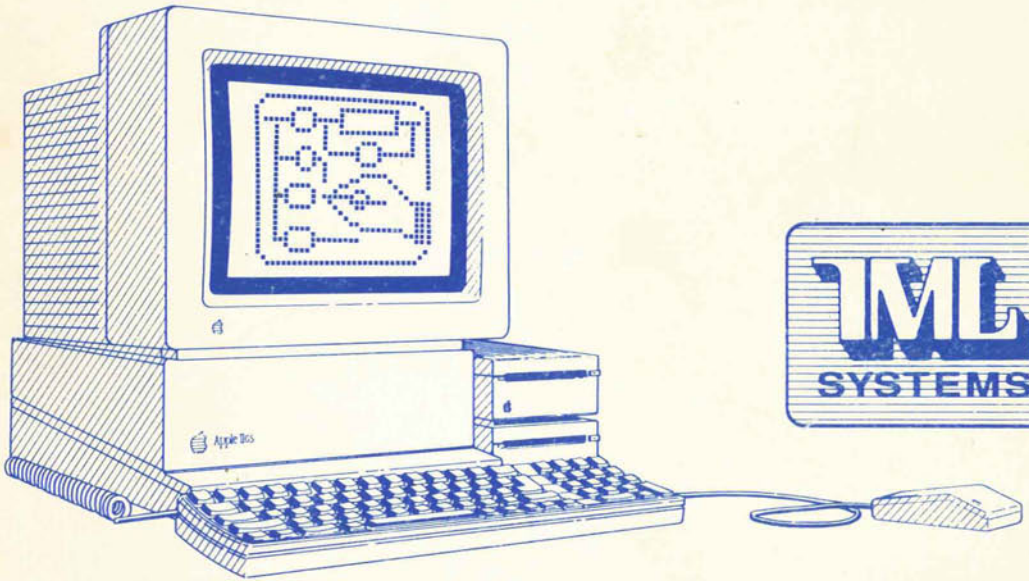
TML Pascal for the APPLE IIGS

User's Guide and Reference Manual



TML SYSTEMS

APW Version

TML Pascal for the Apple IIgs

User's Guide and Reference Manual

Version 1.0, March 1987

4241 Baymeadows Rd., Suite 23
Jacksonville, FL 32217
(904) 636-8592

TML PASCAL LICENSE AGREEMENT

This manual and the software described in it were developed and are copyrighted by TML Systems, Inc. and are licensed to you on a non-exclusive, non-transferable basis. Neither the manual nor the software may be copied in whole or in part except as follows:

1. *You may make backup copies of the software for your use providing that they bear TML Systems' copyright notice.*
2. *You have the right to include the object code provided in the several libraries included with TML Pascal in programs you develop using this software and you also have the right to use, distribute and license such programs to third parties without payment of any further license fees providing that you include the following copyright notice (no less prominently than your own copyright notice) in the software and its documentation: "© 1987 TML Systems, Inc. Certain portions of this software are copyrighted by TML Systems, Inc."*

You may not in any event distribute any of the source files provided as part of this software.
You may only use the software and its documentation at any number of locations or machines so long as there is no possibility of it being used at more than one location or one machine at a time.

CUSTOMER SUPPORT AND PRODUCT UPDATE PLAN

Software Registration. Ycur registration of TML Pascal is ESSENTIAL for you to receive the full benefits of TML Systems' customer services.

Technical Support. We at TML Systems want you to take the greatest advantage of your development tools possible. If you have a technical problem we will be glad to help. Gather ALL the information pertinent to the problem along with your registration number, and call our Technical Support Department at (904) 636-0118 during our normal support hours. You may also write to:

TML Systems, Inc., Technical Support Department
4241 Baymeadows Road, Suite 23, Jacksonville, FL 32217

Remember it is absolutely required that you include your registration number with all correspondence and have it available with you call TML Systems.

Keeping your software up to date. TML Pascal is a very large and sophisticated software package. From time to time, TML Systems will improve its product making it even more powerful and useful to you. You can take advantage of our ongoing development efforts if you have returned your registration card to us. As a registered TML Pascal user, you will receive announcements about major improvements for your software. These announcements will provide you the cost of the update and ordering procedures. Only registered users will receive these update notices and be elgible to purchase the update.

Version	Printing	Date
1.0	First Printing	March 1987

If you have any comments or suggestions regarding either the TML Pascal development system software or this documentation, please send comments to:

TML Systems
4241 Baymeadows Rd., Suite 23
Jacksonville, FL 32217

Your input is extremely valuable in assisting us to continue to provide you with the best development tools possible.

Part I

TML Pascal User's Guide

Table of Contents

Table of Contents

Chapter 1
Introduction

Welcome to *TML Pascal for the Apple IIGS.* The programming language TML Pascal has been designed to meet the needs of the broadest range of programmers possible for the Apple IIGS. The language is solidly based upon the American National Standard for the Pascal language with numerous extensions for programmers acustomed to other Pascal implementations, thus achieving the greatest amount of compatibility possible.

TML Pascal is available in two versions for the Apple IIGS – as a language tool running under the Apple Programmer's Workshop (APW) and as an integrated stand-alone package. This product represents TML Pascal implemented as an APW language tool and therefore can only be used in conjunction with the Apple Programmer's Workshop.

TML Pascal supports Units, random disk I/O, and standard subprograms such as MoveLeft, FillChar, etc. that are found in UCSD implementations of Pascal such as Apple Pascal. And of course language features from the Macintosh version of TML Pascal such as type casting, bit operations, CYCLE and LEAVE statements, Unit bodies, and much more are found in TML Pascal for the Apple IIGS.

TML Pascal for the Apple IIGS has been designed to take specific advantage of and provide access to the new features and capabilities of the Apple IIGS. TML Pascal runs in full 16-bit native mode under ProDOS16 as an APW language tool. Complete and full access in Pascal is provided to every routine in the Apple IIGS Toolbox as well as ProDOS16 and APW. With TML Pascal, you will be able to develop stand-alone ProDOS16 applications, New Desk Accessories, and other APW tools, as well as the ability to easily create definition procedures.

In addition to developing applications which take advantage of the Apple IIGS Toolbox, TML Pascal allows you to develop what we call *plain vanilla* or *textbook* applications. This feature allows you to enter programs directly from textbook examples and compile them. In this evnironment, TML Pascal provides a 20 row by 80 column *console* window in Super HiRes 640 Mode which behaves as a standard CRT terminal.

The TML Pascal User's Guide and Reference Manual have been written to guide you through the use of TML Pascal, however, they are not tutorials. You should be familar with using the Apple IIGS and programming in Pascal, although no specific knowledge of programming the Apple IIGS is necessary.

In order to use TML Pascal, you will require a copy of the Apple Programmer's Workshop (APW) or a compatible product such as ORCA/M for the Apple IIGS, one 3.5 800K floppy disk drive, and a memory expansion card with at least 512K bytes of additional memory for a total of 768K of memory. For development of large applications, two 3.5 800K floppy disks or a hard disk is recommended.

The User's Guide Manual

Chapter 1: Introduction is this chapter. A brief overview of TML Pascal for the Apple IIGS and its documentation is given together with a complete reference of supplemental documentation and texts that can assist your development efforts.

Chapter 2: Setting Up describes the contents of the distribution disk, how to install TML Pascal into the Apple Programmer's Workshop as a language tool, and reviews several common system configurations.

Chapter 3: Getting Started with TML Pascal provides a brief review of the Apple Programmer's Workshop and then takes you step by step through the compilation process necessary to build an application with TML Pascal. File naming conventions are also explained.

Chapter 4: Writing "Plain Vanilla" Applications exposes the capability of TML Pascal to compile *plain vanilla* or *textbook* applications. This capability allows you to take standard Pascal programs and compile them with TML Pascal without having to be familar with the specific details of the Apple IIGS.

Chapter 5: The Apple IIGS Toolbox Interfaces details TML Pascal's access to the extensive collection of ROM and RAM based tools that make the Apple IIGS so unique from previous Apple II's.

Chapter 6: Writing Apple IIGS Applications discusses the techniques of event driven programming and shows how to take advantage of the new Apple IIGS features such as Menus, Windows, QuickDraw, etc. in your own applications.

Chapter 7: Writing Desk Accessories explains the details of designing and writing New Desk Accessories in TML Pascal that can be used from the *Apple Menu* of Apple IIGS applications.

The Reference Manual

The TML Pascal Reference Manual is a complete reference for the Pascal language features implemented by TML Pascal. TML Pascal is based upon the ANS standard for Pascal with numerous extensions for achieving compatibility with the Macintosh version of TML Pascal and UCSD Pascal. Each of the chapters discusses a functional component of the Pascal language.

Chapter 1: Tokens

Chapter 2: Blocks, Scope, and Activations

Chapter 3: Types

Chapter 4: Variables

Chapter 5: Expressions

Chapter 6: Statements

Chapter 7: Procedures and Statements

Chapter 8: Programs and Units

Chapter 9: Input and Output

Chapter 9: Input and Output

Chapter 10: Standard Procedures and Functions

Appendix A: Compiler Error Messages and IOResult Codes provides a summary of the error messages reported by the TML Pascal compiler and the result codes returned by the Pascal function IOResult for I/O operations performed at runtime.

Appendix B: Compiler Directives explains each of the directives which affect the way the TML Pascal interprets the source code it compiles.

Appendix C: Inside TML Pascal offers additional technical information for advanced Pascal programmers, including the memory model, internal representation for data types, calling conventions, and interfacing to assembly language routines.

Appendix D: Comparing TML Pascal with ANS Pascal details the differences between the ANS standard for the Pascal langauge and the implementation of TML Pascal.

Apple IIGS Technical Documentation from Apple Computer, Inc.

While the Apple IIGS provides a new degree of friendliness to the user, the programmer is confronted with the burden of developing software for a much more sophisticated machine. Without the appropriate technical references, the task of programming the Apple IIGS will be nearly impossible. The following paragraphs outline the technical documentation published by Apple Computer for the Apple IIGS. Each of these references is available directly from the Apple Programmer's and Developer's Association (APDA) or from Addison-Wesley.

- *Technical Introduction to the Apple IIGS* is the first book in the suite of technical manuals for the Apple IIGS. It describes all aspects of the Apple IIGS, including its features, general design, and Toolbox.

- *Apple IIGS Hardware Reference* and *Apple IIGS Firmware Reference* cover the hardware details of the Apple IIGS. You will not necessarily need these texts in order to develop applications for the Apple IIGS, however, reading them might provide you with a better insight as to how the machine operates.

- *Programmer's Introduction to the Apple IIGS* provides an excellent introduction to the concepts and guidelines you'll need to know in order to develop quality applications which take specific advantage of the Apple IIGS. While this text does not use TML Pascal, it does review the principles of event-driven programming using the Toolbox and operating system.

- *Apple IIGS Toolbox Reference: Volume 1 and Volume 2* is the complete and authoritative reference for the Apple IIGS's built in set of routines which are collectively known as the Toolbox. For example, the Toolbox contains the software necessary to draw graphical objects on the screen (QuickDraw) and for menus, windows, and sound. The Toolbox supports the Apple desktop user interface and makes developing new and powerful applications much easier to accomplish.

  If you intend to develop applications which take advantage of the Toolbox, then you will find that these two volumes are absolutely necessary. It will be nearly impossible to program the Toolbox effectively without this documentation.

- *Apple IIGS ProDOS 16 Reference* documents the operating system of the Apple IIGS. The details of the System Loader and file manipulation operations are covered in this text.

- *Human Interface Guidelines: The Apple Desktop Interface.* This book documents Apple's standards for the desktop user interface to any program that runs on an Apple IIGS or a Macintosh. If you are writing an application which is to use the desktop user interface, you should study this manual so that you conform to the standards set forth by Apple Computer.

- *Apple Numerics Manual* is the reference for the Standard Apple Numeric Environment (SANE), a full implementation of the IEEE standard for floating-point arithmetic.

In addition to these texts, Apple Computer puslishs a series of *Technical Notes for the Apple IIGS* on a periodic basis. These notes discuss often asked technical questions and other mysteries about the Apple IIGS. The technical notes are available on a subscription basis from the Apple Programmer's and Developer's Association. Below is the address for the Apple Programmer's and Developer's Association.

Apple Programmer's and Developer's Association
290 SW 43rd Street
Renton , WA 98055
(206) 251-6548

Please note that in order to purchase products from APDA you must first be a member. There is a nominal fee for membership in APDA.

In addition to technical documentation on the Apple IIGS, you should also be familar with the Apple Programmer's Workshop (APW) since this version of TML Pascal will only function within the APW as a language tool. For information regarding the operation of APW, consult the *Apple IIGS Programmer's Workshop Reference Manual,* and for programming in 65816 assembler using APW consult the *Apple IIGS Programmer's Workshop Assembler Reference.*

Where to go for more Information

In addition to technical documentation from Apple Computer, you may find one or more of the following texts useful in your development efforts.

The following two books document the Apple IIGS Toolbox. While neither of the books uses TML Pascal for its examples, they still provide a wealth of useful information. In particular, the *Apple IIGS Technical Reference* by Michael Fischer provides exhaustive coverage of the Toolbox, but in a much more readable fashion than Apple Computer's *Apple IIGS Toolbox Reference* volumes.

- *Apple IIGS Technical Reference,* Michael Fischer, Osborne/McGraw-Hill, 1987.

- *The Apple IIGS Toolbox Revealed,* Danny Goodman, Bantam Computer Books, Prentice Hall Press 1986.

The following texts provide an exellent introduction and tutorial to the Pascal langauge.

- *Oh! Pascal!* Michael Clancy and Doug Cooper, W.W. Norton and Company, 1982.

- *Programming in Pascal,* Peter Grogono, Addison-Wesley, 1978.

The three documents listed below are technical references for the Pascal language.

- *Pascal User Manual and Report,* Kathleen Jensen and Nicklaus Wirth, Springer-Verlag, 1985.
- *The American Pascal Standard (with Annotations),* Henry Ledgard, Springer-Verlag, 1985.
- *American National Standard Pascal Computer Programming Language,* ANSI/IEEE 770X2.97-1983, IEEE/Wiley-Interscience, 1983.

And finally, for programmer's integrating 65816 assembly language routines with TML Pascal or just interested in the 65816 processor will find the following two texts outstanding references for the 65816 microprocessor.

- *Programming the 65816, Including the 6502, 65C02 and 65802,* David Eyes and Ron Lichty, A Brady Book, Prentice Hall Press, 1986.
- *65816/65802 Assembly Language Programming,* Michael Fischer, Osborne/McGraw-Hill, 1986.

Acknowlegments

TML Pascal is a trademark of TML Systems, Inc.

Apple is a registered trademark of Apple Computer, Inc.
Apple IIGS and Macintosh are trademarks of Apple Computer, Inc.
ProDOS is a registered trademark of Apple Computer, Inc.

Chapter 2

Setting Up TML Pascal

Before you begin using TML Pascal you should take some precautions to ensure that you protect your software. Since it is impossible to use the TML Pascal distribution disk directly (it does not contain ProDOS16 or APW) it may be sufficient to merely write protect the disk while you use it to install TML Pascal into your copy of APW. However, it is probably wise to make a backup copy of the distribution disk and then place your master distribution disk in a safe place.

Please remember that TML Systems' philosophy of selling quality software at an affordable price with NO COPY PROTECTION can only work if you make it work. As stated in the license agreement, you are permitted to make backup copies of the software for your own archival purposes, but copies of the software may not be given to or used by anyone else.

In order to format a new disk and make a backup copy of the distribution disk, you may use the `SYSUTIL.SYSTEM` program on your Apple IIGS System Disk that came with your Apple IIGS or by using the `INIT` and `COPY` commands available in APW. If you are unfamilar with using either of these tools for making a copy of a disk, then consult their respective manuals. If you are unfamilar with APW and its commands, then you should take some time now to become familar with APW before proceeding. A brief review of APW is given in Chapter 3 of this manual, however, it is in no way a substitute for the APW documentation itself.

Files on the Distribution Disk

The TML Pascal distribution disk contains all files that you will need to build your Pascal development environment. Also included are several example programs to get you well on your way to writing your own applications and desk accessories.

The following is a quick rundown of the files and directories you will find on your TML Pascal distribution disk. As stated above, the TML Pascal distribution disk can not be used alone, you must first install TML Pascal as a language into your copy of APW. The process of installing TML Pascal into APW is detailed in the next section of this chapter and is followed by several sections describing typical system configurations.

```
/TML/

     TMLPASCAL                  The TML Pascal compiler

     TMLPASCALLIB               The TML Pascal runtime libraries

     SYSCMND                    An APW System Command File

     SYSTABS                    An APW System Tabs File

     TOOLINTF/                  The TML Pascal Apple IIGS Tool Interfaces
        RECOMPILE                  An EXEC file to recompile all interfaces

        QDINTF.PAS                 Pascal Units for the GS tools
        GSINTF.PAS
        MISCTOOLS.PAS
        SCHEDULER.PAS
        SOUND.PAS
        NOTESYN.PAS
        TEXTTOOLS.PAS
        APW.PAS
        PRODOS16.PAS
        INTMATH.PAS
        SANE.PAS
        PRINTMGR.PAS
        LISTMGR.PAS
        CONSOLEIO.PAS

        QDINTF.USYM                TML Pascal unit symbol files for each
        GSINTF.USYM                  of the above units
        MISCTOOLS.USYM
        SCHEDULER.USYM
        SOUND.USYM
        NOTESYN.USYM
        TEXTTOOLS.USYM
        APW.USYM
        PRODOS16.USYM
        INTMATH.USYM
        SANE.USYM
        PRINTMGR.USYM
        LISTMGR.USYM
        CONSOLEIO.USYM

     EXAMPLES/                  TML Pascal Source Code Examples

        FIRSTPROG/              The "Hello World" program

           MAKE
           FIRSTPROG.PAS
           FIRSTPROG.ROOT
           FIRSTPROG
```

```
BENCHMARK/                  Simple demo showing Sieve & Selection Sort

    MAKE
    BENCHMARK.PAS
    BENCHMARK.ROOT
    BENCHMARK

CONSDEMO/                   Show use of "Plain Vanilla" console mode

    MAKE
    CONSDEMO.PAS
    CONSDEMO.ROOT
    CONSDEMO

TURTLE/                     Implementation of Apple Pascal's Turtle
                              graphics and demo program.
    MAKE
    TURTLE.PAS
    TURTLE.ROOT
    TURTLE.USYM
    TURTLEDEMO.PAS
    TURTLEDEMO.ROOT
    TURTLEDEMO

GSDEMO/                     Simple desktop multi-window demo

    MAKE
    GSDEMO.PAS
    GSDEMO.ROOT
    GSDEMO

CLOCKNDA/                   A New Desk Accessory in TML Pascal

    MAKE
    CLOCKNDA.PAS
    CLOCKNDA.ROOT
    CLOCKNDA
```

System Configurations

As we have stated before, the TML Pascal distribution disk is not a runnable disk configuration, but rather it contains all the files needed to install TML Pascal into a running copy of the Apple Programmer's Workshop. An Apple IIGS development system using TML Pascal and APW may be configured in numerous ways, however, in the following sections we will address the most likely of configurations – a single 800K floppy disk, two 800K floppy disks, and a hard disk.

If you are already using APW in a configuration you are happy with, then feel free to skip this section and read the section *Installing TML Pascal into APW* below for instructions on how to install TML Pascal into your APW development system.

To build a development environment based upon TML Pascal you will need three different sets of software – TML Pascal software, APW software, and system software from you Apple IIGS System Disk. The following paragraphs outline exactly what software is needed, and how you should install the APW and System software onto your development disk(s). The discussion of how to install TML Pascal is left to the next section. Please note that the sections below will discuss only the *minimal* software requirements. If you have a hard disk for example, you may wish to configure your system with additional tools and utilities that will not be discussed here.

Minimal Software Requirments for TML Pascal development

- *System Software:* From your Apple IIGS System disk.

  Note that the APW distribution disk also contains ProDOS16 and other system software, but does not contain all the RAM based tools, fonts, and other files found on the System Disk. It is usually best to obtain the latest version of the System Disk and use that software.

  The following is the minimal set of software from the Apple IIGS System Disk that you will need.

```
/SYSTEM.DISK/
   PRODOS
   SYSTEM/
      P16
      SYSTEM.SETUP/
         TOOL.SETUP
      TOOLS/
         TOOL014
         TOOL015
         TOOL016
         TOOL020
         TOOL021
         TOOL022
         TOOL023
         other tool files as they become available
      FONTS/
         any font files that you need
```

- *Apple Programmer's Workshop Software:* From your APW disk.

