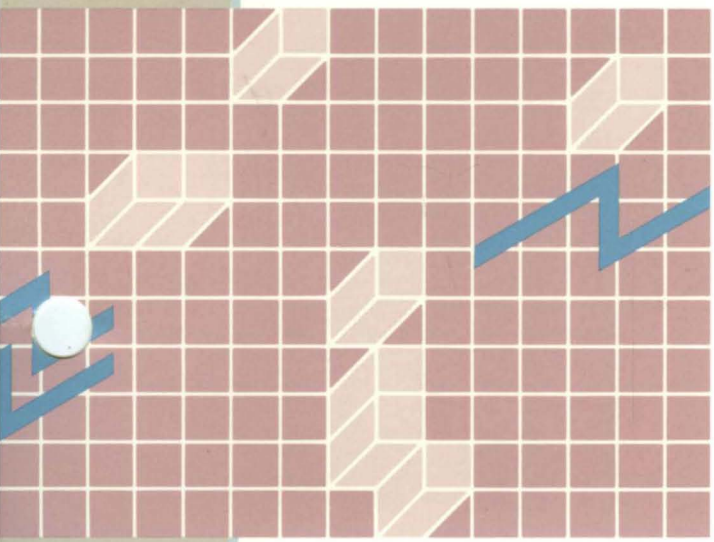
Pascal

Third Edition
Update Notice 1



The enclosed pages update the following manual:

Pascal Reference, Third Edition
A-09-00852-01-A

Insert the pages of this Update Notice into the *Pascal Reference* according to the instructions below. Each new page is dated so that you know which pages are update pages and which are original pages to the *Pascal Reference*. The Summary of Changes on the first page of this Update Notice summarizes the changes made in this notice.

All material discussed in this Update Notice will be incorporated into the next edition of the *Pascal Reference*. To retain a record of this Update Notice, we suggest you insert this cover page immediately after the title page of your *Pascal Reference*.

Volume 1

Find and Remove	**Replace With**
i to iv	i to iv
vi to xii	vii to xii
5-3/5-4	5-3/5-4
5-9/5-10	5-9/5-10
12-15 to 12-18	12-15 to 12-18.2
Index-1 to Index-18	Index-1 to Index-18

Volume 2

i to iv	i to iv
vii/viii	vii/viii
13-25/13-26	13-25/13-26
14-13 to 14-16	14-13 to 14-16
14-47 to 14-50	14-47 to 14-50
14-61/14-62	14-61/14-62
15-25/15-26	15-25/15-26
16-7/16-8	16-7/16-8
17-5/17-6	17-5/17-6
18-1/18-2	18-1/18-2
18-7 to 18-10	18-7 to 18-10
18-15/18-16	18-15/18-16
19-1 to 19-24	19-1 to 19-25
A-1 to A-50	A-1 to A-51

Find and Remove	Replace With
F-3/F-4	F-3/F-4
F-7 to F-10	F-7 to F-10
F-13/F-14	F-13/F-14
H-1 to H-18	H-1 to H-20
Index-1 to Index-18	Index-1 to Index-18

PASCAL REFERENCE MANUAL: VOLUME 1

Third Edition (September 1984) A–09–00852–01–A
Update Notice 1 (December 1987) 09–01363–01

This document was produced using the Document Designer Series.

SUMMARY OF CHANGES

The 9.1 and 10.0 Releases of Pascal are described in this update notice to the the Pascal Manual, third edition (Volume 1, A-09-00852-01-A, and Volume 2, A-09-00868-01-A).

Changes in Update Notice 1 include the removal of functions no longer supported with the Pascal 9.1 release, the addition new Linker procedures, a description of the (optional) Math service for floating point calculation, and the inclusion of other technical corrections.

These changes are summarized below.

- The functions, AIDRQQ and ANDRQQ, are no longer supported (as of the 9.1 software release).

- Error message 2052, Signed Divide by Zero, is reported for all occurrences of integer division by zero (as of the 9.1 software release).

- Object modules must be linked using the Bind command rather than the Link command in order to produce a version 6 run file. (Version 4 run files produced by the Link command are no longer supported as of the 10.0 software release.)

- PasFirst.obj, an assembly language module included with the 10.0 software release, must be the first file specification entered on the Object modules command line of the Bind command.

- You can optionally install the Math service for floating point calculation. When installed, the service emulates the numeric coprocessor.

NOTE

In this manual, the terms CTOS and CTOS operating system refer to either the CTOS operating system, which runs on the Intel 8086 microprocessor, or the CTOS/VM operating system, which runs on the Intel 80X86 series of microprocessors.

RELATED DOCUMENTATION

The following manuals, or related products, are referenced in this manual. It may be helpful to have copies of them on hand when you are using this manual.

The complete Guide to Technical Documentation is provided in the Executive Manual or similar command-line interpreter manual for your operating system.

Utilities
- Executive

System Administration
- CTOS System Administrator's Guide

Operating System
- CTOS Operating System
- CTOS/VM Concepts
- CTOS/VM Reference

Languages
- Assembly Language
- BASIC
- BASIC Compiler
- FORTRAN
- FORTRAN-86
- Level II COBOL
- COBOL Animator
- Workstation C

Programming Tools
- Debugger
- Editor
- Forms
- Linker/Librarian
- Status Codes

UTILITIES

The Executive manual functions as a user's guide and a reference to the available Executive commands. It addresses command execution, file management and protection, and program invocation. It also provides descriptions and details about parameter fields for Executive commands.

SYSTEM ADMINISTRATION

The CTOS System Administrator's Guide

The CTOS System Administrator's Guide describes the administrative aspects of the CTOS and CTOS/VM operating systems. Topics include setting up various operating system types, installing system services, adding application software, and installing perripheral devices. In addition, the manual describes SRP and workstation configuration, system protection, nationalization, and operating system customization (SysGen).

OPERATING SYSTEM

The CTOS Operating System manual describes the CTOS Operating System concepts and operations. Topics include parameters, I/O, memory, messages, timers, system services, virtual code, and interrupts. Additionally, it contains the system structure formats.

CTOS/VM Concepts describes the CTOS/VM Operating System. Topics include parameters, I/O, memory, messages, timers, system services, virtual code, interrupts, and administration.

CTOS/VM Reference, together with CTOS/VM Concepts, describes each operation contained in the System Image and in the standard object module library, CTOS.lib. This manual also contains the format of each system structure.

LANGUAGES

Assembly Language describes programming with assembly language at the symbolic instruction level, including the instruction set. Some knowledge of assembly language and the machine architecture of the associated CPU is recommended before using this manual.

The BASIC manual describes the BASIC language syntax.

The BASIC Compiler manual describes how to use the BASIC Compiler and the BASIC Assembler.

The FORTRAN manual describes the FORTRAN language syntax and includes instructions for using the FORTRAN Compiler in the CTOS environment. Additional topics include how to compile and link FORTRAN programs and how to configure FORTRAN with non-FORTRAN procedures.

The FORTRAN-86 manual describes the FORTRAN-86 language syntax and includes instructions for using the FORTRAN-86 Compiler. Additionally, FORTRAN-86 operation is explained for the DISTRIX and CTOS environments.

The COBOL Animator describes the commands used to debug COBOL programs as they execute. An example is included.

The Pascal Reference manual describes the Pascal programming language. It provides specific information for using the Pascal Compiler and for using Pascal to make CTOS calls.

The Workstation C Programmer's Guide describes enhancements to the C programming language, models of segmentation, library functions, and troubleshooting tips for workstation systems. The manual assumes that its reader is familiar with the C programming language.

PROGRAMMING TOOLS

The Debugger manual introduces and describes how to debug C, FORTRAN, Pascal, and assembly language programs. Previous experience with debugging and knowledge of assembly language are recommended. For debugging COBOL or BASIC, see their respective manuals.

The Editor manual describes how to use the Editor to create or modify an ASCII text file.

The Forms manual describes how to use the Forms Editor to interactively design and edit forms. Also included is a description of how the Forms run time works when it is called from an application program to display forms and to accept user input.

The Linker/Librarian describes the Linker, which links separately compiled object files, and the Librarian, which builds and manages libraries of object modules.

The Status Codes manual contains a comprehensive listing of all status codes that can be generated by a CTOS workstation or a Shared Resource Processor, including bootstrap ROM error codes and CTOS utilization codes. The codes are organized, explained, and interpreted in numerical order.

OTHER

The iAPX Programmer's Reference is an Intel-published manual that describes the iAPX 286 architecture. It includes sections on the base architecture and instruction set, real address mode, memory management, protection, and interrupts. There is also a section on advanced topics.

The 80386 Programmer's Reference is an Intel-published manual that describes the 80386 architecture. It includes sections on applications and systems programming, 16-bit compatibility, and the instruction set.

Having both an INTEGER and a WORD type permits mapping of 16-bit quantities in either of two ways:

- as a signed value ranging from -32767 to +32767
- as a non-negative value ranging from 0 to 65535

However, do not mix WORD and INTEGER values in an expression (although doing so generates a warning rather than an error message). Neither are WORD and INTEGER values assignment-compatible.

CHAR

In this version of Pascal, CHAR values are 8-bit ASCII values. CHAR is an ordinal type. All 256-byte values are included in the type CHAR. In addition, SET OF CHAR is supported. Relational comparisons use the ASCII collating sequence.

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

(See the appendix entitled "Standard Character Set" in your operating system manual for a complete listing of the ASCII character set.)

BOOLEAN

BOOLEAN is an ordinal type with only two (pre-declared) values: FALSE and TRUE. The BOOLEAN type is a special case of an enumerated type, where ORD (FALSE) is 0 and ORD (TRUE) is 1. This means that FALSE < TRUE.

You can redefine the identifiers BOOLEAN, FALSE, and TRUE, but the compiler implicitly uses the former type in Boolean expressions and in IF, REPEAT, and WHILE statements.

No function exists for changing an ordinal type value to a BOOLEAN type value. However, you can achieve this effect with the ODD function for INTEGER and WORD values, or the expression

```
ORD (value) <> 0
```

ENUMERATED TYPES

An enumerated type defines an ordered set of values. These values are constants and are enumerated by the identifiers that denote them.

Examples of enumerated type declarations:

```
FLAGCOLOR = (RED, WHITE, BLUE)
SUITS = (CLUB, DIAMOND, HEART, SPADE)
DOGS = (MAUDE, EMILY, BRENDAN)
```

The type values (for example, RED, CLUB, or MAUDE) do not have to be declared in the CONST section or any other section in the program.

Every enumerated type is also an ordinal type. Identifiers for all enumerated type constants must be unique within their declaration level.

The ORD function, at the standard level, can be used to change enumerated values into INTEGER values; the WRD function changes enumerated values into WORD values.

The values obtained by applying the ORD function to the constants of an enumerated type always begin with zero. Thus, the values obtained for the type FLAGCOLOR, from the example above, are as follows:

```
ORD (RED)   = 0
ORD (WHITE) = 1
ORD (BLUE)  = 2
```

The RETYPE function, at **extend level**, can be used to change INTEGER or WORD values to an enumerated type. For example:

```
IF RETYPE (COLOR, I) = BLUE THEN
  WRITELN ('TRUE BLUE')
```

Enumerated types are particularly useful for representing an abstract collection of names, such as names for operations or commands. Modifying a program by adding a new value to an enumerated type is much safer than using raw numbers, since any arrays indexed with the type or any sets based on the type are changed automatically.

As an extension to standard Pascal, the exponent character can be "D" or "d" as well as "E" or "e", for example, 12.34d56. Note that the D or d exponent character does not indicate double precision, as it does in the FORTRAN language.

This version of Pascal performs floating point operations using the numeric coprocessor chip the Math service, or the Pascal run-time support library numeric coprocessor emulator routines. The numeric coprocessor chip is an option installed only on some workstations. (See the subsection "Linking Your Program" in Section 18, "Using the Compiler," for information on how to link you program if you have a numeric coprocessor chip.) The Math service also is an option. See the Math service Release Notice for inatallation details.

Operations on two REAL4 operands are calculated in REAL4 precision with the numeric coprocessor emulator, but with REAL8 precision if you have a numeric coprocessor chip installed on your workstation.

REAL literals are converted first to REAL8 format and then to REAL4 as necessary (for example, to be passed as a CONST parameter or to initialize a variable in a VALUE section). If you need actual REAL4 constants, you must declare them as REAL4 variables (perhaps adding the READONLY attribute) and assign them a constant in a VALUE section.

Both REAL4 and REAL8 values are passed to intrinsic functions as reference (CONSTS) parameters, rather than as value parameters. The compiler accepts REAL expressions as CONSTS parameters; it will evaluate the expression, assign the result to a stack temporary, and pass the address of the temporary, which is usually more efficient than passing the value itself.

Functions that return REAL values use the long return method. That is, the caller passes an additional, hidden, offset address of a stack temporary, which will receive the result. This applies to all functions returning REAL4 or REAL8 values, both user-defined and intrinsic. (See the subsection "Boolean Expressions," in Section 11, "Expressions," for a description of REAL comparisons that produce an unordered result.)

All results are rounded up to the nearest representable number (with 0.5 rounded up or down to make the next digit even.)

INTEGER4

As with INTEGER and WORD values, INTEGER4 values are a subset of the whole numbers. INTEGER4 values occupy four bytes of storage and range from -MAXINT4 to MAXINT4. MAXINT4 is a predeclared constant with the value of 2,147,483,647 (2**31 - 1). The value -2,147,487,648 (-2**31) is not a valid INTEGER4.

Unlike INTEGER and WORD, the INTEGER4 type is not considered an ordinal type. There are no INTEGER4 subranges and INTEGER4 cannot be an array index or the base type of a set. INTEGER4 values also cannot be used to control FOR and CASE statements.

INTEGER4 is currently an extended numeric type, as is REAL. Values of type INTEGER or WORD in an expression change automatically to INTEGER4 if the expression requires an intermediate value that is out of the range of either INTEGER or WORD. Values of type INTEGER4 do not change to REAL in an expression; you must explicitly use the FLOAT4 function to make the conversion. The functions ROUND4 and TRUNC4 are also available for REAL/INTEGER4 conversion.

To assign a WORD to an INTEGER4, use BYLONG instead of the ORD function, because ORD will sign-extend the sign bit of the WORD. For example:

```
Integer4Var := BYLONG (0, WordExpression);
```

The CASE Statement

The CASE statement is similar to the conditional statement in that it specifies that only one (or none) of a number of statements must be executed.

CASE and OTHERWISE are reserved words.

The syntax of the CASE statement is:

```
CASE <index> OF
  <value1>: <statement1>;
  <value2>: <statement2>;
  ..
  <valueN>: <statementN>
END
```

where

<index>
is any expression of an ordinal type.

> **CAUTION**
>
> The short integer (SINT) data type must not be used as a value for <index>. If used, results are undefined.

<value1>
can be any constant of the same type, or a list of constants separated by commas, or a subrange of the same type. The same applies to <value2>,..<valueN>.

Each constant in the type can be defined by not more than one <value>.

<statement1>
can be any one statement. (A compound statement can be used if you want more than one statement there). It can also be another conditional and case statement. This applies also to <statement2>, ... <statementN>, <statement> in the example below.

At the **extend level**, the CASE statement can also look as follows:

```
CASE <index> OF
  <value1>: <statement1>;
  <value2>: <statement2>;
  ..
  <valueN>: <statementN>;
  OTHERWISE <statement>
END
```

When the CASE statement is executed, <index> is evaluated. If <index> is equal to <value1> (or, if <value1> is a list of constants or a subrange and <index> equals one of the constants specified by <value1>), then <statement1> is executed; the rest of the statements are ignored. Control then passes to the statement following the CASE statement.

If the <index> value is equal to a constant defined by <value2>, only <statement2> is executed. (The same is true for the rest of the <value>'s.)

If <index> is not equal to any of the constants defined by the <value>'s, then one of the following occurs:

- If the OTHERWISE clause is present, <statement> is executed, and control is passed to the statement following the CASE statement. An empty OTHERWISE clause forces control to pass on to the next statement following the CASE statement.

- If the OTHERWISE clause is not present, results are either undefined or can generate a run-time error. (See the discussion of the OTHERWISE clause following the CASE example, below.)

Example:

```
VAR OPERATOR (PLUS,MINUS,TIMES);
    NEXTCH : CHAR;
BEGIN
  .
  .
CASE OPERATOR OF
  PLUS:  X := X + Y;
  MINUS: X := X - Y;
  TIMES: X := X * Y
END;
{OPERATOR is the CASE index.  PLUS, MINUS,}
{and TIMES are CASE constants.  In this}
{instance they are all of the values}
{assumable by the enumerated variable,}
{OPERATOR.}

CASE NEXTCH OF
  'A'..'Z', '_'       : WRITE ('Identifier');
  '+', '-', '*', '/'  : WRITE ('Operator');
  {Commas separate CASE constants}
  {and ranges of CASE constants.}
  OTHERWISE
    WRITE ('Unknown Character')
    {that is, if any other character}
END
```

Note that <index> cannot be an INTEGER4, since INTEGER4 is not an ordinal type.

