with an $INCLUDE file, and the clause "USES FILKQQ" (FILKQQ is the name of the unit containing the FCBFQQ type).

This FCBFQQ declaration is included on the PAS1 diskette in the file FILKQQ.INC.

Procedures and functions with reference FCBFQQ parameters can be called with any file type, including predeclared procedures and functions like CLOSE and READ. An FCBFQQ type variable can be passed to procedures like READLN and WRITELN that require a textfile.

This permits, for example, calling the file system interface routines directly, and other file system activities.

Doing this requires a thorough knowledge of the file system. Appendix B describes the file system interface and file control block in detail. Using this information, one could write the file system interface for another operating system, for example.

DIRECT Files

To use a DIRECT file, the mode field must be set to DIRECT before the file is opened with RESET or REWRITE.

If the file is actually read or written sequentially, the usual READ and WRITE procedures can be used, and READ and WRITE can be used normally on textfiles (ASCII structured files). However, due to the way Pascal files are defined, BINARY structure files generally need GET and PUT instead of READ and WRITE.

READ, in particular, always reads ahead one component in Standard Pascal, which does not produce the desired effect when used with IBM Pascal's SEEK procedure.

The following apply to DIRECT mode files:

- They are always opened for both reading and writing.

- Records can be accessed randomly by record number.

- The line length for ASCII structured files sets the record length, given as a constant in parentheses after the word TEXT. Examples:

```
VAR SMALLBUF:TEXT(2);
    DEFAULTX:TEXT;
    RANDOMTXT:TEXT(132);
```

  Usually, the line length declaration is for DIRECT mode files but it may be used for other TEXT files.

- For DIRECT ASCII files, the line length, whether explicitly set or set by default, is the record length used when reading or writing.

PROCEDURE GET (VAR F)

When using GET(F), the value of EOF(F) before GET(F) can be true or false, since DIRECT mode permits repeate GETs at the end of file.

PROCEDURE PUT (VAR F)

When using PUT(F), the value of EOF(F) before PUT(F) can be true or false.

PROCEDURE REWRITE (VAR F)

REWRITE will allow either reading or writing and the file will be created if it did not already exist. If it did exist, its old value is not lost.

PROCEDURE RESET (VAR F)

RESET will allow either reading or writing but the file will not be created automatically. The initial GET reads record number one.

PROCEDURE EOF (VAR F)

If EOF(F) is true, either the last operation was a WRITE (the file may or may not be positioned at the end in this case) or the last GET reached the end of file. See "EOF in DIRECT Mode" later in this chapter.

PROCEDURE SEEK (VAR F; N:WORD)

The record number parameter N can also be of type INTEGER or WORD. For textfiles (ASCII structure), records are lines; for other files (BINARY structure), records are components. Record numbers start at one (*not* zero). F must have a mode of DIRECT.

SEEK modifies a field in F so that the next GET or PUT applies to record number N.

If F is an ASCII file, SEEK sets the lazy evaluation status "empty."

DIRECT mode must be set explicitly before RESET or REWRITE to use the SEEK procedure on a file.

SEEK followed by GET

If component N exists in the file F, then the current file position is advanced to this component, the value of this component is assigned to the buffer variable F∧, and EOF(F) becomes false. Advancing and assigning may be deferred internally (see above).

If the component N does *not* exist, the value of F∧ becomes undefined. SEEK is usually followed by use of the buffer variable value or READ/READLN for ASCII files. SEEK is not usually followed by READ for BINARY files.

SEEK followed by READ, BINARY files

Since READ(F,B) is "B:=F∧; GET(F)", using READ results in assigning the current buffer variable (whatever

it may be) to B and reading the record following record N. Since this action has nothing to do with value of record N, it should be avoided.

SEEK followed by READ/READLN, ASCII files

This works as one would expect, thanks to lazy evaluation. SEEK(F,N) sets the status to "empty" so the buffer variable access implied by READ actually causes record N to be read to the internal line buffer before any characters are processed by the READ.

SEEK followed by PUT

The value of the buffer variable F^ is written to file F at record number N, the value of F^ becomes undefinec and EOF(F) is set true. SEEK followed by PUT is usually preceded by a setting of the buffer variable value.

SEEK followed by WRITE, BINARY files

Since WRITE(F,B) is "F^:=B; PUT(F)", using WRITE after a SEEK call results in writing B to record N, as one would expect. However, since READ after SEEK doesn't work for BINARY files, users may wish to avoid WRITE after SEEK for BINARY files to preserve syntactic consistency.

SEEK followed by WRITE/WRITELN, ASCII files

This works as one would expect, writing characters to record N. As a general rule, ASCII DIRECT file I/O works as one would expect, but BINARY DIRECT file I/O should use GET and PUT instead of READ and WRITE.

EOF in DIRECT Mode

Once a file is updated (or created) in DIRECT mode, the end-of-file is not exact. Thus the EOF can't be used to detect the true end-of-file in DIRECT mode.

In addition, the file's exact EOF will no longer be detectable in sequential mode.

In ASCII-structured files, some data may be inaccessible using SEQUENTIAL mode, if the file was updated using DIRECT mode and re-accessed using SEQUENTIAL.

CHAPTER 13. COMPILANDS

Contents

A "compiland" is a source file compiled by the compiler. IBM Pascal permits three kinds of compilands: *programs, modules,* and implementations of *units.* Each of these can contain *interfaces* to units as well.

Programs

A Pascal program has the form of a procedure declaration, except for its heading, and the period at the end (see "Procedure and Function Declarations" in Chapter 10). The statements between the BEGIN and END are called the *body* of the program. For example:

```
PROGRAM ALPHA (OUTPUT, AFILE, PARAMETER);
VAR AFILE: TEXT; PARAMETER: STRING(10);
BEGIN
  REWRITE (AFILE); WRITELN (AFILE,PARAMETER);
END.
```

After compilation and linking, the following program can be run in any of the ways shown. Assume the program ALPHA exists in the file ALPHA.EXE:

```
A>ALPHA
AFILE: DATA.FIL
PARAMETER: ABCDEFGHIJ
```

or

```
A>ALPHA DATA.FIL
PARAMETER: ABCDEFGHIJ
```

or

```
A>ALPHA DATA.FIL ABCDEFGHIJ
```

(Remember you must press Enter after each line.)

ALPHA is the program identifier, and it becomes the identifier for a parameterless PUBLIC procedure at a static level "above" all the other identifiers in the program. As a result, a program identifier can be re-declared within a program and the usual scoping rules apply.

Since the program identifier is at the same level as the predeclared identifiers (for example, INTEGER or READ), to use an identifier like INTEGER would produce a duplicate identifier error.

The example program is also given the PUBLIC identifier ENTGQQ which is called during initialization to start program execution. The program body can be called as a PUBLIC procedure from another compiland using the program identifier or ENTGQQ, although this is not recommended.

The program's parameters are the means by which it communicates with its environment. In standard level, variables of FILE type should be parameters since there is no other way for the variable to be assigned an operating system filename.

Program parameters are not at all like procedure parameters. They are not passed to the body of the program, and *must* be declared in the variable declaration part of the block constituting the program.

ASSIGN and READFN can be used to assign filenames without the need for them to appear as program parameters.

If there are no program parameters, and the files INPUT and OUTPUT are *not* used, the form "PROGRAM identifier;" can be used.

The two predeclared files, INPUT and OUTPUT, get special treatment as program parameters. Their value is not set like the other program parameters. They should always be present as program parameters, whether they are used explicitly or implicitly, to suppress a compiler warning if they are used and not present as program parameters.

INPUT and OUTPUT can be redefined, but the textfile I/O procedures and functions will assume the original definition. RESET(INPUT) and REWRITE(OUTPUT) are automatically generated.

Every program parameter variable (except INPUT and OUTPUT) gets a value during program initialization by doing a READFN of one form or another.

Each parameter must be of a type acceptable to READFN; that is, a simple type (INTEGER, WORD, CHAR, BOOLEAN, enumerated, their subranges, REAL), a pointer type, STRING, LSTRING, or a FILE type. Program parameters must be entire variables; no component selection (including .R and .S address types) is permitted.

The READFN call for program parameters internally uses the file INPUT. But before every parameter is read, a special call to the routine PPMFQQ causes READFN to actually get characters returned from the DOS interface routine called PPMUQQ. PPMUQQ gets them from the command line that started the program. Both routines are passed the program's identifier, for use as a prompt, if necessary.

An ambiguity exists when specifying an LSTRING as a program parameter. Assume the following program exists:

```
PROGRAM PARMS(In);
VAR In:LSTRING(10)
    •
    •
    •
END;
```

If, after compilation and linking, the user starts the program as follows:

```
A>parms
```

it is not clear whether the user is entering an LSTRING of length 0, or wishes to be prompted for the LSTRING. IBM Pascal assumes the latter, and will always prompt if an LSTRING appears to be omitted.

For the application programmer who wishes to use advanced command line processing, the Unit U procedure PPM provides this capacity. To use PPM, do not specify any parameters in the program statement.

Modules

Using a modules is a quick way to combine several compilands into one program. A module is a program without a body.

The module header gives the module identifier, which has the same scope as a program identifier. A module has no body and no parameters. The compiland ends with "END.". Here is a sample module:

```
MODULE BETA [PUBLIC];
VAR X:INTEGER;
  PROCEDURE GAMMA;
  BEGIN
   X := 1;
  END;
  FUNCTION DELTA: WORD;
  BEGIN
   DELTA := 123;
  END;
END.
```

If no explicit attributes are given, [PUBLIC] is assumed, so [ ] must be used if no attributes are wanted

Although there is not a module body, a module becomes a parameterless procedure which can be called from other compilands using the module identifier.

In some cases the compiler generates module initialization code which should be executed by the module procedure. This should be done if the module declares any FILE variables, or if the module USES any interfaces that need initialization. The compiler will give a warning "Initialize Module" if it finds either of these while compiling the module. The module should also be initialized if it declares any module variables, such as X above, and uses them in the module.

In these cases, the module should be declared as a parameterless EXTERN procedure in the program compiland and this procedure should be called only once, before the module procedures are used. Example:

```
PROCEDURE MODULNAME; EXTERN;
```

Module variables are not automatically given any attributes and are the same as program variables, except for initialization of FILE variables as indicated.

Module procedures can be called as EXTERN procedures without a module initialization call if the above cases do not apply.

Units

A *unit* contains constants, types, super types, variables, procedures, and functions which are declared in the unit's *interface.* An interface can be used by any compiland or by another interface.

An *implementation* compiland contains the bodies of the procedures and functions in a unit. By separating the interface from the implementation, a program can be written and compiled before the implementation is written, or loaded with several implementations (for example, one in Pascal, one in machine language, etc.).

A large Pascal program is best organized as a main program and a number of units. However, only a program, module, interface, or implementation may use a unit, not an individual procedure or function.

A compiland using an interface *must* start with the source code for the interface, generally a separate file brought in with an $INCLUDE metacommand. This form is more reliable and easier, than putting the interface physically in front of every compiland using it, because there is only one master copy of the interface.

A list of all identifiers exported from the unit is required in the unit clause (optional in a USES clause) to avoid identifier conflicts. Special internal code initializes units and controls interface versions.

A Pascal source file consists of zero or more unit interfaces, separated by semicolons, followed by a program, a module, or an implementation, followed by a period. Each of these is called a "division."

A unit consists of the unit identifier, followed by a list of identifiers in parentheses, called "constituents" of the unit. The identifiers in parentheses are provided by the unit or required by a program, interface, or implementation.

The unit is preceded by the keyword UNIT for a unit provided, and USES for a unit required. All unit identifiers in a source file must be unique. The identifiers in parentheses can be different in the providing and requiring divisions.

Correspondence between provided and required identifiers is by position in the list (similar to formal and actual parameters in procedures). The list is optional in a USES clause, and if not given, the list in the UNIT clause will be used. Naming different identifiers in USES clauses can resolve identifier conflicts between interfaces. Multiple USES clauses such as "USES A; USES B; USES C;" can be combined with the form "USES A, B, C;."

For example:

Program file:

```
(*$INCLUDE:'GRAPHI'*)
PROGRAM PLOTBOX (INPUT, OUTPUT);
USES GRAPHICS (MOVE, PLOT);
BEGIN (*GRAPHICS*)
  MOVE (0, 0);
  PLOT (10, 0); PLOT (10,10);
  PLOT (0, 10); PLOT (0, 0);
END.
```

Interface file GRAPHI:

```
INTERFACE;
  UNIT GRAPHICS (BJUMP, WJUMP);
PROCEDURE BJUMP (X, Y: INTEGER);
PROCEDURE WJUMP (X, Y: INTEGER);
BEGIN
END;
```

Implementation file:

```
(*$INCLUDE:'GRAPHI'*)
(*$INCLUDE:'BASEPL'*)
IMPLEMENTATION OF GRAPHICS;
  USES BASEPLOT; (*identifiers below*)
  PROCEDURE BJUMP;
    BEGIN DRAWLINE (BLACK, X, Y) END;
  PROCEDURE WJUMP;
    BEGIN DRAWLINE (WHITE, X, Y) END;
BEGIN
  DRAWLINE (BLACK, 0, 0)
END.
```

Interface file BASEPL:

```
INTERFACE;
  UNIT BASEPLOT (BLACK, WHITE,
       DRAWLINE);
  TYPE RAINBOW = (BLACK, WHITE,
       RED, BLUE, GREEN);
  PROCEDURE DRAWLINE (C: RAINBOW;
       H, V: INTEGER);
END;
```

Implementation file:

```
(*$INCLUDE: 'BASEPL'*)
IMPLEMENTATION OF BASEPLOT
```

A USES clause can only occur directly after a program, module, interface, or implementation header. When a USES clause is encountered, all constituent identifiers (from the UNIT clause and the USES clause itself) are entered in the symbol table, and those associated with variables, procedures, and functions are associated with the corresponding identifiers in the interface, which become external references for the loader.

In the above example, when the program is compiled, every reference to the procedure PLOT generates an external reference to WJUMP. References to DRAWLINE, however, use the same identifier for the external reference.

Constants and types (including any super array types) in the interface are simply entered in the program's symbol table (with the new identifier, if any) so a type in the interface is identical to the corresponding type in the USES clause.

Record field identifiers are automatically the same in the program, interface, and implementation when the record name is given. Enumerated type constant identifiers must be given explicitly if needed; they are not automatically implied by the enumerated type identifier.

Labels cannot be provided by an interface, since the target label of a GOTO must occur in the same division as the GOTO.

The interface facility could almost be handled by an $INCLUDE. In place of the USES clause would be a file which declares everything EXTERN. The corresponding "implementation" file would declare everything as PUBLIC.

However, the UNIT method has several advantages over the $INCLUDE:

- The identifiers can be changed to avoid conflicts.
- Constants, types and procedures, and function parameters are only given in the interface, avoiding what would be an undetectable mismatch.
- The interface is generally cleaner.

$INCLUDE is useful for combining one or more interfaces in front of a program or implementation.

Interface Division

The interface section starts with the keyword INTERFACE, an optional version number in parentheses, and a semicolon. Next comes the UNIT keyword, the unit identifier, the parenthesized list of exported (constituent) identifiers, and another semicolon.

Any other units required by this interface come next, in USES clauses. Then come the actual declarations for all identifiers given in the interface list, using the usual CONST, TYPE, and VAR sections and procedure and function headings, in any order. No LABEL or VALUE sections are permitted.