  The following is the minimal set of software from the APW Disk that you will need.

```
/APW/
   APW.SYS16
   SYSTEM/
      LOGIN
      SYSCMND
      SYSTABS
      EDITOR
   LANGUAGES/
      LINKED
```

- *TML Pascal Software:*

  See the Section *Installing TML Pascal* below

One 800K Floppy Disk

A single 800K floppy disk is the minimal development configuration possible. While this configuration might prove to be a little cramped for disk space, it still allows for a complete and powerful development system using TML Pascal.

A single 800K floppy disk development system only contains the minimal software as outlined above. With this configuration, you will have approximately 100K of disk space available for developing your applications, however, it will be impossible to develop with both TML Pascal and the APW Assembler from the same disk since there is not enough space for all the files you would need. If you need to develop an application using both Pascal and assembly, then you will need to create two development disks – one for each language, and copy the resultant object code from one disk to the other for linking the application.

In order to create a single 800K development disk, first initialize a new disk and give it the name `/APWWORK`. Then copy the APW and system software exactly as outlined above onto your disk. After doing this you should have a disk with the following contents.

(1) Initialize a new 800K floppy disk and give it the name `/APWWORK`.

(2) Copy the minimal system software outlined above from the Apple IIGS System Disk onto the `/APWWORK` disk.

(3) Copy the minimal APW software outlined above from the APW distribution disk onto the `/APWWORK` disk.

(4) Create the `LIBRARIES/` and `UTILITIES/` directories that APW requires. This can be done using the APW command `CREATE`.

```
CREATE /APWWORK/LIBRARIES
CREATE /APWWORK/UTILITIES
```

(5) Review the `/APWWORK` disk to ensure its contents match the listing below, and then proceed to the section *Installing TML Pascal into APW.*

```
/APWWORK/
    PRODOS
    APW.SYS16
    SYSTEM/
        P16
        LOGIN
        SYSCMND
        SYSTABS
        EDITOR
        SYSTEM.SETUP/
            TOOL.SETUP
        TOOLS/
            TOOL014
            TOOL015
            TOOL016
            TOOL020
            TOOL021
            TOOL022
            TOOL023
            other tool files as they become available
        FONTS/
            any font files that you need
    LANGUAGES/
        LINKED
    LIBRARIES/
    UTILITIES/
```

Two 800K Floppy Disks

With two 800K floppy disks it is quite reasonable to configure the `/APWWORK` disk just as above, and use second floppy disk for the storing the source code to the applications you are developing. Alternatively, you may wish to configure the `/APWWORK` disk to contain both TML Pascal and the APW Assembler. In this case, it is necessary to install the TML Pascal `TOOLINTF/` directory (discussed in *Installing TML Pascal into APW)* on the second floppy disk so that there is sufficient disk space for the APW Assembler and its associated files.

A Hard Disk

A hard disk development system of course offers the greatest amount of flexibility. In this case, we would recommend installing all of the Apple IIGS System Disk and the APW software on the hard disk. However, be sure to use the latest version of the system software from the Apple IIGS System Disk rather than the subset of system software found on the APW disk.

Installing TML Pascal into APW

Now that you have created a development disk containing the necessary system and APW software we will outline the steps necessary to install TML Pascal into the Apple Programmer's Workshop environment as a langauge.

The following instructions assume that you have volume named /APWWORK and that it already contains the necessary System and APW files and directories applicable to your particular system configuration as outlined in the *System Configurations* section above. They also assume that you have booted your Apple IIGS using your properly configured system and are in the APW shell. If your volume is not named /APWWORK, then replace /APWWORK with the name of your volume in the instructions below.

(1) Copy the file TMLPASCAL from the distribution disk into the /APWWORK/LANGUAGES/ directory of your APW development disk. This can be done using the COPY command of APW. For example,

```
COPY   /TML/TMLPASCAL   /APWWORK/LANGUAGES/
```

(2) Copy the file TMLPASCALLIB from the distribution disk into the /APWWORK/LIBRARIES/ directory of your APW development disk. Again, this can be done using the COPY command of APW. For example,

```
COPY   /TML/TMLPASCALLIB   /APWWORK/LIBRARIES/
```

(3) Create a directory named TOOLINTF on you APW disk in the same directory as the LANGUAGES/ and LIBRARIES/ directories are located. For example,

```
CREATE /APWWORK/TOOLINTF
```

Now copy all the files in the /TML/TOOLINTF/ directory that end with the suffix .USYM into the /APWWORK/TOOLINTF directory you just created. This can be done with the following APW command.

```
COPY   /TML/TOOLINTF/=.USYM   /APWWORK/TOOLINTF/
```

If you have sufficient disk space on your APW disk (i.e. you are not using a single 800K disk configuration) then you may choose to copy all the files in the /TML/TOOLINTF/ directory onto your APW disk. These remaining files are the Pascal source code files to the units represented by the files ending in .USYM. It is not necessary to recompile these file, however, you may wish to have them readily available for reference. To copy the entire directory, you could use the APW command.

```
COPY   /TML/TOOLINTF/=   /APWWORK/TOOLINTF/
```

Please remember that the Pascal source code files in this directory may not be reproduced without written permission from TML Systems, Inc.

(4) The SYSCMND file in the /APWWORK/SYSTEM/ directory of your APW disk must be modified to include the new language TMLPASCAL. If you have not made any changes to the SYSCMND file that was delivered on your APW distribution disk, then you may merely replace it with the SYSCMND file on your TML Pascal distribution disk.

If you have made other changes to this file then, you will want to add the following line to the file /APWWORK/SYSTEM/SYSCMND. To do so, you will need to enter the APW editor to edit this file and insert the following line somewhere in the file (usually in alphabetical order).

```
TMLPASCAL       *L              30              The TML Pascal compiler
```

If you do not want TML Pascal to be restartable then you should only enter an *L* instead of **L*.

(5) The SYSTABS file in the /APWWORK/SYSTEM/ directory of your APW disk must be modified to include tabs settings for files which will have the new langauge subtype TMLPASCAL. If you have not made any changes to the SYSTABS file that was delivered on your APW distribution disk, then you may merely replace it with the SYSTABS file on your TML Pascal distribution disk.

If you have made other changes to this file, then you will want to add the following 3 lines to the file /APWWORK/SYSTEM/SYSTABS. To do so, you will need to enter the APW editor to edit this file and insert the following line somewhere in the file (usually in numerical order by language number). TML Systems recommends the following 3 lines.

```
30
000
00010001000100010001000l    ....  00010002
```

(6) Modify the LOGIN exec file in the /APWWORK/SYSTEM/ directory of your APW disk to include the following line. To do this, you will again have to enter the APW editor.

```
PREFIX  7  /APWWORK/TOOLINF
```

As discussed in Chapter 3, TML Pascal uses the ProDOS 16 prefix 7 to search for unit symbol files required for a USES clause when the appropriate file can not be found in prefix specified by the compiler's $P directive. Remember the default prefix for the $P directive is "0/" for the ProDOS16 prefix 0 (the current prefix). If you are not familiar with this terminology, add the line to your LOGIN exec file for now and then study Chapter 3 of the User's Guide and Appendix B of the Reference Manual.

This step is not absolutely required. However, if you do not use this convention you will need to add the compiler's $P directive to all of the examples on your TML Pascal distribution disk before they will compile successfully.

(7) Copy the EXAMPLES/ directory onto your /APWWORK disk or some other work disk.

(8) Reboot the Apple IIGS. Since we have changed (or replaced) the SYSCMND file, it is necessary to reboot the machine in order for the changes to the SYSCMND file to take effect.

You have now successfully installed TML Pascal into your copy of the Apple Programmer's Workshop. The remaining chapters of this manual are intended to show you how to use TML Pascal to develop applications and desk accessories.

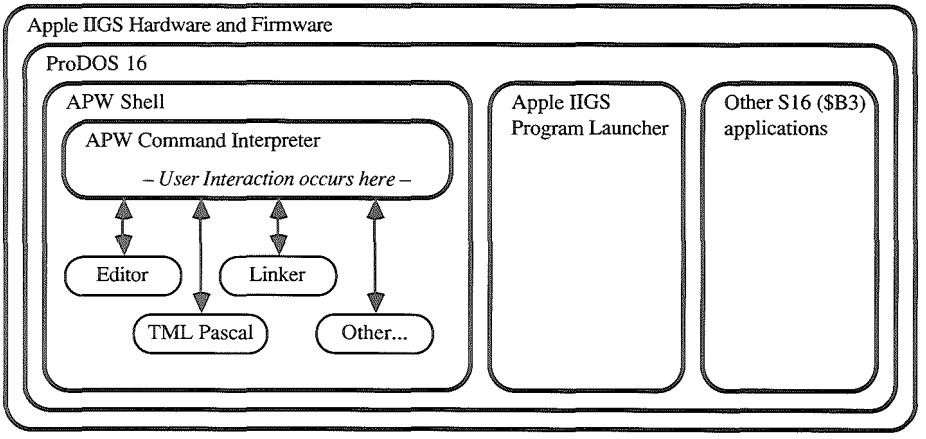
Chapter 3
Getting Started

This chapter reviews the use of TML Pascal within the Apple Programmer's Workshop environment. It assumes that you have successfully created a development environment as discussed in Chapter 2.

Writing your first TML Pascal Program

In this section we provide a brief overview of the Apple Programmer's Workshop and the steps necessary to write our first program – the infamous "Hello World".

The following picture illustrates the relationship between the various components of your Apple IIGS development system. The APW is a application which executes under ProDOS16 and provides numerous built in commands as well as providing for a *shell environment* in which language compilers, utilities, and other tools may execute such as the APW Editor, the APW Linker, and of course TML Pascal. In order to access the commands and other tools in APW, a command line interpreter is provided for interacting with the APW user. The command line interpreter accepts commands from the user and then invokes the appropriate tool to carry out the operation indicated by the command. If several commands must be repeated often, APW allows you to create EXEC files which automate the process of entering several commands, one after the other.



Thus, the process of writing, compiling, and executing a program developed in TML Pascal merely requires that you instruct APW to perform certain operations.

In the following paragraphs we show how to write our first TML Pascal program – "Hello World". For those who would rather not type in this program, it can be found on the TML Pascal distribution disk as the file FIRSTPROG.PAS.

Before creating the FIRSTPROG.PAS source code file, the APW current language must be set to TML Pascal. This is done by typing the command TMLPASCAL. APW uses the current language to set the subtype field of a source code file created by the editor so that when the COMPILE command is issued, APW can detemine which compiler to invoke.

Once the current language type has been set, enter the APW editor so that the source code for FIRSTPROG.PAS can be entered. This is done by entering the command EDIT FIRSTPROG.PAS. From within the editor type in the following five line program.

```
PROGRAM FirstProg(Input,Output);
BEGIN
   Writeln('Hello World');
   Readln;
END.
```

To exit the editor, press the COMMAND and Q keys simultaneously which returns you to the Editor's main menu. From here save the file and then exit.

In order to compile our first Pascal program, you merely need to enter the command:

```
COMPILE FIRSTPROG.PAS
```

This command will invoke the TML Pascal compiler since the language subtype for this file is TMLPASCAL. The TML Pascal compiler does not directly create a stand-alone application, but rather object code. The APW Linker must be used to link the object code generated by the compiler with any runtime libraries that are needed from the TMLPASCALLIB library. To invoke the linker to link our application enter the command:

```
LINK FIRSTPROG KEEP=FIRSTPROG
```

After the link is complete, your disk should contain the application FIRSTPROG as specified by the KEEP parameter to the linker. To run our first TML Pascal program merely type its name.

```
FIRSTPROG
```

Note that the linker has created the application with the ProDOS 16 filetype of EXE ($B5). This file type indicates that the application is a *Shell Load* file. If you would like to be able to run this application other environments, for example the Apple IIGS Program Launcher, you must change its filetype to be S16 ($B3) indicating that it is a standard *Load* file. This is accomplished by using the APW command FILETYPE as shown below.

```
FILETYPE FIRSTPROG S16
```

The following table reviews the steps taken to Edit/Compile/Link and run an application developed in TML Pascal.

APW Command	*Comments*
TMLPASCAL	Set the current APW language to TMLPASCAL.
EDIT FirstProg.Pas	Create a new file FirstProg.Pas whose language is TMLPASCAL since the current language is TMLPASCAL and enter the Editor.
COMPILE FirstProg.Pas	Invoke the TML Pascal compiler to compile the file FirstProg.Pas. The compiler creates the object code output file FirstProg.Root.
LINK FirstProg Keep=FirstProg	Invoke the APW Linker to link the file FirstProg.Root with any necessary runtime library routines and create the application shell load file FirstProg whose filetype is EXE ($B5)
FILETYPE FirstProg S16	Optional step to change the load file's filetype to S16 ($B3) for a stand-alone ProDOS 16 application or to $B9 for a new desk accessory.
FirstProg	Run the newly created application.

A Lesson on Stacks

Pascal is a *recursive* language. By recursive, we mean that Pascal procedures and functions may call themselves (ie. there can be more than one *activation* of a procedure or function at any given time). Because there can be more than one activation of a procedure or function, there must be multiple copies of a procedure's or function's local variables. To accomplish this, Pascal uses a *runtime stack* to allocate storage for local variables as well as other infomation.

The stack does not have an unlimited size. In fact, the default stack size allocated by TML Pascal for applications is only 8K bytes. Thus, a procedure which declares a single array requiring 10,000 bytes of storage will not execute since there would be insufficient space on the stack to allocate storage for the array.

TML Pascal does, however, provide a mechanism for changing the size of the runtime stack with the *$StackSz* directive (see Appendix B of the Reference Manual). The *$StackSz* directive has a single argument which is an integer value specifying the number of bytes to allocate for the runtime stack. For example,

```
{$StackSz 10240 }
```

causes the runtime stack to be created with 10K bytes. There are two restrictions on the use of the *$StackSz* directive. First, the directive must appear before the keyword `PROGRAM`, and second the amount of memory that can be requested for the stack is not unlimited. The Pascal runtime stack is allocated in *Bank 0* of the Apple IIGS memory which has approximately 40K bytes of storage available for use by an application. Since this memory is required for other uses as well, it is wise not to attempt to allocate all of this memory. If an application requires a large amount of storage, it may have to allocate it by

using global variables instead of local variables.

The following two examples demonstrate the use of the *$StackSz* directive. The first exmple fails to run correctly since it does not have sufficient stack space, while the second example will run correctly.

```
PROGRAM Test1;                          { uses default stack size of 8K bytes }

PROCEDURE BigStack;
VAR I: Integer;
    Arr: array[1..5000] of Integer; { requires 10,000 bytes of storage }
BEGIN
   FOR I := 1 TO 5000 DO
      Arr[I] := I;
END;

BEGIN
   BigStack;
END.

{$StackSz 11264 }     { an 11K stack }
PROGRAM Test2;

PROCEDURE BigStack;
VAR I: Integer;
    Arr: array[1..5000] of Integer;  { requires 10,000 bytes of storage }
BEGIN
   FOR I := 1 TO 5000 DO
      Arr[I] := I;
END;

BEGIN
   BigStack;
END.
```

File Naming Conventions

Many diffent files are created and used by TML Pascal and the other APW tools. Thus, a file naming convention has been adopted to help identify the creator and/or contents of otherwise similar files by reading their names. The convention defines a set of file name extensions – a period followed by a few letters – appended to the end of the main part of a file name. Thus, different yet related files are logically associated because they have the same base name.

The following table lists each of the file name extensions that are encountered in developing with TML Pascal and the APW 65816 assembler, what creates the file, and a brief comment about its contents.

File Name	*Created by*	*File Contents*
EXAMPLE.PAS	Editor	is the recommended extension for Pascal source programs.
EXAMPLE.USYM	TML Pascal	is the unit symbol file created by the TML Pascal compiler when compiling a Pascal unit. The file saves the symbol table information about this unit so that when the unit is named in a USES clause its symbol table can be restored.
EXAMPLE.ROOT	TML Pascal	is the object code file. Unlike the APW 65816 assembler, TML Pascal generates all code to the single file ending with the .ROOT suffix.
EXAMPLE.ASM	Editor	is the recommended extension for 65816 assembly source programs.
EXAMPLE.ROOT EXAMPLE.A	Assembler	are the object code files created by the APW 65816 assembler. The first module in a assembly source file is assembled to the file EXAMPLE.ROOT, all subsequent modules are assembled to the file EXAMPLE.A.
EXAMPLE	Linker	is the name of the final application or desk accessory.

ProDOS16 File Types used by APW

The following table lists the various ProDOS 16 filetypes used by files created and manipulated with the Apple Programmer's Workshop and TML Pascal. For a complete list of ProDOS 16 file types consult the *Apple IIGS ProDOS 16 Reference.*

Hex Filetype	*Named Filetype*	*Meaning*	*Comments*
$00	–	Binary File	For TML Pascal's unit symbol files (`.USYM` files).
$04	TXT	Text File	Standard ASCII text files.
$B0	SRC	Source	Source code files for an assembler or compiler.
$B1	OBJ	Object	Object code output from an assembler or compiler and input to the Linker.
$B2	LIB	Library	Library of object code modules grouped by the `MAKELIB` utility. These are usually placed in the `LIBRARIES/` directory.
$B3	S16	Load	Stand-alone ProDOS16 application
$B4	RTL	Run Time Library	Code segments that represent general utilites that can be shared by applications.
$B5	EXE	Shell Load	Stand-alone ProDOS16 application intended to run within a shell (such as APW)
$B6	STR	Startup Load	A load file which is meant to be executed once at system

			start-up, returning to the system when it is done, but whose code remains resident in memory.
$B7	–	Startup Load	A load file which is meant to be executed once at system start-up, returning to the system when it is done, and whose code is removed from memory when completed.
$B8	CDA	Classic DA	Code for a Classic Desk Accessory
$B9	NDA	New DA	Code for a New Desk Accessory
$BA	–	Tool	A RAM based tool such as the Menu or Window Manager.

Where to go from here

Now that you have written your first program in TML Pascal you should continue through the rest of this User's Guide to discover more about writing *Plain Vanilla* programs, Apple IIGS event-driven applications, and of course desk accessories. And even if you already know how to program Pascal, you should also spend some time reading the TML Pascal Reference Manual. You will discover that TML Pascal is full of interesting features that will help you write better and more powerful applications. For example, TML Pascal includes such things as *static* and *univ parameters, type casting, unit bodies,* and much more.

Chapter 4

Writing "Plain Vanilla" Applications

Introduction to Plain Vanilla

Programming the Apple IIGS can sometimes prove to be an intimidating endeavor. For this reason, not only has the TML Pascal compiler been designed for developing Apple IIGS applications and desk accessories using the desktop interface, but it has also been designed to provide for a more traditional Pascal programming environment. That is, the ability to write useful applications without dealing with Windows, Menus, Controls, etc.

This type of application is called a *plain vanilla* or *textbook* application. Plain vanilla applications are still stand-alone ProDOS 16 applications, but they represent the more typical standard Pascal program one might find in a textbook or from more "traditional" computer environments. These programs make little or no use of the Apple IIGS Toolbox and thus allow one to quickly port Pascal programs to the Apple IIGS.

A plain vanilla program executes in the Apple IIGS Super HiRes 640 mode. The compiler generates code which initializes appropriate Apple IIGS tools and creates a single window on the screen which behaves just as a standard CRT monitor with a 20 row by 80 column display. Within this window, you will be able to use the standard Pascal *Readln* and *Writeln* operations, and because it is a Apple IIGS window you will also be able to use Quickdraw graphics.

Plain vanilla programming is especially useful for programmers just learning the Pascal language and then exploring the fascinating features of the Apple IIGS. Advanced programmers will also enjoy this feature of TML Pascal, as well, for building "quick and dirty" tools or testing a new piece of code without having to go through the details of using the Apple IIGS tools.

The Source File

The structure of a *plain vanilla* program is very simple and straightforward. In fact, the only requirement for programming a plain vanilla program is to include the file parameters *(Input,Output)* in the program heading. The compiler uses the presence or absence of these file parameters to distinguish between plain vanilla programs and Apple IIGS applications and desk accessories.

The `FIRSTPROG` example from Chapter 3 is a plain vanilla program.

```
PROGRAM FirstProg(Input,Output);
BEGIN
   Writeln('Hello World');
   Readln;
END.
```

Note the presence of *(Input,Output)* in the program header. If these file parameters had not appeared in the program heading, then the compiler would not have generated code to initialize the Apple IIGS and create the plain vanilla window for displaying the message *'Hello World'*. Thus, this program would fail to execute properly as a non-plain vanilla program.

If we modify the above program to include a for loop as shown below you will discover that as lines are written to the bottom of the window the content of the window is scrolled up to make room for each subsequent line.

```
PROGRAM FirstProg(Input,Output);
VAR I: Integer
BEGIN
   FOR I := 1 TO 50 DO
      Writeln('Hello World');
   Readln;
END.
```

As you might expect, if your program is writing a lot of output, much of it could scroll past the top of the window before you had the opportunity to read it. To help solve this problem, TML Pascal allows you to stop screen output at any time by pressing the mouse button and holding it down. As soon as the mouse button is released, output will resume.