The CASE syntax for <value1>..<valueN> is the same as for RECORD variant declarations. In standard Pascal, a CASE constant is one or more constants separated by commas. At the **extend level**, you can substitute a range of constants, such as 'A'..'Z', for a constant. No constant value can apply to more than one statement.

The **extend level** also allows the CASE statement to end with an OTHERWISE clause. The OTHERWISE clause contains additional statements to be executed in the event that the CASE index value is not in the given set of CASE constant values. One of two things happens if the CASE index value is not in the set and no OTHERWISE clause is present:

- If the range-checking switch ($RANGECK) is on, a run-time error is generated.
- If the range-checking switch is off, the result is undefined.

CAUTION

Depending on optimization, the code generated by the compiler for a CASE statement can be either a "jump table" or series of comparisons (or both). If it is a jump table, a jump to an arbitrary location in memory can occur if the control variable is out of range and the range-checking switch is off.

A semicolon (;) can appear after the final statement in the list, but is not required. The compiler skips over a colon (:) after an OTHERWISE and issues a warning.

REPETITIVE STATEMENTS

Repetitive statements specify repeated execution of a statement. In standard Pascal, these include the WHILE, REPEAT, and FOR statements.

At the **extend level**, there are two additional statements, BREAK and CYCLE, for leaving or restarting the statements being repeated. These statements are functionally equivalent to a GOTO but easier to use.

The WHILE Statement

The WHILE statement repeats a statement zero or more times, until a Boolean expression becomes false. The syntax of the WHILE statement is

```
WHILE <expression> DO <statement>
```

where

<expression>
: is any Boolean expression.

<statement>
: is any statement.

WHILE and DO are reserved words.

<statement> is executed while <expression> is TRUE, that is:

1. <expression> is evaluated.

2. If its value is FALSE, the execution of the WHILE statement is terminated and control passes to the next executable statement.

   If its value is TRUE, <statement> is executed and control returns to step 1.

This page is intentionally left blank.

This index covers both Volumes 1 and 2. Sections 1 through 12 are in Volume 1. Sections 13 through the Glossary are in Volume 2.

Page numbers in boldface indicate the principal discussion of a topic.

PASCAL REFERENCE MANUAL: VOLUME 2

Third Edition (September 1984) A–09–00852–01–A
Update Notice 1 (December 1987) 09–01363–01

This document was produced using the Document Designer Series.

LIST OF FIGURES

LIST OF TABLES

the interrupt associated with it occurs. Furthermore, INTERRUPT procedures take no parameters. (To associate an INTERRUPT procedure with an interrupt see the section entitled "Interrupt Handlers" in your operating system manual.)

Declaring a procedure with the INTERRUPT attribute ensures that the procedure conforms to the constraints of an interrupt handler in which

- a special calling sequence saves all status on the stack
- the status saved includes machine registers and flags, plus any special global compiler data such as the frame pointer
- the saved status is restored upon exit from the procedure

All INTERRUPT procedures must be nested directly within a compiland.

Interrupts are not automatically vectored to INTERRUPT procedures and are neither enabled or disabled by an INTERRUPT procedure.

This version of Pascal does not provide interrupt vectoring or enabling.

An INTERRUPT procedure should usually return normally, in order to continue processing in the interrupted routine. Therefore,

- You should not execute a GOTO that leaves an INTERRUPT procedure.
- All debug checking should be turned off (that is, $DEBUG-, $ENTRY-, and $RUNTIME-).
- Stack overflow cannot be checked even if $STACKCK is on.

The use of INTERRUPT procedures introduces reentrancy into Pascal code: generated code is reentrant, as is the run-time system (except for the heap unit and portions of the file unit.

Note that caution should be used when nonreentrant code is used in INTERRUPT procedures. For example, if the heap allocator is executing

when an interrupt occurs and the INTERRUPT procedure tries to allocate a block from the heap, the structure of the heap could become invalid. This condition causes a run-time error.

It is safest to avoid performing any I/O within the INTERRUPT procedure. Alternatively, you can avoid most problems with I/O in an INTERRUPT procedure by not opening or closing any files (that is, not declaring any local file variables or creating files on the heap) and by not performing input or output with any file that might be in the process of performing I/O when the interrupt occurs.

THE PURE ATTRIBUTE

The PURE attribute applies only to functions, not to procedures or variables. PURE indicates to the compiler's optimizer that the function does not modify any global variables either directly or by calling some other procedure or function.

Example of a PURE declaration:

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

For further illustration, examine these statements:

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

If the function FOO is given the PURE attribute, the optimizer only generates code to compute I*10 once. However, FOO, if it is not declared PURE, can modify I so that I*10 must be recomputed after the call to FOO.

Functions are not considered PURE unless given the attribute explicitly. A PURE function should not

- assign to a nonlocal variable
- have any VAR or VARS parameters (CONST and CONSTS parameters are permitted)
- call any functions that are not PURE

ABS

```
FUNCTION ABS (X: NUMERIC): NUMERIC;
```

An arithmetic function.

Returns the absolute value of X. Both X and the return value are of the same numeric type: REAL4, REAL8, INTEGER, WORD, or INTEGER4. Since WORD values are unsigned, ABS (X) always returns X if X is of type WORD.

ACSRQQ and ACDRQQ

```
FUNCTION ACSRQQ (CONSTS A: REAL4): REAL4; EXTERN;
FUNCTION ACDRQQ (CONSTS A: REAL8): REAL8; EXTERN;
```

Arithmetic functions.

Return the arc cosine of A. Both A and the return value are of type REAL4 or REAL8, as shown.

These functions are from the run-time library and must be declared EXTERN before use.

AISRQQ

```
FUNCTION AISRQQ (CONSTS A: REAL4): REAL4; EXTERN;
```

Arithmetic function.

Returns the integral part of A, truncated toward zero. Both A and the return value are of type REAL4, as shown.

This function is from the run-time library and must be declared EXTERN before use.

ALLHQQ

```
FUNCTION ALLHQQ (SIZE: WORD): WORD; EXTERN;
```

A library routine (heap management function).

Returns zero if the heap is full, one if the heap structure is in error, MAXWORD if the allocator has been interrupted. Otherwise, it returns the pointer value for an allocated variable with the size requested.

Generally, ALLHQQ is used with the RETYPE function. For example:

```
P_VAR := RETYPE (P_TYPE, ALLHQQ (28));
{RETYPE converts the value returned by}
{ALLHQQ (28) to the type P_TYPE.}
{This value is assigned to P_VAR.}

IF WRD (P_VAR) < 2 THEN GO ABORT;
{PVAR is then checked for a heap}
{full or heap structure error.}
```

ALLMQQ

```
FUNCTION ALLMQQ(wants: WORD) : ADSMEM; EXTERN;
```

Allocates a block of 'wants' bytes on the long heap and returns the block address. The block cannot be larger than 64K bytes.

This function is from the run-time library and must be declared EXTERN before use.

ANSRQQ

```
FUNCTION ANSRQQ (CONSTS A: REAL4): REAL4; EXTERN;
```

Arithmetic function.

Returns the integral part of A, which is the result of truncating the sum of A and 0.5. Both A and the return value are of type REAL4, as shown.

This function is from the run-time library and must be declared EXTERN before use.

ARCTAN

```
FUNCTION ARCTAN (X: REAL): REAL;
```

An arithmetic function.

Returns the arc tangent of X in radians. Both X and the return value are of type REAL. To force a particular precision, declare ATSRQQ (CONSTS REAL4) and/or ATDRQQ (CONSTS REAL8) and use them instead.

ASSRQQ and ASDRQQ

```
FUNCTION ASSRQQ (CONSTS A: REAL4): REAL4; EXTERN;
FUNCTION ASDRQQ (CONSTS A: REAL8): REAL8; EXTERN;
```

Arithmetic functions.

Return the arc sine of A. Both A and the return value are of type REAL8 or REAL4, as shown.

These functions are from the run-time library and must be declared EXTERN before use.

ASSIGN

```
PROCEDURE ASSIGN (VAR F : FILE OF..; CONSTS N:
STRING);
```

A file system procedure (**extend level I/O**).

Assigns an operating system filename in a STRING (or LSTRING) to a file F.

See the subsection "Extend Level Procedures" in Section 15, "File-Oriented Procedures and Functions," for a description.

ATSRQQ and ATDRQQ

```
FUNCTION ATSRQQ (CONSTS A: REAL4): REAL4; EXTERN;
FUNCTION ATDRQQ (CONSTS A: REAL8): REAL8; EXTERN;
```

See ARCTAN.

A2SRQQ and A2DRQQ

```
FUNCTION A2SRQQ (CONSTS A, B: REAL4): REAL4;
EXTERN;
FUNCTION A2DRQQ (CONSTS A, B: REAL8): REAL8;
EXTERN;
```

Arithmetic functions.

Return the arc tangent of (A/B). Both A and B, as well as the return value, are of type REAL4 or REAL8, as shown.

These functions are from the run-time library and must be declared EXTERN before use.

BEGOQQ

```
PROCEDURE BEGOQQ; EXTERN;
```

A library routine (initialization).

BEGOQQ is called during initialization, and the default version does nothing. However, you can write your own version of BEGOQQ, if for example, you want to invoke a debugger or to write customized messages to the video display, such as the time of execution.

See also ENDOQQ.

PREALLOCHEAP

```
FUNCTION PREALLOCHEAP (VARS CBALLOC: WORD): WORD;
EXTERN;
```

A library function.

Allocates short-lived memory from the operating system memory pool. This memory is unused after this call. It then can be used for the heap by heap management routines.

This preallocation is useful if your program then calls the operating system directly to allocate short-lived memory. (See the section entitled "Memory Management" in the your operating system manual for further information on memory organization and management.)

Lets you specify how much storage is to be allocated for the short heap. You can then use short-lived memory without worrying about running out of heap space.

CBALLOC Is the count of bytes to allocate for the heap

If cbAlloc is #0FFFF, the maximum possible storage is allocated for the heap

PREALLOCLONGHEAP

```
FUNCTION PREALLOCLONGHEAP (CPARA: WORD) : WORD;
EXTERN;
```

A run-time library function.

Normally, the first call to a long heap allocation routine allocates as much short-lived memory as possible for the short heap and takes all the rest of the short-lived memory for the long heap (to satisfy the current and possible future requests). To avoid the rest of the short-lived memory being allocated for the long heap, you can preallocate the short-lived memory for the long heap using PREALLOCLONGHEAP.

CPARA is the number of paragraphs (number of bytes divided by 16) to be allocated for the long heap. This procedure

- allocates as much short-lived memory as possible for the short heap
- allocates CPARA paragraphs of short-lived memory for the long heap

  If there are less than CPARA paragraphs available, all available short-lived memory is allocated.

  If CPARA = #0FFFF, then all available short-lived memory is allocated.

PRED

```
FUNCTION PRED (X: ORDINAL): ORDINAL;
```

Determines the ordinal "predecessor" to X. The ORD of the result returned is equal to ORD (X) - 1. An error occurs if the predecessor is out of range or overflow occurs. These errors are caught if appropriate debug switches are on.

This function can also be used with INTEGER4.

PRSRQQ and PRDRQQ

```
FUNCTION PRSRQQ (CONSTS A, B: REAL4): REAL4;
EXTERN;
FUNCTION PRDRQQ (CONSTS A, B: REAL8): REAL8;
EXTERN;
```

Arithmetic functions.

The return value is A**B (A to the REAL power of B). Both A and B are of type REAL4 or REAL7, as shown. An error occurs if A < 0 (even if B happens to have an integer value).

These functions are from the run-time library and must be declared EXTERN before use.

PTYUQQ

```
PROCEDURE PTYUQQ (LEN: WORD; LOC: ADSMEM); EXTERN;
```

A library routine (terminal I/O).

Writes LEN characters, beginning at LOC in memory, to the video display.

Example:

```
PTYUQQ (8, ADS 'PROMPT: ');
```

Together with GETYQQ and PLYUQQ, PTYUQQ is useful for doing terminal I/O in a low-overhead environment. These functions are part of a collection of routines called Unit U, which implements the Pascal file system.

PUT

```
PROCEDURE PUT (VAR F : FILE OF ..);
```

A file system procedure.

Writes the value of the file buffer variable F^ to the currently pointed-to component of F and advances the file pointer.

See the subsection "GET and PUT" in Section 15, "File-Oriented Procedures and Functions," for a description.

READ

```
PROCEDURE READ (VAR F : FILE OF ..; P1, P2, ..
PN);
```

A file system procedure.

READ reads data from files. Both READ and READLN are defined in terms of the more primitive operation, GET.

See the subsection "Textfile Input and Output" in Section 15, "File-Oriented Procedures and Functions," for a description.

READFN

```
PROCEDURE READFN (VAR F : FILE OF ..; P1, P2, ..
PN);
```

A file system procedure (**extend level I/O**).

READFN is the same as READ (not READLN) with two exceptions:

- File parameter F should be present (INPUT is assumed but a warning is given.)
- 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 in the READ procedure.

See the subsection "Extend Level Procedures" in Section 15, "File-Oriented Procedures and Functions," for a description.

SUCC

```
FUNCTION SUCC (X: ORDINAL): ORDINAL;
```

A data conversion function.

Determines the ordinal "successor" to X. The ORD of the returned result is equal to ORD (X) + 1. An error occurs if the successor is out of range or overflow occurs. These errors are caught if appropriate debug switches are on.

This function can also be used with INTEGER4.

THSRQQ and THDRQQ

```
FUNCTION THSRQQ (CONSTS A: REAL4): REAL4; EXTERN;
FUNCTION THDRQQ (CONSTS A: REAL8): REAL8; EXTERN;
```

Arithmetic functions.

Return the hyperbolic tangent of A. Both A and the return value are of type REAL4 or REAL8, as shown.

These functions are from the run-time library and must be declared EXTERN before use.

TNSRQQ and TNDRQQ

```
FUNCTION TNSRQQ (CONSTS A: REAL4): REAL4; EXTERN;
FUNCTION TNDRQQ (CONSTS A: REAL8): REAL8; EXTERN;
```

Arithmetic functions.

Return the tangent of A. Both A and the return value are of type REAL4 or REAL8, as shown.

These functions are from the run-time library and must be declared EXTERN before use.

TRUNC

```
FUNCTION TRUNC (X: REAL): INTEGER;
```

An arithmetic function.

Truncates X toward zero. X is of type REAL4 or REAL8, and the return value is of type INTEGER.

Examples

```
TRUNC (1.6) is 1
TRUNC (-1.6) is 1
```

Error message 2136, REAL Indefinite, is reported if ABS (X - 1.0) >= MAXINT. (See Appendix A for a description of the compiler error messages.)

TRUNC4

```
FUNCTION TRUNC4 (X: REAL): INTEGER4;
```

An arithmetic function.

Truncates real X toward zero. X is of type REAL4 or REAL8, and the return value is of type INTEGER4.

Examples:

```
TRUNC4 (1.6) is 1
TRUNC4 (-1.6) is -1
```

An error occurs if ABS (X -1.0) >= MAXINT4.

File variables with the STATIC attribute in procedures and functions are also closed automatically when the procedure or function returns. Files allocated statically at the program, module, or implementation level are automatically closed when the entire program terminates.

If necessary, when a CLOSE is executed, a file being written to has its operating system buffers flushed. However, the buffer variable is not PUT. If a file of type TEXT is being written and the last nonempty line does not end with a line marker, one is added to the end of the last line. If the file has the mode SEQUENTIAL and is being written, an end-of-file is written.

Note that some run-time errors may remove control from the run-time system. In these cases, files being written may not be closed, and the information in them may be lost. 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);

A file system procedure (**extend level I/O**).

Closes and deletes an open file. DISCARD is much like CLOSE except that the file is deleted.

PROCEDURE READFN (VAR F; P1, P2, .. Pn);

A file system procedure (**extend level I/O**).

READFN is the same as READ (not READLN) with two exceptions:

- File parameter F should be present (INPUT is assumed, but a warning is given if F is omitted).
- If a parameter P is of type FILE, a sequence of characters forming a valid file name 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 is like READ, not like READLN, and does not read the trailing line marker. If the first parameter in a READFN call is a file of any type, it is assumed to be the textfile from which characters are read. 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 useful when reading a file name and assigning the file name to a file in one operation.

PROCEDURE READSET
(VAR F; VAR L; LSTRING, CONST S: SETOFCHAR);

A file system procedure (**extend level I/O**).

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, formfeeds, and line markers are always skipped.

Reading ceases at the first line marker, which is never in the type CHAR.

This example assumes the program was invoked with a command form such as:

```
Type
  File name_____________________________________
```

(For details of how to construct command forms, see the New Command command in the Executive Manual.)