Attributes can be given to variables, procedures, and functions. Since the PUBLIC or EXTERN attributes or EXTERN directive is given automatically, attributes which may conflict must not be used.

Normally, the only identifiers declared are the constituents, but declaring other identifiers is permitted If the interface will need a call to initialize the unit, the

keyword BEGIN causes it to be generated. The interface ends with the reserved word END and a semicolon. For example:

```
INTERFACE (3);
UNIT KEYFILE (FINDKEY,INSKEY,DELKEY,KEYREC);
  USES KEYPRIM (KFCB,KEYREC);

  PROCEDURE FINDKEY (CONST NAME: LSTRING;
    VAR KEY: KEYREC; VAR REC: LSTRING);
  PROCEDURE INSKEY (CONST REC: LSTRING;
                    VAR KEY: KEYREC);
  PROCEDURE DELKEY (CONST KEY: KEYREC);
  PROCEDURE NEWKEY (CONST KEY: KEYREC);
BEGIN
END;
```

In this example, KFCB is part of the KEYPRIM unit, but is not exported by the KEYFILE unit, which is allowed. NEWKEY is defined in the interface, but not exported by the KEYFILE unit, also allowed, but pointless since NEWKEY will be unknown in any USES of the unit and in the unit's IMPLEMENTATION.

The Implementation Division

An implementation of an interface starts with the reserved words IMPLEMENTATION OF, the unit identifier, and a semicolon. Next comes a USES clause for units it needs only for its own use; then the usual LABEL, CONSTANT, TYPE, VAR, and VALUE sections, and all procedures and functions mentioned as constituents (which must be in the outer block), or used internally, in any order.

The VALUE and LABEL sections can be used in the implementation, although they are not permitted in the interface.

Labels must only have GOTOs from the body itself (if any), not from procedures and functions within the implementation.

Constants, variables, and types declared in the interface are not re-declared in the implementation, but other "private" ones can be declared.

Procedures and functions that are constituents of the unit do not have their parameter list (it is implied by the interface), or any attributes.

The PUBLIC attribute is given automatically, unless the EXTERN directive is given explicitly.

All procedures and functions in the INTERFACE must be defined in the IMPLEMENTATION; however they can be given the EXTERN directive so several IMPLEMENTATIONS (or an IMPLEMENTATION and assembly code, etc.) can implement a single INTERFACE.

All procedures and functions using the EXTERN directive must appear first; the compiler checks for this.

Units can also be implemented in other languages, such as FORTRAN, or by a mixture of languages. If the interface is not implemented in Pascal, the proper calling sequence attribute must be used in the interface, and of course the user must be familiar with calling sequences and internal representation of parameters.

Several runtime units are implemented partially in Pascal and partially in assembly language. As mentioned, any implementation section that does not implement all interface procedures and functions must declare those not implemented with the EXTERN directive at the start of the implementation.

An implementation, like a program, can have a body; the body is executed when the program using the unit is invoked, so any initialization needed by the unit can be done.

This includes internal initialization, such as file variable initialization, as well as user initialization code. If the source file contains several units, each implementation body is called in the order their USES clauses are found in the source file. The body, as in a program, is a BEGIN. . .END statement.

At initialization time, the version number of the interface that the implementation was compiled with is compared against the version number of the interface that the program was compiled with. The two must be the same. This prevents trying to run a program with obsolete implementations of the units it uses. If no version number is given zero is assumed. For example:

```
IMPLEMENTATION OF KEYFILE;
  USES KEYPRIM (KBLOCK, KEYREC);

  VAR KEYTEMP: KEYREC;

  PROCEDURE FINDKEY;
    BEGIN {code for FINDKEY} END;

  PROCEDURE INSKEY;
    BEGIN {code for INSKEY} END;

  PROCEDURE DELKEY;
    BEGIN {code for DELKEY} END;

  PROCEDURE NEWKEY (CONST KEY: KEYREC);
    BEGIN {code for NEWKEY} END;

  END.
```

The implementation division ends with a period.

APPENDIXES

Contents

APPENDIX A. MESSAGES

Error conditions may be undetected, detected by the compiler, or detected by the runtime system. This appendix gives error messages and codes for errors detected by the compiler and runtime system. Compiler errors are divided into front end (pass one) errors and back end (pass two) errors; runtime errors are divided into file system errors and all other errors.

Front End Errors

Front end error and warning messages include a number as well as a message. Most contain a row of dashes and an arrow to the location of the error, but three (128, 129, and 130) occur after the body in the $SYMTAB listing area. The front end recovers from most errors, but a few are called "panic" errors and the front end only lists the rest of the program. These panic errors also give the message:

Compiler Cannot Continue

and occur in the following conditions:

- Error count set by $ERRORS exceeded
- End of file occurs when not expected
- Identifier scopes nested too deeply
- Cannot find PROGRAM, MODULE, or IMPLEMENTATION keyword
- Cannot find PROGRAM, MODULE, or IMPLEMENTATION identifier

The word "Warning" before a message indicates the intermediate code files produced by the front end are correct, and the condition is not severe or is just considered "unsafe." Other messages indicate true

errors; writing to the intermediate files stops, and these files are discarded when the front end is finished.

The error message **"Compiler"** refers to an internal consistency check which failed; no matter what source program is compiled, there should not be a way to get one of these messages.

101 Invalid Line Number
Line number above 32767; too many lines in source file.

102 Line Too Long Truncated
Source lines are limited to 142 characters.

103 Identifier Too Long Truncated
Any identifier longer than the maximum is truncated.

104 Number Too Long Truncated
Numeric constants are limited to the identifier length.

105 End Of String Not Found
The line ended before the closing quote was found.

106 Assumed String
A double quote is assumed to enclose a string; use single quote.

107 Unexpected End Of File
End of file in a number, metacommand, etc. (while scanning).

108 Meta Command Expected Command Ignored
A $ at the start of a comment was not followed by an identifier.

109 Unknown Meta Command Ignored
A metacommand identifier was unknown, or invalid.

110 Constant Identifier Unknown Or Invalid Assumed Zero
Setting a metacommand to a constant identifier (as in $DEBUG: A) and the identifier is unknown or not constant of the right type.

111 [unassigned]

112 Invalid Numeric Constant Assumed Zero
Setting a metacommand to a numeric constant. The constant has the wrong format or is out of range.

113 Invalid Meta Value Assumed Zero
Setting a metacommand to neither a constant or identifier.

114 Invalid Meta Command
One of +, -, or : is expected following a metacommand.

115 Wrong Type Value For Meta Command Skipped
The metacommand expects a string but an integer was given, or vice versa.

116 Meta Value Out Of Range Skipped
The $LINESIZE integer value was below 16 or above 160.

117 File Identifier Too Long Skipped
The $INCLUDE string value for the filename was too long.

118 Too Many File Levels
Too many $INCLUDE file nesting levels.

119 Invalid Initialize Meta
A $POP metacommand has no corresponding $PUSH metacommand.

120 CONST Identifier Expected
A $INCONST metacommand was not followed by an identifier.

121 Invalid INPUT Number Assumed Zero
The user input invoked by $INCONST was invalid in some way.

122 Invalid Meta Command Skipped
A $IF and its value was not followed by $THEN or $ELSE.

123 Unexpected Meta Command Skipped
A $THEN was found unrelated to any $IF metacommand.

124 Unexpected Meta Command
The metacommand was not in a comment; it was processed anyway.

125 Reserved

126 Invalid Real Constant
A type REAL constant with a leading or trailing decimal point.

127 Invalid Character Skipped
Source file character is not acceptable in program text.

128 Forward Proc Missing
The procedure or function given in the message was declared FORWARD but not found (message occurs in $SYMTAB area).

129 Label Not Encountered
The label given in the message was declared or used in a GOTO, BREAK, or CYCLE but not found (message occurs in $SYMTAB area).

130 Program Parameter Bad
The program parameter given in the message was never declared or has the wrong type for READFN (message occurs in $SYMTAB area).

131 [unassigned]

132 [unassigned]

Overflow Errors can occur in several contexts:

133 Type Size Overflow
The data type implies a structure bigger than 32766 bytes.

134 Constant Memory Overflow
Constant memory allocation has gone above 65534 bytes.

135 Static Memory Overflow
Static memory allocation has gone above 65534 bytes.

136 Stack Memory Overflow
Static frame memory allocation has gone above 65534 bytes.

137 Integer Constant Overflow
A type INTEGER signed constant expression out of range.

138 Word Constant Overflow
A type WORD or other unsigned constant expression out of range.

139 Value Not In Range For Record
Record tag value not in range of variant, in a structured constant, long form NEW/DISPOSE/ SIZEOF, or other application.

140 Too Many Compiler Labels
The compiler needs internal labels; program is too big.

141 Compiler

142 Too many identifier levels
Identifier scope level is over 15 (panic error).

143 Compiler

144 Compiler

145 Identifier Already Declared
An identifier can only be declared once in a given scope level.

146 **Unexpected End Of File**
End of file in statement, declaration, etc. (while parsing).

Common Substitution Mistakes: Get their own special messages, and are corrected with just a warning.

147 : **Assumed =**
148 = **Assumed :**
149 := **Assumed =**
150 = **Assumed :=**
151 [**Assumed (**
152 (**Assumed [**
153) **Assumed]**
154] **Assumed)**
155 ; **Assumed ,**
156 , **Assumed ;**
157 [unassigned]
158 [unassigned]
159 [unassigned]
160 [unassigned]
161 [unassigned]

Missing Symbol: If a particular symbol is expected in the source but not found, it may be inserted and one of the following messages given:

162 **Insert Symbol**
163 **Insert ,**
164 **Insert ;**
165 **Insert =**
166 **Insert :=**

167 **Insert OF**
168 **Insert]**
169 **Insert)**
170 **Insert [**
171 **Insert (**
172 **Insert DO**
173 **Insert :**
174 **Insert .**
175 **Insert ..**
176 **Insert END**
177 **Insert TO**
178 **Insert THEN**
179 **Insert ***
180 [unassigned]
181 [unassigned]
182 [unassigned]
183 [unassigned]
184 [unassigned]

If a particular symbol is expected in the source but is found after some invalid symbols, the invalid ones are deleted with the following two messages:

185 **Invalid Symbol Begin Skip**
186 **End Skip**

187 **End Skip**
The previous error message ended with the phrase "Begin Skip"; this message marks the end of skippe source text.

188 **Section Or Expression Too Long**
Compiler limit; try re-arranging the program or breaking up an expression with assignments to intermediate values.

189 Invalid Set Operator or Function
For example, MOD operator or ODD function with sets.

190 Invalid Real Operator or Function
For example, MOD operator or ODD function with reals.

191 Invalid Value Type For Operator or Function
For example, MOD operator or ODD function with enumerated type.

192 [unassigned]

193 [unassigned]

194 Type Too Long
Use of a variable or type with greater than 32766 bytes.

195 Compiler

196 Zero Size Value
Use of the empty record "RECORD END" as if it had a size.

197 Compiler

198 Constant Expression Value Out Of Range
Array index, subrange assignment, other subrange check.

199 Integer Type Not Compatible with Word Type
Common error indicates confusing signed and unsigned arithmetic; either change the positive signed value to unsigned with WRD () or change the unsigned value (< MAXINT) to signed with ORD ().

200 [unassigned]

201 Types Not Assignment Compatible
Assignment statement or value parameter.

202 Types Not Compatible In Expression
Expression mixing incompatible types.

203 Not Array Begin Skip
Variable followed by a left bracket (or parenthesis) is not array.

204 Invalid Ordinal Expression Assumed Integer Zero
The expression has the wrong type or a type that is not ordinal.

205 Invalid Use of PACKED Components
A component of a PACKED structure has no address (it may not be on a byte boundary); it cannot be passed by reference.

206 Not Record Field Ignored
Variable followed by a dot is not record, address, or file.

207 Invalid Field
Record variable and dot not followed by a valid field.

208 File Dereference Considered Harmful
When the address of a file buffer variable is calculated, the special actions normally done with buffer variables (that is, lazy evaluation for textfiles or concurrency for binary files), cannot be done; the buffer variable at this address may not be valid.

209 Cannot Dereference Value
Variable followed by an arrow is not pointer, address, or file.

210 Invalid Segment Dereference
Variable resides at segmented address, but a default segment address is needed; may need to make local copy of variable.

211 Ordinal Expression Invalid Or Not Constant
A constant ordinal expression was expected.

212 [unassigned]

213 [unassigned]

214 Out of Range For Set 255 Assumed
An element of a set constant must have an ordinal value <= 255.

215 Type Too Long Or Contains File Begin Skip
A structured constant must have 255 or fewer bytes; also, it cannot be, or contain, a file type or an LSTRING type.

216 Extra Array Components Ignored
An array constant has too many components for the array type.

217 Extra Record Components Ignored
A record constant has too many components for the record type.

218 Constant Value Expected Zero Assumed
A value in a structured constant is not constant.

219 [unassigned]

220 Compiler

221 Components Expected For Type
A structured constant needs more components for its type.

222 Overflow 255 Components In String Constant
A string constant must have 255 or fewer bytes.

223 Use NULL
The predeclared constant NULL must be used instead of two quotes.

224 Cannot Assign With Supertype LSTRING
A super array LSTRING cannot be source or target of assignment.

225 String Expression Not Constant
String concatenation with the asterisk only applies to constants.

226 **String Expected Character 255 Assumed**
Somehow a string constant had no characters, perhaps using NULL.

227 **Cannot Assign To Function**
Assignment to the function is not in the scope of the function.

228 **Cannot Assign To Variable**
Assignment to READONLY, CONST, or FOR control variable.

229 **Cannot Use As CONST Parameter Or Address Assumed Zero**
The expression has no address and cannot be reference parameter.

230 **Unknown Identifier Assumed Integer Begin Skip**
Unknown identifier, for which the address is needed.

231 **VAR Parameter Or WITH Record Assumed Integer Begin Skip**
Invalid identifier, for which the address is needed.

232 **Cannot Assign To Type**
Target of assignment is a file or otherwise cannot be assigned.

233 **Invalid Procedure Or Function Parameter Begin Skip**
Error in use of intrinsic procedure or function, such as:

- NEW or DISPOSE first parameter not pointer variable
- Long form NEW/DISPOSE/SIZEOF record tag value not found
- Long form NEW/DISPOSE/SIZEOF super array, too many bounds
- Long form NEW/DISPOSE/SIZEOF super array, not enough bounds
- NEW or SIZEOF super array without giving bounds
- ORD or WRD on a value that is not of an ordinal type
- LOWER or UPPER on an invalid value or type
- PACK or UNPACK on super array, array of files, or mis-packed
- RETYPE first parameter not a type identifier
- RESULT parameter is not a function identifier

234 Type Invalid Assumed Integer
Parameter for READ, WRITE, ENCODE, or DECODE is not of type INTEGER, WORD, REAL, BOOLEAN, enumerated, a pointer; or, for READ and WRITE, type CHAR, STRING, LSTRING; or, for READFN, type FILE.

235 **Assumed File Input**
First READFN parameter is not a file, so INPUT is assumed.

236 **Not File Assumed Text File**
The first parameter to READ or WRITE (or READLN or WRITELN) was assumed to be the file but this assumption was not correct; please give INPUT or OUTPUT explicitly to avoid this message.