Plain vanilla programs are not actually completely stand-alone programs. Because the implementation of the plain vanilla window is created in the Apple IIGS Super HiRes mode using the Window Manager, the window manager RAM based Apple IIGS tool must be in the `SYSTEM/TOOLS/` directory of your boot disk (see Chapter 2). If the RAM based Window Manager tool is not available then plain vanilla programs will fail to execute.

See the *Technical Details* section below for more information regarding the implementation of plain vanilla programs in TML Pascal.

Using ConsoleIO

In addtion to providing simple *Writeln* and *Readln* operations in the plain vanilla window, TML Pascal also provides a special unit named *ConsoleIO* for implementing the traditional UCSD Pascal screen control operations. These include the following operations:

```
FUNCTION  KeyPressed: Boolean;
FUNCTION  ReadChar: Char;
PROCEDURE GotoXY(x,y: Integer);
PROCEDURE EraseScreen;
PROCEDURE ClearEOL;
PROCEDURE InsertLine;
PROCEDURE DeleteLine;
```

In order to use these operations in your program, you merely specify the name ConsoleIO in the program's uses clause. For example,

```
PROGRAM Test(Input,Output);
USES ConsoleIO;
BEGIN
   { Your code goes here }
END.
```

In addition to these operations, the ConsoleIO unit also provides for string to number and number to string conversions and a special routine for using *dithered colors*. For more information about the ConsoleIO unit, see Chapter 5.

Adding Graphics to Plain Vanilla

Because plain vanilla programs actually run in the Apple IIGS Super HiRes 640 mode using QuickDraw and the Window Manager to create the plain vanilla window, it is possible to use QuickDraw graphics in your plain vanilla programs.

In order to use QuickDraw, merely include the unit name QDIntf in your program's USES clause. When your plain vanilla program begins, QuickDraw has already been intialized and ready to use. For example,

```
PROGRAM Test(Input,Output);
USES QDIntf;
BEGIN
   { Your code goes here including use of QuickDraw operations }
END.
```

Some Examples

Your TML Pascal distribution disk contains two additional plain vanilla applications besides the FIRSTPROG demo discussed in Chapter 3. These are BENCHMARK and CONSDEMO. The BENCHMARK example program performs the classical Sieve of Erastothenes and Selection Sort compiler benchmarks which tests a compiler's code generation capability. The CONSDEMO is a more advanced example which shows the use of the ConsoleIO unit and the use of QuickDraw graphics using the QDIntf unit.

Technical Details

This section is for more advanced programmers who would like to know exactly what code is executed to create the plain vanilla environment. After performing the standard initialization required for all Pascal programs and before control is given to the main program of a plain vanilla application, the compiler generates a call to the subroutine _PASTASTART which performs the necessary operations to create the plain vanilla environment.

The following is a source code fragment in Pascal which shows the functionality of the _PASTASTART routine.

```
TLStartUp;                                      { init Tool Locator }
MyMemoryID := MMStartUp;                        { init Memory Manager }

{ Allocate 4 pages of memory in bank 0 for use by GS Tools.
  3 pages for QuickDraw,
  1 page for Event Manager }

ToolsZeroPage :=
    NewHandle(4 * 256,                          { allocate 4 pages }
              MyMemoryID,                       { User ID for memory blocks }
              fixedBank+fixedblk+locked,        { Attributes }
              ptr(0));                          { start in bank 0 }

QDStartUp
    (LoWord(ToolsZeroPage^),                    { low address of the first
                                                  3 zero pages }
     $80,                                       { 640 mode }
     160,                                       { max size of scan line }
     MyMemoryID);                               { Memory user ID }

EMStartUp
   (LoWord(ToolsZeroPage^) + $300,              { low address of the
                                                  4th zero page  }
    20,                                         { event queue size }
    0,                                          { X min clamp }
    640,                                        { X max clamp }
    0,                                          { Y min clamp }
    200,                                        { Y max clamp }
    MyMemoryID);                                { Memory user ID }

{ Now load RAM based tools (and RAM patches to ROM tools!) }
ToolRec.NumTools := 3;
ToolRec.Tools[1].TSNum := 4;                    { QuickDraw // }
ToolRec.Tools[1].MinVersion := 1;
ToolRec.Tools[2].TSNum := 6;                    { Event Manager }
ToolRec.Tools[2].MinVersion := 1;
ToolRec.Tools[3].TSNum := 14;                   { Window Manager }
ToolRec.Tools[3].MinVersion := 0;
LoadTools(ToolRec);                             { Load the tools I need }

WindStartUp(MyMemoryID);                        { init Window Manager }
Refresh(nil);

PlainVanillaWndPtr := NewWindow(NewWindRec); { NewWindRec is defined
                                               appropriately. }
SetPort(GrafPtr(PlainVanillaWndPtr));

SetForeColor(0);
SetBackColor(15);
GotoXY(1,1);
ShowCursor
```

Chapter 5

The Apple IIGS Toolbox Interfaces

This chapter outlines TML Pascal's access to and use of the Apple IIGS Toolbox which is of course the basis for developing event-driven, desktop based applications and desk accessories. The Toolbox is a large collection of software organized into several different functional components called *Tool Sets* or *Managers.* Within each tool set is a collection of routines which provide for the functionality of the tool set. Each tool set is assigned a unique *tool number* and each routine within a tool set is assigned a unique *function number.*

TML Pascal provides access to the Apple IIGS Toolbox via a collection of Pascal units. Each unit defines an assortment of Constant, Type, Procedure, and Function declarations which correspond to one or more of the Apple IIGS tool sets. Thus, whenever an application requires access to any of the Toolbox it merely includes a USES clause which specifies the appropriate Pascal unit which defines the interfaces to the required Toolbox routines. These Pascal units are in the `TOOLINTF/` directory of the TML Pascal distribution disk. For example, the following USES clause makes the QuickDraw II tool set available.

```
USES QDIntf;
```

In addition to the Toolbox interfaces, TML Pascal provides three additional units in its `TOOLINTF/` directory. These are the `PRODOS16`, `APW`, and `CONSOLEIO` units. These units provide access to the ProDOS16 file system calls, the APW shell calls, and the special TML Pascal plain vanilla calls respectively. Each of these units are discussed in the section *Other TML Pascal Units* at the end of this chapter.

Review of the Apple IIGS Tools

With the release of the Apple IIGS System Disk version 1.1, 28 different tool sets have been defined as part of the Apple IIGS Toolbox. Each of these tool sets is listed in the table below together with the TML Pascal unit which defines its interface. Also indicated in the table is whether or not the particular tool set resides in the Apple IIGS's Read Only Memory (ROM) or on disk as a tool file and thus must be loaded into Random Access Memory (RAM) before it may be used.

If a tool set is to reside in RAM, then its corresponding tool file must be available in the `SYSTEM/TOOLS/` directory of the boot disk. The name of a tool file is `TOOLxxx`, where `xxx` is a three digit number corresponding to the tool set's assigned tool number. For example, the Window Manager's tool file has the name `TOOL014`.

Tool Number	*Tool Name*	*Pascal Unit*	*RAM*	*ROM*
1	Tool Locator	GSIntf	-	X
2	Memory Manager	GSIntf	-	X
3	Miscellaneous Tools	MiscTools	-	X
4	QuickDraw II	QDIntf	-	X
5	Desk Manager	GSIntf	-	X
6	Event Manager	GSIntf	-	X
7	Scheduler	Scheduler	-	X
8	Sound Manager	Sound	-	X
9	Apple Desktop Bus	n/a	-	X
10	SANE	SANE	-	X
11	Integer Math	IntMath	-	X
12	Text Tools	TextTools	-	X
13	*Reserved for System Use*	–	-	-
14	Window Manager	GSIntf	X	-
15	Menu Manager	GSIntf	X	-
16	Control Manager	GSIntf	X	-
17	System Loader	Loader	X	-
18	QuickDraw Auxilary Routines	QDIntf	X	-
19	Print Manager	PrintMgr	X	-
20	Line Edit	GSIntf	X	-
21	Dialog Manager	GSIntf	X	-
22	Scrap Manager	GSIntf	X	-
23	Standard File	GSIntf	X	-
24	Disk Utilities	n/a	X	-
25	Note Synthesizer	NoteSyn	X	-
26	Note Sequencer	n/a	X	-
27	Font Manager	GSIntf	X	-
28	List Manager	ListMgr	X	-

Release version 1.0 of TML Pascal does not provide the interfaces to three of the Apple IIGS tool sets. These are indicated above by the *n/a* in the Pascal Unit column of the table. Also, note that while the interfaces to some RAM based tools are provided, they are not available with the Apple IIGS System Disk version 1.1. For example, the interfaces to the Print Manager and List Manager are provided, but their tool files will not be available until a subsequent release of the Apple IIGS System Disk. Contact you Apple Dealer for the latest released system software.

What do the Tools Do?

The following paragraphs provide a very brief synopsis of the functionality of each of the Apple IIGS tool sets. This synopsis is intended only as a very brief introduction to each of the tool sets, and you should consult the *Apple IIGS Toolbox Reference, Volumes 1 and 2* for a detailed discussion of each tool set.

Tool Locator

The Tool Locator is the most important of the Apple IIGS tool sets. Without the Tool Locator, it would be impossible to access any other tool sets. The Tool Locator allows you to load tool sets from disk into RAM and invoke a tool routine as described above without knowing where in memory the routine's code is actually located.

Memory Manager

The Memory Manager is the second most important tool set after the Tool Locator. This tool is entirely responsible for the allocation, deallocation, and repositioning of memory blocks on the Apple IIGS. The manager keeps track of how much memory is free and what parts are allocated and to whom.

Miscellaneous Tools

The Miscellaneous Tools consists mostly of system-level routines that must be available to most other tool sets.

QuickDraw II

QuickDraw II is the tool set that controls the graphics environment of the Apple IIGS and draws simple objects and text. All other tools which create graphical objects such as the Window Manager call the QuickDraw II tool set.

Desk Manager

The Desk Manager is the tool which enables an application to support desk accessories, both classic desk accessories and new desk accessories.

Event Manager

The Event Manager allows applications to monitor and react to a user's actions, such as those involving the mouse and keyboard.

Scheduler

The Scheduler delays the activation of a desk accessory or other task until the resources that the task/desk accessory requires become available.

Sound Manager

The Sound Manager provides access to the Apple IIGS's sound hardware for creating basic sounds.

Apple Desktop Bus

The Apple Desktop Bus is a method and a protocol for connecting input devices, such as keyboards and mice with the Apple IIGS. The routines in this tool set are used to send commands and data between the Apple Desktop Bus Microcontroller and the rest of the system.

SANE

SANE implements Apple's Standard Apple Numeric Environment. It is an extended-precision IEEE 754 and 854 conformant implementation of floating point arithmetic.

Integer Math

This tool set consists of a varied collection of operations for integers. These include multiplication, division, conversions, etc.

Text Tools

The Text Tools provides an interface between Apple II character device drivers, which must be executed in emulation mode, and applications running in native mode.

Window Manager

The Window Manager creates the desktop environment and is responsible for the creation and manipulation of windows.

Menu Manager

The Menu Manager controls and maintains the use of pull-down menus and the items in the menus for an application.

Control Manager	The Control Manager consists of all the routines necessary to manipulate controls. Controls are such things as scroll bars, radio buttons, check boxes, etc.
System Loader	The System Loader is responsible for loading and relocating code such as applications and desk accessories to memory.
QuickDraw Auxilary Routines	This tool contains additional routines which complement the QuickDraw tool set.
Print Manager	The Print Manager allows an application to use QuickDraw routines to print text and graphics to an Imagewriter or LaserWriter.
Line Edit	Line Edit allows a program to present text on the screen and allows a user to edit that text.
Dialog Manager	The Dialog Manager provides the routines which allow an application to create and use dialog boxes and alerts as a means of communication between a user and your program.
Scrap Manager	The Scrap Manager implements the desk scrap, which implements the *Cut, Copy*, and *Paste* operations of an application.
Standard File	The Standard File tool set implements the standard user interface for specifiying a file to be opend or saved.
Note Synthesizer	The Note Synthsesizer is used to create complex musical sounds using the Apple IIGS's sound hardware.
Font Manager	The Font Manager is the tool set which allows your application to make use of different text fonts, font styles, etc.
List Manager	The List Manager is used to create lists which are used to display and allow selection of a variable amount of similar data.

How Calling a Tool Routine Works

This section is intended for advanced programmers who want to understand how a tool routine is actually invoked from Pascal. If you are content with the fact that everything works and that the tool routines are essentially additional built in routines then feel free to skip this section.

As discussed above, TML Pascal provides access to the Apple IIGS Toolbox via a collection of Pascal units which defines the appropriate Pascal interface to each of the routines of a particular tool set. Each tool routine is defined as a either a procedure or function depending upon whether or not the routine returns a value on the stack and may have zero or more parameters. Finally, the procedure or function declaration is completed with the *Tool Directive* (see Chapter 7 of the Reference Manual). The tool directive is a special extension to TML Pascal for the Apple IIGS for the specific purpose of defining interfaces to the Toolbox.

The following procedure declaration is taken from the QDIntf.Pas unit, and is the interface to the MoveTo procedure in the QuickDraw tool set.

```
PROCEDURE MoveTo(h,v: Integer);              Tool 4,58;
```

As you can see, the procedure declaration is completed with the tool directive *Tool 4,58.* The first integer in the tool directive specifies the tool set to which the the routine belongs. In this case, it is tool set number 4 which is the QuickDraw tool set. The second integer is the function number of the routine within the tool set. Every routine within a tool set is assigned a unique function number. The MoveTo routine is assigned number 58. Together, these two integers uniquely identify the MoveTo procedure in the entire Apple IIGS Toolbox.

The Apple IIGS defines a consistent mechanisim for invoking a Toolbox routine. To invoke a Toolbox routine, space for any function result value must first be reserved on the stack followed by pushing the values of any parameters. Then the 65816 X-register must be loaded with the desired Toolbox routine's function number and tool set number such that *X-register = 256 * function number + toolset number.* Finally, a *jump subroutine long* instruction is made to the address $E10000 which then contains a jump into the tool locator which finds the code associated with the desired Toolbox routine and passes control to it. Upon returning from the Toolbox routine, all parameters have been removed from the stack leaving the function result value (if any) on the top of the stack. In addition, the 65816 processor's *carry flag* is set if an error occured during the execution of the Toolbox routine, and if so the 65816 accumulator register contains an error code.

By using TML Pascal's *tool directive* with a procedure or function declaration, the preceeding conventions are obeyed. In addition, TML Pascal will generate a *store accumulator* instruction to the Pascal global variable *ToolErrorNum* (see Chapter 10 of the Reference Manual) so that potential error codes returned by a Toolbox routine can be examined.

Thus, an invocation of MoveTo(16,20) would generate the following 65816 instructions.

```
pea  $0010
pea  $0014
ldx  $3A04                ; 58 * 256 + 4
jsl  $E10000
sta  ToolErrorNum
```

In order to allow programs written in TML Pascal to perform error checking on calls to Toolbox routines, TML Pascal has defined the special function *IsToolError* (see Chapter 10 of the Reference Manual) which examines the current state of the processor's carry flag. The IsToolError function should only be used IMMEDIATELY after a call to a Toolbox routine to ensure that the state of the processor's carry flag has not been corrupted by any intervening operations.

Thus, a program written in TML Pascal might use the following code to detect from an error which occurs in the Toolbox routine MoveTo.

```
MoveTo(16,20);
if IsToolError then
   Temp := ToolErrorNum;
   Writeln('Error occurred in MoveTo, #',Temp);
   end;
```

Note that the value of ToolErrorNum was first saved to the temporary variable Temp before the call to WriteIn. This is because WriteIn itself makes tool calls that would destroy the value of ToolErrorNum associated with the error condition returned by MoveTo.

There are at least three cases where the compiler's generation of the `STA ToolErrorNum` instruction is not required. These are the following:

(1) Many Toolbox routines do not return errors (this is the case in the above example).

(2) An application has otherwise guaranteed that all possible error conditions do not exist.

(3) An application is not effected if an error occurs, proceed regardless (usually poor programming style, but sometimes appropriate).

If these reasons occur often enough in an application, then the generation of the `STA ToolErrorNum` instruction can potentially increase the size of an application unnecessarily. To avoid this possibility, TML Pascal provides the *$ToolErrorChk* directive to turn off and on the generation of this instruction (see Appendix B of the Reference Manual).

For example, the following call to the Toolbox routine MoveTo would not generate the `STA ToolErrorNum` instruction.

```
{$ToolErrorChk-}
MoveTo(16,20);
```

While use of the *$ToolErrorChk* directive can save a considerable amount of code, the programmer must be very careful of its use in order to avoid erroneously checking the value of ToolErrorNum when the directive is turned off, and therefore ToolErrorNum has not been assigned an error code.

Other TML Pascal Units

In addition to providing interfaces to the Apple IIGS Toolbox, TML Pascal provides an additional three units which provide interfaces to the ProDOS 16 operating system of the Apple IIGS, interfaces to the shell environment calls of the Apple Programmer's Workshop, and a collection of utilities for Plain Vanilla programs and a set of string to number conversion routines.

Each of these three units is discussed in the following sections.

The ProDOS16 Unit

ProDOS 16 is the operating system of the Apple IIGS. As such it provides for the manipulation of *volumes, files, prefixes* and other system services. The Pascal unit ProDOS16 provides the interfaces to each of the routines available in ProDOS 16.

If you examine this unit, you will discover that the procedures and functions delcared there are not declared using the TML Pascal *Tool directive.* This is becuase the ProDOS 16 routines are not implemented as a tool set in the Apple IIGS Toolbox. Instead, the ProDOS 16 procedures and functions are accessed in a very special way and therefore are declared with the *External directive.* That is, each of these routines is implemented with *glue code* which changes the call into the proper form from the

standard Pascal calling convention. These glue code routines are provided in the TMLPASCALLIB runtime library.

For more information regarding ProDOS 16 and its routines consult the *Apple IIGS ProDOS 16 Reference.*

The APW Unit

The APW unit defines the interface to the Apple IIGS Programmer's Workshop shell calls. APW is a shell environment in which other programs may execute (such as the TML Pascal compiler), and therefore these programs must interact with the shell. For example, a language tool which operates from within APW must be able to communicate with APW in order to determine which file to compile. APW defines a set of routines for providing this interaction for which the Pascal interfaces are given in the APW unit.

As with the ProDOS16 unit, each of the routines are declared using the compiler's *External directive* where the corresponding code is in the TMLPASCALLIB runtime library.

For more information regarding APW and its routines consult the *Apple IIGS Programmer's Workshop Reference.*

The ConsoleIO Unit

The ConsoleIO unit is a special unit provided with TML Pascal for the purpose of providing a collection of standard utilities for all TML Pascal programs. The ConsoleIO unit provides three classes of utilities. These are

(1) The traditional UCSD Pascal screen control operations,

(2) Setting a dithered color, and

(3) Pascal string to number and number to Pascal string conversion routines.

The following are the screen control operations provided by ConsoleIO. While these routines are typically used by Plain Vanilla programs, they are implemented so that they work correctly in any window of an Apple IIGS program using Super HiRes 640 mode.

```
FUNCTION  KeyPressed : boolean;
FUNCTION  ReadChar : char;
PROCEDURE GotoXY(x, y : integer);
PROCEDURE EraseScreen;
PROCEDURE ClearEOL;
PROCEDURE InsertLine;
PROCEDURE DeleteLine;
```

The following routine is provided in the ConsoleIO unit in order to change the *color* of the QuickDraw pen while in Super HiRes 640 mode. Actually, the routine sets the *pattern* of the pen, but the patterns used cause an effect called dithering which appears to have changed the color of the pen. See the CONSDEMO example on the TML Pascal distribution disk to see the effects of this routine. The reason for providing this routine is to make up to 16 colors available at one time in 640 mode which normally displays only 4 colors. The value of the Color parameter must be between 0 and 15.

```
PROCEDURE SetDithColor(Color: Integer);
```

Finally, the following six routines are provided as utilities for convertig between Pascal string and binary representations of real and integer numbers.

```
FUNCTION  IntToString(i: integer): string;
FUNCTION  LongIntToString(l: LongInt): string;
FUNCTION  RealToString(r: real): string;

FUNCTION  StringToInt(str: string): integer;
FUNCTION  StringToLongInt(str: string): LongInt;
FUNCTION  StringToReal(str: string): real;
```

Chapter 6

Writing Apple IIGS Applications

In this chapter, we move on to show you how to develop event-driven Apple IIGS applications which take advantage of the Apple IIGS Toolbox and present the user of your application with the *desktop metaphor.* Although this chapter covers all the basic principles involved in developing Apple IIGS applications, it is by no means an attempt to discuss how each of the Toolbox routines is used. Apple Computer's documentation of the Toolbox, the *Apple IIGS Toolbox Reference,* spans two large volumes, so you can see we don't have room to discuss every Toolbox routine here.