Also note that the workstation operating system has more sophisticated parameter management facilities than those offered by the Pascal run-time, and can be used to access all subparameters entered in the command form, as well as the command name itself. (See the section entitled "Parameter Management" in your operating system manual for details.)

MODULES

Modules provide a simple, straightforward method for combining several compilable segments into one program. Units, described in the next subsection, provide a more powerful and structured method for achieving the same end.

Basically, a module is a program without a body. The identifier in the module heading has the same scope as a program identifier. The heading can also include attributes that apply to all procedures and functions in the module. There are no module parameters; nor is there a module body. A module ends with the reserved word END and a period.

Example of a module:

```
MODULE BETA [PUBLIC];     (optional attributes)

PROCEDURE GAMMA;
  BEGIN WRITELN ('Gamma') END;

FUNCTION DELTA: WORD;
  BEGIN DELTA := 123 END;

END.                          (no body before END)
```

Table 17-2. Metacommands. (Page 3 of 3)

Metacommand Name	Function
$RANGECK	Checks for subrange validity.
$REAL	Sets the length of the REAL type.
$ROM	Gives a warning on static initialization.
$RUNTIME	Determines the context of run-time errors.
$SIMPLE	Disables global optimizations.
$SIZE	Minimizes the size of code generated.
$SKIP	Skips a specified number of lines or skips to end of page.
$SPEED	Minimizes the execution time of code.
$STACKCK	Checks for stack overflow at procedure or function entry.
$SUBTITLE	Sets the page subtitle.
$SYMTAB	Sends the symbol table to the listing file.
$TITLE	Sets the page title.
$WARN	Gives a warning message in the listing file.

OPTIMIZATION LEVEL

The metacommands shown in Table 17-3 let you control the degree to which optimization is used.

Table 17-3. Optimization Level.

Name	Description
$INTEGER:<n>	Sets the length of the INTEGER type (default is 2.)
$REAL:<n>	Sets the length of the REAL type.
$ROM-	Gives an error on static initialization.
$SIMPLE	Disables global optimizations.
$SIZE	Minimizes the size of code generated. $SIZE is the default setting.
$SPEED	Minimizes the execution time of code.

$INTEGER and $REAL set the length (that is, precision) of the standard INTEGER and REAL data types. $INTEGER can only be set to 2 (the default), for 16-bit integers. However, you can set $REAL to 4 (the default) or 8, to make type REAL identical to REAL4 or REAL8, respectively.

The $SIMPLE metacommand turns off common subexpression optimization. $SIZE and $SPEED currently turn it back on again. $SIZE, $SPEED, and $SIMPLE are all mutually exclusive. The default is $SIZE.

18 USING THE PASCAL COMPILER

You run a Pascal program by first compiling its one or more source modules, using the Linker to link the resulting object files with the Pascal library, and invoking the resulting run file. The run file is usually invoked through the Executive.

The Pascal compiler translates your Pascal source programs into object modules. The compiler provides a source listing, error messages, and a number of compiler metacommands to aid in program development and debugging.

The compiler comes with a set of object libraries to be linked with your code. These libraries provide complete run-time support for input/output, arithmetic functions, and inline code execution by the optional numeric coprocessor that is available with some workstations or with the optional Math service. When you link your program, the Linker automatically accesses these libraries when necessary. (The run-time libraries are discussed in Section 19, "Run Time and Debugging.")

Using the Linker, you can also combine Pascal object modules with those of other languages, for example FORTRAN, to facilitate writing applications that need different languages for different parts.

Pascal supports systems programming by providing access to all operating system services, such as direct (random) access to disk files, interrupt handling, and process creation. Calls also extend the range of services needed by the commercial application programmer: DAM, ISAM, Sort/Merge, and the Forms Run Time.

COMPILING, LINKING, AND RUNNING PASCAL: OVERVIEW

To create and execute a Pascal program,

1. Create and edit the source file. You can use the Editor or the Word Processor to create the source file.

2. Compile the program. The compiler flags syntax errors as it reads your source file. You can place compiler controls called metacommands within your program to generate diagnostic calls for run-time errors. If compilation is successful, the compiler creates a relocatable object file.

3. Use the Linker's Bind command to link compiled object files with the run-time library. A compiled object file is not executable and must be linked with one or more run-time libraries, using the Linker. Separately compiled subroutines in other languages or assembly language programs can also be linked to your program at this time. The Linker produces an executable file called a version 6 run file.

4. Use the Executive Run command to execute the resulting run file. (Alternatively, you can use the Executive command New Command to create a special command that you can use to execute your run file.)

Repeat this process until your program has successfully compiled, linked, and run without errors.

Since compiler metacommands can slow your program down, once the program runs without errors, remove or comment out any metacommands that are no longer necessary, then recompile, relink, and rerun your program.

Pass Two of the compiler produces the object file. When it is complete a message similar to the one below is displayed:

```
Code Area Size = #05EC     ( 1516)
Cons Area Size = #00E6     (  230)
Data Area Size = #0264     (  612)

Pass Two   No Errors Detected
```

The first three lines indicate, first in hexadecimal and then in decimal notation, the amount of space taken by executable code (Code), constants (Cons), and variables (Data). The number of errors given is for Pass Two only.

The third pass produces the object list file and is executed only if you request one.

For a more detailed discussion of the compiler see the subsection "Compiler Structure" below.

LINKING A PASCAL PROGRAM

After the Pascal modules are compiled, you must link the resulting object modules with the Linker to produce an executable version 6 run file.

The Linker is invoked through the Executive, by typing "Bind" (or as many letters as required to make the command unique) into the Executive command form. The following form is displayed:

```
Bind
  Object modules          _____________________
  Run file                _____________________
  [Map file]              _____________________
  [Publics?]              _____________________
  [Line numbers?]         _____________________
  [Stack size]            _____________________
  [Max array, data, code] _____________________
  [Min array, data, code] _____________________
  [Run file mode]         _____________________
  [Version]               _____________________
  [Libraries]             _____________________
  [DS allocation?]        _____________________
  [Symbol file]           _____________________
```

Using the Linker and completing each of the fields of the Bind form are discussed in detail in the Linker/Librarian Manual. The following special features of the Linker are important for use with Pascal:

Object modules

Enter the object file name, PasFirst.obj. PasFirst.obj is an assembly language module included with Pascal 10.0 software that ensures a successful link. Following this module name, enter the name(s) of any other object modules you want linked. Leave a space between each object file specification. (If you have too many entries to fit on the command line, you must place the entries in a file, called an at-file, then place the file name on the command line prefixed by an at sign (@). The use of at-files is discussed in the Executive Manual.)

If your program includes run-time overlays, you must include the file [Sys]<Sys>PasSwp.obj in this field.

If your program includes floating point calculations and you have a numeric coprocessor installed on your workstation, you can include the entry @[Sys]<Sys>Pascal8087.fls. If you do not have a numeric coprocessor installed on your workstation, you can install the Math service. (See the Math service Release Notice for installation details.) When installed, this service emulates the coprocessor. (If you do not have either a numeric coprocessor or the Math service installed and your program uses floating point constants or variables, the numeric coprocessor emulator is automatically linked to your programs.)

If you are linking a minimal Pascal program, include the file [Sys]<Sys>Pasmin.obj. Minimal Pascal is discussed in the subsection "Avoiding the Run-time Library" in Section 19, "Run time and Debugging."

[Stack size] Default for Pascal: 8K

If you wish to change stack size, specify the desired size here.

[Run file mode]

The default for this field is real. If you choose to have your program execute in protected mode, you must select from the other options for this field. The Linker/Librarian Manual describes each of these options in detail. Note also that if your program is to execute in protected mode, it must be written according to the guidelines for compatible programs. These are

described in the Engineering Update for CTOS/VM 2.0. (For additional details on real and protected mode, see the Linker/Librarian Manual.)

[Libraries] When linking a Pascal program, the Linker automatically searches the library [Sys]<Sys>Pascal.Lib (if it exists) for any unresolved external symbols. The library [Sys]<Sys>CTOS.Lib is also searched.

You can specify any additional libraries you wish, for example, if you are linking with subprograms written in other languages, then the libraries for those languages must be specified.

If you are linking a program from which you wish to exclude any references to the Pascal run-time library, it is recommended that you specify the libraries that you do wish to link with your program followed by the word none. For example,

```
[Libraries] [Sys]<Sys>CTOS.Lib none
```

In this case, if any calls are made to the run time, the Linker indicates an unresolved external.

This page is intentionally left blank.

[DS Allocation?] Default for Pascal: Yes

This field is used to minimize the run-time value of DS (the data segment register) by offsetting all references to group DGroup.

Group DGroup consists of 64K bytes or less allocated for constants, data, and stack.

If you specify "Yes", the default, then the entire 64K bytes can be used for your data if necessary. The Code Segment is loaded at the high end of memory, above the data segment. Relative addressing starts at the highest word, no matter how much space is really needed for DGroup.

If you specify "No," however, the data segment takes only the amount of space actually needed and is loaded at the high end of memory, with the Code Segment below it.

For example, if your program uses only 32K of data and you specify "Yes" for [DS Allocation?] then the address of DS is DS:FFFF, whereas if you specify "No" the address is DS:$00032K. Figure 18-1 illustrates this.

Most Pascal applications require [DS Allocation?] to be "Yes." Object module procedures and tasks produced by the Pascal compiler use a single value in DS during their entire execution, and include the group DGroup with DS equal to DGroup. This feature must be used for linking Pascal tasks that make use of the Pascal heap.

Run files linked using Pasmin.obj can have DS Allocation set to either "Yes" or "No."

The third pass, performed by the file [Sys]<Sys>PascalLst.Run, produces the object listing file and is only invoked if you specifically request an object listing when you complete the Pascal command form. During the third pass the files Pasibf.Tmp and Pasibf.Oid are deleted.

All intermediate files contain Pascal records. A common constant and type definition file is used called Pascom.nnm, which defines the intermediate code and symbol table types. A similar file is used during the second and third passes for the intermediate binary file definition.

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 another value, 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, etc.

The symbol table record is complex, with a variant for every kind of identifier (such as, assorted data types, variables, procedures, and functions.)

The compiler itself takes memory, and in addition needs memory for its internal tables. It puts some of these tables into the long heap, the others into the short heap. The long heap is limited only by the computer memory. Exact size of the compiler and memory requirements for the short heap and the compiler stack are detailed in the Pascal Release Notice.

A compilation can sometimes terminate abnormally on the first pass with the error message 'Compiler Out of Memory.' This message usually indicates stack/heap space overflow. Examples of information that is stored in the short heap are: PUBLIC and EXTERN declarations, and TYPE declarations. Reducing the number of the these declarations in your program, can help it to compile successfully. Note, however, that these declarations do not affect the size of the program when it runs. They only affect memory requirements during compilation.

VIRTUAL CODE MANAGEMENT FACILITY

Pascal is compatible with the Virtual Code Management facility. The Virtual Code Management facility is described in detail in the section entitled "Virtual Code Management" (or "Virtual Code Segment Management") in your operating system manual.

As with all applications that use the Virtual Code Management facility, the swap buffer must be allocated and initialized before any overlay is called. You can overlay both portions of the Pascal run-time system and portions of your own program.

To include portions of the run-time system in overlays, the following are necessary:

- Include PasSwp.obj in the Object modules field of the Linker command form.
- Write a procedure called BEGOQQ to perform user initialization. These procedures must be included (in the Object Modules field of the Linker command form) when the Pascal application is linked.

Pascal provides an empty procedure, BEGOQQ as an entry point. You can use it to initialize a swap buffer before any Pascal run-time initialization takes place. You must allocate and initialize the swap buffer in BEGOQQ to ensure that the swap buffer is ready when the Pascal run-time system is invoked.

For example:

```
module misoqq[];

Type adsw = ads of word;

(* InitOverlays is a CTOS function
initializing a swap
 buffer.
*)

Function InitOverlays(pBuf:adsw;cb:word):word;
            extern;
Procedure CheckErc(erc:word); extern;
```

19 RUN TIME AND DEBUGGING

The run-time support libraries contain object modules that can be linked to your program to satisfy unresolved external references. When your Pascal program is linked, the library files [Sys]<Sys>Pascal.Lib and [Sys]<Sys>CTOS.Lib are automatically searched and the appropriate modules are linked to it if necessary.

The run-time support libraries provide all input/output (I/O) support needed to run your programs on your system. If you choose to use floating-point software routines all required arithmetic and interface software is also provided by the run-time libraries. If you have a numeric coprocessor installed on your system, you can specify a special library at link time to take advantage of this chip for your floating-point routines. (See Section 18, "Using the Pascal Compiler," for more information on how to link your program.)

OVERVIEW OF THE PASCAL RUN TIME

Run-time routines linked to a Pascal program are described briefly below. Pascal run-time routines all have six character names and end in the suffix QQ. Run-time routines are discussed in detail in the subsection "Run-Time Architecture" below.

The run file produced by the Linker for a Pascal program has the entry point BEGXQQ, which is a routine written in assembly language. This routine sets the initial stack pointer, the starting address of the heap, and various other routine variables. There is also a call to initialize the Pascal file system. Finally, there is a call to the Pascal program, which is always given the name ENTGQQ.

The Pascal main program continues the initialization process. Every unit mentioned in a USES clause in any interfaces or in the program is initialized by calling it as a procedure, in the order of the USES clauses. Any files declared in the program are initialized by calling NEWFQQ for each one. Finally, any program parameters are read and assigned to their variables, and the actual program code begins.

When the program terminates, the call to ENTGQQ returns to procedure BEGXQQ, which calls ENDXQQ. The Pascal file system is then called to close all open files and to discard all temporary files. A call to Exit in the operating system terminates the program.

Inside a Pascal application, many calls are also made to the Pascal run time to accomplish tasks too complicated to be done by straight generated code. For example, most error checking is accomplished by calling run-time helpers. You can identify these calls by their names: all run-time routines have six character names ending in QQ.

Note that run-time routines are not reentrant. Therefore, if one application creates several processes that execute concurrently a piece of code written in Pascal, care must be taken that only one of them is executing Pascal run-time code at any one time.

All CTOS facilities are available for use from Pascal. Interfaces to routines are described in your operating system manual and examples of the use of the operating system from Pascal are given in this manual in Appendix F, "Using Pascal as a Systems Programming Language."

DEBUGGING

Pascal programs may be run under the control of the Debugger. (Note that the term Debugger here does not refer to Pascal error handling routines, but to the Debugger available with the standard software for your workstation.) To pass control to the Debugger, use CODE-GO rather than GO when you invoke your program. Using the Debugger is described in detail in the Debugger Manual.

The use of symbol files and object list files is very helpful in the debugging of Pascal programs.

The symbol file gives you the addresses of public variables for your program. The symbol file is created by the Linker when your program is linked. The name of the symbol file has the extension ".Sym".

The entry point into the main program is ENTGQQ (a public variable).

The object list file is a symbolic assembler-like listing of the object code that lists addresses of the instructions relative to the start of the program or module.

The example below shows code from an object list file for the Pascal statement i := i+1; where i is an integer.

```
L5:
     **  000011      MOV         AX,I
     **  000014      INC         AX
     **  000015      MOV         I,AX
```

The L5 indicates that this statement is on line 5 of the program. The numbers on the left side of the code indicate the hexadecimal offset from the beginning of the code segment for the particular instruction. For example, the MOV AX,I instruction begins at CS:11, where CS is the current code segment address.

RUN-TIME ROUTINES

The Pascal run-time entry point and variable names all have six characters, the last three of which consist of a unit identification letter followed by the letters "QQ".

Table 19-1 shows the current unit identifier suffixes.

Table 19-1. Unit Identifier Suffixes. (Page 1 of 2).

Suffix	Unit Function
AQQ	Reserved
BQQ	Compile time utilities
CQQ	Encode, decode
DQQ	Double precision real
EQQ	Error handling
FQQ	Pascal file system (Unit F)
GQQ	Generated code helpers
HQQ	Heap allocator
IQQ	Reserved
JQQ	Reserved
KQQ	FCB definition
LQQ	STRING, LSTRING
MQQ	Reserved
NQQ	Reserved

Table 19-1. Unit Identifier Suffixes. (Page 2 of 2).

Suffix	Unit Function
PQQ	Reserved
RQQ	Real (single precision)
SQQ	Set operations
TQQ	Reserved
UQQ	Operating system file system
VQQ	Reserved
WQQ	Reserved
XQQ	Initialize/terminate
YQQ	Special utilities
ZQQ	Reserved

MEMORY ORGANIZATION

Memory on the CPU is divided into segments, each containing up to 64K bytes. The Linker also puts segments into classes and groups. All segments with the same class name are loaded next to each other. All segments with the same group name must reside in one area up to 64K bytes long; that is, all segments in a group can be accessed with one segment register.

Pascal uses the medium model of computation, that is, it uses multiple code segments, but only one data segment, called DGroup. Memory is allocated within DGroup for all static variables, constants that reside in memory, the stack, and the short heap.