237 **Assumed Input**
INPUT was not given as a program parameter.

238 **Assumed Output**
OUTPUT was not given as a program parameter.

239 **LSTRING Expected**
Target of a READSET, ENCODE, or DECODE must be an LSTRING.

240 [unassigned]

241 **Invalid Segment Variable**
Variable resides at segmented address, but a default segment address is needed. May need to make local copy of variable.

242 **File Parameter Expected Skip Statement**
READSET expects a textfile parameter.

243 **Character Set Expected**
READSET expects a SET OF CHAR parameter.

244 Unexpected Parameter Begin Skip
EOF, EOLN, and PAGE do not take more than one parameter.

245 Not Text File
EOLN, PAGE, READLN and WRITELN only apply to textfiles.

246 [unassigned]

247 Invalid Function
[not applicable]

248 Size Not Identical
Warning given in RETYPE; may or may not work a intended.

249 Procedural Type Parameter List Not Compatible
The parameter lists for formal and actual procedural parameters are not compatible; number of parameters is different, function result type or parameter type is different, or attributes wrong.

250 Reserved

251 Unexpected Parameter Begin Skip
Procedure or functional has no parameters, but left parenthesis was found.

252 Cannot Use Procedure or Function As Parameter
Intrinsic procedure or function cannot be passed as parameter.

253 Parameter Not Procedure Or Function Begin Skip
Procedural parameter expected, need procedure or function here.

254 Supertype Array Parameter Not Compatible
Actual parameter is not same or derived super type as formal.

255 Compiler

256 VAR Or CONST Parameter Types Not Compatible
Actual and formal reference parameter types must be identical.

257 Parameter List Size Wrong Begin Skip
Too few or too many parameters; only skips if too many.

258 Invalid Procedural Parameter To EXTERN
The actual procedure or function is invoked with intra-segment calls, and so cannot be passed to an external code segment. Give the PUBLIC attribute to the procedure or function to fix this.

259 Invalid Set Constant For Type
Set not constant, base types not identical, or constant too big.

260 Unknown Identifier In Expression Assumed Zero
The identifier is undefined (misspelled?) in an expression.

261 Identifier Wrong In Expression Assumed Zero
General identifier error in an expression; for example, file type ID.

262 Assumed Parameter Index Or Field Begin Skip
After error 260 or 261, anything in parenthesis or square brackets, or a dot followed by an identifier, is skipped.

263 [unassigned]

264 [unassigned]

265 Invalid Numeric Constant Assumed Zero
Decode error in an assumed INTEGER (or WORD) literal constant.

266 [unassigned]

267 Invalid Real Numeric Constant
Decode error in an assumed type REAL literal constant.

268 Cannot Begin Expression Skipped
Symbol cannot start an expression, so it has been deleted.

269 Cannot Begin Expression Assumed Zero
Symbol cannot start an expression, so zero has been inserted.

270 Constant Overflow
DIV or MOD by the constant zero (INTEGER or WORD).

271 **Word Constant Overflow**
Unary minus on a WORD operand (try NOT word + 1).

272 **Word Constant Overflow**
WORD constant minus WORD constant giving negative result.

273 [unassigned]

274 [unassigned]

275 **Invalid Range**
Lower bound of subrange is greater than upper bound (for example, 2..1).

276 **CASE Constant Expected**
Expecting a constant value for CASE statement or record variant.

277 **Value Already in Use**
In CASE statement or record variant, value has already been assigned (as in CASE 1..3: XXX; 2: YYY; END).

278 **Reserved**

279 **Label Expected**
In a BREAK, CYCLE, or GOTO statement, or starting a statement, or in a LABEL section, the expected label was not found.

280 **Invalid Integer Label**
Non-decimal notation (for example, 8#77) is not allowed in labels.

281 Label Assumed Declared
This label did not appear in the LABEL section.

282 [unassigned]

283 Expression Not Boolean Type
The expression following IF, WHILE, or UNTIL must be BOOLEAN.

284 Skip To End Of Statement
Skipping an unexpected ELSE or UNTIL clause.

285 Compiler

286 ; Ignored
A semicolon before ELSE is always an error and is skipped.

287 [unassigned]

287 [unassigned]

288 : Skipped
A colon after OTHERWISE is always an error and is skipped.

289 Variable Expected For FOR Statement Begin Skip
A variable identifier must come after FOR.

290 [unassigned]

291 FOR Variable Not Ordinal Or Static Or Declared In Procedure
The FOR statement control variable must *not* be:

- Type REAL or other non-ordinal type
- The component of an array, record, or file type
- The referent of a pointer type or address type
- In the stack or heap, unless locally declared
- Non-locally declared, unless in static memory
- A reference parameter (VAR or VARS parameter)

292 Skip To :=
In a FOR statement, the assignment is expected here.

293 Reserved

294 GOTO Considered Harmful
The $GOTO metacommand is on, and here is a GOTO.

295 [unassigned]

296 Label Not Loop Label
The BREAK or CYCLE label is not before a FOR, WHILE, or REPEAT.

297 Not In Loop
The BREAK or CYCLE statement is not in a FOR, WHILE, or REPEAT.

298 Record Expected Begin Skip
A WITH statement expects a record variable.

299 [unassigned]

300 Label Already In Use Previous Use Ignored
This label has already appeared in front of a statement.

301 Invalid Use of Procedure Or Function Parameter
A procedure parameter used as a function, or vice versa.

302 [unassigned]

303 Unknown Identifier Skip Statement
The identifier is undefined (misspelled?) starting statement.

304 Invalid Identifier Skip Statement
General identifier error starting statement; that is, file type ID.

305 Statement Not Expected
A MODULE or uninitialized IMPLEMENTATION with a main BEGIN...END.

306 Function Assignment Not Found
Somewhere in the function's body, its value must be assigned.

307 Unexpected END Skipped
END was unexpected; perhaps a missing BEGIN, CASE, or RECORD.

308 Compiler

309 Attribute Invalid
Attribute only for procedures and functions given for variable or vice versa, or invalid attribute mix like PUBLIC and EXTERN.

310 Attribute Expected
Left bracket indicated attributes, but this is not an attribute.

311 Skip To Identifier
This symbol was skipped to get to the identifier which follows.

312 Identifier Expected
List of identifiers expected, but this is not an identifier.

313 Reserved

314 Identifier Expected Skip To ;
A new identifier to be declared was expected but not found.

315 **Type Unknown Or Invalid Assumed Integer Begin Skip**
Parameter or function return type not identifier, undeclared, or value parameter or function return with file or super array.

316 **Identifier Expected**
No identifier after PROCEDURE or FUNCTION in parameter list.

317 [unassigned]

318 **Compiler**

319 **Compiler**

320 **Previous Forward Skip Parameter List**
The parameter list and function return type are not repeated when a forward (or interface) procedure or function is defined.

321 **Reserved**

322 **Reserved**

323 **Invalid Attribute In Procedure Or Function**
Nested procedure or function cannot have attribute or be EXTERN.

324 **Compiler**

325 **Already Forward**
FORWARD cannot be used twice for the same procedure or function.

326 **Identifier Expected For Procedure Or Function**
Keywords PROCEDURE or FUNCTION must be followed by an identifier.

327 **Invalid Symbol Skipped**
FORWARD or EXTERN directives are never used in interfaces.

328 **EXTERN Invalid With Attribute**
An EXTERN procedure cannot have the PUBLIC attribute.

329 **Ordinal Type Identifier Expected Integer Assumed Begin Skip**
An ordinal type identifier is expected for a record tag type.

330 **Contains File Cannot Initialize**
A file in a record variant, while allowed, is considered unsafe and is not initialized automatically with the usual NEWFQQ call.

331 **Type Identifier Expected Assume Character**
General message; this identifier is not a type identifier.

332 **Reserved**

333 **Not Supertype Assumed String**
This looks like a super array type designator but type identifier is not a super array type so STRING super array type is assumed.

334 Type Expected Integer Assumed
General message; a type clause or type identifier is expected.

335 Out Of Range 255 For LSTRING
An LSTRING designator cannot have an upper bound over 255.

336 Cannot Use Supertype Use Designator
Super array type must be reference parameter or pointer referent.

337 Supertype Designator Not Found
All upper bounds must be given in a super array designator.

338 Contains File Cannot Initialize
A super array of a file type, while allowed, is considered unsafe and is not initialized automatically with the usual NEWFQQ call.

339 Supertype Not Array Skip To ; Assumed Integer
The keyword SUPER is always followed by ARRAY in a type clause.

340 Invalid Set Range Integer Zero To 255 Assumed
The base type of a set must be within the subrange 0..255.

341 File Contains File
A file type cannot contain a file type, directly or indirectly.

342 PACKED Identifier Invalid Ignored
The PACKED keyword must be followed by one of ARRAY, RECORD, SET, or FILE; it cannot be followed by a type identifier.

343 Unexpected PACKED
The PACKED keyword only applies to structured types (above).

344 [unassigned]

345 Skip To ;
Semicolon expected at end of declaration (not at end of line).

346 Insert ;
Semicolon expected at end of declaration (at end of line).

347 Reserved

348 UNIT Procedure Or Function Invalid EXTERN
In an IMPLEMENTATION, any interface procedures and functions not implemented must be declared EXTERN at the beginning of the IMPLEMENTATION, but this EXTERN occurs later.

349 [unassigned]

350 Not Array Begin Skip
Variable in VALUE section followed by square bracket not array.

351 Not Record Begin Skip
Variable in VALUE section followed by dot is not a record type.

352 Invalid Field
In VALUE section identifier assumed to be field is not in record.

353 Constant Value Expected
In VALUE section variable can only be initialized to constant.

354 Not Assignment Operator Skip To ;
The assignment operator was not found in a VALUE section.

355 Cannot Initialize Identifier Skip To ;
Symbol in VALUE section is not variable declared at this level in fixed (STATIC) memory or has EXTERN attribute.

356 Cannot Use Value Section
Put a VALUE section in the IMPLEMENTATION, not the INTERFACE.

357 Unknown Forward Pointer Type Assumed Integer
The identifier for the referent of a reference type declared earlier in this TYPE (or VAR) section was never declared itself.

358 Pointer Type Assumed Forward
In this TYPE section, a pointer or address type occurred in which the referent type was already declared in an enclosing scope, but the identifier fo the referent type was declared again later in the same TYPE section. For example:

For example:

TYPE A=WORD; PROCEDURE B; TYPE C=^A;
A=REAL;

Message says the forward type is used in this case (for example, REAL).

359 Cannot Use Label Section
Put a LABEL section in the IMPLEMENTATION, not the INTERFACE.

360 [unassigned]

361 Constant Expression Expected Zero Assumed
In a CONST section, the expression is not constant.

362 Attribute Invalid
In a VAR section, PUBLIC with EXTERN.

363 [unassigned]

364 Contains File Initialize Module
File variables must be initialized, so when a file variable is declared in a module, the module must be called (as a parameterless procedure) to initialize these files.

365 Reserved

366 UNIT Identifier Expected Skip To ;
USES was not followed by the identifier of a unit.

367 Initialize MODULE to Initialize UNIT
A USES clause triggers a unit initialization call, but to invoke this call, the module must be called as a procedure.

368 Identifier List Too Long Extra Assumed Integer
In a USES clause with a list of identifiers, more identifiers were found in the list than are constituents of the interface.

369 End Of UNIT Identifier List Ignored
In a USES clause with a list of identifiers, fewer identifiers were found in the list than are constituents of the interface.

370 [unassigned]

371 UNIT Identifier Expected
After the phrase INTERFACE; UNIT an identifier was not found.

372 Compiler

373 Identifier In UNIT List Not Declared
One of the identifiers in the interface UNIT list was not declared in the body of the interface.

374 Program Identifier Expected
No identifier after PROGRAM or MODULE keyword (panic error).

375 **UNIT Identifier Expected**
No unit identifier after IMPLEMENTATION OF (panic error).

376 **Program Not Found**
PROGRAM, MODULE, or IMPLEMENTATION OF keywords not found (panic error). Can occur if source file is not a Pascal compiland.

377 **File End Expected Skip To End**
The assumed end of compiland was processed, but there is more.

378 **Program Not Found**
The main body of a PROGRAM or initialized IMPLEMENTATION, or final END of a MODULE or other IMPLEMENTATION, was not found.

Back End Errors

There are two kinds of errors given by the back end (optimizer and code generator): user errors and internal errors. There are very few user errors; all are concerned with limitations that cannot be detected by the front end.

A large number of internal consistency checks are done in the back end, but naturally these should always be correct and never give an internal error. Both user and internal back end errors cause an immediate stop. Both give an error number and approximate listing line number.

Back End User Errors

1. **Attempt to divide by zero.**
   For example: A DIV 0.

2. **Overflow during integer constant folding.**
   For example: MAXINT + A + MAXINT.

3. **Expression too complex/Too many internal labels.**
   Try breaking up expression with intermediate value assigns.

Back End Internal Errors

These errors have the format:

***** Internal Error NNN**

NNN is the internal error number, which ranges from 1 to 999. There is little that can be done when an internal error occurs, except report it to your authorized IBM Personal Computer dealer. Perhaps try changes to the program near the line where the error occurred.

File System Errors

Errors caught at runtime can be divided into file system errors and all other errors. File system errors will be described first. File system error codes range from 1000 to 1999. Codes from 1000 to 1099 are for operating system errors (from unit U), and 1100 to 1199 for Pascal file system errors (from unit F). Unit F errors are given below.

File system errors all have the format:

error type error in file *filename*

followed by the error code.

The *error type* field is based on the ERRS field of the file control block. The letters in parentheses show which units (U and F) can generate the error (see Appendix B). The error codes are as follows:

CODE	CONDITION
0	No error
1	Reserved
2	Reserved
3	Operation (UF)
4	Reserved
5	Reserved
6	Reserved
7	File name (UF)

CODE	CONDITION
8	Device full (U)
9	Reserved
10	File not found (U)
11	Reserved
12	Reserved
13	File not open (F)
14	Data format (F)
15	Line too long (UF)

Unit U Errors

1000 Write Error When Closing File

1001 Reserved

1002 Filename Extension With More Than 3 Characters

1003 Error During Creation Of New File

1004 Error During Opening Of Existing File

1005 Filename With More Than 8 Or Zero Characters

1006 Reserved

1007 Total Filename Length Over 12 Characters

1008 Write Error When Advancing To Next Record

1009 File Too Big (over 65535 logical sectors)

1010 Write Error When Seeking To Direct Record

1100 ASSIGN Or READFN Of Filename To Open File

1101 Reference To Buffer Variable Of Closed File

1102 Textfile READ OR WRITE Call To Closed File

1103 READ When EOF Is True (Sequential Mode)

1104 READ To REWRITE File, OR WRITE to RESET File (Sequential Mode)

1105 EOF Call To Closed File

1106 GET Call To Closed File

1107 GET Call When EOF is True (SEQUENTIAL Mode)

1108 GET Call To REWRITE File (SEQUENTIAL Mode)

1109 PUT Call To Closed File (SEQUENTIAL Mode)

1110 PUT Call To RESET File (SEQUENTIAL Mode)

1111 Line Too Long In DIRECT Textfile

1112 Decode Error In Textfile READ BOOLEAN

1113 Value Out Of Range In Textfile READ CHAR

1114 Decode Error in Textfile READ INTEGER

1115 Decode Error in Textfile READ SINT (Integer subrange)

1116 Decode Error in Textfile READ REAL

1117 LSTRING Target Not Big Enough in READSET

1118 Decode Error In Textfile READ WORD

1119 Decode Error In Textfile READ BYTE

1120 SEEK Call To Closed File

1121 SEEK Call To File Not In DIRECT Mode

1122 Encode Error (Field Width>255) In Textfile WRITE BOOLEAN

1122 Encode Error (Field Width>255) In Textfile WRITE INTEGER

1122 Encode Error (Field Width>255) In Textfile WRITE REAL

1122 Encode Error (Field Width>255) In Textfile WRITE WORD

Other Runtime Errors

Non-file system error codes range from 2000 to 2999. In some cases metacommands control whether errors are checked; in other cases they are always checked. The metacommand controlling a check, if any, is given in the list below.