Throughout this chapter, the example program GSDEMO.PAS from the TML Pascal distribution disk will be referenced. While reading this chapter, you might find it useful to have a listing of the program, and have used the program to see what features it supports.

Event-Driven Programming

The concept of event-driven programming is a somewhat different approach to programming than you may be acustomed. Conventional programming strategies usually has an application consist of a linear sequence of actions which are executed one after the other. Action 1 is executed first, followed by action 2 and then action 3, etc. Event-driven programming is the opposite of this approach. Instead, all actions are possible at a given time, and the next action depends upon some event, usually in response to a user's interaction with the program.

Given this strategy, the basic structure of every Apple IIGS application is nearly identical. The main program usually consists of the following statements.

```
BEGIN
   StartUpGSTools;
   { routines to initialize menus, windows, etc. }
   MainEventLoop;
   ShutDownGSTools;
END.
```

The `StartUpGSTools` and `ShutDownGSTools` procedures are responsible for loading and initializing the tool sets from the Apple IIGS Toolbox which are to be used in the application, and then shutting them down before program termination (see *Using the Apple IIGS Toolbox* below). The MainEventLoop procedure is responsible for detecting and then responding to events. The typical structure of this procedure is as follows:

```
procedure MainEventLoop;
var     Event: EventRecord;
        code:  integer;
begin
   Event.TaskMask := $1FFF;          { allow TaskMaster to do everything }
   Done := false;
   repeat
      code := TaskMaster(-1, Event);
      case code of
         { Event Manager Events }
         NullEvent:   ;
         mouseDown:   ;
         mouseUp:     ;
         keyDown:     ;
         autoKey:     ;
         updateEvt:   ;
         activateEvt: ;
         switchEvt:   ;
         deskaccEvt:  ;

         { Task Master Events }
         wInDesk:      ;
         wInMenuBar:   ProcessMenu(Event.TaskData);
         wInContent:   ;
         wInDrag:      ;
         wInGrow:      ;
         wInGoAway:    ;
         wInZoom:      ;
         wInInfo:      ;
         wInFrame:     ;
      end;
   until Done;
end; { of MainEventLoop }
```

Each of the alternatives in the `MainEventLoop` case statement represents an event that *TaskMaster* can detect. Depending on the features of your application, some of these events may or may not occur. For a complete discussion of these events and how to respond to them you should read the Event Manager and Window Manager chapters of the *Apple IIGS Toolbox Reference.* The TML Pascal Source Code Library is also an excellent source of Apple IIGS example applications showing the use of the Toolbox.

While this chapter does not show how to respond to each of the possible events, we will discuss how an application should respond to a `wInMenuBar` event. The `wInMenuBar` is retuned by *TaskMaster* whenever the mouse has been pressed down over the menu bar and then released over an item of a pull-down menu.

As you can see from the `ProcessMenu` procedure below the long integer `codeWord` contains the *Menu Number* in the high order word and the *Item Number* in the low order word. The procedure uses the item number to determine which menu item was selected and then performs the appropriate action. The menu number is used at the end of the procedure to unhilight the menu in the menu bar.

```
procedure ProcessMenu(codeWord : Longint);
var     menuNum:    Integer;
        itemNum:    Integer;
begin
   menuNum := HiWord(codeWord);
   itemNum := LoWord(codeWord);

   if itemNum = AboutItem then begin
      { code to display About Box }
      end

   else if itemNum = QuitItem then
      Done := true;

  HiliteMenu(false,menuNum);
end; { of ProcessMenu }
```

Using the Apple IIGS Toolbox

As discussed above, every Apple IIGS application must have the procedures `StartUpGSTools` and `ShutDownGSTools` for the purpose of initializing the Apple IIGS tool sets used by the application and then shutting them down. Every tool set used by the application must be loaded and started and then shut down, tool sets which are not used by the application do not need to be loaded or started.

The order of loading and starting as well as shutting down the tool sets used in an application is well defined and must be followed. The following `StartUpGSTools` procedure from the GSDEMO.PAS application shows the required order of initialization. If additional tool sets besides those shown are used, then they should be specified in the `ToolRec` and started up after the Desk Manager routine `DeskStartUp`.

```
procedure StartUpGSTools;
{ This routine initializes each of the //GS Tools required for every
  application.
}
var     ToolRec:  ToolTable;
begin
   TLStartUp;                                   { init Tool Locator }
   MyMemoryID := MMStartUp;                     { init Memory Manager }
   MTStartUp;                                   { init Misc Tools }

   { Allocate 7 pages of memory in bank 0 for use by GS Tools.
     3 pages for QuickDraw,
     1 page for Event Manager,
     1 page for Menu Manager,
     1 page for Control Manager,
     1 page for Line Edit }
```

```
ToolsZeroPage :=
    NewHandle(6 * 256,                      { allocate 7 pages }
              MyMemoryID,                   { User ID for memory blocks }
              fixedBank+fixedblk+locked,    { Attributes }
              ptr(0));                      { start in bank 0 }

QDStartUp
    (LoWord(ToolsZeroPage^),                { low address of the first
                                              3 zero pages }
     ScreenMode,                            { 640 mode }
     160,                                   { max size of scan line }
     MyMemoryID);                           { User ID for memory blocks }


EMStartUp
   (LoWord(ToolsZeroPage^) + $300,      { low address of 4th zero page }
    20,                                 { event queue size }
    0,                                  { X min clamp }
    MaxX,                               { X max clamp }
    0,                                  { Y min clamp }
    200,                                { Y max clamp }
    MyMemoryID);                        { User ID for memory blocks }

{ Now load RAM based tools (and RAM patches to ROM tools!) }
ToolRec.NumTools := 9;
ToolRec.Tools[1].TSNum := 4;            { QuickDraw // }
ToolRec.Tools[1].MinVersion := 1;
ToolRec.Tools[2].TSNum := 5;            { Desk Manager }
ToolRec.Tools[2].MinVersion := 1;
ToolRec.Tools[3].TSNum := 6;            { Event Manager }
ToolRec.Tools[3].MinVersion := 1;
ToolRec.Tools[4].TSNum := 14;           { Window Manager }
ToolRec.Tools[4].MinVersion := 0;
ToolRec.Tools[5].TSNum := 15;           { Menu Manager }
ToolRec.Tools[5].MinVersion := 1;
ToolRec.Tools[6].TSNum := 16;           { Control Manager }
ToolRec.Tools[6].MinVersion := 1;
ToolRec.Tools[7].TSNum := 21;           { Dialog Manager }
ToolRec.Tools[7].MinVersion := 0;
ToolRec.Tools[8].TSNum := 20;           { Line Edit }
ToolRec.Tools[8].MinVersion := 0;
ToolRec.Tools[9].TSNum := 22;           { Scrap Manager }
ToolRec.Tools[9].MinVersion := 0;
LoadTools(ToolRec);                     { Load the tools I need }

if isToolError then begin
   { add code to ask for Boot disk to be inserted... }
   end;

WindStartUp(MyMemoryID);                { init Window Manager }
Refresh(nil);
```

```
    CtlStartUp                                  { init Control Manager }
       (MyMemoryID,                             { User ID for memory blocks }
        LoWord(ToolsZeroPage^) + $400);         { low address of 5th zero page }

    MenuStartUp                                 { init Menu Manager }
       (MyMemoryID,                             { UserID for memory blocks }
        LoWord(ToolsZeroPage^) + $500);         { low address of 6th zero page }

    ScrapStartUp;
        { init Scrap Manager }

    LEStartUp                                   { init Line Edit }
       (LoWord(ToolsZeroPage^) + $600,          { low address of 6th zero page }
        MyMemoryID);                            { User ID for memory blocks }

    DialogStartUp                               { init Dialog Manager }
       (MyMemoryID);                            { User ID for memory blocks }

    DeskStartUp;                                { init Desk Manager }

end; { of StartUpGSTools}
```

Just as there is a required order for loading and initializing the tool sets used in an application, there is a required order to shut the tools down as well. The `ShutDownGSTools` procedure from the GSDEMO.PAS application shows the required order of initialization. If additional tool sets besides those shown were started, then they should be shut down in the reverse order of start up before the Desk Manager is shut down.

```
procedure ShutDownGSTools;
{ Shut down each of the //GS Tools initialized previously before exitting
  this application.
}
begin
   GrafOff;
   DeskShutDown;
   DialogShutDown;
   LEShutDown;
   ScrapShutDown;
   MenuShutDown;
   WindShutDown;
   CtlShutDown;
   EMShutDown;
   QDShutDown;
   MTShutDown;
   MMShutDown(MyMemoryID);
   TLShutDown;
end; { of ShutDownGSTools}
```

Supporting Desk Accessories

Apple IIGS applications should also support *New Desk Accessories* (see Chapter 7) by way of the *Apple Menu*. New desk accessories are "mini-applications" which run from within a window of an Apple IIGS application. As you might expect, the Toolbox contains a tool set which does most all the work necessary to support desk accessories in an application – the Desk Manager. The following paragraphs outline the responsibilities of an Apple IIGS application supporting desk accessories.

In order for an application to support desk accessories, it must first ensure that the following Apple IIGS tool sets have been loaded and started.

QuickDraw II
Event Manager
Window Manager
Menu Manager
Control Manager
Scrap Manager
Line Edit
Dialog Manager

The reason for these particular tool sets is that desk accessories, by convention, are permitted to assume that all of these tool sets are initialized and available for use. If a desk accessory requires any other tool set, it must load and startup the tool set itself. The Desk Manager tool set must also be started so that its calls are available in the application.

In order to add the set of currently installed desk accessories to the apple menu of an application, the application's *SetUpMenus* procedure should call the *FixAppleMenu* Menu Manager routine before the *FixMenuBar* routine. Consider the following code fragment from `GSDEMO.PAS`.

```
AppleMenuStr := '>>@\N300\0==About GSDemo...\N301\0==-\N302D\0..';
InsertMenu(NewMenu(@AppleMenuStr[1]),0);
FixAppleMenu(300);
```

The technique for opening, closing, and running of desk accessories during the execution of your application depends upon whether or not your application is using *TaskMaster* (a Window Manager routine). If your application is using *TaskMaster,* then your application needs no additional code to support the full functionality of desk accessories. However, if your application is not using *TaskMaster*, but instead is using *GetNextEvent* and processing all events on its own, then your application must perform the following operations.

(1) Call *OpenNDA* when the user selects a desk accessory in the apple menu.

(2) Call *SystemTask* frequently (at least every time through the event loop).

(3) Call *SystemClick* when a mouse down event occurs in a system window.

(4) Call *SystemEdit* when a desk accessory is active and the user selects Undo, Cut, Copy, Paste, or Clear from the application's Edit Menu.

(5) Call *CloseNDA* or *CloseNDAbyWinPtr* when the user selects Close from the application's File Menu.

Definition Procedures (DefProcs)

Often times, the Apple IIGS Toolbox routines must call a procedure which is actually part of your application. These types of procedures (and sometimes functions) are given the name *Definition Procedures* or *DefProcs* for short. The reason for the name *definition procedure* is that these routines are generally used to allow the application to provide a custom definition of some generic operation. For example, there are *Menu Definition Procedures* which allows an application to provide customized drawing procedures for drawing the representation of menus – perhaps a menu that contains a palatte of colors rather than a list of items. As you might expect, the Toolbox also allows for definition procedures of windows, controls, lists, etc.

Another component of the Toolbox where an application must use definition procedures (and the most likely), is with the *NewWindow* tool routine of the Window Manager. The NewWindow routine has a single record parameter which is the *NewWindowParamBlk*. This record defines all the information the Window Manager needs to draw and maintain the new window. Three of the fields of the record require definition procedures. The following Pascal record declaration shows the fields of the *NewWindowParamBlk* that require definition procedures.

```
NewWindowParamBlk =
   record
      ...
      wFrameDefProc: ProcPtr;
      wInfoDefProc:  ProcPtr;
      wContDefProc:  ProcPtr;
      ...
   end;
```

The *wContDefProc* routine, for example, is called by the Window Manager whenever it detects that the display of the content of the window must be updated due to a region of the window, which was previously hidden, becoming visible.

As you might expect, the procedure calling conventions for a Toolbox routine to a definition procedure are different than normal Pascal procedures. Therefore, it is necessary for the application to signal to the TML Pascal compiler that a particular procedure is in fact a definition procedure and should use the calling conventions of Toolbox routines. To accomplish this, the compiler's *$DefProc* directive is used. The directive must appear immediately before every procedure which is a definition procedure.

There is one additional consideration that must be addressed for definition procedures – the addressing of global variables. Typically, global variables are addressed using the 65816's *absolute addressing mode* versus the less efficient *absolute long addressing mode* since TML Pascal ensures that the 65816's *Data Bank Register* points to the memory bank containing the program's global variables. However, in the case of definition procedures, TML Pascal's convetion may not be obeyed by a particular Toolbox routine (ie. the Toolbox routine has changed the value of the Data Bank register). Thus, it is necessary to force TML Pascal to use *absolute long addressing mode* for addressing global variables in a definition procedure to ensure they are referenced correctly.

The following is the definition procedure *Window2Content* from GSDEMO.PAS.

```
{$DefProc }                { signal the following is a defproc }
{$LongGlobals+}            { force absolute long addressing of globals }

PROCEDURE Window2Content;
VAR i: Integer;
BEGIN
   for i := 1 to 10 do begin
      MoveTo(i*11+20,i*9+10);
      DrawString('TML Pascal is Great!');
      end;
END;

{$LongGlobals-}            { retore to absolute addressing of globals }
```

Large Programs and Segmentation

The Apple IIGS limits the size of a program's code and data segments to 64K bytes. Code segments contain the application's code, while data segments contain the storage required for the application's global variables. The reason for this size restriction is that a segment must not cross the boundries of a *bank* of memory. On the Apple IIGS, a bank of memory is 64K bytes. Thus, in order to develop applications which have more than 64K bytes of code or 64K bytes of data, the program must be *segmented*. Normally, TML Pascal creates one code segment and one data segment for an application. To obtain more than one segment, the compiler's *$CSeg* and *$DSeg* directives must be used.

Code Segmentation

Code segments are named so that the Linker can organize the different pieces of code together based on their code segment names. The default code segment name is *main*. In order to change the name of the current code segment, the TML Pascal *{$CSeg segname }* compiler directive is used. When a *{$CSeg segname }* directive appears in a program or unit, the code for all subsequent procedures and functions is placed in the new code segment. To restore code segmentation back to the default segment, merely place the *{$CSeg main }* directive in your program.

For more information regarding the use of the *{$CSeg segname }* directive see Appendicies B and C of the Reference Manual.

Data Segmentation

Data segments are named just as code segments are so that the Linker can organize the different pieces of data together based on their data segment names. The default code segment name is *~global*. In order to change the name of the current data segment, the TML Pascal *{$DSeg segname }* compiler directive is used. When a *{$DSeg segname }* directive appears in a program or unit, the data for all subsequent global variable declarations is placed in the new data segment. To restore data segmentation back to the default segment, merely place the *{$DSeg ~global }* directive in your program.

Unless a program absolutely requires a large amount of global storage, the *{$DSeg segname }* should not be used. The reason for this is that all global storage allocated outside of the *~global* data segment is

addressed using less efficient addressing modes than data allocated in the *~global* data segment.

For more information regarding the use of the *{$DSeg segname }* directive see Appendicies B and C of the Reference Manual.

Chapter 7

Writing Desk Accessories

Introduction

Desk Accessories are "mini-applications" which can be run from within Apple IIGS applications. There are actually two types of desk accessories – *Classic Desk Accessories* and *New Desk Accessories.*

Classic desk accessories (CDA's) are desk accessoires designed to execute in a non-desktop, non-event based environment. Unlike new desk accessories, a CDA gets full control of the machine during what is basically an interrupt state. Classic desk accessories are invoked by pressing the APPLE, CONTROL, and ESCAPE keys simultaneously. TML Pascal provides no direct means of writing classic desk accessories, although an enthusiastic developer could easily do so.

New desk accessories (NDA's) are desk accessories designed to execute in a desktop, event-driven envrionment. NDA's run in a window and get control when that window is the frontmost window on the desktop. New desk accessories are made available by applications which support them via the *Apple Menu.* TML Pascal provides direct support for implementing new desk accessories in Pascal, and is the topic of this chapter.

Getting Started

Since NDA's operate in the desktop environment of Apple IIGS applications, you may assume that the following Apple IIGS tools have been loaded and initialized:

- QuickDraw
- Event Manager
- Window Manager
- Menu Manager
- Control Manager
- Scrap Manager
- LineEdit
- Dialog Manager

Other tools may also be available, but you can not assume that they have been loaded and initialized. If a new desk accessory requires other tools, it must load and initialized them itself.

The Source File

The source code for a new desk accessory is quite different than a normal program. In particular, a NDA does not have a main program, but rather contains four special procedures – DAOpen, DAClose,

DAAction, and DAInit.

In addition to the four special procedures that every NDA must have, three additional pieces of information must also be provided – the service period, the event mask, and its menu name. This information is specified in TML Pascal with the *$DeskAcc* compiler directive.

```
{$DeskAcc period eventMask menuName  }
```

The service period defines how often the NDA should be "called" with the DARun action code (see below) in order to service the NDA's functionality. A period of 1 is 1/60th of a second, a period of 2 is 1/30th of a second, etc. A period of $FFFF is never. If a NDA displays the current time, then it would specify a service period of 60 so that it could update its display every second.

The event mask defines which events should be handled by the desk accessory. These values are a subset of those used by Apple IIGS applications using GetNextEvent or TaskMaster, and are listed below for reference from the GSIntf unit. Of the six listed below, the update and activate events are always passed to the desk accessory regardless of the event mask, however, the remaining four event types must be specified explicitly. If all events should be handled by the desk accessory then an event mask of -1 (or $FFFF) should be specified.

```
CONST   MDownMask   = 2;
        MUpMask     = 4;
        KeyDownMask = 8;
        AutoKeyMask = 32;
        UpdateMask  = 64;
        ActivMask   = 256;

        EveryEvent  = -1;       { $FFFF }
```

Finally, the menu name is the name for the desk accessory which should appear in the apple menu of an application supporting desk accessories.

As mentioned above, this information is specified with the compiler's *$DeskAcc* compiler directive. This directive must appear as the first line of the program before the keyword `PROGRAM`. For example, the following direcitve specifies a service period of 1 second, that all events should be handled by the desk accessory and the menu name for the desk accessory is "Clock".

```
{$DeskAcc 60 -1 Clock }
```

Desk accessories also function differently than normal applications with respect to addressing global variables. TML Pascal allocates the storage necessary for global variables in a *data segment.* A data segment is loaded to memory just as the code for a desk accessory is loaded to memory. However, the 65816 *Data Bank Register* is not set to point to the bank of memory which contains the desk accessory's data segment for its global variables. Since the compiler can not be sure where the storage for a desk accessory's globals variables will be allocated, it must always use the 65816's *absolute long addressing mode* when referencing global variables. Since this is not the normal case for TML Pascal programs, the compiler must be instructed to do so with its *$LongGlobals+* direcitve.

Thus, we arrive at the basic structure for a new desk accessory written in TML Pascal.

```
{$DeskAcc 60 -1 Clock }
```

```
{$LongGlobals+}
PROGRAM MyClockNDA;

FUNCTION DAOpen: WindowPtr;
BEGIN
   { Code for DAOpen }
END;

PROCEDURE DAClose;
BEGIN
   { Code for DAClose }
END;

PROCEDURE DAAction(Code: Integer; Param: LongInt);
BEGIN
   { Code for DAAction }
END;

PROCEDURE DAInit(Code: Integer);
BEGIN
   { Code for DAInit }
END;

BEGIN
   { No main program allowed }
END.
```

The following sections defines each of the four required desk accessory routines and outlines each of their responsibilites. The TML Pascal distribution disk contains the source code to a simple Clock desk accessory.

The DAOpen Function

This function is called as the result of an application calling the Desk Manager routine *OpenNDA*. This routine should check if the desk accessory has already been opened, and if it has then return without performing any action. Otherwise, the function should create the window for the desk accessory, making it a system window and return the window pointer to the created windows as the function's result.