DGroup is addressed using the DS (current data) or SS (current stack) segment register. Normally, DS and SS contain the same value, although DS may be changed temporarily to some other segment and changed back again. SS is never changed; its segment registers always contain abstract "segment values" and the contents are never examined or operated on. Long addresses, such as ADS variables, use the ES segment register for addressing.

Memory in DGroup is normally allocated from the top down; that is, the highest addressed byte has DGroup offset 65535, and the lowest allocated byte has some positive offset. This allocation means offset zero in DGroup may address a byte in the code portion of memory, in the operating system below the code, or even below absolute memory address zero. (In the latter case the values in DS and SS are "negative.")

DGroup has two parts:

- a fixed-length upper portion containing static variables and constants
- a variable-length lower portion containing the heap and the stack

After your program is loaded, during initialization (in ENTXQQ), the fixed upper portion is placed as high as possible to make room for the lower portion. If there is enough memory, DGroup is expanded to the full 64K bytes; if there is not enough room for this, it is expanded as much as possible.

Figure 19-1 illustrates memory organization as described above.

Note that memory organization appears differently than as shown in Figure 19-1, if, when you link your program, you set the field "[DS Allocation?] to "No." In that case the Data segment is not expandable and is loaded above the Code segment. (See the subsection "Linking Your Program" in Section 18, "Using the Compiler," for an explanation of DS Allocation.)

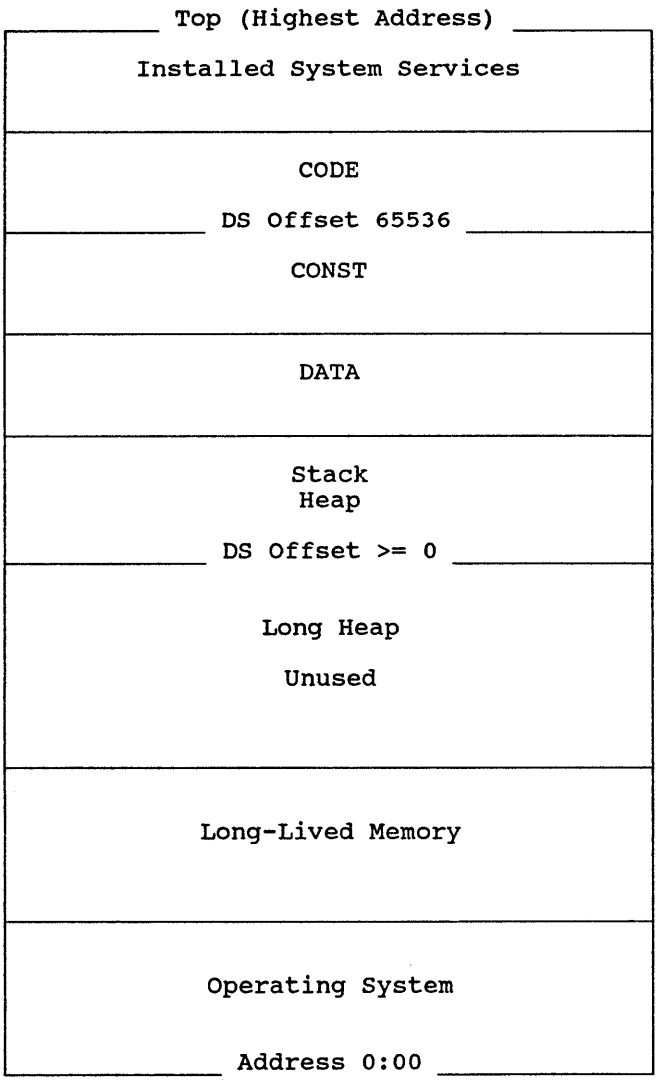


Figure 19-1. Memory Organization, Single Partition Operating System.

INITIALIZATION AND TERMINATION

Every executable file contains one, and only one, starting address. As a rule, when object modules are involved, this starting address is at the entry point BEGXQQ in the module PASMAX. A program (as opposed to a module or implementation) has a starting address at the entry point ENTGQQ. BEGXQQ calls ENTGQQ.

The following discussion assumes that a main program along with other object modules is loaded and executed. However, you can also link a main program in assembly or some other language with other object modules. In this case, some of the initialization and termination done by the PASMAX module may need to be done elsewhere.

When a program is linked with the run-time library and execution begins, several levels of initialization are required. The levels, in the order in which they occur, are the following:

- machine-oriented initialization
- run-time initialization
- program and unit initialization

The general scheme is shown in Table 19-2.

Table 19-2. Pascal Program Structure.

PASMAX module

```
ENDXQQ: (Aborts come here)
        Call ENDOQQ
        Call ENDYQQ
        Call ENDUQQ
        Call ENDX87
        Exit to operating system

BEGXQQ: Set stack pointer, frame pointer
        Initialize PUBLIC variables
        Set machine-dependent flags,
        registers, and other values
        Call INIX87
        Call INIUQQ
        Call BEGOQQ
        Call ENTGQQ (Execute program)
        Call ENDXQQ (Termination)
```

INTR module

```
INIX87: Real processor initialization

ENDX87: Real processor termination
```

UNIT U module

```
INIUQQ: Operating system specific file unit
        initialization

ENDUQQ: Operating system specific file unit
        termination
```

MISO module

```
BEGOQQ: (Available for other user
        initialization procedures)

ENDOQQ: (Available for other user
         termination procedures)
```

Program module

```
ENTGQQ: Call INIFQQ
        If $ENTRY on, CALL ENTEQQ
        Initialize static data
        Initialize units
        FOR program parameters DO
          Call PPMFQQ
        Execute program
        If $ENTRY on, CALL EXTEQQ
```

Machine Level Initialization

The entry point of a load module is the routine BEGXQQ, in the module PASMAX. BEGXQQ does the following:

- Initializes constant and static variables. The initial stack pointer is put into PUBLIC variable STKBQQ and is used to restore the stack pointer after an interprocedure GOTO to the main program.

- Sets the frame pointer (that is, the pointer to the current procedure) to zero.

- Initializes a number of PUBLIC variables to zero or NIL. These include

  - RESEQQ, a machine error context

  - CSXEQQ, a source error context list header

    PNUXQQ, an initialized unit list header

  - HDRFQQ, an open file list header

- Sets machine dependent registers, flags, and other values.

- Sets the short heap control variables. BEGHQQ and CURHQQ are set to the lowest address for the heap: 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 initial heap is empty.) The stack and the heap grow together, and the PUBLIC variable STKHQQ is set to the lowest legal stack address (ENDHQQ, plus a safety gap).

  The long heap is initialized when the user calls a long heap routine.

- If the program uses REAL numbers, calls INIX87, the real processor initializer. This routine initializes the numeric coprocessor or sets numeric coprocessor emulator interrupt vectors, as appropriate.

- Calls INIUQQ, the file unit initializer. If the file unit is not used and you do not want it loaded, a dummy INIUQQ routine that only returns must be loaded. Pasmin.Obj provides an empty INIFQQ instead of calling INIUQQ.

- Calls BEGOQQ, the escape initializer. In a normal load module, an empty BEGOQQ that only returns is included. However, this call provides an escape mechanism for any other initialization. For example, it could initialize tables for an interrupt driven profiler or to initialize overlay management.

  If you want a nonempty initialization, you must write your own BEGOQQ routine. (See Appendix F, "Using Pascal as a Systems Programming Language," for an example of a module that uses BEGOQQ to allocate and initialize a swap buffer.)

- Calls ENTGQQ, the entry point of your program.

- Calls ENDXQQ, the termination procedure.

Program Level Initialization

Your main program continues the initialization process. First, the file system, a parameterless procedure called INIFQQ, is called. If you link your program with Pasmin.Obj, an empty INIFQQ is provided.

After the file initialization, if the metacommand $ENTRY is on during compilation, ENTEQQ is called to set the source error context. Next, each file at the program level gets an initialization call to NEWFQQ.

After static data initialization comes unit initialization. Every USES clause in the source, including those in INTERFACEs, generates a call to the initialization code for the unit.

Units may or may not contain initialization code. If the interface contains a trailing pair of BEGIN and END statements, initialization code in the implementation is presumed. Units are initialized in the order that the USES clauses are encountered.

Finally, any program parameters are read or otherwise initialized, and your program begins. Except for INPUT and OUTPUT, PPMFQQ is called for each parameter 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, assigning it to the parameter. The parameter's identifier is passed to PPMFQQ for use as a prompt. PPMFQQ first calls PPMUQQ to get the text of any parameters from the command form. If PPMUQQ returns an error, then PPMFQQ does the prompting and reads the response directly.

User unit initialization is much like user program initialization. The following actions occur:

- error context initialization, if $ENTRY metacommand was on during compilation
- variable (file) initialization
- unit initialization for USES clause
- user unit initialization

Calls to initialize a unit can come from more than one unit. The unit interface has a version number, and each 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. This has the effect of

```
IF INUXQQ (useversion, ownversion, intrec,
           unitid)
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 "useversion" to INUZQQ. The interface version number in the implementation itself is passed as "ownversion" to INUXQQ. INUXQQ generates an error if the two are unequal.

INUZQQ also maintains a list of initialized units. INUXQQ returns true if the unit is found in the list, or else puts the unit in the list and returns false. The list header is PNUXQQ. A list entry passed to INUXQQ as "initrec" is initialized to the address of the unit's identifier (unitid) plus a pointer to the next entry.

User modules (and uninitialized implementations of units) may have initialization code, much like a program and unit implementation's initialization code, but without user initialization code or INUXQQ calls.

The initialization call for a module or uninitialized unit cannot be issued automatically. When the module is compiled, a warning is given if an initialization call is 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.

Program Termination

Program termination occurs in one of three ways:

- The program may terminate normally, in which case the procedure ENDXQQ is called.

- The program may abort because of an error condition, either with a user call to ABORT or a run-time 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 as an external procedure and called directly.

ENDXQQ first calls ENDOQQ, the escape terminator, which normally just returns to ENDXQQ. Then ENDXQQ calls ENDYQQ, the generic file system terminator. ENDYQQ closes all open Pascal files, using the file list headers HDRFQQ and HDRVQQ. ENDXQQ calls ENDUQQ, the file unit terminator.

Finally, ENDXQQ calls ENDX87 to terminate the real number processor (numeric coprocessor or emulator.) As with INIUQQ and INIFQQ, if your program requires no file handling, you can declare empty parameterless procedures for ENDYQQ and ENDUQQ. The main initialization and termination routines are in module PASMAX. Procedure BEGOQQ is in the module MISOALT1; ENDUQQ is in RICUQQ; and ENDYQQ is in MISY.

Using the Initialization and Termination Points in Your Program

The routines BEGOQQ and ENDOQQ are provided by the run-time library as entry points for you to use. The program example that follows uses these entry points to display the date and time.

```
{$debug-}

Program UserInitAndTermination (Output);
    {This program sample describes how to use the
    initialization and termination entry points
    that the run time provides for the user.

    The nubs provided for initialization and
    termination are labeled 'BEGOQQ' and 'ENDOQQ'
    respectively.

    Since these entry points are defined by the
    run-time library, this compiland must be
    linked with Pascal.Lib.}


Type
    pbType = ads of word;
    DateTimeType = array [1..2] of word;
    ExpDateTimeType = array [1..4] of word;


Var [public]
    lsDateTime :lstring(30);
    DateTime   :DateTimeType;
    ExpDateTime:ExpDateTimeType;
```

```
{Definition of CTOS externals to be used:}

Var [extern]
    bsVid:array [1..130] of byte;
    {'bsVid' is an open video bytestream declared
    in CTOS.Lib}

Function FormatTime
    (plsDateTimeRet:pbType;
    pExpDateTime:pbType ) :word; extern;

Function GetDateTime
    (pDateTimeRet:pbType ) :word; extern;

Function ExpandDateTime
    (dateTime  :DateTimeType;
    pExpDateTime:pbType )   :word; extern;

Function WriteBsRecord (
    pBswa       :pbType;
    pbRec       :pbType;
    cbRec       :word;
    pbCbRet     :pbType)  :word; extern;

Procedure CheckErc
    (erc        :word );      extern;

Procedure BEGOQQ[public];
        var cbRet :word;
        begin
        {This procedure will be called by the run-
        time initialization.  It will display a
        banner with the date/time}
        CheckErc (GetDateTime (ads DateTime));
        CheckErc (ExpandDateTime (DateTime, ads
              ExpDateTime));
        CheckErc (FormatTime (ads lsDateTime, ads
              ExpDateTime));
        CheckErc (WriteBsRecord (ads bsVid,
              ads 'Program initialization at ',
              26, ads cbRet));
        CheckErc (WriteBsRecord (ads bsVid,
              ads lsDateTime[1], lsDateTime.len,
              ads cbRet));
        CheckErc (WriteBsRecord (ads bsVid, ads
              #0a, 1, ads cbRet));
        end;
```

```
Procedure ENDOQQ[public];
    var cbRet :word;
    begin
        {This procedure will be called by the run-
        time termination before the first
        executable statement of the program. It
        will display a banner with the date/time.
        Note that if the CTOS calls 'Exit' or
        'ErrorExit' are used the run-time
        termination is circumvented.}
    CheckErc (GetDateTime (ads DateTime));
    CheckErc (ExpandDateTime (DateTime, ads
        ExpDateTime));
    CheckErc (FormatTime (ads lsDateTime, ads
        ExpDateTime));
    CheckErc (WriteBsRecord (ads bsVid, ads
        'Program termination at ', 23, ads
        cbRet));
    CheckErc (WriteBsRecord (ads bsVid, ads
        lsDateTime[1], lsDateTime.len, ads
        cbRet));
    CheckErc (WriteBsRecord (ads bsVid, ads #0a,
        1, ads cbRet));
    end;

begin{start of program, after run-time
    initialization}
    Writeln;
    Writeln ('Hello');
    Writeln;
end.
```

ERROR HANDLING

Run-time errors are detected in one of four ways:

- The user program calls EMSEQQ (that is, ABORT).
- A run-time routine calls EMSEQQ.
- An error checking routine in the error module calls EMSEQQ.
- An internal helper routine calls an error message routine in the error unit which, in turn, calls EMSEQQ.

Handling an error detected at run-time usually involves identifying the type and location of the error and then terminating the program. The error type has three components

- a message
- an error number (Pascal error code)
- an error status code (operating system return code)

The message describes the error and the number can be used to look up more information. The error status value is undefined, although for file system errors it may be an operating system return code. However, the error status value may also be used for other special purposes. Table 19-3 shows the general scheme for error code numbering.

An error location has two parts:

- machine error context
- source program context

The machine error context is the program counter, stack pointer, and stack frame pointer at the point of the error. The program counter is always the address following a call to a run-time routine (for example, a return address.)

Table 19-3. Error Number Classification.

Range	Classification
1- 999	Reserved for user ABORT calls
1000-1099	Unit U file system errors
1100-1199	Unit F file system errors
1200-1299	Unit V file system errors
1300-1999	Reserved
2000-2049	Heap, stack, memory
2050-2099	Ordinal and long integer arithmetic
2100-2149	Real and double real arithmetic
2150-2199	Structures, sets, and strings
2200-2399	Reserved
2400-2449	Unused
2450-2499	Other internal errors
2500-2999	Reserved

The source program context is optional; it is controlled by metacommands. If the $ENTRY metacommand is on the program context consists of

- the source file name of the compiland containing the error
- the name of the routine in which the error occurred (program, unit, module, procedure, or function)

- the line number of the routine in the listing file
- the page number of the routine in the listing file

If the $LINE metacommand is also on, the line number of the statement containing the error is also given. Setting $LINE also sets $ENTRY.

Machine Error Context

By default, run-time routines are compiled with the $RUNTIME metacommand set. This generates special calls for each run-time routine at the entry and exit points so that, for any error that occurs in a run-time routine, the location of that error is in the user program. The entry call, BRTEQQ, saves the context (frame pointer, stack pointer, and program counter) at the point where the run-time routine is called by the user program. The exit call restores the context. The run-time entry helper, BRTEQQ, uses the run-time values shown in Table 19-4.

Table 19-4. Run-Time Values in BRTEQQ.

Value	Description
RESEQQ	Stack pointer
REFEQQ	Frame pointer
REPEQQ	Program counter offset
RECEQQ	Program counter segment

The first thing that BRTEQQ does is examine RESEQQ. If this value is not zero, the current run-time routine was called from another run-time 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. 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 run-time exit helper, ERTEQQ, has no parameters. It determines the caller's stack pointer (again, from the frame pointer) and compares it against RESEQQ. If these values are equal, the original run-time routine called by your program is returning, so RESEQQ is set back to zero.

EMSEQQ uses RESEQQ, REFEQQ, REPEQQ and RECEQQ to display the machine error context.

Source Error Context

Giving the source error context involves extra overhead, since source location data must be included in the object code in some form. This is done with calls that set the current source context as it occurs. These calls can also be used to break program execution as part of the debug process. The overhead of source location data, especially line number calls, can be significant. Routine entry and exit calls, while requiring more overhead individually, are much less frequent, so the overall overhead is less.

The procedure entry call to ENTEQQ passes two VAR parameters: the first is a LSTRING containing the source file name; the second is a record that contains the following:

- the line number of the procedure (a WORD)
- the page number of the procedure (a WORD)
- the procedure or function identifier (an LSTRING)

The file name is that of the compiland source (the main source file name, not the names of any $INCLUDE files.) If one name is given in an INTERFACE and another in a USES clause, the USES identifier is used. The line and page are those designated by the procedure header.

Entry and exit calls are generated for the main program, unit initialization, and module initialization, in which case the identifier is the program, unit, or module name, 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 the PUBLIC variable CLNEQQ. Since the current routine is always available (because $LINE implies $ENTRY), the compiland source file name and the name of the routine containing the line are available along with the line number. Line number calls are generated just before the code in the first statement on a source 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 modules ERRE and PASE. The source error context entry points ENTEQQ, EXTEQQ, and LNTEQQ are in the debug module DEBE.

AVOIDING THE USE OF RUN-TIME ROUTINES

You may wish to write programs with Pascal that are specifically designed to use a minimum amount of memory. To do so, you should not use Pascal features that call run-time routines in your source code, and should avoid linking your program to the run-time library.

Use of the file input/output, real numbers, and sets all involve the run-time library routines. Units involve use of the run-time library, although use of modules does not. Use of the $DEBUG metacommand also brings in the run time. Section 14, "Available Procedures and Functions," indicates which procedures and functions are implemented through the run-time library.

The Pascal run-time modules linked with a Pascal program may occupy from 36.5 to 70K bytes of memory. Out of that, 4 to 5.5K bytes are taken by the run-time data. Run-time data, the user's data, the stack, and the short heap all share one memory segment (64K bytes). For more information, see your current Pascal Release Notice.

You can suppress linking the run-time library by explicitly specifying the module [sys]<sys>PasMin.obj in the object module line of the Linker command form. In this case, your program must provide the run-time support that is normally provided by the Pascal run time. This includes file and memory management and also all the run-time services that use the file and memory management (for instance, the numeric coprocessor emulator). If you do link in PasMin.Obj, you can enter either "Yes" or "No" for [DS Allocation?].

A useful technique when avoiding the run-time library support is to enter "none" as the last parameter for the [Libraries] field of the Linker command form. This ensures that [Sys]<Sys>Pascal.Lib is not linked to your program and the run time cannot be accessed. Any calls made to the run time then appear as unresolved external references. (See the subsection "Linking a Pascal Program" in Section 18, "Using the Pascal Compiler," for an example of how to complete the [Libraries] field.)

EXAMPLES

Each sample program below performs the same function. The first program does not use the run-time routines.

Example 1: Min.Pas