2000..2049 Memory Errors

Since the stack and the heap grow toward each other, these errors are all related; for example, a stack overflow can cause a "Heap Is Invalid" error if $STACKCK is off and the stack overflows.

2000 Overflow
While calling a procedure or function, the stack (frame) ran out of memory. Checked if $STACKCK+, and in some other cases.

2001 No Room In Heap
While in the NEW (GETHQQ) procedure, not enough room was found in the heap for a new variable. Always caught.

2002 Heap Is Invalid
While in the NEW (GETHQQ) procedure, the allocation algorithm discovered the heap structure is wrong. Always caught.

2003 Reserved

2031 NIL Pointer Reference
DISPOSE or $NILCK+ found a pointer with a NIL (for example, 0) value.

2032 Uninitialized Pointer
DISPOSE or $NILCK+ found an uninitialized (value 1) pointer. Pointers will only get this value if $INITCK is on.

2033 Invalid Pointer Range
DISPOSE or $NILCK+ found a pointer that does not point into the heap or is otherwise invalid. Might have pointed to DISPOSEd block which was removed from heap and given back to system.

2034 Pointer To disposed VAR
DISPOSE or $NILCK+ found a pointer to a heap block that has been disposed. Calling DISPOSE twice for same variable is invalid.

2035 Long DISPOSE Sizes Unequal
When the long form of DISPOSE was used, the actual length of the variable did not equal the length based on the tag values given.

2050..2099 Ordinal Arithmetic

2050 No CASE Value Matches Selector
In a CASE statement without an OTHERWISE clause, none of the branch statements had a CASE constant value equal to the selector expression value. Checked if $RANGECK is on.

2051 Unsigned Divide By Zero
WORD value divided by zero; checked if $MATHCK+.

2052 Signed Divide By Zero
INTEGER value divided by zero; checked if $MATHCK+.

2053 Unsigned Math Overflow
WORD result outside 0..MAXWORD; checked if $MATHCK+.

2054 Signed Math Overflow
INTEGER result outside -MAXINT..MAXINT; checked if $MATHCK+.

2055 Unsigned Value Out Of Range
Assignment or value parameter in which the source value is out of range for the target value; target can be a subrange of WORD (including BYTE), or CHAR, or an enumerated type. Can also occur in SUCC and PRED functions, and when the length of an LSTRING is assigned. These are checked with $RANGECK+. Also occurs when an array index is out of bounds and the array has an unsigned index type; this is checked with $INDEXCK+.

2056 Signed Value Out Of Range
As above, but applies to INTEGER type and its subranges.

2100..2149 Type REAL Arithmetic

2100 REAL Divide By Zero
REAL value divided by zero; always checked.

2101 REAL Math Overflow
REAL value too large for representation; always checked.

2102 SIN Or COS Argument Range
SIN or COS function argument is too large to get a meaningful result.

2103 EXP Argument Range
EXP function in which argument is too large for result to fit in representation.

2104 SQRT Of Negative Argument
Square root function on argument < zero; always caught.

2105 LN of Non-Positive Argument
Natural log function on argument <= zero; always caught.

2106 TRUNC/ROUND Argument Range
Converting a REAL outside the range of INTEGER; always caught.

2131 Tangent Argument Too Small
TANRQQ function argument so small result invalid; always caught.

2132 Arcsin Or Arccos Of REAL > 1.0
ASNRQQ or ACSRQQ argument greater than one; always caught.

2133 Negative Real To Real Power
RSRRQQ function with first argument below zero; always caught.

2150..2199 Structured Type Errors

2150 String Too Long In COPYSTR
COPYSTR intrinsic source string is too large for target string; always caught.

2151 LSTRING Too Long In Intrinsic Procedure
Target LSTRING is too small in INSERT, DELETE, CONCAT, or COPYLST intrinsic procedure; always caught.

2180 SET Element Greater Than 255
Value in constructed set above maximum; always caught.

2181 SET Element Out Of Range
Value in set assignment or set value parameter too large for target set; checked if $RANGECK is on.

2200..2999 Other Errors

2400 Reserved

2450 Unit Version Number Mismatch
During UNIT initialization, the user (one with the USES clause) and implementation of an interface were discovered to have been compiled with unequal interface version numbers; always caught.

APPENDIX B. FILE SYSTEM INTERNALS

The File Control Block

IBM Pascal is designed to be easily interfaced with the IBM Personal Computer DOS operating system. This Appendix describes this common interface in detail.

The interface consists of a file control block (FCB) declaration, and a set of procedures and functions (named "unit U") called from the runtime to do I/O.

This interface supports three access methods, referred to as *terminal, sequential,* and *direct.*

Every file has an associated FCB (file control block). The FCB record type begins with a number of standard, operating system independent fields, the last of which is the name of the file.

Following these standard fields are operating system dependent fields. Included here are buffers, DOS control blocks, and so on, generally all operating system data for the file.

The advanced Pascal user can access FCB fields directly as explained below.

Two special FCBs are always available in Pascal, corresponding to the user console input and output. These files are usually ASCII structure TERMINAL

mode. However, there are also the Pascal predeclared files INPUT and OUTPUT which the user can reassign and generally treat like any other files. The Pascal files are called INPFQQ and OUTFQQ.

For Pascal files, the FCB ends with the buffer variable, conceptually the current file component. This means the length of a Pascal file is the length of the fixed portion plus the buffer variable length.

Pascal file variables can occur in static memory, on the stack as local variables, or in the heap as heap variables. Pascal generated code initializes FCBs when they are allocated and CLOSEs them when they are de-allocated. An FCB can be created or destroyed; however, one is never moved or copied.

The IBM Pascal compiler front end must know the length of an FCB. It reads this value during initialization from a special file called PASKEY, which resides on the PAS1 diskette. Every reference to a Pascal file buffer generates a call to a unit U routine.

As mentioned, unit U refers to the operating system interface routines. The IBM Pascal specific file routines are called unit F.

Generated code has calls to unit F, which in turn calls unit U routines.

The IBM Pascal system uses a naming convention for IBM Personal Computer Linker names. The compiler and run time system use the same conventions.

All linker globals are six alphabetic characters, ending with QQ (this avoids conflicts with user program global names).

The fourth letter indicates a general class, so xxxFQQ is always part of the generic Pascal file unit, and xxxUQQ is part of the operating system interface unit.

File system error conditions may be detected at the lower unit U level, detected at the higher unit F level, or undetected.

When a unit U routine detects an error, it sets a flag in the FCB and returns with a non-zero result.

When unit F detects an error, or discovers unit U has detected one, either an immediate runtime error message and program termination occur, or if the user has set error trapping by setting the TRAP flag (in the FCB), unit F just returns to the calling program.

Unit F will not pass a unit U routine a file with an error condition.

This condition (also called an "error not caught") has undefined results. Runtime errors which cause a program termination, use the standard IBM Pascal error handling system, which gives the context of the error

and provides entry to the debugging system which can be turned on with the $DEBUG metacommand.

File Structures and Modes

Files come in two basic structures, and three modes. The structures are ASCII (Pascal TEXT type with CHAR components), and BINARY (Pascal FILE OF some type).

The TXTF field in the FCB is true for ASCII files, and false for BINARY files. In theory, an ASCII file can contain any 8 byte character, but in practice, IBM Pascal preempts two character values, CHR(13) for end-of-line and CHR(26) for end-of-file.

The modes supported are SEQUENTIAL, TERMINAL, and DIRECT (these values are of Pascal type FILEMODES).

TERMINAL mode is for display/keyboard and printers; SEQUENTIAL mode means a diskette file with variable length records accessed serially, and DIRECT mode means a disk file with fixed length records accessed by logical record numbers.

SEQUENTIAL mode is the default, and TERMINAL mode is set automatically if a file is opened to the display/keyboard or a printer device.

The Pascal user can set DIRECT or any other mode by setting the file's MODE field (for example 'filevariable.MODE := SEQUENTIAL').

The six kinds of files implied by two structures and three modes are often grouped as follows:

- ASCII structure SEQUENTIAL mode
- ASCII structure TERMINAL mode
- ASCII structure DIRECT mode
- BINARY structure SEQUENTIAL mode
- BINARY structure TERMINAL mode
- BINARY structure DIRECT mode

Files with mode TERMINAL usually have ASCII structure, but BINARY structure is also permitted.

When reading in TERMINAL mode with ASCII structure, the user types an entire line before it is processed, allowing the usual line editing features (such as backspace or cancel) before the Enter key is pressed.

When reading in TERMINAL mode with BINARY structure, keystrokes are returned directly, without waiting for a carriage return or echo.

This TERMINAL/BINARY combination is very useful for non-echoed input (since the user is responsible for any echoing) and one character responses to prompts (see GET versus READ).

Files with SEQUENTIAL or TERMINAL mode and ASCII structure contain records (called lines in Pascal) with a variable number of characters from zero (empty) upward.

There are no particular limits on the set of 8 bit bytes allowed as characters, except DOS uses two characters to terminate records making these byte values impossible to read (CHR(13) for end-of-line and CHR(26) for end-of-file). ASCII record and file boundaries are called "hard," that is, boundary information is always present in the file in some form.

Files with SEQUENTIAL or TERMINAL mode and BINARY structure contain records with fixed length in Pascal (that is, the file's component type size). The boundaries between BINARY SEQUENTIAL records are detectable (hard), and the length of a record read must be the same as the length of the same record written or an error occurs.

In Pascal, once a SEQUENTIAL or TERMINAL file is opened for reading (RESET) or writing (REWRITE), only read and write operations (respectively) are permitted, and an EOF is written when a file opened for writing is CLOSEd.

Files with mode DIRECT contain records with a fixed length, so generally there is no difference between an ASCII record and a BINARY record and any byte value can be used as data.

The Pascal user sets the record size of a BINARY DIRECT file implicitly as the length of one file component. The record size of an ASCII DIRECT file is set with the special type declaration "TEXT(nnn)."

The Pascal user sets the record number with SEEK (file,n). If no record number is given, the next higher record number is used.

IBM Pascal automatically extends a DIRECT file when a record is written with a new highest record number.

It is an error to read a record that has never been written, but this error is not caught if the record number is below the maximum allocated. Reading a record with a number above the maximum is an end of file condition.

It is an error to write a DIRECT file with one size and read it with another.

Reading or writing past the declared size of a record is an error caught; reading or writing less than a full record is permitted, and, if writing, unit F will blank pad for ASCII records and affect the rest of the record in an undefined way for BINARY records.

Special Features

The various I/O related operations in a Pascal program are compiled into calls to routines in the generic file unit, referred to as unit F.

Although not recommended, unit F or unit U routines can be called directly in a Pascal program, by declaring them EXTERN.

In Pascal, a variable with any FILE type is identical to an FCB record, and in fact any FILE variable can be treated as having the record type FCBFQQ.

For example, if FREAL is declared type FILE OF REAL, in addition to accessing the current buffer variable as FREAL^ the file error trapping flag can be accessed as FREAL.TRAP. An actual parameter of any FILE type or the type FCBFQQ can be passed to a formal VAR parameter of the identical FILE type or the identical FCBFQQ type.

A partial version of the record type FCBFQQ is pre-declared, but the full version can be re-declared in a user program (or within a procedure, etc.).

However, as with all IBM Pascal VAR or CONST parameters, if both the formal and actual parameters have the type identifier FCBFQQ these must be the same type.

Generally Pascal users who redeclare the type FCBFQQ will $INCLUDE the INTERFACE and a USES clause for unit FILKQQ which contains the standard declaration code.

This technique must be used with caution (see Chapter 12, "File System"). The includable interface file (FILKQQ.INC) is resident on the PAS1 diskette for the user's convenience.

The FCB fields available to the Pascal user with the default FCBFQQ type are the error trapping flag TRAP, the error status ERRS, and the filemode MODE.

TRAP can be set true to enable error trapping; if true when an error is detected, no further unit U calls (except CLSUQQ and CLDUQQ) using the FCB are executed, unless the user first clears ERRS.

If TRAP is false a file error invokes the usual runtime error termination. ERRS is a standard user-oriented error code from 0 to 15 (see "Error Handling" in this chapter).

The file is defined to be in an error condition if ERRS is non-zero. Although the Pascal user can set ERRS to zero to re-enable unit U calls using the file, the status of a file in error is undefined.

Clearing ERRS will allow recovery from format errors in TERMINAL file input, the most important case. (See "Error Handling" in this chapter.) MODE can be set to the filemode wanted.

When RESET or REWRITE are called, MODE is copied to CMOD, OPNUQQ is called to open the file (which may change CMOD), then CMOD is copied back to MODE. Changing the MODE field of an open file has no effect.

IBM Pascal also supports temporary files, in that a file assigned the name CHR (0) is given a unique temporary file name with a call from REWRITE (REWFQQ) to the special routine TFNUQQ.

A temporary file is deleted when it is closed. The FCB field TMPF is true for a temporary file, false for all other files.

Of course, a temporary file can be created by REWRITE, written, then RESET and read, but any implicit or explicit CLOSE (including program termination) will delete it.

The IBM Pascal system uses a method called *lazy evaluation* to facilitate interactive terminal input. This means input data is not evaluated until it is needed.

Unit F applies lazy evaluation to all ASCII files, transparent to unit U (it does no harm with non-TERMINAL ASCII files).

It uses the FCB field FULL, as follows:

- GET (really GETFQQ) advances to the next character only if FULL is false, and always returns FULL false.

- BUFFQQ (see below) advances to the next character only if FULL is false, and always returns FULL true.

- Unit F calls BUFFQQ to make sure FULL is true.

In addition, the generated code has a call to BUFFQQ before every reference to a textfile buffer variable.

Pascal also contains the concept of a "program parameter;" these look like parameters to the main program in a Pascal source file.

Program parameter values are received from the operating system with a call to PPMUQQ, described below. Its corresponding unit U routine is EOFUQQ.

Error Handling

Errors may be detected in either the Pascal runtime (unit F), or operating system interface routines (unit U).

However, an error condition is never processed by unit U; these routines just pass an error code back to unit F.

The Pascal user may indicate that I/O errors should be trapped by setting FILE.TRAP:=TRUE for a file.

When an error occurs, if error trapping is on additional I/O to a file stops and the user program is expected to handle (or ignore) it. If error trapping is off, unit F will end the run by calling the standard error handler ABORT (EMSEQQ).

Within unit F, if ERRS is or becomes non-zero, no unit U routines involving the file except CLSUQQ, CLDUQQ, and ENDUQQ are called, although the buffer variable and/or user input variables may be changed in undefined ways.