The following is a source code fragement showing the basic structure of the DAOpen function.

```
FUNCTION DAOpen: WindowPtr;
{  The variables myWindOpen, myWindPtr, and myWind are globals }
BEGIN
   if not myWindOpen then begin
      myWindOpen := true;
      myWindPtr := NewWindow(myWind);
      SetSysWindow(myWindPtr);
      DAOpen := myWindPtr;
      end;
END;
```

The DAClose Procedure

The DAClose procedure should shut down the desk accessory when it is open. It should also work, without creating an error situation, if it is called when the desk accessory is not actually open.

```
PROCEDURE DAClose;
{  The variables myWindOpen and myWindPtr are globals }
BEGIN
   if myWindOpen then begin
      CloseWindow(myWindPtr);
      myWindOpen := false;
      end;
END;
```

The DAAction Procedure

The DAAction procedure is the routine which does all the work associated with the desk accessory between the time that it has been opened until it is closed. The DAAction procedure has two parameters – a *Code* which indicates what type of action to perform and a *Param* whose meaning depends upon the *Code* parameter. There are nine potential values for the *Code* parameter, each of which must be implemented by the DAAction procedure. These operations are summarized in the following table together with the meaning of the *Param* parameter in each case.

Action	Description
DAEvent	An event relevant to the desk accessory has occured. *Param* points to the event record describing the event.
DARun	The time period specified as the service period has expired. *Param* has no meaning.
DACursor	This code is passed to a desk accessory if it is the frontmost window each time SystemTask is called. The purpose is to allow the desk accessory to change the cursor when it is over the NDA's window. *Param* has no meaning.
DAMenu	This is passed to a desk accessory if an item from a system menu is selected. *LoWord(Param)* is the Menu ID and *HiWord(Param)* is the Item ID.
DAUndo DACut DACopy DAPaste DAClear	Each of the following 5 codes are passed to a desk accessory if the application determines that the user has selected one of these edit commands from theEdit menu. The DAAction procedure should assign the value of 1 in the *Code* parameter if the action was handled, otherwise a value of 0 should be assigned.

The following source code fragement shows the basic structure of a DAAction procedure.

```
PROCEDURE DAAction(Code: Integer; Param: Longint);
{  The variable myWindPtr is globals }
VAR currPort: GrafPtr;
BEGIN
   case Code of
```

```
        DAEvent: begin
           if EventRecordPtr(param)^.what = updateEvt then begin
              BeginUpdate(myWindPtr);
              { code to update window }
              EndUpdate(myWindPtr);
              end
           end;
        DARun: begin
           currPort := GetPort;
           SetPort(GrafPtr(myWindPtr));
           { code to "run" the DA }
           SetPort(currPort);
           end;
        DACursor: begin
           { code to update the cursor }
           end;
        DAMenu: begin
           { code to respond to a menu selection }
           end;
        DAUndo: begin
           { code to perform an Undo for the DA }
           Code := 1;
           end;
        DACut: begin
           { code to perform a Cut for the DA }
           Code := 1;
           end;
        DACopy: begin
           { code to perform a Copy for the DA }
           Code := 1;
           end;
        DAPaste: begin
           { code to perform a Paste for the DA }
           Code := 1;
           end;
        DAClear: begin
           { code to perform a Clear for the DA }
           Code := 1;
           end;
END;
```

The DAInit Procedure

The DAInit procedure is called when the Desk Manager routines DeskStartUp and DeskShutDown are called by an application to initialize and shut down all NDAs. The value of the *Code* paramter indicates under which circumstance the routine is being called. If *Code = 0* then DAInit is being called due to a DeskShutDown call, otherwise the call is due to a DeskStartUp call. In either case, this routine should contain the necessary initialization and shutdown code for the desk accessory.

```
PROCEDURE DAInit(Code: Integer);
{  The variable myWindOpen is global }
```

```
BEGIN
   if Code = 0 then begin
      { A DeskShutDown Call, check that the DA window is closed }
      end
   else begin
      { a DeskStartUp Call, init the myWindOpen flag }
      myWindOpen := false
      end
END;
```

Installing a Desk Accessory

Now that you have successfully created a desk accessory it must be properly installed in the SYSTEM/DESK.ACCS/ directory of the boot disk so that desktop based applications supporting desk accessories may access it. The installation of a new desk accessory is basically a three step process as outlined below.

(1) New desk accessories are programs (ProDOS16 load files) defined to have the file type of $B8. Thus, the file type of the newly created desk accessory must be changed to $B8 using the APW command FILETYPE. For example,

```
FILETYPE MYCLOCKNDA $B8
```

(2) The Apple IIGS Desk Manager requires that all desk accessories be placed in the special system directory SYSTEM/DESK.ACCS/. Thus, it is necessary to copy the desk accessory load file into this directory using the APW COPY command.

```
COPY MYCLOCKNDA /APWWORK/SYSTEM/DESK.ACCS/
```

(3) Finally, the Apple IIGS must be rebooted. During the boot process of the Apple IIGS, the special directory SYSTEM/DESK.ACCS/ is searched for the currently installed desk accessories. Since this process is only done at boot time it is necessary to reboot the machine in order for it to recognize the new desk accessory.

Part II

TML Pascal Reference Manual

Table of Contents

Preface

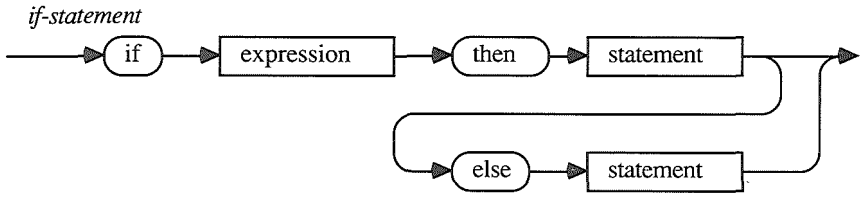
About this Manual

This Reference Manual is a complete description of the implementation of the Pascal language provided by the TML Pascal compiler. The manual is not a tutorial on programming Pascal, but rather it is intended for programmers who have some knowledge of the Pascal language.

The implementation very nearly follows the ANS specification of the Pascal language with some omissions and several extensions to conform with other popular variations of Pascal. Most notable these include TML Pascal v2.0 for the Apple Macintosh, and Apple Computer's MPW Pascal.

The manual is organized in 10 chapters and 4 appendicies. The 10 chapters dicuss in detail each of the language features implemented in TML Pascal in nearly the same organization as the ANS standard, thus it should be easy to compare the two documents. The appendicies discuss the details of the TML Pascal implementation, error messages, compiler directives, and a comparison of TML Pascal to the ANS standard.

Throughout the manual are syntax diagrams for each of the Pascal language constructs. For example, the following diagram shows the syntax of the if statement.



The diagram begins on the left, then following the arrows through the diagram shows the sequence of syntactic elements which make up the if statement. Elements in an oval or a circle denote Pascal special-symbols, and elements in rectangular boxes denote syntactic constructs which are described in another diagram.

Chapter 1

Tokens

Lexical *tokens* are the smallest units of text in a Pascal program. The tokens of Pascal are classified into *special-symbols, identifiers, directives, unsigned-numbers, labels,* and *character-strings.* Aside from character-strings the representation of any letter (upper-case, lower-case, font, etc.) is insignificant to the meaning of the program.

The text of a Pascal program consists of tokens and *separators*, where a separator is either a *blank* (the space or tab characters) or a *comment.* Two adjacent tokens must be separated by one or more separators if each token is an identifier, number, or word-symbol.

Special Symbols

Special-symbols are tokens having special meanings and are used to delimit the syntactic units of the language.

The following single characters are special-symbols:

```
+ - * / = < > [ ] . , ( ) : ; ^ @ { }
```

The follwing *character-pairs* are special-symbols:

```
<> <= >= := .. (* *)
```

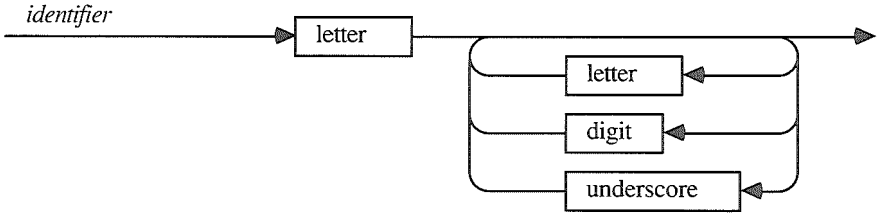
The following*word-symbols* (or *reserved-words*) are special-symbols:

and	**else**	**interface**	**program**	**until**
array	**end**	**label**	**record**	**uses**
begin	**file**	**mod**	**repeat**	**var**
body	**for**	**not**	**set**	**while**
case	**function**	**of**	**string**	**with**
const	**goto**	**or**	**then**	
div	**if**	**otherwise**	**to**	
do	**implementation**	**packed**	**type**	
downto	**in**	**procedure**	**unit**	

Identifiers

Identifiers serve to denote constants, types, variables, procedures, functions, programs, units and fields in records. An identifier can be of any length so long as it fits on a single line, however, only the first 255 characters are significant. Corresponding upper- and lower-case letters are equivalent in identifiers. No

identifier can have the same spelling as a word-symbol.



Examples of standard identifiers in TML Pascal:

Succ Exit MaxInt Writeln

Directives

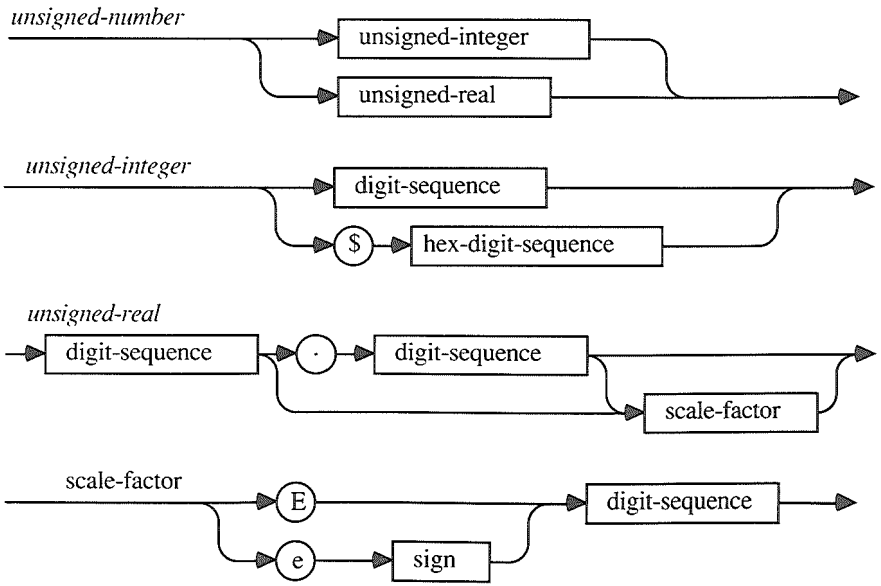
Directives are identifiers that have special meanings in the context of a procedure declaration or function declaration. They can otherwise be used as identifiers in all other contexts.

The directives available in TML Pascal are:

EXTERNAL FORWARD INLINE TOOL

Numbers

Unsigned-integers in decimal or hexidecimal (hexidecimal integers have the $ character as a prefix) notation represent constants of the data types *integer* and *longint*. Unsigned-reals in decimal notation represent constants of the data type *Extended*. The letter 'E' or 'e' preceding a scale factor means *times ten to the power of*.



Examples of Numbers:

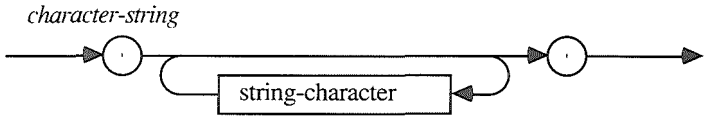
1 +100 -0.1 $A05D 5.329E4

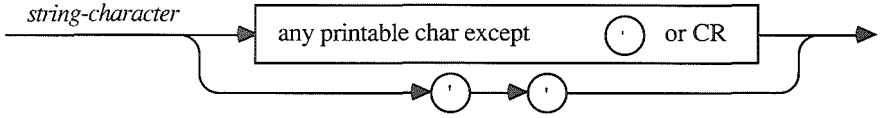
Labels

A label is a digit-sequence whose value may be any of the integers. Leading zeroes in a label are insignificant, e.g. the labels *1* and *0001* are considered equivalent. Labels are used with **goto** statements, described in Chapter 6.

Character-Strings

A character-string is a sequence of zero or more printing characters all on the same line in a program and enclosed by apostrophes. The maximum number of characters that can be in a character-string is 255. A character-string with nothing between the apostrophes denotes a *null-string* value.





A character-string represents a value of a string type. As a *string type*, a character-string is compatible not only with other string types, but also *char types* and *packed string types* .

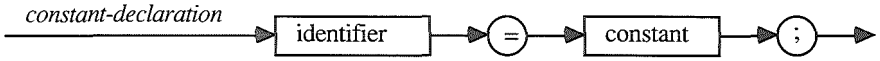
All string-type values have a *length* attribute. In the case of a character-string, the length is fixed; it is equal to the actual number of characters in the string as enclosed within apostrophes. A pair of adjacent apostrophes within a character-string is regarded as a single apostrophe and thus counts as a single character in the string's length.

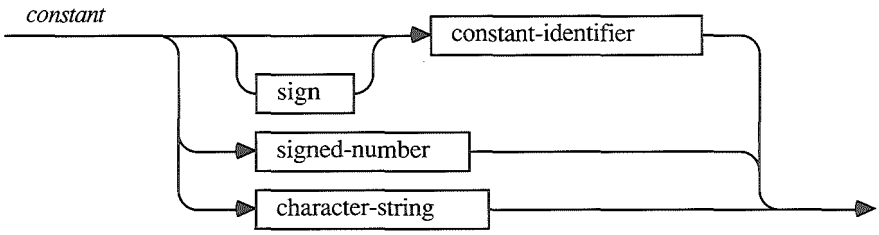
Examples of character-strings:

'A' ';' 'Pascal' 'THIS IS A STRING' 'Don''t worry!' '''' ''

Constant Declarations

A constant-declaration defines an identifier to denote a constant, within the block that contains the declaration.





A signed-number may be an integer or real number.

Comments and Compiler Directives

The constructs:

{ any text not containing right-brace }
(* any text not containing star-right-parenthesis *)

are called *comments.*

The substitution of a blank for a comment or a comment for a blank does not alter the meaning of the program. That is, a comment, as a separator, may appear anywhere in a program where a blank may appear.

Comments of the form {...} may be nested within comments of the form (* ... *), and vice versa, however, no other nesting of comments is available. The occurrence of the special symbol } within a {...} comment, or the special symbol *) within a (* ... *) comment always terminates the comment.

A compiler directive is a comment that contains a $ (dollar-sign) character immediately after the { or (* that begins a comment. The $ character is then followed by one or more letters which represent a specific compiler directive. Compiler directives serve to affect the behavior of the compiler. Each of the compiler directives and their affects are described in Appendix B.

Examples of compiler directives:

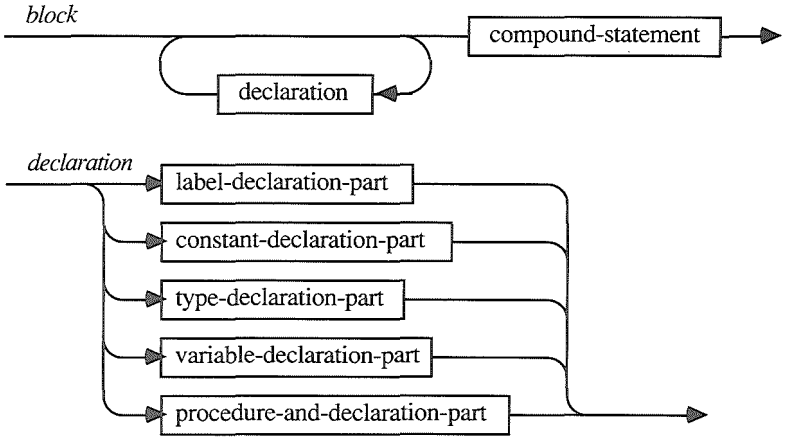
*{$DefProc} {$LongGlobalst} (*StackSize 10240*)*

Chapter 2

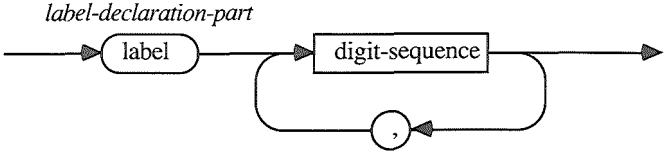
Blocks, Scope, and Activations

Definition of a Block

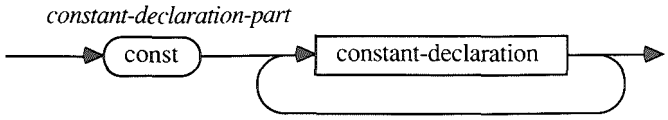
The block is the fundamental unit of Pascal source code. A *block* consists of a *declaration part* and a *statement part.* The *declaration part* consists of zero or more declarations which may appear in any order. The *statement part* is a compound statement and follows the declarations. Every block is part of a procedure declaration, a function declaration, a program, or a unit. All identifiers and labels that are declared in the declaration part of a block are *local* to that block. The program block contains all other blocks; therefore, declarations in the program block are termed *global.*



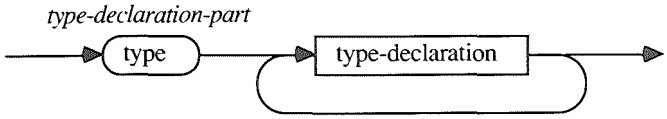
The *label declaration part* declares labels that mark statements in the corresponding statement part. Each label must mark exactly one statement in the statement part.



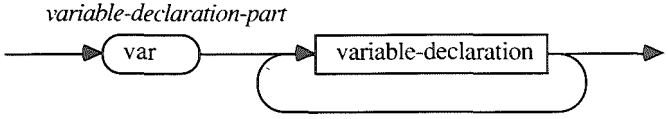
The *constant declaration part* contains *constant declarations* (see "Constant Declarations" in Chapter 1) local to the block.



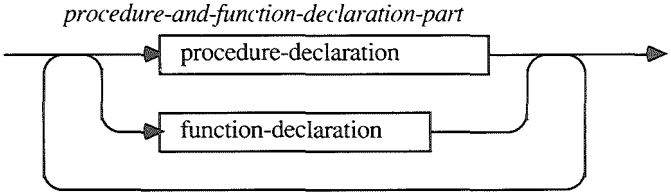
The *type declaration part* contains *type declarations* (see Chapter 3) local to the block.



The *variable declaration part* contains *variable declarations* (see Chapter 4) local to the block.



The *procedure and function declaration part* contains all *procedure* and *function* declarations local to the block (see Chapter 7).



Rules of Scope

Scope of a Declaration

The appearance of an identifier or label in a declaration defines the identifier or label, that is, the identifier or label is associated with its *meaning* at the point of declaration. All other applied occurrences of the identifier or label must be within the *scope* of this declaration. The scope of a declaration is the block that contains the declaration, and all blocks enclosed by that block except as explained in the sections below.

Redeclaration in an Enclosed Block

Suppose that *outer* is a block, and that *inner* is another block declared within *outer*. If an identifier declared in block *outer* has the same spelling as an identifier declared in block *inner*, then block *inner* and all blocks enclosed by *inner* are excluded from the scope of the declaration in block *outer*.

Position of Declaration within Its Block

The declaration of an identifier or label must precede all applied occurrences of that identifier or label in the program text. That is, identifiers and labels cannot be used until they are declared.

There is one exception to this rule: in a type declaration, the domain type of a pointer type can be an identifier that has not yet been declared. In this case, the identifier must be declared somewhere in the same declaration part as the pointer type.

Redeclaration Within a Block

An identifier or label cannot be declared more than *once* within a block, unless it is declared within a contained block, or it if it appears in the field-list of a record declaration.

A record field identifier is declared within a record type. It is meaningful only in combination with a reference to a variable of that record type. Therefore, a field identifier can be declared within the same block as another identifier with the same spelling, as long as it has not been declared previously in the same field-list. An identifier that has been declared can be used again as a field identifier in the same block.

Identifiers of Standard Objects

TML Pascal provides a set of standard (predeclared) constants, types, procedures, and functions that behave as if they were declared in a block that contains the entire program. Their scope is the entire program or unit (See Chapters 9 and 10, where each of TML Pascal's standard identifiers are documented).

Scope of Unit Interface and Unit Specification Identifiers

Programs, Units, Unit Specifications, and Unit Bodies containing a **uses** clause are provided the identifiers belonging to each of the units in the **uses** clauses. These identifiers act if as if they were declared in the same block where the **uses** clause appears.