```
{$debug-}

Program TypeFile_NoRunTime;

    {This program does not use any elements of the
    Pascal run-time system.  ByteStreams are used
    in place of Pascal I/O and CTOS parameter
    management is used instead of Pascal parameter
    management.  Also the metacommand '$debug-' is
    included to turn off the run-time error
    checking.}

Const
        modeRead=#6d72;
        modeWrite=#6d77;

Type
        pbType  =ads of word;
        ppType  =ads of pbType;
        sdType  =record
                   pb   [00]:pbType;
                   cb   [04]:word;
                   end;
        pSdType =ads of sdType;


Function RgParam (
    iParam,
    jParam :word;
    pSdRet :pSdType ) :word; extern;

Function OpenByteStream (
    pBswa      :pbType;
    pbFileSpec :pbType;
    cbFileSpec :word;
    pbPassword :pbType;
    cbPassword :word;
    mode       :word;
    pbBuffer   :pbType;
    cbBuffer   :word ) :word; extern;

Function ReadByte (
    pBswa :pbType;
    pByte :pbType ) :word; extern;
```

```
Function WriteByte (
    pBswa :pbType;
    b     :byte ) :word; extern;

Function CloseByteStream (
    pBswa :pbType ) :word; extern;

Procedure CheckErc (
    erc :word ) ; extern;


Var     [public]
        erc,
        cbRet   :word;
        bswa    :array [1..130] of byte;
        bsBuffer:array [1..1024] of byte;
        b       :byte;

Var     [extern]
        bsVid   :byte;   {open video bytestream
                from CTOS.Lib}


Procedure Init[public];
    var sd :sdType;
    begin
    CheckErc (RgParam (1,0, ads sd));
    {get 1st Executive paramameter, the file to be
    typed, and open it}
    CheckErc (OpenByteStream (ads bswa,
                sd.pb,
                sd.cb,
                ads ' ',
                0,
                modeRead,
                ads bsBuffer,
                1024));
    end;

Procedure TypeFile[public];
    begin
    While true do
        begin
        erc := ReadByte (ads bswa, ads b);
        if erc<>0 then break;{end of file}
        CheckErc (WriteByte (ads bsVid, b));
        end;
    CheckErc (CloseByteStream (ads bswa));
    end;
```

```
begin   {program start}
    Init;
    TypeFile;
end.
```

Example 2: Max.Pas

```
Program                          TypeFile_UsingRunTime
(Input,OutPut,lsFileSpec);

    {This program types the file specified by the
    first parameter of a command form
    ('lsFileSpec'), to the Video}


Var     [public]
        inputFile,
        outputFile :file of byte;
        lsFileSpec :lstring(91);
        b          :byte;

Procedure Init[public];
    begin
    {the Pascal initialization run time loads
    lsFileSpec, see "program" statement with the
    first field of the Executive command form}

    inputFile.trap := true;{trap I/O errors}
    Assign (inputFile, lsFileSpec);
    Reset (inputFile);
    Assign (outputFile, '[vid]');
    Rewrite (outputfile);
    end;


Procedure TypeFile[public];
    begin
    While true do
        begin
        Read (inputFile, b);
        if inputFile.errs <> 0 then break;{end of
              file}
        Write (outputFile,b);
        end;
    end;

begin{program start}
    Init;
    TypeFile;
end.
```

APPENDIX A: COMPILER ERROR MESSAGES

This section lists error messages generated by the Pascal compiler. For operating system status messages and error codes see the Status Codes Manual.

ERRORS DETECTED BY THE FRONT END (PARSER/SEMANTIC ANALYZER

Front end error and warning messages include a number as well as a message, and most contain a row of dashes and an arrow to the location of the error. The front end recovers from most errors. However a few such errors are called panic errors, in which case the front end only lists the rest of the program. 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 too deeply nested.
- 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 be no way to get one of these messages. The comment in this list refers to the compiler routine containing the call.

FRONT END ERROR LIST

Decimal Value	Meaning
101	Invalid Line Number Line number is above 32767; there are too many lines in the source file.
102	Line Too Long Truncated Source lines are currently 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 (") or an accent mark (') is assumed to enclose a string; use a single quote (') instead.
107	Unexpected End of File End of file appears in a number, or metacommand, etc. [while scanning].
108	Metacommand Expected Command Ignored A $ at the start of a comment is not followed by an identifier.
109	Unknown Metacommand Ignored A metacommand identifier was unknown or invalid in this version.

Decimal Value	Meaning
110	Constant Identifier Unknown or Invalid Assumed Zero A metacommand is set 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 A metacommand is set to a numeric constant (as in $DEBUG: 1) and the constant has the wrong format or is out of range.
113	Invalid Meta Value Assumed Zero A metacommand is set to neither a constant or identifier.
114	Invalid Metacommand One of +, -, or : is expected following a metacommand.
115	Wrong Type Value for Metacommand Skipped The metacommand expects a string but an integer is given, or vice versa.
116	Meta Value Out of Range Skipped o The $LINESIZE integer value was below 16 or above 160. o The $REAL:N integer value was not 4 or 8. o The $INTEGER:N integer value was not 2.
117	File Identifier Too Long Skipped The $INCLUDE string value for the filename was too long.

Decimal Value	Meaning
118	Too Many File Levels There are 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 Metacommand Skipped A $IF and its value was not followed by $THEN or $ELSE.
123	Unexpected Metacommand Skipped A $THEN, $ELSE, or $END was found unrelated to a $IF metacommand.
124	Unexpected Metacommand The metacommand was not in a comment; it was processed anyway.
125	Assumed Hexadecimal A # was led without a "16" warning.
126	Invalid Real Constant A type REAL constant was used with a leading or trailing decimal point.
127	Invalid Character Skipped Source file character is not acceptable in program text.

Decimal Value	Meaning
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]

NOTE

The following 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.

Decimal Value	Meaning
136	Stack Memory Overflow Stack frame memory allocation has gone above 65534 bytes.
137	Integer Constant Overflow A type INTEGER or other, signed constant expression out of range.
138	Word Constant Overflow A type WORD or other unsigned constant expression is out of range.
139	Value Not in Range for Record Record tag value is not in range of variant, in a structured constant, a long form NEW/DISPOSE/SIZEOF, or other application.
140	Too Many Compiler Labels The compiler needs internal labels; the program is too big.
141	Compiler [in BOUNDS] This refers to an internal consistency check which failed; no matter what source program is compiled, this message should not occur. The compiler is in error, not your source program. The comment in this list refers to the compiler routine containing the call.
142	Too Many Identifier Levels Identifier scope level is over 15. (This is a compiler panic error. See explanation at the front of this section.)

Decimal Value	Meaning
143	Compiler [in DECLEVL] This refers to an internal consistency check which failed; no matter what source program is compiled, this message should not occur. The compiler is in error, not your source program. The comment in this list refers to the compiler routine containing the call.
144	Compiler [in LOOKUP7; often a PASKEY file format error] If this error occurs, you can rename the file [Sys]<Sys>Paskey to another name (thus, saving it) and try to recompile your program. However, this error refers to an internal consistency check that failed; no matter what source program is compiled, this message should not occur.
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 a statement, declaration, etc. [while parsing].

NOTE

The following common substitution mistakes get their own special messages, and are corrected with just a warning.

147	: Assumed =
148	= Assumed :
149	:= Assumed =

Decimal Value	Meaning
150	= Assumed :=
151	[Assumed (
152	(Assumed [
153	) Assumed]
154	] Assumed)
155	; Assumed ,
156	, Assumed ;
157 to 161	[Unassigned]

NOTE

If a particular symbol is expected in the source but not found, it may be inserted with one of the following messages.

162	Insert Symbol [this message should not occur; it is a minor compiler error]
163	Insert ,
164	Insert ;
165	Insert =
166	Insert :=
167	Insert OF
168	Insert]
169	Insert)
170	Insert [
171	Insert (

Decimal Value	Meaning
172	Insert DO
173	Insert :
174	Insert .
175	Insert ..
176	Insert END
177	Insert TO
178	Insert THEN
179	Insert *
180 to 184	[Unassigned]

NOTE

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 skipped source text.

188 Section or Expression Too Long

Compiler limit; try rearranging the program or breaking up long expressions by assigning intermediate values to temporary variables.

Decimal Value	Meaning
189	Invalid Set Operator or Function These include, for example, MOD operator or ODD function with sets.
190	Invalid Real Operator or Function These include, for example, MOD operator or ODD function with reals.
191	Invalid Value Type for Operator or Function These include, for example, MOD operator or ODD function with enumerated type.
192	[Unassigned]
193	[Unassigned]
194	Type Too Long A variable or type with greater than 32766 bytes is used.
195	Compiler [in SIZEOFT, {B}] This refers to an internal consistency check that failed; no matter what source program is compiled, this message should not occur. The compiler is in error, not your source program. The comment in this list refers to the compiler routine containing the call.
196	Zero Size Value Use of the empty record "RECORD END" as if it had a size.
197	Compiler [in ALLOCAT, {B}] This refers to an internal consistency check that failed; no matter what source program is compiled, this message should not occur. The compiler is in error, not your source program. The comment in this list refers to the compiler routine containing the call.

Decimal Value	Meaning
198	Constant Expression Value out of Range Check array index, subrange assignment, other subrange check.
199	Integer Type Not Compatible with Word Type A common error that 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 Check assignment statement or value parameter; see the subsection "Type Compatibility" in Section 4, "Introduction to Data Types."
202	Types Not Compatible in Expression Expression mixes incompatible types; see the subsection "Type Compatibility" in Section 4, "Introduction to Data Types."
203	Not Array - Begin Skip A variable followed by a left bracket (or parenthesis) is not an 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.

Decimal Value	Meaning
206	Not Record Field Ignored A variable followed by a dot is not a record, address, or file.
207	Invalid Field A record variable and dot are 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. (See Section 7, "Files," and Section 15, "File-Oriented Procedures and Functions.")
209	Cannot Dereference Value A variable followed by a caret is not a pointer, address, or file.
210	Invalid Segment Dereference A variable resides at a segmented address, but a default segment address is needed. You may need to make a local copy of the variable.
211	Ordinal Expression Invalid or Not Constant A constant ordinal expression was expected.
212	[Unassigned]
213	[Unassigned]

Decimal Value	Meaning
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 [in STRCONS] This refers to an internal consistency check which failed; no matter what source program is compiled, this message should not occur. The compiler is in error, not your source program. The comment in this list refers to the compiler routine containing the call.
221	Components Expected for Type A structured constant needs more components for its type.

Decimal Value	Meaning
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 the 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	Invalid Address of Function Assignment or other address reference to the function value is not in the scope of the function. This error also occurs when RESULT is used outside the scope of the function.
228	Cannot Assign to Variable Assignment to READONLY, CONST, or FOR control variable.
229	[Unassigned]
230	Unknown Identifier Assumed Integer - Begin Skip Unknown identifier, for which the address is needed.

Decimal Value	Meaning
231	VAR Parameter or WITH Record Assumed Integer - Begin Skip Invalid identifier, for which the address is needed.
232	Cannot Assign to Type The target of assignment is a file or otherwise cannot be assigned.
233	Invalid Procedure or Function Parameter - Begin Skip An error in the use of an intrinsic procedure or function, such as the following: o The first parameter to NEW or DISPOSE is not a pointer variable. o The long form of a NEW/DISPOSE/ SIZEOF record tag value was not found. o The long form of a NEW/DISPOSE/ SIZEOF super array, has too many bounds. o The long form of a NEW/DISPOSE/ SIZEOF super array, does not have enough bounds. o A NEW or SIZEOF super array was not given bounds. o ORD or WRD was performed on a value that is not of an ordinal type. o LOWER or UPPER was performed on an invalid value or type. o PACK or UNPACK was performed on a super array, array of files. o The first parameter to RETYPE is not a type identifier. o A RESULT parameter is not a function identifier.

- A CODEBYTE parameter value is greater than 255.
- An intrinsic is used which is not available in this version.
- ORD or WRD of an INTEGER4 value out of range.
- A HIWORD or LOWORD parameter is not ordinal or INTEGER4.

Decimal Value	Meaning
234	Type Invalid Assumed Integer o A parameter to READ, WRITE, ENCODE, or DECODE is not of type INTEGER, WORD, REAL, BOOLEAN, enumerated, or pointer. o A parameter to READ and WRITE is not of type CHAR, STRING, or LSTRING. o A parameter to READFN is not of type FILE. o A program parameter does not have a "readable" type; in this case the error occurs at the BEGIN keyword for the main program.
235	Assumed File INPUT The 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 The target of a READSET, ENCODE, or DECODE must be an LSTRING.
240	[Unassigned]

Decimal Value	Meaning
241	Invalid Segment Variable The variable resides at a segmented address, but a default segment address is needed. You may need to make local copy of the variable.
242	File Parameter Expected - Begin Skip 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 Use of the intrinsic function WRD is invalid.
248	Size Not Identical The warning is given in RETYPE; it may or may not work as intended.
249	Procedural Type Parameter List Not Compatible The parameter lists for formal and actual procedural parameters are not compatible. The number of parameters is different: the function result type or parameter type is different: or the attributes are wrong.

Decimal Value	Meaning
250	Cannot Use Procedure with Attribute You cannot call an INTERRUPT procedure, directly or indirectly.
251	Unexpected Parameter - Begin Skip The procedure or function has no parameters, but a left parenthesis was found.
252	Cannot Use Procedure or Function as Parameter An intrinsic procedure or function cannot be passed as parameter.
253	Parameter Not Procedure or Function - Begin Skip A procedural parameter was expected; you need a procedure or function here.
254	Supertype Array Parameter Not Compatible Actual parameter is not same or derived super type as formal.
255	Compiler [in ACTUALS] This refers to an internal consistency check which failed; no matter what source program is compiled, this message should not occur. The compiler is in error, not your source program. The comment in this list refers to the compiler routine containing the call.
256	VAR or CONST Parameter Types Not Identical Actual and formal reference parameter types must be identical.

Decimal Value	Meaning
257	Parameter List Size Wrong - Begin Skip Too few or too many parameters were used; skips only if too many.
258	Invalid Procedural Parameter to EXTERN The actual procedure or function is invoked with intrasegment 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 The set is not constant, the base types are not identical, or the constant is too big.
260	Unknown Identifier in Expression Assumed Zero The identifier is undefined (or misspelled) in an expression.
261	Identifier Wrong in Expression Assumed Zero A general identifier error in an expression has occurred; for example, file type id.
262	Assumed Parameter Index or Field - Begin Skip After error 260 or 261, anything in parentheses or square brackets, or a dot followed by an identifier, is skipped.
263	[Unassigned]
264	[Unassigned]
265	Invalid Numeric Constant Assumed Zero A decode error in an assumed INTEGER (or WORD) literal constant.
266	[Unassigned]

Decimal Value	Meaning
267	Invalid Real Numeric Constant A decode error in an assumed type REAL literal constant.
268	Cannot Begin Expression Skipped A symbol cannot start an expression, so it has been deleted.
269	Cannot Begin Expression Assumed Zero A 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 a WORD constant gives a negative result.
273	[Unassigned]
274	[Unassigned]
275	Invalid Range The lower bound of a subrange is greater than upper bound (e.g., 2..1).
276	CASE Constant Expected A constant value is expected for a CASE statement or record variant.
277	Value Already in Use In a CASE statement or record variant, a value has already been assigned (as in CASE 1..3: XXX; 2: YYY; END).

Decimal Value	Meaning
278	Invalid Symbol ".." was used in a CASE or record variant.
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 Nondecimal notation (e.g., 8#77, etc.) 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 An unexpected ELSE or UNTIL clause was skipped.
285	Compiler [in STATEMT {B}] This refers to an internal consistency check which failed; no matter what source program is compiled, this message should not occur. The compiler is in error, not your source program. The comment in this list refers to the compiler routine containing the call.
286	; Ignored A semicolon before ELSE is always in error, and is skipped.

Decimal Value	Meaning
287	[Unassigned]
288	: Skipped A colon after OTHERWISE is always in 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 o type REAL, INTEGER4, or other non-ordinal type o the component of an array, record, or file type o the referent of a pointer type or address type o in the stack or heap, unless locally declared o nonlocally declared, unless in static memory o a reference parameter (VAR or VARS parameter) o a variable with a segmented ORIGIN attribute
292	Skip to := In a FOR statement, the assignment is expected here.

Decimal Value	Meaning
293	GOTO Invalid The GOTO or label here involves an invalid GOTO statement.
294	GOTO Considered Harmful The $GOTOCK 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 was used as a function, or vice versa.
302	[Unassigned]
303	Unknown Identifier Skip Statement The starting statement identifier is undefined (or misspelled).

Decimal Value	Meaning
304	Invalid Identifier Skip Statement A general identifier error starts a statement; for example, 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 An END was unexpected; perhaps a missing BEGIN, CASE, or RECORD.
308	Compiler [in CONTEXT {B}] This refers to an internal consistency check which failed; no matter what source program is compiled, this message should not occur. The compiler is in error, not your source program. The comment in this list refers to the compiler routine containing the call.
309	Attribute Invalid An attribute valid only for procedures and functions was given for variable or vice versa, or an invalid attribute mix such as PUBLIC and EXTERN was used.
310	Attribute Expected A left bracket indicates attributes, but this is not an attribute.
311	Skip to Identifier This symbol was skipped to get to the identifier which follows.

Decimal Value	Meaning
312	Identifier Expected A list of identifiers is expected, but this is not an identifier.
313	[Unassigned]
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 appears after a PROCEDURE or FUNCTION in a parameter list.
317	[Unassigned]
318	Compiler internal error.
319	Compiler internal error.
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	Not EXTERN A procedure or function with the ORIGIN attribute must be EXTERN.
322	Invalid Attribute with Function or Parameter An INTERRUPT procedure cannot have parameters or be a function.

Decimal Value	Meaning
323	Invalid Attribute in Procedure or Function A nested procedure or function cannot have attributes or be EXTERN.
324	Compiler internal error.
325	Already Forward FORWARD cannot be used twice for the same procedure or function.
326	Identifier Expected for Procedure or Function The 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 Assumed Integer This error occurs when an ordinal type identifier is expected.

Decimal Value	Meaning
332	Invalid Type Declaring the WORD type.
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 This is a 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 ; An Integer is assumed. The keyword SUPER is always followed by ARRAY in a type clause.
340	Invalid Set Range Integer 0 to 255 The base type of a set must be within the subrange 0..255.

Decimal Value	Meaning
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. (See above.)
344	[Unassigned]
345	Skip to ; A semicolon is expected at the end of a declaration (not at end of line).
346	Insert ; Semicolon expected at end of declaration (at end of line).
347	Cannot Use Value Section with ROM Memory Setting $ROM on prevents the use of a VALUE section.
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]

Decimal Value	Meaning
350	Not Array - Begin Skip A variable in a VALUE section followed by square bracket not array.
351	Not Record - Begin Skip A variable in VALUE section followed by a dot is not a record type.
352	Invalid Field In the VALUE section an identifier assumed to be a field is not in the record.
353	Constant Value Expected In the VALUE section a variable can only be initialized to a constant.
354	Not Assignment Operator Skip to ; The assignment operator was not found in a VALUE section.
355	Cannot Initialize Identifier Skip to ; A symbol in the VALUE section is not a variable declared at this level in fixed (STATIC) memory, or has the ORIGIN or EXTERN attribute.
356	Cannot Use Value Section Put the 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.

Decimal Value	Meaning
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 for the referent type was declared again later in the same TYPE section. For example: TYPE A=WORD; PROCEDURE B; TYPE C=^A; A=REAL;Message says the forward type is used in this case (such as, REAL).
359	Cannot Use Label Section Put a LABEL section in the IMPLEMENTATION, not the INTERFACE.
360	Forward Pointer to Supertype The referent of a reference type declared in this TYPE section is a super array type; the supertype declaration must come earlier.
361	Constant Expression Expected Zero Assumed In a CONST section, the expression is not constant.
362	Attribute Invalid In a VAR section, PUBLIC or ORIGIN with are used with EXTERN, or ORIGIN in attribute brackets after the VAR keyword.
363	[Unassigned]
364	Contains File Initialize Module File variables must be initialized. Thus, when a file variable is declared in a module the module must be called (as a parameterless procedure) to initialize these files.

Decimal Value	Meaning
365	Origin Variable Contains File Cannot Initialize File variables must be initialized, but ORIGIN variables are never initialized, so the user must initialize this file.
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; a UNIT identifier was not found.
372	Compiler error This error occurs when the keyword UNIT is missing in an interface.

Decimal Value	Meaning
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 appears after the PROGRAM or MODULE keyword. (This is a compiler panic error. See explanation at the front of this section.)
375	UNIT Identifier Expected No unit identifier after IMPLEMENTATION OF. (This is a compiler panic error. See explanation at the front of this section.)
376	Program Not Found PROGRAM, MODULE, or IMPLEMENTATION OF keywords not found (panic). Can occur if source file is not a Pascal compiland.
377	File End Expected Skip to End The assumed end of the compiland was processed, but there is more.
378	Program Not Found The main body of a PROGRAM or initialized IMPLEMENTATION, or the final END of a MODULE or other IMPLEMENTATION, was not found.

ERRORS DETECTED BY THE BACK END (OPTIMIZER/CODE GENERATOR)

The following program errors are detected by the back end:

- Attempt to divide by zero. For example,