Therefore, on entry to any unit U routine (except those mentioned), there should never be any error conditions present in the FCB.

If an error occurs in a unit U routine, it sets ERRS to an error status code (1..15, below), and ESTS to any WORD value (such as the operating system return code).

The unit F routines are responsible for producing an error message if error trapping is off for the file, or not invoking any unit U routines again (except those mentioned above) if error trapping is on and ERRS remains non-zero. Unit U routines should never look at ERRS or ESTS.

The run time error message is produced by calling EMSEQQ, passing ERRC, ESTS, and a message containing the filename and a short description based on ERRS.

The error code ERRS is available as a general operating system independent code (see Appendix A).

FCB Declarations in Detail

FNLUQQ=21–CONST length of a DOS filename

SCTRLNTH=512–CONST length of disk sector

DOSEXT–Type

This is the extended portion of the DOS FCB. Not used by Pascal but provided for the advanced machine language user.

DOSFCB–Type

The FCB used by DOS to do actual I/O.

DEVICETYPE–Type

The type of device that the file resides on. LDEVICE is the RS232 port.

FILEMODES–TYPE Enumerated

Used to describe the mode of a file. New modes can be added on the end. Current constant values are SEQUENTIAL, TERMINAL, and DIRECT with the ordinal values 0, 1, and 2 respectively.

FCBFQQ–TYPE file control block; fields:

TRAP–error trap flag. Set true by the user to enable error trapping, or false (default) to cause normal error termination. See "Error Handling" discussed previously.

ERRS–Pascal error status. If zero, no error; values from 1 to 15 indicate the general class of error. Can be set to zero to re-enable I/O on a file. See "Error Handling" discussed previously.

MODE–Pascal user filemode. Set by the user to indicate a filemode wanted before calling RESET or REWRITE; set by unit F to indicate the mode actually obtained. To avoid mistakes by casual users, MODE does not contain the actual filemode value; it is in CMOD.

MISC–unused (reserved).

ERRC–generic error code. This error code is part of the standard error code range used generally by Pascal. Invalid unless ERRS is non-zero; see "Error Handling," discussed previously. Write-only in units F and U; given in runtime error message.

For I/O errors the code ranges from 1000 to 1999. (See Appendix D, "Runtime Structure" for more information.)

- 1000..1099: unit U errors
- 1100..1199: unit F errors
- 1200..1999: unused

ESTS–unused (reserved).

CMOD–current file mode. Actual mode of the file, used by all units; value is SEQUENTIAL (0), TERMINAL (1), or DIRECT (2). Must be set before OPNUQQ call, which may change it from SEQUENTIAL to TERMINAL. Does not change as long as file is open.

Set indirectly by Pascal user by setting MODE before RESET or REWRITE.

TXTF–current structure. Actual structure of the file, used by all units; true if ASCII, false if BINARY. Must be set before OPNUQQ or NEWUQQ. Does not change as long as file is open. Set indirectly by type of file, TEXT or FILE OF type.

Pascal generated code calls NEWFQQ when a file is allocated passing TXTF (and SIZE) as parameters.

SIZE–record size. For DIRECT mode, the size of a record in bytes, excluding any DOS overhead (such as record terminator). Must be set before OPNUQQ or NEWUQQ. Does not change as long as file is open.

Known at compile time from component length (BINARY) or TEXT(nnn) type (ASCII); passed (along with TXTF) in a generated call to NEWFQQ when the file is open.

Also applies to SEQUENTIAL and TERMINAL modes, as the component size of a BINARY file.

MISB–unused (reserved).

OLDF–existing file flag. True if file must already exist when OPNUQQ called; false if a new file can be created. For Pascal, set true for RESET, false for REWRITE.

INPT–input/output flag. True if reading, false if writing. Set true by RESET and false by REWRITE. For Pascal files with SEQUENTIAL or TERMINAL modes, is only changed when file is closed. Files with DIRECT mode can change the INPT value at any time. Always true for GETUQQ call; always false for PUTUQQ, PERUQQ, and PCCUQQ calls.

Used by OPNUQQ to open file initially for reading or writing; used by IOCUQQ to change from writing to reading or vice versa; used by FBRUQQ and CLSUQQ to write end-of-file if was writing. Maintained by unit F.

RECL, RECH–DIRECT record number. Value from 1 to $(2^{16})-1$; high order bit of RECH always zero. Set by Pascal user with SEEK. Unit F will call SEKUQQ in these cases. If not set explicitly, automatically incremented by GET or PUT after a call to PERUQQ (in which case SEKUQQ is not called). Undefined value unless filemode is DIRECT. Changed only by Unit F and SEKUQQ in Unit U.

USED–current record byte counter. Number of bytes read or written in current record; applies to all filemodes. Can range from zero to SIZE.

Can be used by unit U as offset to next byte to read or write. If zero, indicates a new DIRECT file record number. Changed only by unit F.

Because Pascal BINARY records are always of length SIZE and read or written with one call to GET or PUT, USED is set to zero before the GET/PUT call and set to SIZE afterwards. If USED is zero, EORF should be true, and vice versa.

LINK–next FCB in list. Used by unit F to maintain a list of open files. Head of list is global variable HDRFQQ. If zero, last FCB in list. Ignored by unit U.

BADR–ADRMEM. Address of buffer variable at the end of the FCB.

TMPF–Pascal temporary file. Set indirectly by the Pascal user by ASSIGN (file,CHR(0)); only set true during REWRITE, in which case TFNUQQ is called to set a temporary filename.

Causes the file to be deleted (by calling CLDUQQ) when the user explicitly CLOSEs it or at program termination; file is not deleted within a later RESET or REWRITE.

FULL–buffer status. Lazy evaluation status for textfile read; true if evaluated, false if evaluation deferred. BUFFQQ always returns with FULL true; if it was false, unit F goes to the next character. GETFQQ always returns with FULL false; if it was already false, unit F goes to the next character.

MISA–unused (reserved).

OPEN–open flag. If true, file is open. Maintained by unit F only; ignored by unit U. Set true after calling OPNUQQ and false after calling CLSUQQ/CLDUQQ.

FUNT–unused.

ENDF–unused.

REDY–unused (reserved).

BCNT–byte count from GET. Number of bytes in current record copied to the target address by GETUQQ. Ranges from zero to the target length requested.

GETUQQ may copy additional bytes beyond the current record to a maximum of the target length requested, but these are not counted in BCNT and will be returned by a later GETUQQ call.

Unit F adds BCNT to USED after every GETUQQ call. No defined value if writing file. Could be used by unit U in this case.

EORF–end of record flag. Only applies to ASCII SEQUENTIAL/TERMINAL files. If true, file is at record boundary, either start of file or end of record.

Set true by GETUQQ if end of record encountered (in which case BCNT is less than length wanted), else set false by GETUQQ.

Also set true by OPNUQQ, PERUQQ, FBRUQQ, and IOCUQQ; and set false by PUTUQQ and PCCUQQ. Also used by unit F for DIRECT and/or BINARY files; must not be used by unit U in this case. If EORF is true, USED should be zero, and vice versa.

EOFF–end of file flag. Set true by GETUQQ if end of file read. Causes GETUQQ to do nothing if true on entry. Set false by OPNUQQ when opening file to read; if file is empty, first GETUQQ will set it true. Set true by OPNUQQ call if opening for write and remains true as long as file being written.

Set true by IOCUQQ if INPT false (switch read-to-write), but not changed if switching write-to-read. Set false by FBRUQQ if INPT is true. Set true by TFDUQQ.

DOS Specific Fields

This area is designed to contain the operating system's file data: buffers, counters, flags, and so on, as well as any variables needed by unit U.

It starts on a word boundary and should end on a word boundary. During the time a file is opened, an FCB is never moved or copied, so the operating system can use addresses of these fields safely.

BUFF—Pascal File Buffer

Current file buffer variable in Pascal; can be accessed by the Pascal user as *filevar*. The length is always SIZE bytes.

Including the FCB Declaration

The following is an example of how the FCB declaration can be implemented:

```
{&INCLUDE: 'A:filkqq.inc'}
PROGRAM EXAMPLE(OUTPUT);
    USES FILKQQ;
  VAR F:TEXT;
  BEGIN

  ASSIGN(F,'FILE.DAT');
  REWRITE(F);
  IF F.TXTF<>TRUE THEN
    WRITE('Something is wrong');
END.
```

DOS Interface Routines

Interface routines in unit U can be divided into several groups.

First are the overall initialization and termination routines, INIUQQ and ENDUQQ.

Next described are the file initialization and termination routines, OPNUQQ, CLSUQQ, and CLDUQQ.

Then come the data transfer and position routines, GETUQQ, PUTUQQ, PERUQQ, PCCUQQ, and SEKUQQ.

Last in the group are the terminal access routines, PTYUQQ, GTYUQQ, and PLYUQQ; and the filename routines, PFNUQQ and GFNUQQ. Pascal also uses the routines BUFUQQ, NEWUQQ, TFNUQQ, and PPMUQQ.

The declaration for these interface routines can be included in the same fashion as the FCB declaration described above. They are contained in a file on the PAS1 diskette called FILUQQ.INC.

INIUQQ: Initialization

The entry point for any Pascal load module is BEGXQQ, in the module ENTX. The ENTX module source is on the PAS1 diskette for the user's study. It gives various entry points, data areas, and a general memory layout, and is in the file ENTX6S.ASM. BEGXQQ initializes some runtime static variables, initializes the heap, calls INIUQQ to initialize unit U, calls BEGOQQ (the user initial escape routine) and calls the main Pascal program, which has the entry point ENTGQQ. A Pascal main routine calls INIFQQ to initialize unit F.

INIUQQ must also set the filename of the user console input device in INPUQQ and the filename of the user console output device in OUTUQQ. These are eight character strings, and will be used in a PFNUQQ call with any trailing blanks truncated.

FNSUQQ must be initialized to the set of characters allowed in a filename, to be used when the user reads a filename via READFN or as a program parameter.

FNSUQQ is represented as a 32 byte (256 bit) area, with each bit on if the corresponding character is allowed (the high order bit of the first byte corresponds to CHR(0)). These three variables may be set at load time with a VALUE section or the equivalent instead of within INIUQQ at run time.

After calling INIUQQ, INIFQQ will open the user console input and output FCBs, which are statically allocated variables named INPFQQ and OUTFQQ.

ENDUQQ: Termination

When a Pascal program terminates (either normally or from EMSEQQ due to an error), it calls ENDXQQ, which calls ENDYQQ to close all open files and delete any temporary files, using the linked list of FCBs (including the console files). ENDYQQ is conceptually part of unit F. ENDYQQ then calls ENDOQQ (the user escape termination routine), and finally ENDUQQ.

ENDUQQ might do a file system reset, or whatever is required by DOS. ENDUQQ may or may not return; that is, it can be a normal procedure and return, or it can itself exit to the operating system.

Note that all files are closed when ENDUQQ is called.

OPNUQQ: File Open

OPNUQQ does the actual DOS file open. CMOD contains the intended filemode: SEQUENTIAL, TERMINAL, or DIRECT.

If the file or device opened is really a terminal or printer but CMOD is not TERMINAL, OPNUQQ should

change it to TERMINAL if it was SEQUENTIAL, or giv
an error if it was DIRECT.

The Pascal user sets the MODE field before calling RESET or REWRITE, which is copied to CMOD before calling OPNUQQ and copied back again afterwards. Note that unit U routines always use CMOD, never MODE.

TXTF is true for an ASCII file, and false for a BINARY file; it is probably of no use to OPNUQQ (users sometimes want to write a file with one structure and later read it with another). SIZE contains the length of a DIRECT file record, and for Pascal the length of a buffer variable.

When OPNUQQ opens a DIRECT file, if the record size of the opened file is not equal to SIZE this is an error condition.

OPNUQQ must examine the OLDF field to determine whether the file should be present.

If OLDF is true and the file is not found, this is an error (some unit U's may always create the file and not catch this error).

If OLDF is false the file will be created. This could mean creating a file that did not exist before or overwriting an existing file. OLDF is set true for any RESET, false for any REWRITE.

INPT is true for an open to read-only (RESET mode SEQUENTIAL or TERMINAL), and false for an open to write or read-and-write (other calls; REWRITE mode SEQUENTIAL or TERMINAL is write-only).

The user sets the filename using ASSIGN, READFN, or for a file program parameter as part of program initialization (another form of READFN); it must be set before calling RESET or REWRITE.

The Pascal user can assign a "name" of CHR(0) to indicate a temporary file, in which case REWRITE will set TMPF true and call TFNUQQ to get a temporary filename.

If an error occurs during OPNUQQ, the file is assumed to be closed, so if an error is discovered after the file has been opened it is up to OPNUQQ to close it.

OPNUQQ must initialize some fields used by GET and the PUTs. It sets BCNT to zero and EORF true indicating a record boundary.

If writing (INPT false), set EOFF true; if reading, set EOFF false.

CLSUQQ, CLDUQQ: File Close

To close or delete an open file, the user can call CLOSE or DISCARD. Files allocated on the stack or the heap

are automatically CLOSEd when the file variable is de-allocated (and deleted if TMPF is true).

When a program terminates, all open files are automatically closed by ENDYQQ. CLSUQQ closes an open file, and CLDUQQ closes and deletes an open file.

If an error occurs, the file is assumed to be closed. For CLSUQQ, if INPT is false and CMOD is SEQUENTIAL the user was writing a sequential file and an end-of-file record must be generated.

GETUQQ: Read Bytes

GETUQQ reads bytes from a file; on entry INPT is true. Some of its actions are common to all files, and some depend on the mode and structure. It is passed the address and length of a buffer into which bytes are copied. The address is segment-plus-offset. This length is called the "buffer length" or "length requested" below. If EOFF is true on entry, or the length requested is zero, just return (these are not unit U error conditions).

If all the file's bytes have been transferred, set EOFF true and return (but see additional notes below on setting EOFF). A newly opened empty file has EOFF false after OPNUQQ; the first GETUQQ call must set it true (since there are no records in the file GETUQQ does not need to return once with EORF set).

Otherwise bytes are transferred from the file to the buffer until the requested length is transferred, an error occurs, or the end of record or end of file is encountered. The number of bytes transferred must be set in BCNT, which can range from zero to the buffer length.

Putting meaningless bytes into the buffer at positions from BCNT+1 to the buffer length is allowed; changing bytes past the buffer length is not allowed.

For SEQUENTIAL and DIRECT files, return with the bytes actually transferred.

For ASCII structure SEQUENTIAL and TERMINAL files, every record (including the last one) must have a "hard" record boundary, described here as an end of record "mark." EORF is set true by GETUQQ for ASCII structure SEQUENTIAL and TERMINAL mode files if the end of record "mark" is read and false if it is not.

If the last byte of the record is the last byte of the length requested, EORF is set false and BCNT is the buffer length; on the next GETUQQ call EORF is set true and BCNT is set to zero. If a record is empty BCNT is set to zero and EORF to true.

If the last byte of the file is transferred, EORF must be returned true before EOFF is returned true, so if this

last byte fills the buffer GETUQQ will return once with BCNT equal to the buffer length and both EORF and EOFF false; then once with BCNT zero, EORF true, and EOFF false; and finally once with BCNT arbitrary, EORF unchanged, and EOFF true.