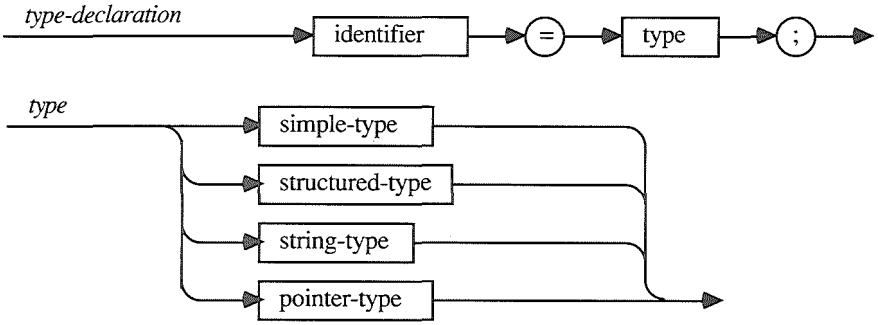
Activations

The execution of a block is referred to as an *activation* of a block. At any given time, a block can have zero or more activations. If a block is not currently being executed, then it has zero activations. If a block is being executed, then there is at least one activation. When a block has more than one activation, it is said to be *recursive*.

Chapter 3

Types

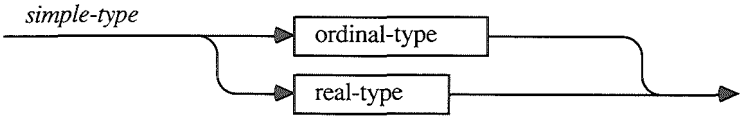
When you declare a variable, you must give its *type.* The type of a variable determines the set of values that the variable can assume and the operations that can be performed upon it. A type declaration introduces an identifier to denote a type.



When an identifier occurs on the left side of a type declaration, it is declared as a type identifier for the block in which the type declaration occurs. A type identifier's scope does not include itself, except for pointer types.

Simple Types

All the simple types define ordered sets of values.



An integer type identifier is one of the standard identifiers *Integer* or *LongInt.* A real type identifier is one of the standard identifiers *Real, Single, Double, Comp* or *Extended.* See "Numbers" in Chapter 1 for how to denote constant integer and real type values.

Ordinal Types

Ordinal types are the subset of the simple types that have the following special characteristic:

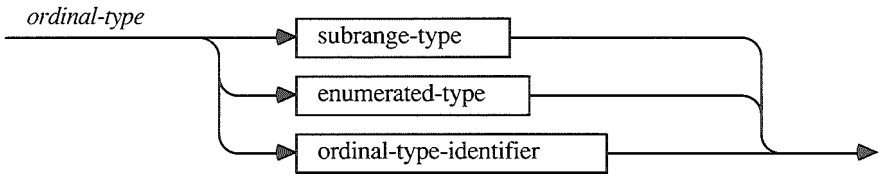
- The possible values of an ordinal type are an ordered set and every value has an *ordinality*,

  which is an integral value. Except for integer types, the first value of every ordinal type has ordinality 0, the next has ordinality 1, etc. For integer types, the ordinality of a value is the value itself. Every value of an ordinal type except the first has a *predecessor* based on the ordering of the type, and every value of an ordinal type except the last has a *successor* based on the ordering of the type.

- The standard function *ord* and *ord 4* can be applied to any value of an ordinal type, and it returns the ordinality of the value.

- The standard function *pred* can be applied to any value of an ordinal type, and it returns the ordinality of the value.

- The standard function *succ* can be applied to any value of an ordinal type, and it returns the ordinality of the value.



TML Pascal has four predefined ordinal types: *Integer, LongInt, Boolean,* and *Char.* In addition, there are two classes of user defined ordinal types: enumerated types and subrange types.

Standard Ordinal Types

Integer Integer type values are a subset of the whole numbers. An integer type variable can have a value within the range *-maxint-1..maxint,* that is, -32,768 to 32,767. The standard *Integer* constant *maxint* is defined as 32,767. The range encompasses 16-bit, two's complement integers.

LongInt LongInt type values are also a subset of the whole numbers. A longint type variable can have a value within the range *-maxlongint-1..maxlongint.* The standard *LongInt* constant *maxlongint* is defined as 2,147,483,647. The range encompasses the 32-bit, two's complement integers.

Arithmetic operations with integer type operands use *Integer* (16-bit) or *LongInt* (32-bit) precision according to the following rules:

- Integer constants in the range of type *Integer* are considered to be of type *Integer.* Other integer constants are considered to be of type *LongInt.*

- When both operands of an operator (or the single operand of a unary operator) are of type *Integer*, 16-bit precision is used, and the result is of type *Integer* (truncated to 16-bits if necessary). Similarly, if both operands are of type *LongInt*, 32-bit precision is used, and the result is of type *LongInt.*

- When one operand is of type *LongInt*, and the other is of type *Integer*, the *Integer* operand is first converted to *LongInt*, 32-bit precision is used for the operator, and the result is of type *LongInt*

- The expression on the right side of an assignment statement is evaluated independently of the left side.

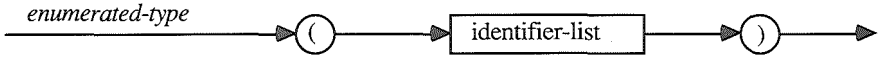
An *Integer* value may be explicitly converted to a *LongInt* by using the standard function ord4 described in Chapter 10.

Boolean Boolean type values are denoted by the predefined constant identifiers *false* and *true*, where *ord(false) = 0*, and *ord(true) = 1*. Values of type *Boolean* are required by the Pascal if statement, repeat statement, and while statement.

Char The char type has a set of values that are the ASCII characters. The function call *Ord(Ch)*, where *Ch* is a *Char* value, returns the ordinality of *Ch*. A string constant of length 1 may be used to denote a constant *Char* value Any value of type *Char* may be generated via the standard function *Chr*.

Enumerated Types

An enumerated type defines an ordered set of values by enumerating a collection of identifiers that denote these values. The ordering of these values is determined by the sequence in which the identifiers are listed. That is, for two enumeration identifiers *x* and *y*, if *x* precedes *y* then the ordinal value of *x* is less than the ordinal value of *y*.



When an identifier occurs within the identifier list of an enumerated type, it is declared as a constant for the block in which the enumerated type is declared. The type of this constant is the enumerated type in which it is declared. The ordinality of an enumerated constant is its position in the identifier list, where the ordinality of the first enumerated constant in the list is always 0.

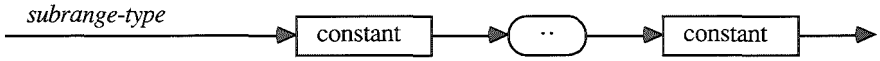
Examples of enumerated types:

```
suit  =  ( club, diamond, heart, spade )
color =  ( red, yellow, green, blue )
```

Given these declarations, *yellow* is an enumerated constant of type *color* with ordinality 1, *spade* is an enumerated constant of type *suit* with ordinality 3, and so on.

Subrange Types

A subrange type defines a subset of the values of some ordinal type called the host type. The definition of a subrange type specifies the least and the largest value in the subrange.



Both constants in a subrange type must be of the same ordinal type. Subrange types of the form *a..b* require that *a* is less than or equal to *b*.

A variable of subrange type possesses all the properties of variables of the host type, with the restriction that its value must always be one of the values in the range defined by the subrange type.

Examples of subrange types:

1..100
-128..127
spade..heart

Real Types

The real types have sets of values that are subsets of the real numbers, which can be represented in floating point notation using a fixed number of digits. In general, a floating point notation of a value *n* is comprised of a set of three values *m, b,* and *e* such that $m * b^e = n$, where *b* is always 2 and both *m* and *e* are integral values within the real type's range. These *m* and *e* values further prescribe the real types's range and precision.

There are four standard real types in TML Pascal: *Single, Double, Comp* and *Extended.* In addition, the standard identifier *Real* is defined to be equivalent to the type *Extended.* The real types differ in the range and precision of values they can represent.

Table 3-1
Real Types

Type identifier	*Memory size*	*Magnitude*
Single	4 bytes	approx 1.4E-45 to 3.4E38
Double	8 bytes	approx 5.0E-324 to 1.7E308
Real, Extended	10 bytes	approx 1.9E-4951 to 1.1E4932

The *Comp* type holds only integral values within the range $-2^{63}+1$ to $2^{63}-$, which is approximately -9.2E18 to 9.2E18.

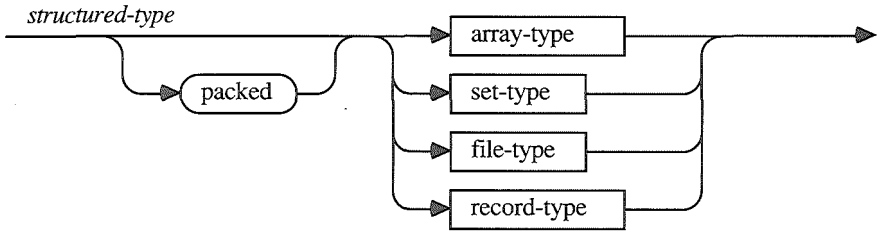
All real type values are converted to *Extended* before any operations are performed on them, and the results of such operations are always of type *Extended.* An *Extended* value may always be used were a *Single, Double,* or *Comp* value is required, provided that the value falls withing the required range.

IMPLEMENTATION NOTE

All real values are converted to the type *Extended* by the compiler before calculations are performed so that maximum accuracy can be obtained. Thus, calculations on data stored as the type *Extended* result in faster and more compact code that calculations on data stored in other representations. The smaller representations should be used when data storage space is more critical than execution speed.

Structured Types

A structured type is characterized by its structuring method and by the type(s) of its components . The type of a component may itself be structured. There is no inherent limit on the number of levels to which types can be structured.



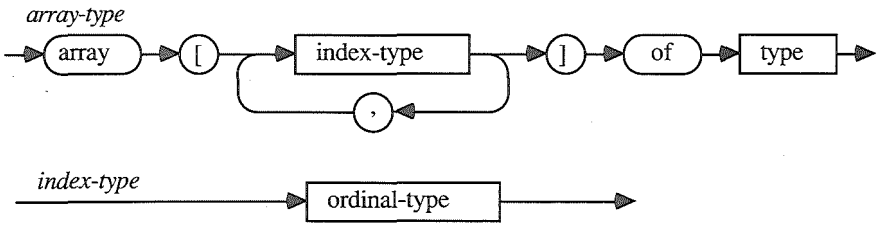
The use of the word **packed** in the declaration of a structured type indicates that storage organization of all values of that type should be compressed to economize storage, even if this causes the access of the component of a variable of this type to be less efficient. Note that you cannot use components of packed variables as actual variable parameters to procedures and functions.

IMPLEMENTATION NOTE

TML Pascal only supports packing to byte boundries. Bit level packing is not implemented. For more information regarding storage allocation and data representation see Appendix C.

Array Types

An array type defines a structured type which has a fixed number of components that are all of the same type.



The type that follows the word **of** is the component type of the array. The number of elements is determined by one or more *index types,* one for each dimension of the array. The index type must be an ordinal type. There is no inherent limit on the number of dimensions an array type can have.

IMPLEMENTATION NOTE

TML Pascal restricts the size of an array to 32,767 bytes of storage.

An array type of the form

packed array [1..n] **of** char

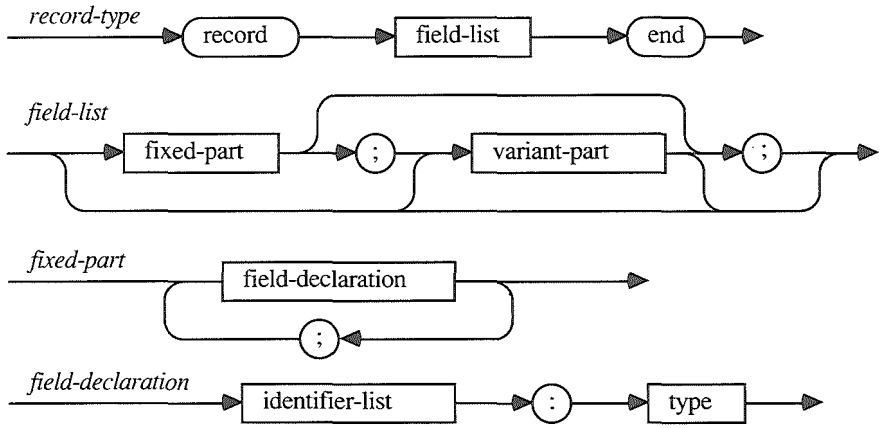
is referred to as a packed string type. A packed string type has certain properties not shared by other array types (see "Identical and Compatible Types" later in this chapter).

Examples of array types:

***array** [1..100] **of** real*
***packed array** [color]**of** boolean*

Record Types

A record type consists of a fixed collection of components called fields, each of which may be a different type. For each component, the record type specifies the type of the field and an identifier that names it.

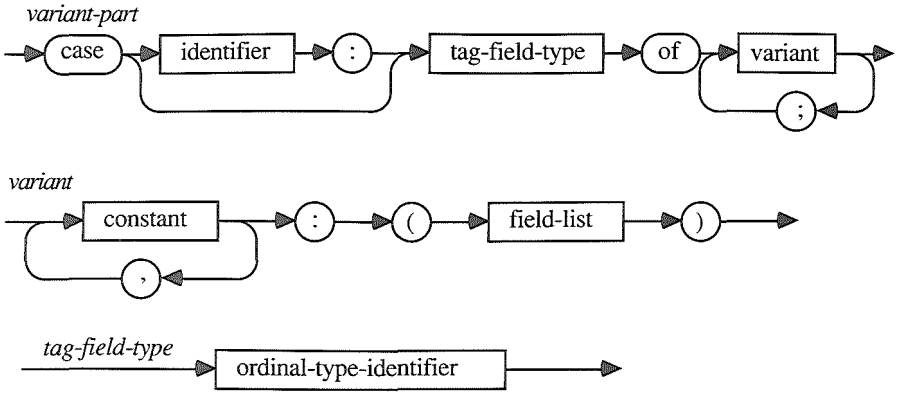


The fixed part of a record type specifies a field list that is always accessible in a variable of the record type, giving an identifier and a type for each field. Each of these fields contains data that is always accessed in the same way.

Exampleof a record type:

```
record
   year:      integer;
   month:     1..12;
   day:       1..31;
end
```

A variant part consists of several alternative field lists which are allocated in the same memory space of a record variable, thus allowing data in this space to be accessed in more than one way. Each of the lists of fields is called a *variant*. The variants "overlay" each other in memory, and all fields of all variants are accessible at all times.



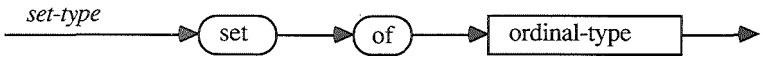
Each variant is introduced by one or more constants. All of the constants must be distinct and must be of an ordinal type that is compatible with the tag field type. The variant part allows for an optional identifier that denotes a *tag field*. If a tag field is present, it is considered a field of the previous fixed part.

Examples of record types with variants:

```
record
    name , firstName: string [80];
    age: 0..99;
    case married: boolean of
        true:   (spousesName: string [80]);
        false: ()
end

record
    x,y: Real;
    case kind: figure of
        rectangle: (height,width: real);
        triangle:   (side1,side2,angle: real);
        circle:     (radius: real);
end
```

Set Types

A set type has a range of values that is the powerset of some ordinal type, called the base type. Each possible value of a set type is a subset of the possible values of the base type.



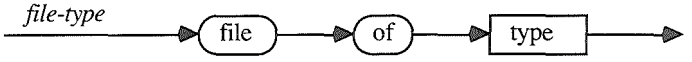
IMPLEMENTATION NOTE

TML Pascal restricts the base type to not more than 256 possible values. If the base type is a subrange of *integer*, it must be in the limits 0..255. For more information regarding storage allocation and data representation see Appendix C.

Every set type can hold the value *[ ]*, called the *empty set.*

File Types

A file type is a structured type consisting of a linear sequence of components that are all of one type, the component type. The component type may be any type that is not a file type or a structured type that contains a file type component. The number of components is not fixed by the file type declaration.



The standard file type *Text* denotes a special packed file of characters organized into lines. Files of type *Text* are supported by special I/O procedures discussed in Chapter 9.

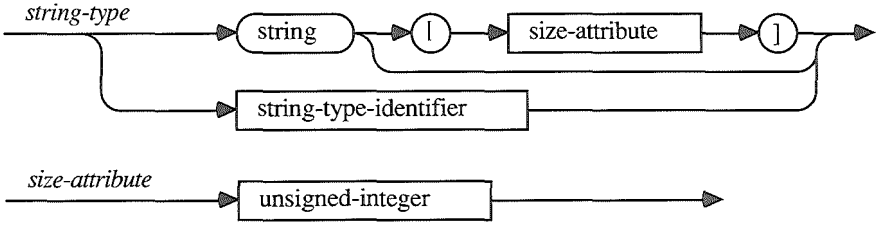
IMPLEMENTATION NOTE

Due to the representations of types in TML Pascal a *file of char* accesses file components which are 16-bit words, whereas a *packed file of char (or Text)* accesses file components which are 8-bit bytes. For more information regarding storage allocation and data representation see Appendix C.

String Types

A string type value is a sequence of characters with a dynamic length attribute and a constant size attribute from 1 to 255. The constant size is a maximum limit on the length of any value of this type. If an explicit size attribute is not given for a string type, then it is given a size of 255 by default.

The current value of the length attribute of a string type value is returned by the standard function *Length.* A *null string* is a string type value with a dynamic length of zero.

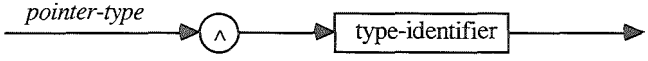


The ordering relationship between any two string values is determined by lexical comparison based on the ordering relationship between character values in corresponding positions in the two strings. When the two strings are of unequal lengths, each character in the longer string that does not correspond to a

character in the shorter one compares "higher"; thus the string value 'attribute' is greater than the value 'at'. Two strings must always have the same lengths to be equal.

Pointer Types

A pointer type defines a set of values that point to *dynamic variables* of a specified type called the *base type*. A pointer type variable contains the memory address of a dynamic variable.



If the base type is an undeclared identifier, it must be declared in the same type declaration part as the pointer type.

You can assign a value to a pointer variable with the *New* procedure, the @ operator, or the *Pointer* function. The *New* procedure allocates a new memory area in the heap for a dynamic variable and stores the address of that area of memory in the pointer value. The @ operator directs the pointer variable to the memory area containing any existing variable. The *Pointer* function points the pointer variable to a specific memory address.

The predeclared constant identifier *Nil* represents a pointer valued constant that is a possible value of every pointer type. Conceptually, *Nil* is a pointer that does not point to anything.

Identical and Compatible Types

Two types may or may not be *identical*, and identity is required in some contexts. Other times, even if not identical, two types need only be *compatible*, and other times *assignment compatibility* is required.

Type Identity

Identical types are required only in the following contexts:

- Between actual and formal variable parameters .
- Between actual and formal result types of functional parameters .

Two types, t_1 and t_2, are *identical* if one of the following is true:

- t_1 and t_2 are the same type identifier.
- t_1 is declared to be equivalent to a type identical to t_2.

Compatibility of Types

Compatibility is required in most contexts where two or more entities are used together. Specific instances where type compatibility is required are noted elsewhere in this manual.

Two types are *compatible* if any of the following are true:

- Both are identical.
- One is a subrange of the other.
- Both are subranges of identical types.
- Both types are set types with compatible base types.
- Both are string types.
- Both are of type packed string type and have the same number of components.

Assignment Compatibility

Assignment compatibility is required whenever a value is assigned to something, either explicitly (as in an assignment statement) or implicitly (as in passing value parameters).

A value of type t_2 is assignment compatible with a type t_1 if any of the following are true:

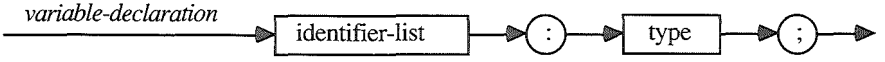
- t_1 and t_2 are identical types and neither is a file type nor a structured type that contains a file type component.
- t_1 is a real type and t_2 is an integer type.
- t_1 and t_2 are compatible ordinal types, and the value of type t_2 is within the range of possible values of t_1.
- t_1 and t_2 are compatible set types, and all the members of the value of type t_2 are within the range of possible values of the base type of t_1.
- t_1 is a string type or a char type and t_2 is a string type of a quoted character constant
- t_1 is a packed string type with *n* components and the value of type t_2 is a string type of a quoted character constant and has a length of *n*.

It is an error if assignment compatibility is required and none of the above is true.

Chapter 4
Variables

Variable Declarations

A variable declaration is used to allocate and associate a piece of storage with a particular type. A variable is an entity in which value(s) are stored. Each identifier in the indentifier list of a variable declaration denotes a distinct variable possessing the type of the variable declaration.