  ```
  A DIV 0.
  ```

- Overflow during integer constant folding. For example,

  ```
  MAXINT+A+MAXINT.
  ```

- Expression too complex or too many internal labels.

  (Try breaking up the expression by using assignments to temporary variables.)

The optimizer and code generator perform a large amount of internal consistency checking. When one of these checks encounters an unexpected condition, the result is an internal error generated by the module where the inconsistency was discovered.

Such errors should generally not occur. When they do occur, we request that they be reported promptly. Since it may be difficult to analyze such reports unless they include the complete source code involved, please include the complete source code in a machine readable form.

The format of an optimizer error message is as shown below:

```
*** Internal Error  <error number>
 Near Line <source line number>
 Contact Technical Support
```

where <error number> is an internal error number and <source line number> is the last source line number seen by the optimizer. The error may not have occurred exactly at this line, but it is likely to be within a few lines following this line. The <source line number> corresponds to the line numbers on the listing generated by the front end.

Module OPTIM (Status Numbers 0 to 99)

1 Bad ICode file format (PRSDEC10).

2 Bad symbols file format; cannot find function return variable (READ_SYMTAB).

3 Multiple symbols file entries for symbol that is not a procedure or function (READ_SYMTAB).

4 Forward reference to an Icod4 number (XLATE).

5 ICode reference to a missing symbol (XLATE_SYM).

6 Duplicate ICode numbers in same block (ENTER_XLATE).

7 Invalid or unexpected operand for ADDR ICode (PHASE1).

8 Invalid addressing mode for ADDR ICode (PHASE1).

9 Invalid or unexpected operand for DRRR ICode (PHASE1).

10 Invalid or unexpected operand for DRFR ICode (PHASE1).

11 Invalid symbol type for UPPR ICode (PHASE1).

12 Invalid addressing mode for ASMS/ASVS (PHASE1).

13 Bad tree format; assignment target tree does not have a SYMR node as its leftmost lead (DEL_TARGET).

14 Unknown ICode value (SUREX).

15 Bad statement list returned from SPLITTREE (OPTIM - main program).

16 Bad statement list returned from PHASE1 (OPTIM).

17 Bad statement list returned from CHECK_LENGTH (OPTIM).

18 Bad statement list returned from PHASE2 (OPTIM).

19 Bad statement list returned from PHASE3 (OPTIM).

20 Bad statement list returned from MD_XFURM (OPTIM).

21 Bad statement list returned from SUREX (OPTIM).

Module GEN6 (Status Numbers 100 to 199)

100 Static nesting level < 0 (NESTLEV).

101 Invalid or unexpected operand for OFFR ICode (MD_XFORM).

102 Invalid flag values for CONR (MD_XFORM).

103 Invalid or unexpected operand for UPPR ICode (MD_XFORM).

104 Invalid symbol type for UPPR operand (MD_XFORM).

105 Too many levels of indirection for UPPR operand (MD_XFORM).

106 Invalid addressing mode for VALP ICode (MD_XFORM).

107 Invalid or unexpected operand for LVAP ICode (MD_XFORM).

108 Multiple definition of an internal label (GENDONE).

109 Cannot load long constant value with a length > 4 (CASELONR).

110 Invalid offset value for OFSR ICode (CASEOFSR).

111 Register table entry or use count for OFSR ICode is bad (CASEOFSR).

112 Invalid nesting for procedure/function call (CALLPF).

113 Invalid function return length (CASECALP).

114 Bad use count for SFRT ICode operands (CASESFRT).

115 Symbol type is invalid, must be a variable (CLASS).

116 Operand use count is already 0 (COUNTUSE).

117 User label must begin a basic block (DEF_ULAB).

118 Duplicate definition of user label (DEF_ULAB).

119 Address flag missing for LONR ICode (EMITIMM).

120 Address flag missing for SYMR ICode (EMITIMM).

121 Variable must be static (EMITIMM).

122 Symbol type must be variable (EMITIMM).

123 Invalid ICode type (EMITIMM).

124 Cannot save a multibyte value (EMPTYREG).

125 Invalid register contents (EMPTYREG).

126 Symbol type must be variable (GENREF).

127 Missing address flag for long constant reference (GENREF).

128 Invalid ICode type (GENREF).

129 Value must be in an index register (GENREF).

130 Value must be in an index register (GENREFI).

131 No registers available for allocation (GETREG).

132 Register BX already in use (IMBXES).

133 Register must be SI or DI (INDREF).

134 Symbol must be variable (INDREF).

135 Missing address bit for long constant reference (LOADR).

136 Symbol must be a variable (LOADR).

137 Invalid ICode type (LOADR).

138 Symbol type must be a label (LONGGOTO).

139 Register residence flags do not match register table contents (MOVER).

140 Value must be in some register (REGN).

141 Invalid operand register specified by template (REGSPEC).

142 Operand's register residence flag does not match the specified register (REGSPEC).

143 Operand must be in a register (X_BINOP).

144 Unexpected opcode value (X_BINOP).

145 Contents of BX do not match operand (X_CHKBXES).

146 Invalid ICode operator (X_CMPI).

147 Invalid ICode operand (X_CMPI).

148 Invalid variable kind (must be static) or address bit missing (X_CMPI).

149 Invalid ICode operator; must have two operands (X_COMOPR).

150 Desired register already in use (WANTREG).

151 Desired register already in use (X_DONE).

152 Index must already be in a register (X_DONEA).

153 Invalid register contents (X_DONEA).

154 Symbol must be a variable (X_DONEA).

155 Invalid ICode operand (X_DONEA).

156 Invalid condition code for IF template (X_IFCOND).

157 Invalid condition code for IFOPR template (X_IFOCOND).

158 Register BX contents are wrong (X_INREGS).

159 Source register is empty (X_MOVREG).

160 Register residence flag for operand is bad (X_MOVREG).

161 Invalid register designated; cannot access high half of register (X_SELFH).

163 Invalid ICode for assignment target (X_STOR).

164 Invalid ICode operator; must have two operands (X_REVOPR).

165 Invalid opcode value (X_UNIOP).

166 Invalid register specification (X_XCHG).

167 Cannot exchange registers containing part of a multiregister value (X_XCHG).

168 Register residence flag does not match register table contents (X_XCHG)

169 Register residence flag does not match register table contents (X_XCHG).

170 Register table contents do not match their associated register residence flags (INTERPRET).

171 Operand must be a CONR node (INTERPRET).

172 No match for this operand class in the templates for this ICode (SCANCLASS).

173 Register BX is already in use (INTERPRET).

174 Use count was not decremented properly (INTERPRET).

175 Use count was not decremented properly (INTERPRET).

176 Error in template processing (INTERPRET).

177 Invalid register specification; cannot access high/low half of the register (INTERPRET).

178 Invalid or unexpected template operator (INTERPRET).

179 Invalid length for OFFR ICode; must be length 1, 2, or 4 (GEN_SUBTREE).

180 Symbol table entry for RTPP ICode does not match the current procedure/function (GEN_SUBTREE).

181 Symbol table entry for RTPP ICode must be a procedure or function (GEN_SUBTREE).

182 Invalid or unexpected ICode value (GEN_SUBTREE).

Module SUBR (Status Numbers 200 to 299)

200 Value too large to convert to WORD type, BOOT compiler only (WRDTOINT).

201 Missing address bit for assignment target (TARGCHECK).

202 Invalid ICode for assignment target (TARGCHECK).

203 Unexpected opcode value, BOOT compiler only (GET_OPCFLAGS).

204 Invalid opcode flag value (GETTYP).

205 Invalid opcode flag value (GETTYP).

Module FOLD (Status Numbers 300 to 399)

300 Invalid operand count, must have two operands (FOLD_CONS).

301 Invalid constant values for operands to the NOTB ICode (FOLD_CONS).

Module CHKLEN (Status Numbers 400 to 499)

400 Operand length cannot be 0 (CHECKLEN).

401 Length of operands must match if both are greater than 0 (CHECKLEN).

402 Operand length must be -1, 1, or 2 (CHECKLEN).

403 Operand length must be -1, 1, or 2 (MUST1OR2).

404 New length must be 1 or 2 (COERCE).

405 Assignment target must be variable or function (TARG_LEN).

406 Invalid ICode for assignment target (TARG_LEN).

407 Invalid symbol type for SYMR ICode (CHECK_LENGTH).

408 Assignment target length must be 4 for AS48 (CHECK_LENGTH).

409 Invalid addressing for VAXP operand (CHECK_LENGTH).

410 Unexpected ICode value (CHECK_LENGTH).

Module CTL6 (Status Numbers 500 to 599)

500 Code generator-computed code size does not match the computed code size.

501 Invalid class override, CS_DTYP record (BINPS).

502 Invalid symbol type, CS_SYM record (BINPS2).

503 Internal label reference to an undefined label, CS_CJMP record (BINPS2).

504 Internal label reference to an undefined label, CS_ILAB record (BINPS2).

505 Internal label location does not match current location counter, CS_DILB record (BINPS2).

506 User label reference to an undefined label, CS_ULAB record (BINPS2).

507 User label location does not match current location counter, CS_DULB record (BINPS2).

508 P-code procedure/function entry address does not match current location counter, CS_PFBEG/CS PROB record (BINPS2).

509 Procedure/function entry address does not match current location counter, CS_PFBEG/CS PROB record (BINPS2).

510 Unknown binary interpass file record type (BINPS2).

Module DUMP86 (Status numbers 600 to 699)

600 Unexpected interpass record type (GETBYTE).

601 Unexpected end of data (GETDATA).

602 Invalid data size (GETDATA).

603 Invalid data size (GETDATA).

604 Unexpected end of data (GETDISP).

605 Unexpected interpass record type (GETDISP).

606 Unexpected end of data (GETLABEL).

607 Invalid label type, must be short label (GETLABEL).

608 Unexpected interpass record type (GETLABEL).

609 Invalid opcode (WRITEOP).

610 Invalid opcode, no PUSH CS opcode exists (PUSHPOPSEG).

611 Cannot do sign extension on operands for logical operators AND, OR, XOR (BINARYOPS).

612 Invalid mode value (LOADPTR).

613 Invalid opcode value (SHIFTOPS).

614 Unused opcode (GROUPC).

615
to
626 Unused opcode (DUMP86)

Module DUMP (Status Numbers 700 to 799)

700 Invalid opcode value (OPNAME).

701 Unknown working value (DMP1ID).

702 Unexpected symbol type (DMP1ID).

703 Invalid operator mode value (WRIMOD).

704 Unexpected ICode value (DMPNOD).

705 Unexpected interpass record type (DMPBREC).

RUN-TIME ERROR MESSAGES

Errors detected at run time are either file system errors or other program exceptions. File system errors are described first.

FILE SYSTEM ERRORS

File system error codes range from 1000 to 1999 and are based on the ERRC field of the file control block.

852-013

File system errors are reported in the following format:

If <error code> is in the range 1000 to 1099, then the error was detected by the CTOS operating system and <status code> is a CTOS status code. See the Status Codes Manual for interpretation of status codes.

If <error code> is in the range 1100 to 1999, then the error was detected by the Pascal file system. These error codes are explained below:

Decimal Value	Meaning
1100	ASSIGN or READFN of file name to open file.
1101	Reference to buffer variable of closed textfile.
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).

Decimal Value	Meaning
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.
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 (word subrange).
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.
1123	Encode error (field width > 255) in textfile WRITE INTEGER.

Decimal Value	Meaning
1124	Encode error (field width > 255) in textfile WRITE REAL.
1125	Encode error (field width > 255) in textfile WRITE WORD.
1126	Decode error in textfile READ INTEGER4.
1127	Encode error in textfile WRITE INTEGER4.

The <error type> field of the file system error report is based on the ERRS field of the file control block. Error types are described below:

0 (no error).

1 Hard data. Hard data error.

2 Device name. Invalid device or volume name.

3 Operation. Invalid operation: GET if EOF, RESET a printer, etc.

4 File system. File system internal error.

5 Device offline. Device or volume no longer available.

6 Lost file. File no longer available.

7 File name. Invalid syntax, name too long, etc.

8 Device full. Disk full, directory full, etc.

9 Unknown device. Device or volume not found.

10 File not found.

11 Protected file.

12 File in use.

13 File not open.

14 Data format. Data format, decode, or range error.

15 Line too long. Buffer overflow.

OTHER RUN-TIME ERRORS

Nonfile 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 to 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.

Decimal Value	Meaning
2000	Stack Overflow While calling a procedure or function, the stack ran out of memory. Checked if $STACKCK+ and in some other cases.
2001	No Room in Heap Not enough room is available in the heap for a new variable. This error is always detected.
2002	Heap Is Invalid While allocating memory in the heap for a new variable, an error in the heap structure was found. This error is always detected.
2003	Heap Allocator Interrupted An interrupt procedure was invoked that interrupted NEW and called NEW again. The heap allocator modifies the heap; thus it is a critical section.
2004	Allocation Internal Error An unexpected error return occurred while requesting additional heap space from the operating system. Contact technical support.

Decimal Value	Meaning
2031	Nil Pointer Reference
	DISPOSE or $NILCK+ found a pointer with a NIL value.
2032	Uninitialized Pointer
	DISPOSE or $NILCK+ found an uninitialized pointer. Pointers are given this value only if $NILCK is on.
2033	Invalid Pointer Range
	DISPOSE or $NILCK+ found a pointer that does not point into the heap or is otherwise invalid. The pointer may have pointed to a DISPOSED block that was removed from the heap.
2034	Pointer to Disposed Var
	DISPOSE or $NILCK+ found a pointer to a heap block that has been disposed. Calling DISPOSE twice for the 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 to 2099 Ordinal Arithmetic

Decimal Value	Meaning
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. This is checked if $RANGECK+ is used.

Decimal Value	Meaning
2051	Unsigned Divide by Zero WORD value divided by zero.
2052	Signed Divide by Zero INTEGER value divided by zero.
2053	Unsigned Math Overflow A WORD result occurred outside 0..MAXWORD. This is checked if $MATHCK+ is used.
2054	Signed Math Overflow An INTEGER result occurred outside -MAXINT..MAXINT. This is checked if $MATHCK+ is used.
2055	Unsigned Value Out of Range Assignment of a value parameter in which the source value is out of range for the target value. The target can be a subrange of WORD (including BYTE), or CHAR, or an enumerated type. This error can also occur in SUCC and PRED functions, and when the length of an LSTRING is assigned. These are checked with $RANGECK+. Another time this error occurs is 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 This is the same as 2055, but applies to the INTEGER type and its subranges.

Decimal Value	Meaning
2057	Uninitialized 16-Bit Integer Used
	An INTEGER or 16-bit INTEGER subrange variable is used without being assigned first, or such a variable has the invalid value, -32768. This condition is checked with $INITCK+.
2058	Uninitialized 8-Bit Integer Used
	A SINT or 8-bit INTEGER subrange variable is used without being assigned first, or such a variable has the invalid value -128. This condition is checked with $INITCK+.

2100 to 2149 Type REAL Arithmetic

Decimal Value	Meaning
2100	REAL Divide by Zero
	A REAL value was divided by zero. This condition is always detected.
2101	REAL Math Overflow
	A REAL value is too large for representation. This condition is always detected.
2104	SQRT of Negative Argument
	A square root function is used on an argument < 0. This condition is always detected.
2105	LN of Non-Positive Argument
	A natural log function is used on an argument <= 0. This condition is always detected.

Decimal Value	Meaning
2106	TRUNC/ROUND Argument Range Results from converting a REAL outside the range of INTEGER. This condition is always detected.
2131	Tangent Argument Too Small The tangent argument is so small that the result is invalid. This condition is always detected.
2132	Arcsin or Arccos of REAL > 1.0 The arcsin or arccos argument is greater than one. This condition is always detected.
2133	Negative Real Raised to a Real Power An invalid argument in exponentiation. This condition is always detected.
2135	REAL Math Underflow The significance of a REAL expression was reduced to zero.
2136	REAL Indefinite (uninitialized or previous error) The REAL value called "indefinite" was encountered; this can occur if $INITCK was on and an uninitialized real variable was used, or if a previous error set a variable to indefinite as part of its masked error response.

2150 to 2199 Structured Type Errors

Decimal Value	Meaning
2150	String Too Long in COPYSTR A COPYSTR intrinsic source string is too large for target string. This condition is always detected.
2151	LSTRING Too Long in Intrinsic Procedure A target LSTRING is too small in INSERT, DELETE, CONCAT, or COPYLST intrinsic procedure. This condition is always detected.
2180	Set Element Greater Than 255 A value in a constructed set is above maximum. This condition is always detected.
2181	Set Element Out of Range A value in a set assignment or set value parameter is too large for the target set. This condition is detected with $RANGECK+.

2200 to 2249 INTEGER4 Arithmetic Errors

Decimal Value	Meaning
2200	INTEGER4 Divide by Zero
2201	INTEGER4 Math Overflow
2234	INTEGER4 Zero to Negative Power

2250 to 2999 Other Errors

Decimal Value	Meaning
2450	Unit Version Number Mismatch

During unit initialization, the user (the one with the USES clause) and the implementation of an interface were discovered to have been compiled with unequal interface version numbers. This condition is always detected.