Note that if GETUQQ returns with BCNT less than the length requested, EORF must be true; transferring less than the length requested when additional bytes remain in the record (such as up to a physical record boundary) is not allowed.

Note that if the returned BCNT is less than the length requested then GETUQQ has either reached a "hard" record boundary or the end of the file.

In the end of file case, if any bytes are returned EOFF must be returned false; the next GETUQQ call returns with EOFF true.

The USED field contains the number of bytes transferred so far in the current record. GETUQQ's callers always set USED to zero if they are starting a new record and add BCNT to USED if they are continuing a read of the current record.

Therefore USED is available as a byte offset from the start of the current record to the next byte to transfer as well as a count of bytes already transferred.

If CMOD is DIRECT, RECH/RECL contain the current record number. If USED is zero and EORF is true, this record number has been incremented, or, if SEKUQQ was called, it has been changed. If USED is not zero and EORF is false, these fields have not changed. If USED is zero and EORF is not true, or USED is non-zero and EORF is true, there is an inconsistency error.

The SIZE field is used by unit F such that no more than SIZE data bytes will ever be requested from a record; for example, USED plus the buffer length will never be greater than SIZE. This is why GETUQQ never sets EORF true for DIRECT mode reads. Note that unit U never changes RECH/RECL, SIZE, or USED.

TERMINAL input always implies reading from a user keyboard, but actions depend on the setting of TXTF. If TXTF is true, GETUQQ does a "high level" read from the terminal.

DOS standard intra-line editing applies. To properly support intra-line editing, a call to GETUQQ should read an entire user line into an internal buffer, waiting for the Enter key or the equivalent.

However, if line buffering is too difficult for unit U, bytes can be transferred directly into the caller's buffer, using BCNT as the current index for backspace and other edits. If the buffer length is reached before Enter is pressed, GETUQQ returns with EORF false. GETUQQ calls with a buffer length of one are common.

For TERMINAL files with BINARY structure, GETUQQ does a "low level" read a character at a time from the keyboard, if possible without echo or intra-line editing or waiting for an entire line.

This raw input mode permits the program to interpret keystrokes as commands or do its own special editing features. Usually in this case the requested length will be one.

PUTUQQ: Write Bytes

PUTUQQ writes bytes to a file; on entry INPT is false. It is passed the address and length of a buffer from which bytes are copied. The address is segmented; the length may be zero. The entire length of the buffer is written, unless an error occurs. The bytes are always written to the current record.

On entry the USED field contains a count of bytes already written to the record; as in GETUQQ this can be used as an offset within the record to the next byte position to write.

EORF must be set false for SEQUENTIAL and TERMINAL files with ASCII structure, and must not be changed in other cases. BCNT and EOFF are not needed by PUTUQQ. EOFF will be true. BCNT is not used by unit F when writing so it could be used internally by unit U.

If CMOD is DIRECT, RECH/RECL contain the current record number. If USED is zero and EORF is true on entry, position the file to this record before writing data if necessary; the record number has either been incremented or SEKUQQ was called to change it. If USED is not zero and EORF is false, the record number has not changed.

The SIZE field is used by unit F such that no more than SIZE data bytes will ever be written to a record; for example, USED plus the buffer length will never be greater than SIZE. Note that unit U never changes RECH/RECL, SIZE, or USED.

Except for DIRECT record positioning, PUTUQQ operates the same way with all file modes and structures.

PERUQQ: End Record

PERUQQ writes an end of record "mark" to a file; on entry INPT is false. Since SEQUENTIAL and TERMINAL mode ASCII structure files have "hard" record boundaries, PERUQQ must mark the end of the record for SEQUENTIAL files and TERMINAL files.

Note that for TERMINAL files, PCCUQQ will always be called next. In this case PERUQQ can send a carriage return, and PCCUQQ can send 0, 1, or 2 line feeds to the terminal or printer.

However, in some cases PERUQQ could send carriage return instead (see PCCUQQ below). DIRECT mode and BINARY structure files may have "hard" or "soft" record boundaries, so PERUQQ may mark the end of the record or do nothing.

EORF must be set true for SEQUENTIAL and TERMINAL files with ASCII structure, but must not be changed in files with DIRECT mode or BINARY structure.

USED always contains the number of bytes in the record. Empty record (USED zero) can occur, except with ASCII DIRECT files.

For ASCII DIRECT mode, USED will always equal SIZE since unit F has padded the record with blanks, but for BINARY DIRECT mode, USED may be from zero to SIZE (if less than SIZE, the content of the rest of the record is undefined).

RECH/RECL still contains the DIRECT file record number. PERUQQ will always be called after one or more PUTUQQ calls for all file modes and structures to end every record, including before the file is closed with CLSUQQ.

PCCUQQ: Put Carriage Control

PCCUQQ corresponds to the PAGE procedure. On entry, INPT is false. It may be called for any file mode or structure, but must ignore all of them except for ASCII TERMINAL files. Also it can only occur at the start of a record, so USED is zero and EORF is true.

Unit F calls PERUQQ and then PCCUQQ at the end of a line.

The one character argument is ' ' for single space and '1' for a new page. The effect of other character values is undefined (generally do a PUTUQQ of the character). Unit F uses '1' for the PAGE command, and ' ' otherwise.

Pascal PAGE calls to non-TERMINAL files just write a form feed to file with a PUTUQQ call.

SEKUQQ: DIRECT File Position

SEKUQQ is passed a new DIRECT file record number (RECH/RECL pair). The file is at a record boundary, so EORF is true. SEKUQQ is only called when the user changes the record number explicitly with SEEK.

Normally, after a PERUQQ call to a DIRECT file, the FCBs record number is automatically incremented to the next record number, and in this case SEKUQQ is not called. SEKUQQ can either actually do the positioning, or internally set a flag so that the next GETUQQ or PUTUQQ call does the positioning.

Unit F sets the record number RECH/RECL to 1 when the file is opened, before the OPNUQQ call.

PTYUQQ: Console Output

Direct user console output routine; does not use any FCB. PTYUQQ is passed the length and segment-plus-offset of a string of characters to write to the console. If the length is zero, ends the line on the console (that is, writes carriage return and line feed).

There is no error return, so if any error occurs it is up to unit U to handle it somehow (PTYUQQ is called by the error message output routine EMSEQQ).

GTYUQQ: Console Input

Direct user console input routine; does not use any FCB. GTYUQQ is passed the length and segment-plus-offset of a buffer for an input string of characters; it returns the number of characters actually read, up to a maximum of the buffer length. The buffer length may be zero. As in GETUQQ, the contents of the buffer after the characters read up to the maximum length is undefined and can be changed. It always reads an entire user input line; any characters typed beyond the end of the buffer length should be ignored.

As with PTYUQQ, there is no error return, so any errors must be handled in unit U somehow.

PLYUQQ: End of Line

This function is very similar to PERUQQ but it writes an end of line to the user's display. It is used in conjunction with PTYUQQ and GTYUQQ.

PFNUQQ: Set the Filename

Passed the location and length of a filename to be assigned to the file. The file is guaranteed closed by unit F. This will be the filename to use for all future OPNUQQ calls; note that the file may be closed and re-opened with this same filename, or the filename may be reset with a PFNUQQ (or TFNUQQ) call between the CLSUQQ and the OPNUQQ.

There is no error return from PFNUQQ; if an error can be detected, set the ERRS field and return non-zero from the upcommand OPNUQQ call.

GFNUQQ: Get the Filename

Function passed the location and length of a buffer to receive the filename; returns the length of the filename (up to the maximum of the length of the buffer). GFNUQQ is only called for error messages, so returning zero (that is, not returning a filename) is permitted if it is difficult to obtain the filename.

NEWUQQ: New FCB

When a file variable is allocated, NEWFQQ is called, which sets the initial standard FCB fields and calls NEWUQQ. SIZE and TXTF are set (and will not change), but other FCB field values are undefined. NEWUQQ will not often be needed; most initialization work should be done in OPNUQQ.

A call to NEWFQQ is generated by the compiler at the start of the program for statically allocated files (after the INIFQQ/INIUQQ call but before program parameters assigning file names are read), at the start of a procedure or function for local stack file variables, and after a NEW for heap file variables.

TFNUQQ: Temp Filename

Create a new temporary filename, and assign it to the FCB like a PFNUQQ call. If no unique name can be created, set ERRS to 7.

PPMUQQ: Program Parameter

Used to get program parameter values. Expected to fill an LSTRING with the parameter value. Parameter values are just text; does not decode integers, check filename syntax, etc.

If the parameter string is longer than expected, return a non-zero error code and unit F will reprompt directly to the display.

Called once per parameter requested, in order, which means the unit U may need a parameter counter variable (initialized in INIUQQ). Can always just return a non-zero value to make unit F write the prompt and read the response itself. This is very useful for reading a command line which might contain an LSTRING of length 0.

> **Note:** The first two parameters of this function are unused and reserved.

Including the Unit U Declaration

```
{$INCLUDE:'A:FILKQQ.INC'}
{$INCLUDE:'A:FILUQQ.INC'}

PROGRAM EXAMPLE:
    USES FILUQQ.INC;
  VAR L:LSTRING(255);
    ERROR:WORD;

  BEGIN
    ERROR:=PPMUQQ(0, ADR NULL, L);
    WRITELN('This is on the command line:',L)
  END.
```

APPENDIX C. COMPILER STRUCTURE

Overview

The IBM Pascal compiler is divided into the "front end" and the "back end." The front end reads the Pascal source program, and writes the listing file, symbol table file, and intermediate code file. The back end is divided into three parts: the optimizer, code generator, and link text emitter.

The *optimizer* reads the symbol table and intermediate code files, and optimizes the intermediate code.

The *code generator* transforms the intermediate code to target machine code, and writes this code to the intermediate binary file.

The *link text emitter* reads the intermediate binary file and writes the final object file for later input to a linker.

The back end also writes a listing of the generated code; this listing is sometimes written by a "disassembler" module called by the code generator and sometimes by the link text emitter.

Both the front end and the back end are written in Pascal, in a source format that can be transformed into either relatively standard Pascal or System level IBM Pascal. All three intermediate files are Pascal record types.

The front and back ends include a common constant and type definition file called PASCOM, which defines the intermediate code and symbol table types. The back ends use a similar file for the intermediate binary file definition.

The symbol table record is relatively complex, with a variant for every kind of identifier (assorted data types, variables, procedures and functions, and so on). The intermediate code (or *Icode*) record contains an Icode number, opcode, and up to four arguments; an argument can be the Icode number of another Icode to represent expressions in tree form, or something else (such as a symbol table reference, constant, or length).

The intermediate binary code record contains several variants for absolute code or data bytes, public or external references, label references and definitions, and so on.

The Front End

The front end can be divided into the scanner; low level utilities; intermediate level utilities for identifiers, symbols, Icodes, memory allocation, and type compatibility; and finally three high level sections for processing expressions, statements, and declarations.

The front end is driven by recursive descent syntax analysis, using a set of procedures such as EXPRESS (for expressions), STATEMT (for statements), TYPEDEC (for type declarations), and so on.

The front end maintains one lookahead symbol; while not absolutely necessary for parsing correct programs, it is very useful in error recovery. Syntax errors are processed by a procedure which forces the current symbol to one of a set of symbols legal at a given point.

If the current symbol is wrong but the following one is correct, the current symbol is deleted; if not, the correct symbol is inserted. However, common substitution mistakes, such as confusing "=" and ":=", merely give a warning message.

The scanner is relatively large, since it must process the compiler directives (the metalanguage) and produce a listing with error messages, data about variables, and other information for the user.

Intermediate code is written to the Icode file as soon as it is generated; there is no reason to keep it around. The symbol table is built as a binary tree for identifiers with pointers to semantic records; at the end of every block all new semantic records are written to the symbol table file.

As soon as an error is detected, all writing to intermediate files stops, since the code may not be acceptable to the back end (detecting a warning does not invalidate the intermediate files).

The front end reads the file PASKEY (on the PAS1 diskette) to initialize the predeclared identifiers, like INTEGER, READ, MAXINT, and SIN. PASKEY can be divided into four sections:

- The first contains special information, in particular the number of bytes in a file control block and primitive type identifiers.
- The second lists all the intrinsic procedure and function identifiers (those that are transformed by the front end in special ways).
- The third section contains constants, types, and external procedures and functions using normal IBM Pascal syntax.
- The fourth contains one or more INTERFACE and USES clauses for predeclared procedures and functions.

The optimizer reads the interpass files in a different order: first the symbol table for a block is read, then the intermediate code for the block.

Optimization is performed on the basis of a "basic block," that is, either a block of intermediate code up to the first internal or user label, or up to a fixed maximum number of Icodes, whichever comes first. Within this block the statements and expressions can be reordered and condensed, as long as the intent of the programmer is unchanged. For example:

```
A [J, K] := A [J, K] + 1;
  (* J := J - 1; *)
IF A [J, K] = MAX THEN PUNT;
```

The array address **A [J, K]** need be calculated only once. However, if the commented-out assignment to J is in the program, the array address in the IF statement must be partially re-calculated. This optimization is called common sub-expression elimination. The optimizer will also re-order expressions so that the most complicated parts are done first, when more registers for temporary values are available.

It does several other optimizations, such as constant folding missed by the front end, strength reduction (changing multiplies and divides to shifts when possible) peephole optimization (removing add of zero, multiply by one, etc.), and changing A:=A+1 to an internal increment memory Icode.

The optimizer works by building a tree out of the intermediate codes for every statement, and transforming the list of statement trees in various ways. There are seven internal passes per basic block:

1. Build statement trees out of the Icode stream.

2. Preliminary transforms, set address/value flags.

3. Length checking and type coersions.

4. Constant and address folding, expression reorder.

5. Peephole and strength reduction optimizations.

6. Machine dependent transforms (e.g. pointer to ADR).

7. Common subexpression elimination.

Finally, the optimizer calls the code generator to translate the basic block from tree form to target machine code.

The code generator must translate these trees into actual machine code. It uses a series of templates to generate more efficient code for special cases.

For example, there is a series of templates for the add operator. The first one checks for an add of the constant one; if found, it generates an increment. If it gives up, the next template gets control; it checks for an add of any constant, and if found generates an add immediate.

The final template in the series must handle the general case; it gets the operands into registers (by recursively calling the code generator itself), then generates an add register instruction. There is a series of templates for every operation.

The code generator must also keep track of register contents, keep track of several memory segment addresses (code, static, variables, constant data, etc.), allocate temporary variables, and so on.

APPENDIX D. RUNTIME STRUCTURE

Overview

A successful Pascal compilation produces an "object" file, which can be linked with other object files to produce an "executable" file. Object files can be derived from user Pascal programs, modules, or implementations; user code not written in Pascal; or routines in standard runtime modules that support facilities such as error handling, heap variable allocation, or input/output.

The most important runtime module has to do with initialization and termination; also important is the runtime error handling module. The remaining runtime modules are concerned with particular facilities such as reals, sets, files, and so on. File system runtime information is in Appendix B of this book.