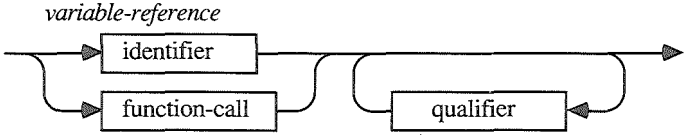
The occurrence of an identifier within the identifier list of a variable declaration declares it as a variable identifier for the block in which the declaration occurs. The variable can then be referred to throughout the block, unless the identifier is redeclared in an enclosed block. Redeclaration creates a new variable using the same identifier, without affecting the value of the original variable.

Examples of variable declarations:

```
x, y, z: real;
c: color;
p1,p2: person;
today: date;
operator: (plus, minus, times);
digit: 0..9;
coord: polar;
done, error: boolean
```

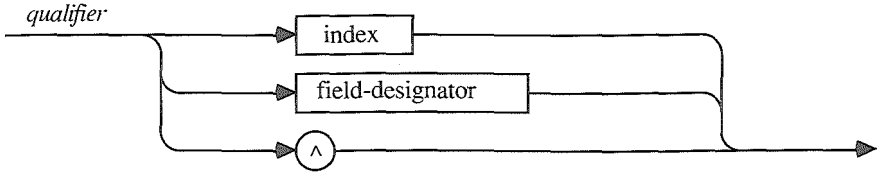
Variable References

A variable reference denotes either an entire variable, a component of a structured or string type variable, a dynamic-variable pointed to by a pointer type variable, or a variable reached through a function call.



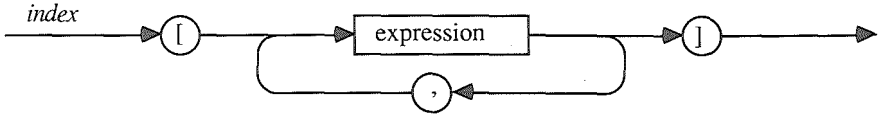
Qualifiers

A variable reference is a variable identifier followed by zero or more *qualifiers* which modify the meaning of the varialbe reference.



Arrays, Strings, and Indexes

A specific component of an array variable is denoted by a variable reference that refers to the array variable, followed by an index qualifier that specifies the component. A specific character within a string variable is denoted by a variable reference that refers to the string variable, followed by an index qualifier that specifies the character position.



Examples of indexed arrays:

```
m[i,j]
a[i+j]
```

Each expression in the index selects a component in the corresponding dimension of the array. The number of expressions must not exceed the number of index types in the array declaration. The index expression must be assignment compatible with the corresponding index type.

When indexing a multi-dimension array, multiple indexes or multiple expressions within an index can be used interchangeably. For example,

```
MyMatrix [I] [J]
```

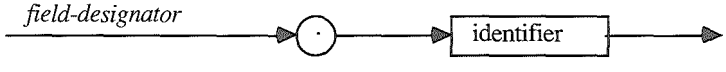
has the same meaning as

```
MyMatrix [I,J]
```

A string variable can be indexed with a single index expression, whose value must be within the range *0..n,* where *n* is the declared size of the string. Indexing a string accesses one character of the string value. The first character of a string variable (index 0) contains the dynamic length of the string.

Records and Field Designators

A specific field of a record variable is denoted by a variable reference that refers to the record variable, followed by a field designator that specifies the field.



Examples of field designators:

```
today.year
p2^.pregnant
```

In a statement within a **with** statement, a field designator does not have to be preceded by a variable reference to its containing record.

Pointers and Dynamic Variables

The value of a pointer variable is either *nil*, or a value that points to a dynamic variable.

The dynamic variable pointed to by a pointer variable is referenced by writing the pointer symbol ^ after the pointer variable.

Dynamic variables and pointer values that point to them are created by the standard procedure *New*. Additionally, the @ operator and the standard procedure *Pointer* may be used to create pointer values that are not in fact pointers to dynamic variables, but are treated as such.

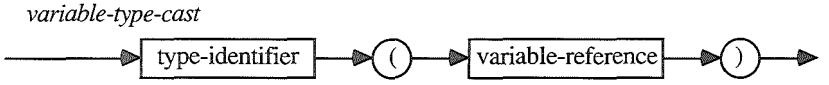
The constant *nil* does not point to any variable. It is an error if you access a dynamic variable when the pointer's value is *nil* or undefined.

Examples of references to dynamic variables:

```
p1^
p1^.sibling^
```

Variable Type Casts

A variable reference of one type can be changed into a variable reference of another type through a mechanism called a variable type cast.



When a varaible type cast is applied to a variable reference, the variable reference is treated as an instance of the type specified by the type identifier. The size of the variable (that is, the number of bytes of storage it occupies) must be the same as the size of the type denoted by the type identifier. A variable type cast may be followed by one or more qualifiers just as a variable reference.

Examples of variable type casts:

```
type   point = record
                 v,h: integer;
               end;
var    p: point;
       l: longint;
begin
  p := point(l);
  l := longint(p);
  longint(p) := longint(p) + $00020002;
end;
```

Chapter 5
Expressions

Expressions denote values. The simplest expression is merely a variable reference, however, most expressions consist of *operators* and *operands*. Most Pascal operators are *binary*, that is, they have two operands. The remaining operators are *unary* and have only one operand. When more than one operator appears in an expression, precedence rules are applied to determine which operands are associated with which operators. For example, the expression:

```
a+b*c
```

can be interpreted as either (a+b)*c or a+(b*c). The precedence rules make the interpretation unambiguous:

- When an operand appears between two operators of different precedence, it is bound to the operator with the higher precedence.
- When an operand is written between two operators of the same precedence, it is bound to the operator to the left.
- A parenthesized expression is always evaluated before it is applied as an operand.

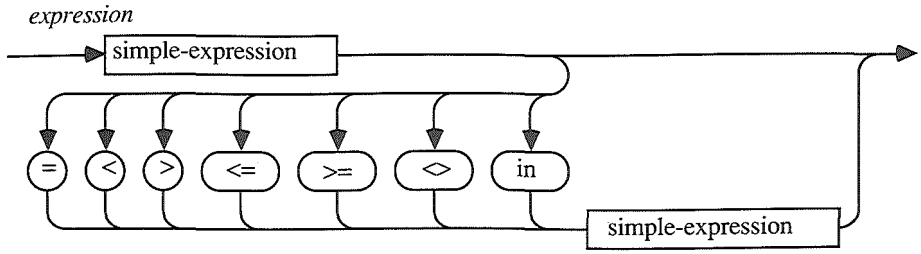
Table 5-1
Precedence of Operators

Unary operators	@, not
Multiplying operators	*, /, div, mod, and
Adding operators	+, -, or
Relational operators	=, <>, <, >, <=, >=, in

Thus, a+b*c is interpreted as a+(b*c), since * has a higher precedence than +. Note that a+b-c is interpreted as (a+b)-c, since + and - have the same precedence.

The precedence rules follow from the syntax of expressions, which are built from factors, terms, and simple expressions.

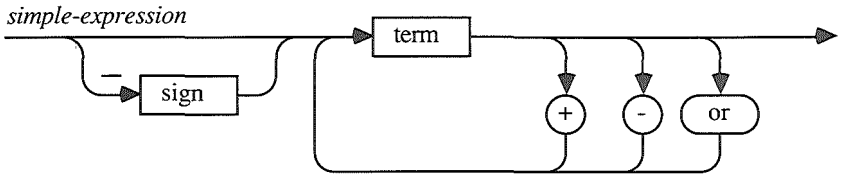
The syntax of an expression is made up of *relational* operators applied to simple expressions:



Example of expressions:

x = 1.5
c **in** *hue1*
done <> error
p ≤ q

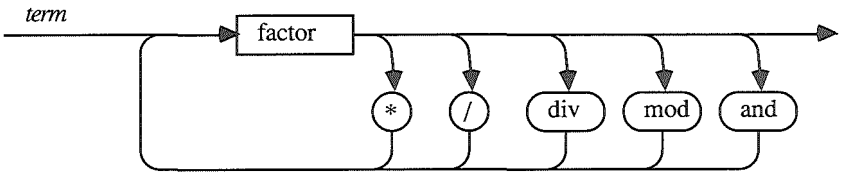
The syntax of a simple expression is made up of *adding* operators and *signs* applied to terms:



Examples of simple expressions:

x+y
-x
hue1 + hue2
b or c

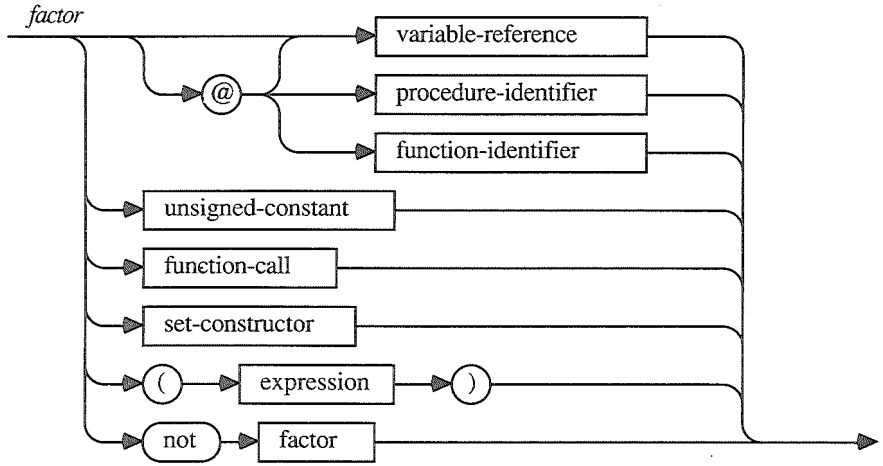
The syntax of a term is made up of *multiplying* operators applied to factors:



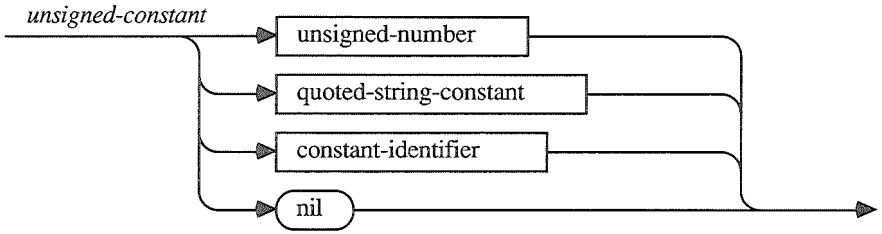
Examples of terms:

*x*y*
e/(1 - e)
done and error

The syntax for a factor is made up of the following basic constructs:



An unsigned constant has the following syntax:



Examples of factors:

x	*(variable reference)*
@x	*(pointer to a variable)*
15	*(unsigned constant)*
'hello world'	*(unsigned constant)*
(x+y+z)	*(sub expression)*
sin (x/2)	*(function call)*
not q	*(negation of a boolean)*
['A'..'F','a'..'f')	*(set construction)*

Operators

The operators are classified as arithmetic operators, boolean operators, set operators, relational operators, and the @ operator.

Arithmetic Operators

The following two tables show the types of operands and results for the binary and unary arithmetic

operators respectively.

Table 5-2
Binary Arithmetic Operations

Operator	*Operation*	*Operand type*	*Result type*
+	addition	integer or real type	same integer or real type (1)
-	subtraction	integer or real type	same integer or real type (1)
*	multiplication	integer or real type	same integer or real type (1)
/	division	integer or real type	real type (2)
div	integer division	integer type	same integer type
mod	modulo	integer type	same integer type

(1) If mixed integer and real type operands then integer operand is converted to real and result type is real.
(2) Integer operands are always converted to real even if both operands are integer.

Table 5-3
Unary Arithmetic Operations

Operator	*Operation*	*Operand type*	*Result type*
+	identity	integer or real type	same integer or real type
-	sign-negation	integer or real type	same integer or real type

If both operands of the +, -, *, **div**, or **mod** operators are of the same integer type (*Integer* or *LongInt*), the result is always of the same integer type. If one of the operands is type *LongInt* and the other is type *Integer,* then the integer operand is first converted to *LongInt* and the result type is *LongInt*. In either case, the resultant value is determined by the normal mathematical rules for integer arithmetic. It is an error if the value of the result is outside the range *-maxint-1..maxint* or *-maxlongint-1..maxlongint.* for *Integer* and *LongInt* result types respectively.

If one of the operands of the +, -, or * operators is of any real type, the result is always of type *Extended,* and has a value that is an approximation of the normal mathematical result. The result of the / operator is always type *Extended.*

If the operand of the identity or sign negation operator is of an integer type, the result is always of the same integer type and the absolute value of the result is always identical to the absolute value of the operand.

If the operand of the identity or sign negation operator is of a real type, the result is always of type real and the absolute value of the result is always identical to the absolute value of the operand.

Boolean Operators

The types of operands and results for Boolean operations are shown in the following table.

Table 5-4
Boolean Operations

Operator	*Operation*	*Operand type*	*Result type*
or	disjunction	boolean	boolean
and	conjunction	boolean	boolean
not	negation	boolean	boolean

The result of a *boolean* operation is determined by the normal rules of boolean logic, e.g. *a and b* evaluates to *true* if and only if both *a* and *b* are true.

Set Operators

The types of operands and results for set operations are shown in Table 5-5.

Table 5-5
Set Operations

Operator	*Operation*	*Operand type*	*Result type*
+	union	compatible set types	if identical operands types then same type as operands, else an anonymous set type defined by min-set-value..max-set-value
-	difference	compatible set types	
*	intersection	compatible set types	

The results of the set operations are determined by the normal rules of set logic. i.e.

- An ordinal value *c* is in the set a+b if and only if *c* is in a *or* in b.
- An ordinal value *c* is in the set a-b if and only if *c* is in a *and not* in b.
- An ordinal value *c* is in the set a*b if and only if *c* is in a *and* in b.

Relational Operators

The types of operands and results for relational operations are shown in the following table.

Table 5-6
Relational Operations

Operator	*Operand type*	*Result type*
= <>	compatible simple, pointer, set, string or packed-string types	boolean
< >	compatible simple, string or packed-string types	boolean
<= =>	compatible simple, set, string or packed-string types	boolean
in	left operand: any ordinal type T right operand: a set-of-T type	boolean

Comparing Ordinals

When the operands of =, <>, <, >, >=, or <= are of an ordinal type, they must be of compatible types unless one of the operands is a real type then the other is allowed to be an integer type. The result is the mathematical relation of their ordinalities. When comparing real types, the results may not be as expected since the representation of a real value is only an approximation.

Comparing Strings

When the relational operators =, <>, <, >, <=, or >= are used to compare strings, they are compared according to their lexicographic ordering. Note that any two string values can be compared since all string values are compatible. Additionally, a *Char* value is compatible with a string type value, and when the two are compared, the *Char* value is treated as a string type value with length one. When a packed string type value with *n* components is compared with a string type value, it is treated as a string type value with length *n*.

Comparing Packed Strings

The relational operators =, <>, <, >, <=, and >= can also be used to compare two values of a packed string type if both have the same number of components. If that number is *n*, then the result is the same as if the values were string type with each having a *length* of *n*.

Comparing Sets

If a and b are set operands then

- a=b is *true* if and only if every member of *a* is a member of *b* and every member of *b* is a member of *a*; otherwise, a<>b.
- a<=b is *true* if and only if every member of *a* is also a member of *b*.
- a>=b is *true* if and only if every member of *b* is also a member of *a*.

Thus, a=b, and a<>b denote the equivalence and non-equivalence of the sets *a* and*b* respectively, and a<=b and a>=b denote the inclusion of *a* in *b* and the inclusion of *b* in *a* respectively.

Comparing Pointers

The relational operators = and <> may be applied to compatible pointer type operands. Two pointers are equal if and only if they denote the same object.

Testing Set Membership

The *in* operator returns *true* if the value of the ordinal type operand is a member of the set type operand; otherwise it yields the value *false.* The type of the left operand must be compatible with the base type of the right operand.

The @ Operator

A pointer value that points to a variable, procedure, or function can be created with the @ operator. The operand and result types are shown in Table 5-7.

@ is a unary operator taking a single variable reference or a procedure or function identifier as its operand and computing the value of its pointer. If the operand to the @ operator is a variable reference, then the pointer value is the address in memory where the variable is stored. If the operand to the @ operator is a procedure or function identifier, then the pointer value is the procedure or function's*entry point.* The type of the value is equivalent to the anonymous pointer type of the pointer constant *nil,* i.e. it can be assigned to any pointer variable.

Table 5-7
Pointer Operations

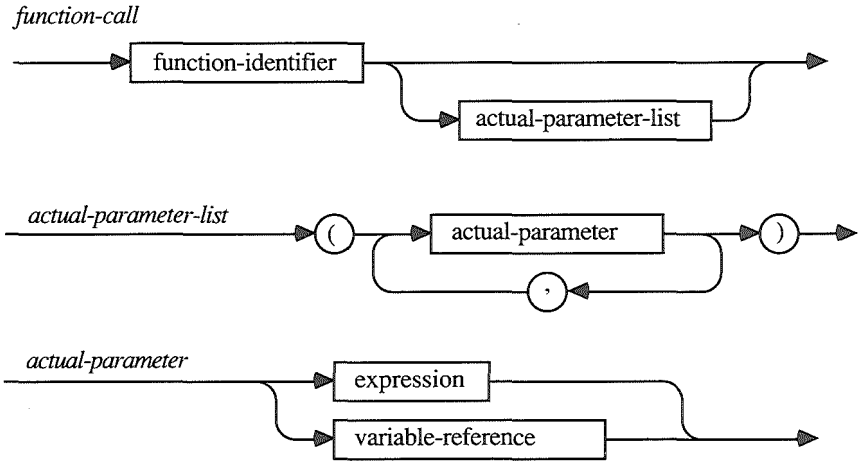
Operator	*Operation*	*Operand type*	*Result type*
@	pointer formation	variable-reference, or procedure or function identifier	the anonymous pointer type of the constant *nil*

IMPLEMENTATION NOTE

The @ operator should only be used in conjunction with procedures and functions declared in the declaration part of the program or unit (global declarations) when the resulting pointer value is passed to an Apple IIGS ROM routine. Procedures and functions declared in the declaration part of another procedure or function (nested declarations) have a different calling convention than those in the declaration part of the program which is not compatible with the Apple IIGS ROM routines. Furthermore,TML Pascal's *DefProc* compiler option should be used when the resulting pointer value is passed to an Apple IIGS ROM routine. See the Appendix "Inside TML Pascal" for a discussion of Pascal calling conventions and the use of the *DefProc* compiler option.

Function Call

A function call specifies the activation of the block associated with the function identifier. The result returned by the function activation is subsequently used as an expression value. If the function has any formal parameters, then the function designator must contain a corresponding list of actual parameters. Each actual parameter is substituted for the corresponding formal parameter.

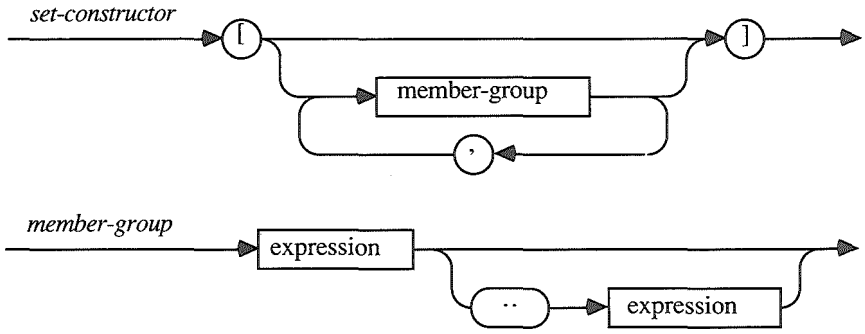


Examples of functions calls:

```
sum(a,63)
sin (x+y)
eof(f)
ord(f^)
```

Set Constructors

A set constructor denotes a value of a set type, and is formed by writing expressions within [brackets]. Each expression denotes a value of the set.



The notation *[ ]* denotes the empty set, which is assignment compatible to every set type. Any member group *x..y* denotes as set members all values in the range *x..y*. If the value of *x* is greater than the value of *y*, then *x..y* denotes no members and *[x..y]* denotes the empty set.

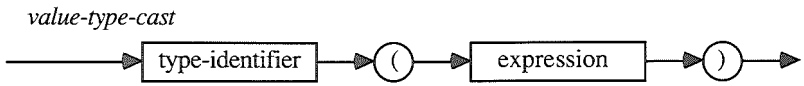
All expression values in the member groups of a particualr set constructor must be of compatible ordinal types. If *a* is the smallest ordinal value in the resulting set, and if *b* is the largest ordinal value in the resulting set, then the base type of the resulting set is *a..b*.

Examples of set constructors:

```
[red, c, green]
[1, 5, 10..k mod 12, 23]
['A'..'Z', 'a'..'z', chr(xcode)]
```

Value Type Casts

The type of an expression can be changed to another type through a value type cast.