```
CONST

    ercOk       = 0;
    modeAppend  = RETYPE(WORD, 'ma');
    modeModify  = RETYPE(WORD, 'mm');
    modeRead    = RETYPE(WORD, 'mr');
    modeWrite   = RETYPE(WORD, 'mw');
```

Pascal data types are not totally adequate for use with the CTOS operating system; therefore, data types that are roughly equivalent were chosen for Syslit.Edf in the example above. The semantics of the data types used in Syslit.Edf are shown below:

ErcType	2-byte unsigned integer; contains error status returned from a CTOS facility. Error status of 0 is no error.
FlagType	1-byte unsigned integer. 0 means flag is off, and 1 means flag is on.
FhType	2-byte unsigned integer; contains a file handle (number) that uniquely identifies open files for the file system.
LfaType	4-byte unsigned integer; contains a logical file address (number) that identifies an offset from the beginning of a file.
ModeType	2-byte string; contains two characters that indicate a file's access mode for the file system.
POINTER	4-byte segmented address; contains two words, of which the low word is the relative address within a segment, and the high word is the segment base address.
QUAD	4-byte unsigned integer; contains a number in the range 0 to 4,294,967,295 (used for arithmetic involving logical file addresses). Note that INTEGER4 does not satisfy this range.

Table F-1 shows the CTOS type and the equivalent Pascal type.

Table F-1. Pascal Data Types for Use with CTOS.

CTOS Type	Equivalent Pascal Type
ercType	WORD
pbType or pointer	ADS of WORD or ADS of BYTE The current compiler allows your program to access a byte value using ADS of BYTE. (Previously, compilers always transferred a word). In protected mode, using ADS of BYTE avoids the general protection fault, which would occur if the word containing the byte value to be accessed extended beyond the memory allocated for the segment. For details on protected mode memory management, see the iAPX 286 Programmer's Reference Manual and the 80386 Programmer's Reference Manual.
flagType	BOOLEAN {00h = false, 01h = true}
fhType	WORD
modeType	WORD
lfaType or quadType DWORD	INTEGER4 INTEGER4 is not strictly an Lfa or a Quad, since the most significant bit is used as a sign, but it works for positive numbers.

```
SCBType = record
            SysBuildType    [00]:byte;
            OsType          [01]:byte;
            saMinLL         [02]:paraType;
            saCurrLL        [04]:paraType;
            saCurrSL        [06]:paraType;
            saMaxSL         [08]:paraType;
            saMemMax        [10]:paraType;
            end;
pSCBType  = ads of SCBType;

VersionType =  lstring(30);
  {version is an 'sb' string, a.k.a, lstring}

pVersionType = ads of VersionType;

{definitions of CTOS externals:}

Function GetpStructure (
    structCode          :word;
    ph                  :word;
    {partition handle}
    ppStructureRet      :ppType )     :word; extern;

Procedure CheckErc (
    erc                 :word ); extern;


Procedure DumpCTOSVersion     [public];
  Const oVersion = #254;
    {oVersion is the relative address of the
    pointer to the pointer to the version. It can
    be found in the System Common Address Table
    (SCAT) described in your operating system
    manual.  The segment address for all
    fields in the SCAT is zero}

    var
      pVersion :pVersionType;
        {pointer to CTOS version}
      Version  :VersionType;
    begin

    {GetpStructure takes as arguments a structure
    code or relative address of a structure
    defined in the SCAT, a partition handle (if
    zero then the handle of the partition the
    program is running in), and the address of the
    address to be returned}
```

```
    CheckErc (GetpStructure (oVersion, 0,
              ads pVersion));
    Version := pVersion^;
      {deference pointer to our lstring}

      {note: deferencing structure pointers
      requires the run time. Structure pointers
      can be deferenced without the run time on a
      field by field basis}

    Writeln ('CTOS version    ',Version);
    end;



Procedure DumpMemoryMap         [public];
  Const oSCB = #2C8;
  {relative address of the pointer to the System
  Configuration Block}

  var     sOsMemory,
          sMaxMemory,
          sParagraph    :integer4;
          pSCB          :pSCBType;
          {pointer to SCB}
  begin
  CheckErc (GetpStructure (oSCB, 0, ads pSCB));
  sParagraph := 16;
  sOsMemory  := sParagraph * pSCB^.saMinLL;
  sMaxMemory := sParagraph * pSCB^.saMemMax;
  Writeln ('OS memory     ', sOsMemory, ' bytes');
  Writeln ('Total memory ', sMaxMemory, ' bytes');
  end;



begin
    DumpCTOSVersion;
    DumpMemoryMap;
end.
```

CONTROL OF THE VIDEO DISPLAY

You can control the video display using one of three different methods: Video Bytestreams, Direct Video Access (Video Access Method) through CTOS, or the Forms package.

Using Forms is described in detail in the Forms Manual. Examples of using Forms with Pascal are available as application notes from technical support.

The section entitled "Video," in your operating system manual describes the Video Access Method (VAM) in detail. In addition, an example showing the use of VAM is included at the end of this section. Direct Video Access has the advantage that it does not use the Pascal run-time library.

The remainder of this section describes video byte streams and shows how to control the video display from your Pascal program by writing a multibyte escape sequence to the display. This allows you to use WRITE and WRITELN to send an escape sequence to the screen in Pascal in the same way that you can use OpenByteStream in CTOS. In this way, a program can

- control character attributes (blinking, reverse video, underscoring, half-bright)
- control screen attributes (reverse video, half bright)
- fill a rectangle with a single character
- control scrolling of lines
- direct video display output to any frame
- control pausing between full frames of data
- control the keyboard LED indicators
- erase to the end of the current line or frame

A multibyte escape sequence consists of the video display escape character, a command character, and parameters. The video display escape character is CHR(255). To print an escape character, precede it with another escape character.

The following pages give the format for escape sequences that control the various features of the video display. Note that these formats show the asterisk (*) as the concatenation operator, but the asterisk can only be used to create constant string expressions. Variable string expressions with concatenation, should use the LSTRING intrinsic CONCAT.

ERROR CONDITIONS IN ESCAPE SEQUENCES

An escape character sequence is in error if the command characters or parameters are unrecognized or the parameters are inconsistent.

The following program turns on the cursor, writes the message "This is a test," and waits for input:

```
PROGRAM Test (INPUT, OUTPUT);
VAR
  LS : LSTRING (128);

BEGIN
  LS := CHR(255) * 'vn';
  Write (LS, 'This is a test');
  ReadLn;
END.
```

VIDEO DISPLAY COORDINATES

Pascal interprets some parameters as x and y coordinates on the video display.

A value of 255 for x or y specifies, respectively, the last column or line of the frame.

If the value of x or y is less than 255 and greater than the last column or line, then the parameters are in error.

Format 2

```
CHR(255) * 'V'<parameter>
```

where

<parameter>
is N or F.

Format 2 is used to make the cursor visible if the <parameter> is N or to make the cursor invisible if the <parameter> is F.

FILLING A RECTANGLE: THE 'F' COMMAND

```
CHR(255) * 'F' * <character>
* CHR(<Xposition>)
* CHR(<Yposition>)
* CHR(<width>) * CHR(<height>)
```

where

<character>
is any character;

<Xposition>, <Yposition>, <width>, and <length>
are integer expressions.

The 'F' command is used to fill a rectangle on the video display with <character>. The currently enabled character attributes are given to each character in the rectangle. A <character> always specifies a character in the standard character set.

The coordinates (<Xposition>,<Yposition>) specify the upper left corner of the rectangle. A value of 255 for <width> and <height> specifies, respectively, the remaining width or height of the frame.

CONTROLLING LINE SCROLLING: THE 'S' COMMAND

```
CHR(255) * 'S'
* CHR(<firstline>)
* CHR(<lastline>)
* CHR(<count>) * '<direction>'
```

where

<direction>
is D or U.

If the <direction> is D, the 'S' command is used to scroll down a portion of the frame beginning at line <firstline> and extending to (but not including) <lastline>. The <count> lines are scrolled and the top <count> lines of the frame portion are filled with blanks.

If the <direction> is U, the 'S' command is used to scroll up a portion of the frame beginning at line <lastline> and extending to (but not including) <firstline>. The <count> lines are scrolled and the bottom <count> lines of the frame portion are filled with blanks.

DIRECTING VIDEO DISPLAY OUTPUT: THE 'X' COMMAND

```
CHR(255) * 'X' * CHR(<frame>)
```

The 'X' command is used to direct video output to the <frame>'th frame of the video display.

The video display is divided into frames. (See the section entitled "Video," in your operating system manual for a discussion of video frames.)

The main frame is the default frame.

If <frame> is 1, the 'X' command is used to direct video output to the Status Frame at the top of the video display.

If <frame> is 2, the output is directed to the line that separates the Status Frame from the main frame.

CONTROLLING PAUSING BETWEEN FULL FRAMES: THE 'P' COMMAND

```
CHR(255) * 'P<parameter>'
```

where

<parameter>
is N or F.

EXAMPLES SHOWING THE USE OF MODULES AND UNITS

The following two examples both perform the same job, converting a temperature in Celsius to Fahrenheit. Both use an external function.

Example 1 uses a module to declare the function, while Example 2 uses a unit.

All the examples in this Appendix were compiled and run with 10.0 level software.

EXAMPLE 1

The two files are separately compiled, then linked to create the run file Pe1.Run. Pe1.Pas contains the main program, and Pe2.Pas contains a module that declares a function that changes temperature from Celsius to Fahrenheit.

Instructions for compiling, linking, and running the two compilands are given in the subsection "Instructions for Compiling, Linking, and Running Example 1."

Main Program: Pe1.Pas

Files Pe1.Pas and Pe2.Pas must be compiled separately and linked together.

This program converts Celsius temperature to Fahrenheit. It prompts the user to enter the Celsius temperature, then converts that to Fahrenheit, and displays the result on the screen. It then prompts the user for another Celsius temperature, and so on. The program terminates when the user enters a number less than -200.

The program uses an external function, Fahrenheit, to compute the Fahrenheit temperature. That function is declared in a separate compiland in the file Pe2.Pas.