Pascal runtime entry points and variables conform to a naming convention: all names are six characters, and the last three are a unit identification letter followed by the letters "QQ". The current unit identifier letters are:

A Not used
B Compile time utilities
C Encode/decode
D Not used (reserved)
E Error handling

F	Pascal file system
G	Generated code helpers
H	Heap allocator
I	Generated code helpers
J	Generated code helpers
K	FCB definition
L	String/LSTRING
M	Not used (reserved)
N	Not used (reserved)
O	Miscellaneous "other" routines
P	Not used (reserved)
Q	Not used (reserved)
R	Real (single precision)
S	Set operations
T	Not used (reserved)
U	DOS file system
V	Not used (reserved)
W	Not used (reserved)
X	Initialize/terminate
Y	Special utilities
Z	Not used (reserved)

Initialization and Termination

Every executable file contains one and only one starting address. As a rule, when Pascal object modules are involved, this starting address is at the entry point BEGXQQ in module ENTX. A Pascal program (as opposed to a module or implementation) has a starting address at the entry point ENTGQQ. BEGXQQ calls ENTGQQ.

In the discussion below, the assumption is made that a Pascal main program along with other object modules

is loaded and executed. However, one can also link a main program in machine language with other object modules in Pascal, in this case some of the initialization and termination done by the ENTX module may need to be done elsewhere.

When a user program is linked with the runtime library and execution begins, several levels of initialization are required. First comes machine oriented initialization, then Pascal runtime initialization, then user program and unit initialization.

Machine Level Initialization

The entry point of a Pascal load module is the routine BEGXQQ, in the module EXTXQQ. BEGXQQ does the following:

- Set the stack pointer. The initial stack pointer is put into public variable STKBQQ, and used to restore the stack pointer after an inter-procedure GOTO to the main program.
- Set the frame pointer, always initially zero.
- A number of public variables must be initialized to zero or NIL. These include:
  - RESEQQ, machine error context (user stack pointer)

- CSXEQQ, source error context list header

- PNUXQQ, initialized unit list header

- HDRFQQ, Pascal open file list header

- Set any other machine dependent registers, flags, etc.

- Set the heap control variables. BEGHQQ and CURHQQ are set to the lowest address for the heap, and the word at this address is set to a heap block header for a free block the length of the initial heap. ENDHQQ is set to the address of the first word after the heap. The length of the initial heap is two bytes. The stack and the heap grow together, and public variable STKHQQ is set to the lowest legal stack address, which is ENDHQQ plus a safety gap.

- Call INIUQQ, the DOS specific file unit initializer. If the file unit is not used and the user does not want it loaded, a dummy INIUQQ entry point must be loaded which just returns.

- Call BEGOQQ, the escape initializer. In a normal load module, a default BEGOQQ is included which just returns. However, this call provides an escape mechanism for any other initialization. For example, it could initialize tables for an interrupt driven profiler, or a runtime debugger.

- Call ENTGQQ, the entry point of the user's Pascal program.

Program Level Initialization

The user's main program continues the initialization process. First the file system is called, INIFQQ.

Next, if $ENTRY was on ENTEQQ is called to set the source error context. Then comes the static data initialization. This includes VALUE section initialization and $INITCK initialization. Static data is initialized by the load process directly.

At the same time every file at the program level gets an initialization call to NEWFQQ.

Next comes unit initialization. Every USES clause in the source, including those in INTERFACEs, generates a call to the initialization code for the unit, in the order that the USES clauses are encountered.

Finally any program parameters are read (or otherwise initialized), and the user program begins.

PPMFQQ is called for every parameter (except INPUT and OUTPUT) to set the parameter's string value as the next line in the file INPUT.

Then one of the READFN routines "reads" and decodes the value into the parameter. PPMFQQ is passed the parameter's identifier to be used as a prompt.

PPMFQQ first calls PPMUQQ to get any command line parameter text; if PPMUQQ gives an error return PPMFQQ does the prompting and reads the response directly.

User unit initialization is much like user program initialization; an ENTEQQ call if $ENTRY was on, variable initialization, unit initialization for every USES clause, and finally the user's unit initialization code. A call to initialize a unit may come from several other units.

The unit interface has a version number, and every initialization call must check that the version number in effect when the unit was used in another compilation is the same as the version number in effect when the unit implementation itself was compiled.

Except for this, unit initialization calls after the first one should have no effect; that is, a unit's initialization code should be executed only once.

Both version number checking and single initial code execution are handled with code automatically generated at the start of the body of the unit, which has the effect of:

IF INUXQQ (useversion, ownversion) THEN RETURN

The interface version number used by the compiland using the interface is always passed as a value parameter to the implementation initialization code. This is passed as the "useversion" to INUXQQ. The interface version number in the implementation itself is passed as "ownversion" to INUXQQ.

INUXQQ gives an error if the two are unequal. It also maintains a list of every unit initialized. INUXQQ returns true if the unit is found on the list, or else puts the unit on the list and returns false. The list header is PNUXQQ, and a list entry contains a way to identify the unit plus a pointer to the next entry.

User modules also have initialization code, the same as a program and unit implementations (variables, USES units), but without user initialization code or INUXQQ calls. However, the module's initialization call cannot be issued automatically. When the module is compiled, a warning is given if an initialization call will be required; that is, if there are any files declared, or USES clauses.

To initialize a module, declare the module name as an external procedure and call it at the beginning of the program.

Termination

Program termination occurs in one of three ways:

- The program may terminate normally, in which case the main program returns to BEGXQQ, and BEGXQQ transfers control to ENDXQQ.

- The program may end due to an error condition, either with a user call to ABORT, or a runtime call to an error handling routine; in either case an error message, error code, and error status are passed to EMSEQQ, which does whatever error handling it can and calls ENDXQQ.

- ENDXQQ can be declared an external procedure and be called directly.

    1. ENDXQQ calls ENDOQQ, the escape terminator, which normally just returns.

    2. ENDXQQ calls ENDYQQ, the generic file system terminator. ENDYQQ closes all open files, using the file list headers HDRFQQ and HDRVQQ.

    3. ENDXQQ calls ENDUQQ, the operating system specific file unit terminator.

ENDUQQ need not return; for example, it can exit to the operating system. If ENDUQQ returns, ENDXQQ exits to the operating system. As with INIUQQ, INIFQQ, and INIVQQ, users who do not require any file handling will need to declare empty parameterless procedures for ENDYQQ and ENDUQQ.

As mentioned, the main initialization and termination routines are in module ENTX. Stubb procedures for BEGOQQ, ENDOQQ, and other miscellaneous entry points are in module MISY.

Error Handling

Runtime errors are detected in one of four ways:

- By the user program, which calls ABORT (EMSEQQ).
- By a runtime routine, which calls EMSEQQ.
- By an error checking routine in the error module itself, which calls EMSEQQ.
- By an internal helper routine, which calls an error message routine in the error unit, which calls EMSEQQ.

Handling an error detected at runtime usually involves telling the user the type and location of the error and terminating the program. The error type has three components: a message to the user, an error number, and an error status.

The message describes the error, and the number can be used to look up more information in a manual.

The error codes themselves can be found in Appendix A. They are assigned as follows:

1.. 999 Available for use with the ABORT procedu
1000..1099 Unit U file system errors
1100..1199 Unit F file system errors
1200..1999 Reserved
2000..2049 Heap, stack, memory
2050..2099 Ordinal arithmetic
2100..2149 Real arithmetic
2150..2199 Structures; sets and strings
2200..2399 Reserved
2400..2449 Reserved
2450..2499 Other internal errors
2500..2999 Reserved

The error location has two parts: the machine error context, and the source program context.

The machine error context is the program counter, stack pointer, and frame pointer at the point of the error. The program counter is always the address following a call to a runtime routine (for example, a return address).

The source program context is optional, controlled by metacommands. If $ENTRY is on, it consists of the source filename of the compiland containing the error, the routine name in which the error occurred (program, unit, module, procedure, or function), and the line and page number of the routine in the listing file. If $LINE is also on, the line number of the statement containing the error is also given. Setting $LINE also sets $ENTRY.

Machine Error Context

Runtime routines are always compiled with $RUNTIME set. This causes special calls to be generated at entry and exit points of the runtime routine, which save the error context: frame pointer, stack pointer, and program counter at the point where a runtime routine is called that leads to an error.

Runtime routines may call other runtime routines, but the error location is always in the user program; for example, in a program that was not compiled with $RUNTIME set.

The runtime entry helper, BRTEQQ, is passed the offset to the saved caller frame pointer as a value parameter.

First it examines RESEQQ; if this value is not zero, the current runtime routine was called from another runtime routine and the error context has already been set, so it just returns. If RESEQQ is zero, however, the error context must be saved. The caller's stack pointer is determined from the current frame pointer and stored in RESEQQ.

The address of the caller's saved frame pointer and return address (program counter) in the frame is determined from the offset passed to BRTEQQ. Then the caller's frame pointer is saved in REFEQQ. The caller's program counter (for example, BRTEQQ's caller's return address) is saved; the offset in REPEQQ and the segment (if any) in RECEQQ.

The runtime exit helper, ERTEQQ, has no parameters. It determines the caller's stack pointer (again, from the frame pointer) and compares it against RESEQQ. If they are equal, the original runtime routine called by the user's program is returning, so RESEQQ is set back to zero.

EMSEQQ uses RESEQQ, REFEQQ, REPEQQ, and RECEQQ to display the machine error context. The machine error context is always displayed, and there is little extra overhead in keeping track of it.

Source Error Context

Giving the source error context involves extra overhead since source location data must be included in the objec code in some form. This is done with calls which set the current source context as it occurs.

These calls could also be used to break program execution as part of the symbolic debug process. The overhead of source location data, especially line numbe calls, can be significant. Routine entry and exit calls, while having more overhead, are much less frequent and so the overhead is less significant.

The procedure entry call to ENTEQQ passes two VAR parameters: the first is an LSTRING containing the source filename; the second is a record containing the line number and page number (both words) and procedure or function identifier (an LSTRING).

The filename is that of the compiland source; that is, the main source filename, not the names of any $INCLUDE files. The procedure identifier is the full identifier used in the source, not the linker name. If one name is given in an INTERFACE and another in a USES clause, the USES identifier is used.

Entry and exit calls are also generated for the main program, unit initialization, and module initialization, in which case the identifier is the program, unit, or module respectively.

The procedure exit call to EXTEQQ does not pass any parameters. It pops the current source routine context off a stack maintained in the heap.

The line number call to LNTEQQ passes a line number as a value parameter. The current line number is kept in public variable CLNEQQ. Since the current routine is always available ($LINE implies $ENTRY), the compiland source filename and routine containing the line are available along with the line number.

Line number calls are generated just before the code starting a statement which is the first statement starting on the line. The statement can, of course, be part of a larger statement. The $LINE+ metacommand should be placed at least a couple of symbols before the start of the first statement intended for a line number call ($LINE- also takes effect "early").

Most of the error handling routines are in module ERRE The source error context entry points ENTEQQ, EXTEQQ, and LNTEQQ are in module DEBE.

Heap Allocation

Variables allocated with the NEW or ALLHQQ calls come from an area of memory called the heap. These variables are not moved once allocated (permitting address references to them), but can be deallocated using the DISPOSE call.

A heap variable resides in a heap "block." A heap block contains a header word followed by data words. The header word low order bit is one if the block is free and zero if it is allocated; the rest is the even length in bytes without the header. Length zero blocks are permitted.

Three pointers to locations in the heap, referring to header words, are maintained by the heap allocator:

BEGHQQ: start of the heap (<=CURHQQ,<ENDHQQ)

CURHQQ: current heap block
(>=BEGHQQ,<#ENDHQQ)

ENDHQQ: end of the heap (>BEGHQQ, >CURHQQ)

The heap is a contiguous area of memory starting at BEGHQQ of length ENDHQQ - BEGHQQ. Memory at address ENDHQQ is never accessed by the heap allocator.

The Pascal procedure NEW generates a call to GETHQQ, which is passed the number of bytes wanted and returns a pointer to the allocated variable. GETHQQ just calls ALLHQQ (which has the same calling sequence); ALLHQQ returns zero if the heap is full or one if the heap structure is in error, and GETHQQ calls an error procedure if one of these errors occurs.

Heap allocation in ALLHQQ is basically first fit, roving pointer. If the block at CURHQQ is allocated, advance CURHQQ. If it is free and big enough, return a pointer to the block, and free any extra space at the end. If it is free but too small, advance CURHQQ, but if the following block is free, collapse them into one free block. If CURHQQ ever goes above ENDHQQ, the heap structure is in error and ALLHQQ returns one.

If CURHQQ is equal to ENDHQQ, the first time this happens, restart from BEGHQQ. However, if the topmost block in the heap was free CUTHQQ is called first to permit releasing this ending free block, usually making this area available to the stack. The second time CURHQQ equals ENDHQQ the current heap is full. In this case GROHQQ is called, which may extend the heap by increasing ENDHQQ or decreasing BEGHQQ in

a system dependent way. Then CURHQQ is reset to BEGHQQ and a third search made. If this is not successful, ALLHQQ returns zero.

Heap deallocation (DISPOSE) is done in-line or with a helper by setting the low order bit of the header. BEGXQQ does heap initialization by setting BEGHQQ, CURHQQ, and ENDHQQ. The generic heap routines ALLHQQ and GETHQQ are in module HEAH. The routines GROHQQ and CUTHQQ are in module MISHM.

Other Runtime Modules

Code Generator Helpers

Some operations considered "primitive" by a code generator may generate too much inline code to be practical; in this case a call to a runtime entry point gets generated instead. These code generation "helpers" can be thought of as increasing the power of the machine; they generally do not use the standard calling sequence.

The module MISG contains all the code generator helpers.

Ordinal Encode/Decode

The encoding and decoding routines are all called EN?CQQ or DE?CQQ, where the "?" is a letter indicati

the type. For the types INTEGER (letters I and J), WORD (letters W and X), enumerated (letters E and F), and BOOLEAN (letter B) these routines are in module CODC. Some of these are written in IBM Pascal.

Real Number Support

The module REAR provides the primitives to add, subtract, multiply, divide, negate, absolute, and do comparisons. All other modules are written to be independent of the actual real number internal format.

These are UTLR (two utilities), CNVR (integer/real conversion), TNSR (transcendental functions), REAC (encode and decode), and RIOF (read and write). The additional transcendental functions are in module RFAR.

Set and String Operations

Those set operations that require runtime support are in module SETS, which is written in IBM Pascal. One exception: the IN operator's entry point CINSQQ is found in MISG.

STRING and LSTRING operations that require runtime are in module LSTL, also written in IBM Pascal. Most of these are implemented by using the MOVE and FULL routines, which are in assembly language.

APPENDIX E. PASCAL STANDARD AND IBM FEATURES

Summary of IBM Pascal Features

This list summarizes the extensions added to Standard Pascal.

Syntactic and Pragmatic

- **Metalanguage**
  - **Metacommands:**

$BRAVE	$LINE	$PUSH/$POP
$DEBUG	$LINESIZE	$RANGECK
$ENTRY	$LIST	$RUNTIME
$ERRORS	$MATHCK	$SHIP
$GOTO	$MESSAGE	$STACKCK
$IF..$END	$NILCK	$SUBTITLE
$INCLUDE	$OCODE	$SYMTAB
$INCONST	$PAGE	$TITLE
$INDEXCK	$PAGEIF	$WARN
$INITCK	$PAGESIZE	

- **Extra Listing**

  Flags for Jumps, Globals, Identifier level, Control level header, trailer, general listing format textual error and warning messages.