```
Program  CelsiusToFahrenheit(Input,Output);

VAR celsTemp : REAL;

FUNCTION Fahrenheit (celsius : REAL) : REAL;
EXTERN;

BEGIN

REPEAT

(* Prompt the user for input. *)

write ('Enter Celsius temperature');
write ('       ( -200 or less to exit): ');

(* Read the response.*)

readln(celsTemp);

IF celsTemp <= -200 THEN BREAK; (* Check for
                                   sentinel value*)

(* Convert Celsius temperature to Fahrenheit and
display the result. *)

writeln;
writeln(celsTemp:6:3,' C = ',
                  Fahrenheit(celsTemp):6:3,' F');
writeln;

UNTIL FALSE

END.
```

Module: Pe2.Pas

Files Pe1.Pas and Pe2.Pas must be compiled separately and linked together.

The file Pe2.Pas, which follows, contains a module declaring the function Fahrenheit.

```
module Fah;

FUNCTION Fahrenheit(cels:REAL) : REAL;

(* This function converts Celsius temperature to
Fahrenheit.
ON ENTRY: cels is temperature in degrees Celsius.
RETURN: The function returns temperature in
degrees Fahrenheit. *)

BEGIN

Fahrenheit := cels * (9/5) + 32

END; (* End of Fahrenheit.*)

END. (* End of module.*)
```

Instructions for Compiling, Linking, and Running Example 1

To invoke the compiler, type "Pascal" into the Executive command form. Complete the Pascal command form as shown below, then press GO:

```
Pascal
  Source file          Pe1.Pas
  [Object file]
  [List file]
  [Object list file]
```

After the program has compiled, compile the module the same way, but complete the command form as shown below:

```
Pascal
  Source file          Pe2.Pas
  [Object file]
  [List file]
  [Object list file]
```

Then link the resulting object files, Pe1.Obj and Pe2.Obj. In addition to these files, you must link the object file, PasFirst.obj, which is included with the 10.0 software. To do this, invoke the Linker through the Executive, by typing "Bind" (or as many letters as required to make the command unique) into the Executive command form. Then, complete the Bind command form as shown below:

```
Bind
  Object modules       PasFirst.Obj Pe1.Obj Pe2.Obj
  Run file             Pe1.Run
  [Map file]
  [Publics?]
  [Line numbers?]
  [Stack size]
  [Max array, data,
     code]
  [Min array, data,
     code]
  [Run file mode]
  [Version]
  [Libraries]
  [DS allocation?]
  [Symbol file]
```

For details on the PasFirst.obj object file, see "Linking a Pascal Program" in Chapter 18.

The resulting run file, Pe1.Run, can be invoked by completing the Run command form as shown below. Remember, to terminate the program, enter a Celcius temperature of less than -200.

```
Run
  Run file          Pe1.Run
  [Case]            _____
  [Parameter 1]     _____
  [Parameter 2]     _____
  [Parameter 3]     _____
  [Parameter 16]    _____
```

EXAMPLE 2

The three files shown below perform the same job as the files shown in Example 1.

Pe3.Pas contains the main program, and Pe4.Pas contains a unit that declares a function that changes temperature from Celsius to Fahrenheit.

These two files are separately compiled, then linked to create the run file Pe3.Run.

The interface file, Pei.Inf (the third file shown below), is used by both Pe3.Pas and Pe4.Pas. It is not compiled separately.

Instructions for compiling, linking, and running the two compilands are given in the subsection "Instructions for Compiling, Linking, and Running Example 2."

Main Program: Pe3.Pas

This file, Pe3.Pas, must be compiled separately and linked together with Pe4.Pas. Both Pe3.Pas and Pe4.Pas use an interface in the file Pei3.Inf. The File Pei3.Inf cannot be compiled separately.

Pe3.Pas and Pe4.Pas implement the same program as the files Pe1.Pas and Pe2.Pas in Example 1, but here we use a unit instead of a module to implement the function Fahrenheit.

The program converts Celsius temperature to Fahrenheit. It prompts the user to enter the Celsius temperature, then it converts it to Fahrenheit, and displays the result on the screen. It then prompts the user for another Celsius temperature, and so on. The program terminates when the user enters a number less then -200.

The program uses an external function, Fahrenheit, to compute the Fahrenheit temperature. That function is declared in a separate compiland in the file Pe4.Pas.

```
(* $INCLUDE:'Pei3.Inf'    --- interface file.*)

Program  CelsiusToFahrenheit(Input,Output);

USES Fah(Fahrenheit);

VAR celsTemp : REAL;

BEGIN

REPEAT

(* Prompt the user for input.*)

write ('Enter Celsius temperature');
write ('       ( -200 or less to exit): ');

(* Read the response.*)

readln(celsTemp);

IF celsTemp <= -200 THEN BREAK; (* Check for
                                   sentinel value*)

(* Convert Celsius temperature to Fahrenheit and
display the result.*)

writeln;
writeln(celsTemp:6:3,' C = ',
                Fahrenheit(celsTemp):6:3,' F');
writeln;

UNTIL FALSE

END.
```

Unit: Pe4.Pas

Pe4.Pas must be compiled separately and linked together with Pe3.Pas. Both files use an interface in the file Pei3.Inf. File Pei3.Inf cannot be compiled separately.

```
(*This file contains an implementation of unit Fah
*)

(* $INCLUDE:'Pei3.Inf'    --- interface file.*)

IMPLEMENTATION OF Fah;

FUNCTION CompFah;   (* (cels : REAL) : REAL *)

(* This function converts Celsius temperature to
Fahrenheit.
ON ENTRY: cels is temperature in degrees Celsius.
RETURN:  The  function  returns  temperature  in
degrees Fahrenheit.
*)

BEGIN

CompFah := cels * (9/5) + 32

END; (* End of CompFah.*)

END. (* End of module.*)
```

Interface: Pei3.INF

Interface for the unit Fah. This file is INCLUDEd into the files Pe3.Pas and Pe4.Pas. This file is not a compiland (it is not compiled separately).

```
INTERFACE (2);   (* 2 is a version number.*)

UNIT Fah(CompFah);

FUNCTION CompFah(cels : REAL) : REAL;

END;
```

Instructions for Compiling, Linking, and Running Example 2

Invoke the compiler, as described in the subsection "Instructions for Compiling, Linking, and Running Example 1," above, and compile Pe3.Pas and Pe4.Pas each separately. Complete the command form as shown below:

```
Pascal
  Source file          Pe3.Pas
  [Object file]        _____
  [List file]          _____
  [Object list file]   _____

Pascal
  Source file          Pe4.Pas
  [Object file]        _____
  [List file]          _____
  [Object list file]   _____
```

Then link the resulting object files, Pe3.Obj and Pe4.Obj. In addition to these files, you must link the object file, PasFirst.obj, which is included with the 10.0 software. To do this, invoke the Linker through the Executive, by typing "Bind" (or as many letters as required to make the command unique) into the Executive command form. Then, complete the Bind command form as shown below:

```
Bind
  Object modules       PasFirst.Obj Pe3.Obj Pe4.Obj
  Run file             Pe4.Run
  [Map file]           _____
  [Publics?]           _____
  [Line numbers?]      _____
  [Stack size]         _____
  [Max array, data,
     code]             _____
  [Min array, data,
     code]             _____
  [Run file mode]      _____
  [Version]            _____
  [Libraries]          _____
  [DS allocation?]     _____
  [Symbol file]        _____
```

For details on the PasFirst.obj object file, see "Linking a Pascal Program" in Chapter 18.

The resulting run file, Pe3.Run, can be invoked by completing the Run command form as shown below. Remember, to terminate the program, enter a Celsius temperature of less than -200.

```
Run
  Run file          Pe3.Run
  [Case]
  [Parameter 1]
  [Parameter 2]
  [Parameter 3]
  [Parameter 16]
```

EXAMPLE 3: BINARY TREE SEARCH

The following example shows a more complicated Pascal program than the examples given above. The program reads a file of characters, orders them (by their ASCII value), and prints them out in order. It stores the characters in an ordered binary tree and traverses the tree in order. Characters are read until it reaches the end of file or a period character (.) whichever comes first. The program uses an additional program parameter, Keyfile, as well as the file Input and Output.

The entire example consists of two compilands (a main program and a module that defines procedures) and an $INCLUDEd file that is not compiled separately.

Instructions for compiling, linking, and running the program appear below.

MAIN PROGRAM: MAINTREE.PAS

This file has the main program for the trees example. The program reads a file of keys, builds an ordered binary tree out of them, then traverses the tree in order, displaying the keys.

```
PROGRAM DisplayOrderedKeys(input,output,keyFile);

(* $INCLUDE:'Tree.Dcl' --- TYPE declarations.*)

VAR keyFile: KeyFileType; (* input file of keys.*)

(* External procedures and functions.*)

FUNCTION BuildTree (VAR keyFile : KeyFileType) :
                                TreeNodePtr;EXTERN;

(*This function builds a tree and returns
 a pointer to the tree.
PARAMETER: keyFile --- the file where the keys
           are.
RETURN: the function returns a pointer to the tree
        built.
*)


PROCEDURE TraverseTreeInOrder(root: TreeNodePtr;
          PROCEDURE Action(key:KeyType)); EXTERN;

(* This procedure traverses a tree in order while
calling a procedure to process each key.
PARAMETERS: root --- pointer to tree root,
            Action --- procedure to process each
            key.
*)

PROCEDURE DisplayKey(key:KeyType); EXTERN;

(* This procedure displays a key on the screen.
PARAMETER: key --- key to display.
*)

(* Internal procedure.*)
```

```
PROCEDURE DisplayTree(root:TreeNodePtr);

(* This procedure displays the keys ordered by
their value on the screen. It writes a heading,
then the keys.
PARAMETER : root --- pointer to the tree root.
*)

BEGIN

(* Write the heading.*)

writeln;
writeln('   ORDERED KEYS');
writeln;

(*Display the keys.*)

TraverseTreeInOrder(root,DisplayKey);

writeln                (* New line at the end*)

END;

BEGIN (* Main program*)

reset(keyFile);

DisplayTree(BuildTree(keyFile));

writeln;
writeln('Program terminated.')

END.
```

MODULE: TREEMODULE.PAS

```
(* This module contains procedures to build and
display trees.*)

module trees[];

(* $INCLUDE:'Tree.Dcl' *)

CONST
    SentinKey = '.'; (* Sentinel key value.*)

FUNCTION GetNewNode(VAR root:TreeNodePtr;
                    key: KeyType) : TreeNodePtr;

(* This function finds a place in a tree where a
key should be inserted, creates a node for
the key and inserts the node into the tree.
It does not fill the node fields.
PARAMETERS: root --- a pointer to the tree root,
            key --- the key.
RETURN: The function returns a pointer to the new
        node.
Note that root can be changed if the tree is
empty. *)

BEGIN

IF root = NIL          (* If tree is empty*)
THEN
    BEGIN
    new(root);         (* root points to new node*)
    GetNewNode := root     (* return the pointer*)
    END

ELSE                        (* tree is not empty *)

    IF key <= root^.nodeKey
    THEN                  (* Insert new node into*)
                          (*     left sub-tree*)

        GetNewNode := GetNewNode(root^.left,key)

    ELSE                   (* Into right sub-tree*)

        GetNewNode := GetNewNode(root^.right,key)

END;

(*--------------------------------------*)
```

```
PROCEDURE FillNode(node : TreeNodePtr;
                       key: KeyType);

(* This procedure initializes new node fields:
left and right pointers to NIL, the key to 'key'.
PARAMETERS: node --- pointer to the node,
            key --- the key.
*)

BEGIN

WITH node^ DO
    BEGIN
    left := NIL;
    right := NIL;
    nodeKey := key
    END

END;

(*-------------------------------------*)

PROCEDURE InsertKey(VAR root : TreeNodePtr;
                    key : KeyType);

(* This procedure inserts a key into a tree.
PARAMETERS: root --- pointer to tree root,
            key --- the key.
*)

BEGIN

FillNode(GetNewNode(root,key),key)

END;

(*-----------------------------------*)
```

```
FUNCTION BuildTree (VAR keyFile : KeyFileType) :
                                TreeNodePtr [PUBLIC];

(*This function builds a tree and returns
 a pointer to the tree.
PARAMETER: keyFile --- the file where the keys
           are.
RETURN: the function returns a pointer to the tree
        built.
*)

VAR
    key : KeyType;        (* holds current key. *)
    root : TreeNodePtr;   (*pointer to tree root.*)

BEGIN

root := NIL;

REPEAT          (*Loop reading keys and inserting*)
                (*them into the tree.             *)

    IF (EOF(keyFile)) THEN BREAK; (*Stop reading*)
                                  (*keys if reached*)
                                  (*  end of file. *)

    (* read a key and insert it into the tree.*)


    read (keyFile,key);
    InsertKey (root,key);

UNTIL key = SentinKey;

BuildTree := root

END;

(*-----------------------------------*)
```

```
PROCEDURE TraverseTreeInOrder(root: TreeNodePtr;
          PROCEDURE Action(key:KeyType)) [PUBLIC];

(* This procedure traverses a tree in order while
calling a procedure to process each key.
PARAMETERS: root --- pointer to tree root,
            Action --- procedure to process each
            key.
*)

BEGIN

IF root <> NIL
THEN
    BEGIN
    (* Traverse left sub-tree*)
    TraverseTreeInOrder(root^.left,Action);
    (* Process root key*)
    Action(root^.nodeKey);
    (* Traverse right sub-tree*)
    TraverseTreeInOrder(root^.right,Action)
    END

END;

(*--------------------------------------*)

PROCEDURE DisplayKey(key:KeyType) [PUBLIC];

(* This procedure displays a key on the screen.
PARAMETER: key --- key to display.
*)

BEGIN

write(key)

END;

END.
```

INCLUDED DECLARATION FILE: TREE.DCL

```
(* Declarations for the tree example *)

TYPE
    KeyType = CHAR;          (* Type of tree key*)
    KeyFileType = FILE OF KeyType;
    TreeNodePtr = ^TreeNode; (* Pointer to tree*)
    TreeNode = RECORD
                nodeKey : KeyType;
                left    : TreeNodePtr;
                right    : TreeNodePtr
               END;
```

INSTRUCTIONS FOR COMPILING, LINKING, AND RUNNING EXAMPLE 3

Example 3 is compiled exactly as Example 2, except that the two compilands are TreeMain.Pas and TreeModule.Pas. The file Tree.Dcl is included automatically in both files because the $INCLUDE metacommand is used in both source files.

After you have compiled, link the resulting object files, TreeMain.Obj and TreeModule.Obj. In addition to these files, you must link the object file, PasFirst.obj, which is included with the 10.0 software. (For details on the PasFirst.obj object file, see "Linking a Pascal Program" in Chapter 18.)

To invoke the Linker through the Executive, type "Bind" (or as many letters as required to make the command unique) into the Executive command form. Then, complete the Bind command form as shown below:

```
Bind
  Object modules          @Filename
  Run file                TreeMain.Run
  [Map file]
  [Publics?]
  [Line numbers?]
  [Stack size]
  [Max array, data, code]
  [Min array, data, code]
  [Run file mode]
  [Version]
  [Libraries]
  [DS allocation?]
  [Symbol file]
```

Note that, in this example, the object file names (PasFirst.Obj, TreeMain.Obj, and TreeModule.obj) are in the at-file, Filename. (For details on how to use at-files, see the Executive Manual.)

The resulting run file, TreeMain.Run, can be invoked by completing the Run command form as shown below. The parameter Inputfile is any file containing ASCII characters that you choose to use. Inputfile must be the name of a real file in your directory.

```
Run
  Run file          TreeMain.Run
  [Case]
  [Parameter 1]     Inputfile
  [Parameter 2]
  [Parameter 3]
  [Parameter 16]
```

This index covers both Volumes 1 and 2. Sections 1 through 12 are in Volume 1. Sections 13 through the Glossary are in Volume 2.

Page numbers in boldface indicate the principal discussion of a topic.

Convergent

2700 North First Street
San Jose, CA 95150–6685

Printed in USA