- **Syntactic Shorthand**
  - "!" as comment to end of line
  - [. . .] equivalent to BEGIN. . .END

- **Nondecimal Number**
  - Numeric constants with #, 2#, 8#, 10#, 16#
  - DECODE/READ takes #, ENCODE/READ with N of 2, 8, 10, 16

- **Extended CASE range**

  Applies to CASE statements and record variants; OTHERWISE for all other values; A..B for range of values.

Data Types and Modes

- **WORD** type; **WRD** function; **MAXWORD** constant

- **Address types ($SYSTEM level)**
  - ADR and ADS types and operators
  - VARS parameter

- **SUPER array types**

- **STRING pre-declared type**

- **LSTRING super type**
  - NULL constant
  - .LEN field

- **Explicit byte offsets in records**

- **CONST parameter**
- **Structured (array, record, and set) constants**
- **Extended functions** returning any assignable type; variable selection on values returned from functions
- **Attributes:**

PUBLIC	EXTERNAL	READONLY
EXTERN	STATIC	PURE

Operators and Intrinsics

- **Extended Operators:**
  - Bitwise Logical: AND OR NOT XOR
  - Set Operators: < and >
- **Constant Expressions:**
  - Numeric, Ordinal, Boolean expressions in type clauses
  - String Constant concatenation with * operator
  - Other Constant functions LOWER, UPPER, SIZEOF, RETYPE
- **Extended Intrinsics:**

RESULT	UPPER	DECODE
EVAL	LOBYTE	ENCODE
SIZEOF	HIBYTE	ABORT
LOWER	BYWORD	

- **System Intrinsics:**

  MOVEL FILLC MOVESL
  MOVER RETYPE MOVESR

- **String Intrinsics:**

  – STRING or LSTRING:
  POSITN SCANNE
  SCANEQ COPYSTR

  – LSTRING type only:
  CONCAT DELETE
  INSERT COPYLST

- **REAL library functions**

- **Pascal library functions:**

  UADDOK ALLHQQ TICS
  SADDOK TIME BEGXQQ
  UMULOK DATE ENDXQQ
  SMULOK

Control Flow and Structure

- **Control flow statements:**

  BREAK, CYCLE, and RETURN

- **Sequential control operators:**

  AND THEN and OR ELSE in FOR, WHILE, REPEAT

- **VALUE section to initialize static variables**

- **Mixed order and multiple CONST, TYPE, VAR, VALUE sections** allowed

- **UNIT INTERFACE and IMPLEMENTATION:**
  - Interface version numbers, version checking
  - Guaranteed unique unit initialization

Input/Output and Files

- **Extended I/O:**
  - Textfile line length declaration, TEXT (nnn)
  - READ Boolean, pointer, STRING, LSTRING
  - WRITE pointer, LSTRING
  - Negative M value to justify left instead of right
  - Temporary files
  - DIRECT mode files, SEEK procedure
  - ASSIGN, CLOSE, DISCARD, READSET, READFN procedures
  - FILEMODES type and constants, F.MODE access
  - Error trapping, F.TRAP and F.ERRS access

- **System I/O:**

  Full FCBFQQ type equivalent to FILE types.

IBM Pascal and Standard Pascal

ISO Pascal defines many error conditions; however, an implementation can "handle" an error by stating in the documentation that the error is not caught. These errors "not caught," and other differences between ISO Pascal and IBM Pascal, are in this Appendix. The following should be noted concerning the Standard level of IBM Pascal and the ISO/ANSI/IEEE Pascal standard:

- IBM Pascal allows "@" as a substitute for the "^" - a minor extension.

- The underscore "_" is allowed in identifiers, a minor extension.

- In some cases a separator is not required between a number and an identifier or keyword; for example, "100mod" is accepted as "100 mod" without error.

- Normally a component of a PACKED structure cannot be passed as a reference parameter, but we specifically permit passing a CHAR element of a PACKED ARRAY [1..n] OF CHAR as a reference parameter (passing other packed components gives the usual error).

- The textfile linemarker character is not supposed to be in the set of CHAR values, but we permit all 256 8-bit values as CHAR values.

- IBM Pascal permits a variant record declaration in which not all tag values are given; for example, RECORD CASE INTEGER OF 1: (a:b) END. In the standard a variant must be given for all possible tag values.

- In general we ignore the error of using an identifier and then re-declaring it in the same scope. For example, CONST X=Y; VAR Y:CHAR; has two meanings for Y in the same scope. IBM Pascal generally uses the latest definition for an identifier. However, there is one ambiguous case; if a type F00 is declared in one scope and an inner scope declares TYPE P = F00; F00 = type; then F00 has two meanings and the programmer intent is ambiguous. In this case the later definition of F00 is used and a warning generated.

- Some standard procedures and functions cannot be passed as parameters; this is allowed in the standard. A new directive EXTERN is allowed; new directives are permitted in ISO Pascal. Also, a new predeclared function FLOAT is allowed; new predeclared functions are permitted in ISO Pascal.

- In the WRITE and WRITELN procedures, the field width "M" can be less than zero; in ISO Pascal this is an error; in IBM Pascal, M < 0 is treated as if M = ABS(M) but field expansion takes place from the right rather than the left. All textfile READ(LN) and WRITE(LN) parameters can take both M and N parameters; if not needed they are ignored. The form "V::N" is allowed. When writing an INTEGER, the N parameter sets the output radix.

- The standard does not allow a variable created with the long form of NEW to be assigned, used in an expression, or passed as a parameter, but this is difficult to check for at compile time and expensive to check at runtime. We allow assignments to these variables using the actual length of the target variable, and the ISO Pascal error is not caught.

- The short form of DISPOSE may be used on a structure allocated with the long form of NEW. The standard does not allow this; a variable allocated with the long form of NEW should only be released with the long form of DISPOSE, and all tag fields should never change between the calls.

- The ISO standard defines a number of errors having to do with variant records which we do not catch. The ISO standard declares that when a "change of variant" occurs (such as when a new tag value is assigned) all the variant fields become undefined, but we do not set the fields uninitialized when a new tag is assigned and so we ignore the use of a variant field with an undefined value.

- A DISPOSE of a record passed as a reference parameter or used by an active WITH statement is an error in the ISO standard not caught by our compiler. Also, passing a record used by an active WITH statement as a reference parameter is not caught. Finally, changing the position of a file while its buffer variable is an active WITH record or reference parameter is an error not caught.

- When using the default files INPUT and OUTPUT in READ, READLN, WRITE, and WRITELN, if a list of data items is given, the first item cannot contain a file, as in READ (OUTPUT^).

- IBM Pascal semantics for DIV and MOD do not agree with the ISO standard. Programs intending to be portable should not use DIV and MOD unless both operands are positive.

- Super arrays provide much the same function as the conformant array concept.

- ISO Pascal requires the control variable of a FOR loop to be local to the immediate block. Our compiler does not detect an assignment to the control variable if the assignment occurs in a procedure or function called within the FOR loop.

- We allow CHR to take any ordinal type. ISO requires the CHR argument to be INTEGER.

APPENDIX F. IBM PASCAL SYNTAX

This is the formal definition of IBM Pascal syntax, as well as documentation of the recursive descent structure of the compiler.

Syntax

Syntax is described with the following metasyntax:

- Words in upper case, and special characters not used in the metasyntax stand for themselves. Note that none of the special characters used in the metasyntax are also used in Pascal program syntax.

  Examples: PROGRAM VAR BEGIN END + - [ ]

- Words in lower case stand for a syntactic class. Usually they are also identifiers for procedures or functions in the compiler that parse the syntactic class.

  Examples: identifier constant term expression

- The left brace and right brace enclose a group of items. If a plus follows the right brace, the items can be repeated but at least one must appear. If a minus follows the right brace, the item is optional but cannot be repeated. If a star follows the right brace, the item is optional and can also be repeated.

  Note that the plus, minus, or star occur right next to the right brace; spaces between the right brace and a special character mean the special character is part of the program syntax.

This gives four possibilities:

{item}	Item must appear once (exactly one time)
{item} +	Item list must appear (one or more times)
{item} -	Optional single item (zero or one time)
{item} *	Optional, list items (zero or more times)

Examples:

{VAR} {digit} + {+|-} - {letter} *

The vertical bar between items shows alternate choices. Examples:

- The vertical bar between items shows alternate choices. Examples:

  program | module | implementation

- An item followed by a backslash and a special character stands for one or more items separated by the character. Note that the separator character only appears between items, and *not* after the final item. Examples:

  identifier\, statement\; constant\,

Primitive Classes (Scanner Portion of Compiler)

These classes do not identify compiler routines. A *letter* is one of the 52 upper and lower case alphabetics. A *digit* is one of the digits 0..9, or letters A..F or a..f. A *char* is one of the 256 characters, except for ' (quote).

```
ident::= letter{letter | digit | _}*

string::='{char | ' '}+'

number::={{digit}+ #}-{digit}

realnm::={digit}+ .{digit} {{E | e}{+ | -}-{digit}+}-
```

Major Classes (Major Body of Compiler)

```
[main] ::= {getintf mainhed getuses {head body} +}

getintf::={INTERFACE{(number)}- ;
   UNIT ident(ident\,) ;
   getuses{declare | header}*
   {BEGIN}-END;}-

mainhed::=PROGRAM ident{(ident\,)}-;|
   MODULE ident; |
   IMPLEMENTATION OF ident;

getuses::={USES{ident{(ident\,)}- ;}+ }*
```

```
head::={declare header}

declare::={LABEL{getlabl}\ , ; |
   CONST{ident = express}\;  ; |
   TYPE {ident = typedec}\;  ; |
   VAR getattr
   {{ident getattr}\, : typedec}\;  ; |
   VALUE{ident
   {{[ordcons\,]} | {.ident}}*
   := express}\;  ;}*

ordcons::= express

getlabl::={ident | number}

getattr::={[{attrs1 ordcons
   {: ordcons}- }\, ] }-

attrs1::={PUBLIC | EXTERN | EXTERNAL |
   STATIC | PURE | READONLY}

typedec::={^ | ADR OF | ADS OF}-
   {{ident{( number \ , )}-} |
   {ordtype} |
   {SUPER}-{PACKED}- |
   {ARRAY [{ordtsub | {ordcons.. *}\,]
   OF typedec} |
   {RECORD fieldls END} |
   {SET OF ordtsub} |
   {FILE OF typedec}} |

ordtsub::= ordtype

ordtype::= ident | ( ident \ , ) |
   ordcons .. ordcons
```

```
fieldls::={ident\, {[ordcons]}-
   :typedec {;}-} *
   {CASE{ident{[ordcons]}-
   :}- ident
   OF{ordlist ( fieldls )}\;}-
   {;}-

ordlist::={{ordcons{..ordsubr}}\,  :}

ordsubr::= ordcons

header::={{PROCEDURE | FUNCTION} ident formals;
   {{FOWARD | EXTERN | EXTERNAL} ;}-}-

formals::={({{VAR | CONST | VARS}-
   ident\, : ident}
   | {{PROCEDURE | FUNCTION}
   ident formals}}\;)}-
   {: ident}- getattr

body::={END | {BEGIN statemt END};}{. | ;}

statemt::={{ident | number} :  }*
   ident selectp := express |
   ident actuals |
   BEGIN statemt \ ; END |
   [ statemt \ ; ] |
   IF boolexp THEN statemt{ELSE statemt} - |
   WITH{ident selectp}\, DO statemt |
   FOR{STATIC} ident := express
      {TO | DOWNTO} express DO statemt |
   REPEAT statemt \ ; UNTIL boolexp |
   WHILE boolexp DO statemt |
   {BREAK | CYCL}{getlabl}- |
   GOTO getlabl |
   RETURN |
```

```
    CASE ordexpr OF
      {ordlist statemt}\;{;}-
      {OTHERWISE statemt\;{;}-} END |
      {}

boolexp::={express{{AND
    THEN | OR ELSE} express}*}

ordexpr::= express

selectp::= [ ordexpr\,] | . ident | ^

express::= simple{{< | <= | > | >= | = | <> |
    IN} simple}*

simple::={+ | -}- term{{+ | - | OR |
    XOR} term}*

term::= factor{{* | / |
    DIV | MOD | AND} | factor}*

factor::={ident |
    ident actuals |
    chrcons | strcons |
    ident setcons |
    number | realnm |
    (express) |
    NOT factor |
    NIL |
    ADR ident |
    ADS ident |
```

INDEX

Special Characters

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Product Comment Form

PASCAL 6172272

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in anyway it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Comments:

If you wish a reply, provide your name and address in this space.

Name _____

Address _____

City _____ State _____

Zip Code _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Please do not staple

Tape

The Personal Computer
Software Library

Product Comment Form

PASCAL 6172272

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in anyway it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Comments:

If you wish a reply, provide your name and address in this space.

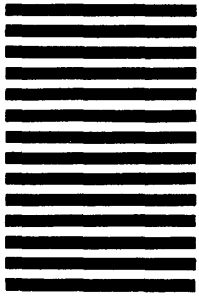
Name _____

Address _____

City _____ State _____

Zip Code _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Please do not staple

Tape

Product Comment Form

PASCAL 6172272

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in anyway it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Comments:

If you wish a reply, provide your name and address in this space.

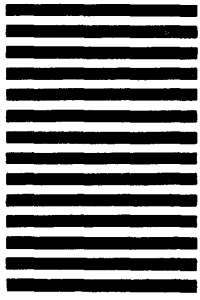
Name _____

Address _____

City _____ State _____

Zip Code _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Please do not staple

Tape

The Personal Computer
Software Library

Product Comment Form

PASCAL 6172272

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in anyway it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Comments:

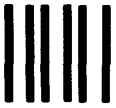
If you wish a reply, provide your name and address in this space.

Name _____

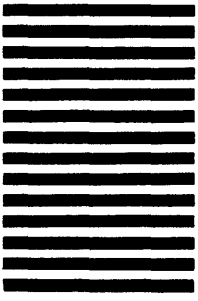
Address _____

City _____ State _____

Zip Code _____



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Please do not staple

Tape

Continued from inside front cover

SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

IBM does not warrant that the functions contained in the program will meet your requirements or that the operation of the program will be uninterrupted or error free.

However, IBM warrants the diskette(s) or cassette(s) on which the program is furnished, to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your receipt.

LIMITATIONS OF REMEDIES

IBM's entire liability and your exclusive remedy shall be:

1. the replacement of any diskette(s) or cassette(s) not meeting IBM's "Limited Warranty" and which is returned to IBM or an authorized IBM PERSONAL COMPUTER dealer with a copy of your receipt, or
2. if IBM or the dealer is unable to deliver a replacement diskette(s) or cassette(s) which is free of defects in materials or workmanship, you may terminate this Agreement by returning the program and your money will be refunded.

IN NO EVENT WILL IBM BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE SUCH PROGRAM EVEN IF IBM OR AN AUTHORIZED IBM PERSONAL COMPUTER DEALER HAS BEEN ADVISED OF THE POSSIBLITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

GENERAL

You may not sublicense, assign or transfer the license or the program except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is void.

This Agreement will be governed by the laws of the State of Florida.

Should you have any questions concerning this Agreement, you may contact IBM by writing to IBM Personal Computer, Sales and Service, P.O. Box 1328-W, Boca Raton, Florida 33432.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

International Business Machines Corporation

P.O. Box 1328-W
Boca Raton, Florida 33432

6172272
Printed in USA