The expression argument must be of an ordinal type or pointer type. The result of the type cast is of the specified type, and its ordinal value is obtained by converting the expression. The syntax of a value type cast is almost identical to that of a variable type cast. However, value type casts operate on values, not varaibles, and can therefore not participate in variable references. That is, a value type cast may not have qualifiers appear on the left side of an assignment statement, or as an actual parameter where the formal parameter is declared as a VAR parameter.

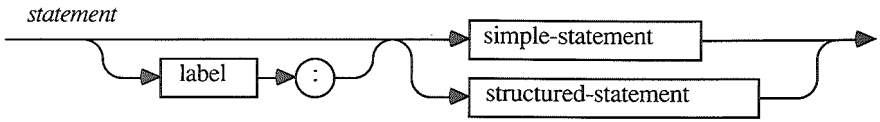
Examples of value type casts:

```
Integer('c')
Ptr($89F2)
Boolean(0)
```

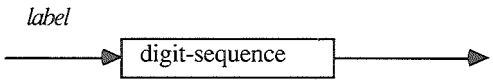
Chapter 6

Statements

Statements describe algorithmic actions that can be executed. There are two classes of statements – simple statements and structured statements. Statements may be prefixed by a label and a labeled statement can be referenced by **goto** statements.

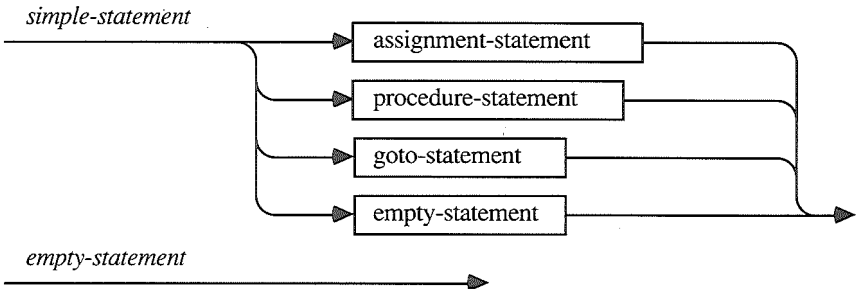


A label is a non-negative integer constant, and must first be declared in a label declaration .



Simple Statements

A simple statement is a statement that does not contain any other statement. The empty statement is a simple statement which contains no symbols and denotes no action.

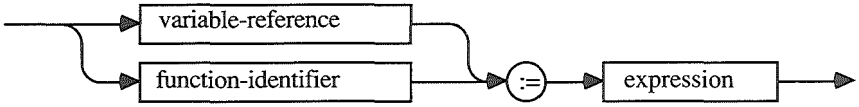


Assignment Statement

The assignment statement can be used to perform either of two actions:

- To replace the current value of a variable by a new value as specified by an expression.
- To specify an expression whose value is to be returned by a function.

assignment-statement



The expression must be assignment compatible with the type of the variable or the result type of the function. The function identifier must be the function identifier of the enclosing function block.

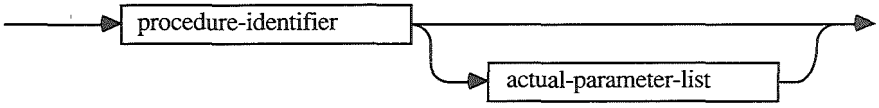
Examples of assignment statements:

```
x := y+z
p := (1<=i) and (i<100);
i  := sqr(k) - (i * j);
hue1 := [blue,succ (c)]
```

Procedure Statement

A procedure statement specifies the activation of the procedure block denoted by the procedure identifier. If the procedure has any formal parameters, then the procedure statement must contain a matching list of actual parameters. Each actual parameter is substituted for the corresponding formal parameter as part of the procedure call.

procedure-statement



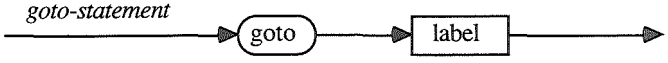
Examples of procedure statements:

```
PrintHeading;
Transpose (a,n,m);
Find (name, address)
```

Goto Statement

A **goto** statement causes the statement prefixed by the label that is referenced in the **goto** statement to be the next statement executed. The following is the syntax of a **goto** statement.

goto-statement

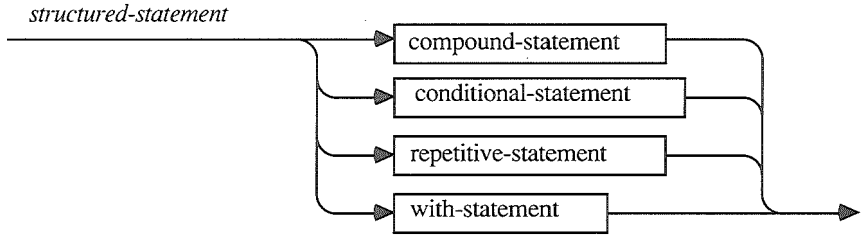


The following rules must be observed when using a **goto** statement.

- The label referenced by a **goto** statement must be in the same block as the **goto** statement. In other words, it is not possible to jump into or out of a procedure or function.
- Jumping into a structured statement from outside that structured statement can have undefined effects and is illegal. However, TML Pascal does not detect the occurrence of such a **goto** statement.

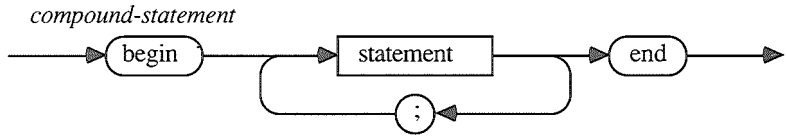
Structured Statements

Structured statements are made up of other statements that are to be executed either conditionally (conditional statements), repeatedly (repetitive statements), or in sequence (compound statement or with statement).



Compound Statements

The compound statement specifies the execution of a sequence of statements in the order in which they are written. The compound statement is treated as one statement in contexts where only a statement is allowed.

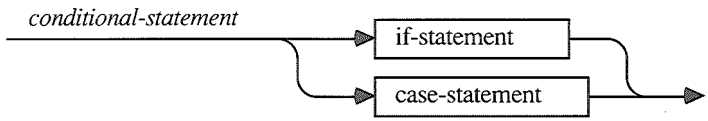


Exampleof a compound statement:

```
begin
    z := x;
    x := y;
    y := z
end
```

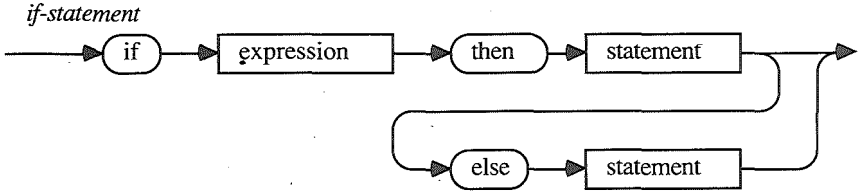
Conditional Statements

A conditional statement selects one or none of its component statements for execution.



If Statements

The syntax of an if statement is as follows:



The expression of the **if** statement must yield a result of the standard type *Boolean.* If the expression yields the value *True,* then the statement following the **then** is executed. If the expression yields *False,* and the **else** part is present, the statement following the **else** is executed; if the **else** part is not present, then execution proceeds with the next statement following the **if** statement.

This syntax for the **if** statement allows for the following ambiguity.

if expression1 **then if** expression2 **then** statement1 **else** statement2

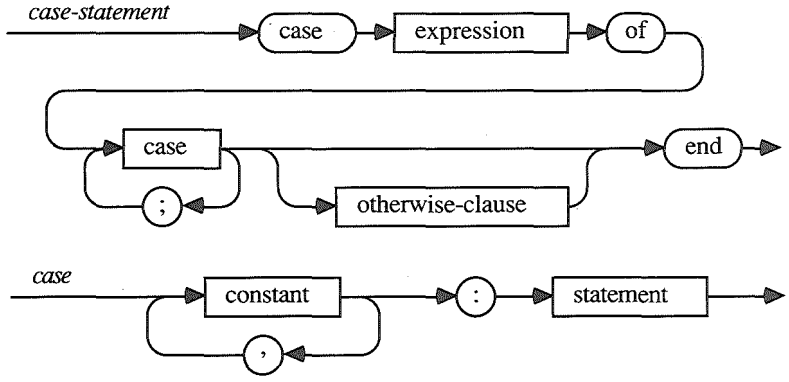
In this case, the **else** is always associated with the closest **if** that is not already associated with an **else**.

Examples of if statements:

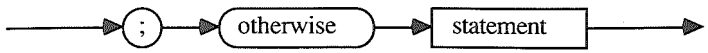
```
if x < 1.5 then
  z := x+y
else
  z := 1.5

if p1 <> nil then
  p1 := p1^.father;
```

Case Statements

The **case** statement consists of an expression (the selector) and a list of statements. Each statement is prefixed with one or more constants (called case constants), or with the reserved word **otherwise**. All the **case** constants must be distinct and must be of an ordinal type that is compatible with the type of the selector expression.



otherwise-clause



The **case** statement executes the statement prefixed with the case constant that equals the value of the selector. If no such **case** constant exists and an **otherwise** clause is present, the statement following the word **otherwise** is executed; if no **otherwise** clause is present then execution continues with the statement following the case statement.

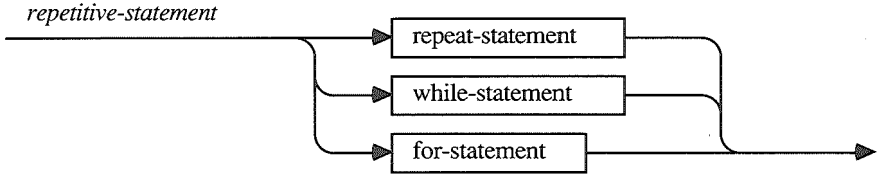
Examples of case statements:

```
case operator of
    plus:    x := x+y;
    minus:   x := x-y;
    times:   x := x*y
end

case i of
    1 :x    x := sin(x);
    2 :x    x := cos(x);
    3,4,5 : x := exp(x);
otherwise
    x := ln(x)
end
```

Repetitive Statements

Repetitive statements specifies a group of statements to be executed repeatedly.

repetitive-statement

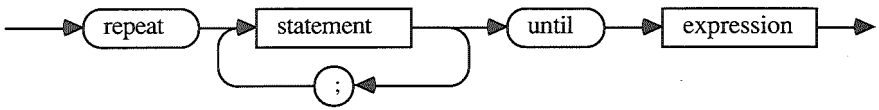


If the number of repetitions is known beforehand, the **for** statement is an appropriate statement to use, otherwise the **while** or **repeat** statements are used.

Repeat Statements

A **repeat** statement contains an expression that controls the repeated execution of a sequence of statements contained within the **repeat** statement.

repeat-statement



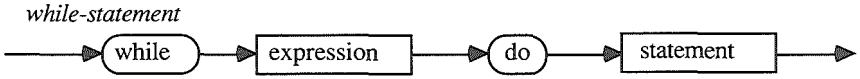
The expression must yield a result of the standard type *Boolean*. The statements between the symbols **repeat** and **until** are repeatedly executed in sequence until, at the end of a sequence, the expression yields the value *True*. The sequence of statements is executed at least once, because the expression is evaluated *after* the execution of each sequence.

Examples of repeat statements:

```
repeat
    k := i mod j;
    i := j;
    j := k
until j = 0

repeat
    process (f^);
    get(f)
until eof(f)
```

While Statements

A **while** statement contains an expression that controls the repeated execution of a statement.



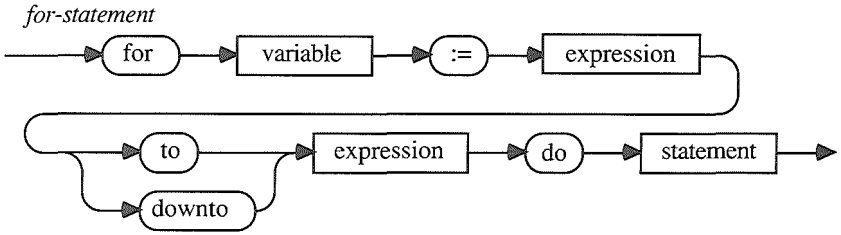
The expression must yield a result of the standard type *Boolean*. The expression is evaluated before the contained statement is executed. The contained statement is repeatedly executed as long as the expression yields the value *True*. If the expression yields *False* at the beginning, the statement is not executed.

Examples of while statements:

```
while a[i] <> x do
    i := i+1

while i>0 do
    begin
        if odd[i] then
            z := z*x;
        i := i div 2;
        x := sqr (x)
    end
```

For Statements

The **for** statement causes a statement to be repeatedly executed while a progression of values is assigned to a variable call the control variable.



The control variable must be a variable identifier (without any qualifier) denoting a variable that is declared either in the declaration part of a program or unit (global) or in the declaration part of the block containing the **for** statement (local). The control variable must be of an ordinal type, and the *initial-value* and *final-value* must be of a type assignment compatible with this type. On entering a **for** statement, the *initial-value* and the *final-value* are determined once (and only once) for the remainder of the execution of the **for** statement.

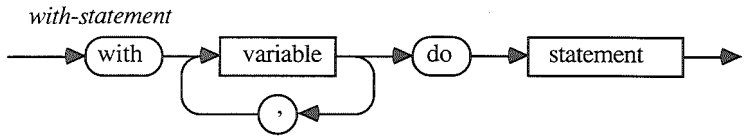
It is illegal for the control variable to appear as a variable reference of an assignment statement or the control variable of another **for** statment in the statement of the **for** statement. After a **for** statement is executed, the value of the control variable is equal to the final value, unless the execution of the **for** statement was terminated by a **goto** out of the **for** statement.

Examples of for statements:

```
for i := 2 to 63 do
    if a[i] > max then
        max := a[i]

for C := red to blue do Check(C);
```

With Statements

The **with** statement is a shorthand method for specifing the fields of a record. Within a **with** statement, the fields of one or more specific record variables can be referenced using only their field identifiers. The syntax of a **with** statement is as follows:



The occurrence of a variable in the **with** statement must denote a record variable. Within a **with** statement, each variable reference is first checked whether it can be interpreted as a field of the record variable. If so, it is always interpreted as such, even if a variable with the same name is accessible.

Examples of a with statement:

```
with date do
    if month = 12 then
        begin
            month := 1;
            year := year + 1
        end
    else
        month := month + 1
```

This is equivalent to:

```
if date.month = 12 then
    begin
        date.month := 1;
        date.year := date.year + 1
    end

else
        date.month := date.month + 1
```

When more than one record variable reference appears in a **with** statement as follows

with var_1, var_2, ... var_n do
    statement

it is considered equivalent to following sequence of nested **with** statements:

```
with var1 do
    with var2 do
        ....
            with varn do
                statement
```

Thus, if var_n in the above statements is a field of both var_1 and var_2, it is interpreted to mean $var_2.var_n$ anc not $var_1.var_n$.

Chapter 7

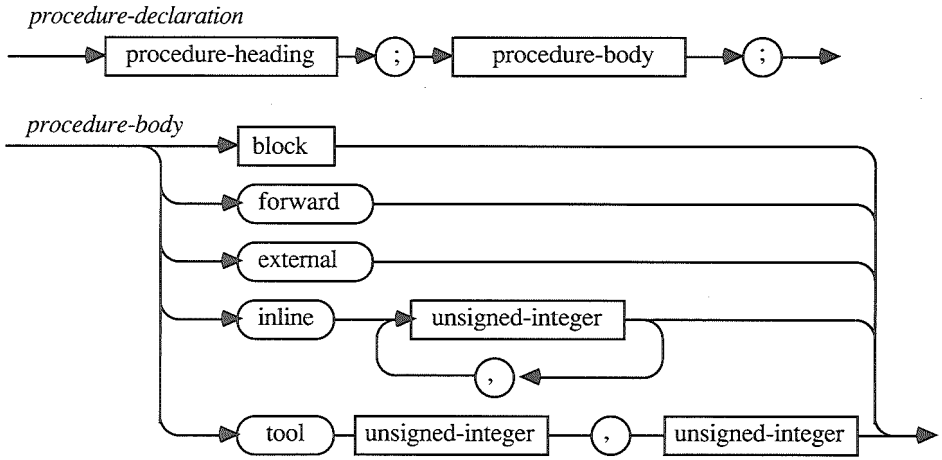
Procedures and Functions

Procedures and functions allow you to nest additional blocks in the main program block or define blocks within a unit. Procedures and functions are also known together as *subprograms.* Each procedure or function has a heading followed by a block or a special directive. A procedure is activated by a procedure statement, and a function is activated by the evaulation of an expression that contains a function call.

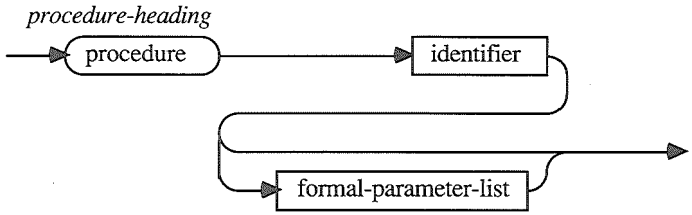
This chapter describes the different types of procedures and functions and their parameters.

Procedure Declarations

A **procedure** declaration associates an identifier with a block as a procedure so that it can be activated by a procedure statement.



The procedure heading specifies the identifier for the procedure, and the formal parameters (if any).



A procedure is activated by a **procedure** statement, which gives the procedure's identifier and any actual parameters required by the procedure. The statements to be executed upon activation of the procedure are specified by the statement part of the procedure's block. If the procedure's identifier is used in a procedure statement within the procedure's block, the procedure is executed *recursively*. That is, it calls itself while it is executing.

Example of a procedure declaration:

```
procedure Num2String (N: Integer; var S: string);
var
  V: Integer;
begin
  V := Abs (N);
  S := '';
  repeat
      S := Concat(Chr (N mod 10 + Ord ('0')) ,S);
      N := N div 10;
  until N = 0;
  if N < 0 then S := Concat('-',S);
end;
```

Instead of the block in a procedure or function declaration, a **forward**, **external**, **inline**, or **tool** directive may be given in its place.

Forward Declarations

A **procedure** declaration that has the directive **forward** instead of a block is called a *forward declaration*. Somewhere after the forward declaration, the procedure is actually defined by a *defining declaration* – a procedure declaration that uses the same procedure identifier and includes a block. The defining declaration may repeat the formal parameter list, but if the formal parameter list is repeated it must be identical to the forward declaration. The forward declaration and the defining declaration must be in the same declaration part, but need not be contiguous; that is, other procedures, functions, types, variables, etc. can be declared between them and can call the procedure that has been declared forward.

The forward declaration and the defining declaration constitute a complete declaration of the procedure. The procedure is considered to be declared at the place of the forward declaration.

Example of a forward declaration:

```
procedure Proc2 ( m,n : integer);       forward;

procedure Proc1 ( x, y : real);
begin
      ...
      Proc2 ( 4, 5 );
end;

procedure Proc2 (m,n : integer);
begin
      ...
      Proc1 ( 8.3, 2.4 );
end;
```

External Declarations

A procedure declaration, whose body is declared *external,* defines the Pascal interface to routines assembled or compiled in a language other than TML Pascal. The external code for the procedure must be available at link time. For more information regarding external procedures and developing in another language see the Appendix "Inside TML Pascal".

Example of an external declaration:

```
procedure GotoXY ( x,y : integer);        external;
```

Inline Declarations

The inline directive allows you to write machine code in place of a procedure's block. The code may only consist of a sequence of integer constants which each represent a single byte of machine code. When the procedure is called, the compiler generates the machine code specified by the inline directive. If the procedure has any parameters, they are pushed onto the stack before the code is generated.

Procedures are usually declared inline to implement very small amounts of code. For example, the following procedure would clear the 65816 Interrupt Disable flag by generating the CLI instruction.

Example of an inline declaration:

```
procedure GenCLI;      inline $58;
```

Tool Declarations

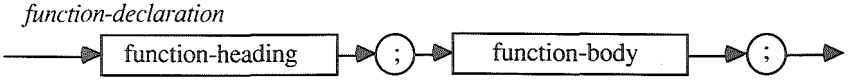
The tool directive is used to define the body of a procedure to be one of the Apple IIGS ROM tool routines. The Apple IIGS ROM tools are divided into several *toolsets* and then into individual *tool routines.* Each toolset is identified by a unique *tool-number,* and each tool routine within a given toolset is assigned a unique *function number.* Using this special method of describing an Apple IIGS ROM routine provides TML Pascal with the information needed to generate a call into the ROM. For example, the MoveTo procedure in the QuickDraw toolset (tool-number 4) is assigned the function numer 58. Thus, the tool declaration is a follows:

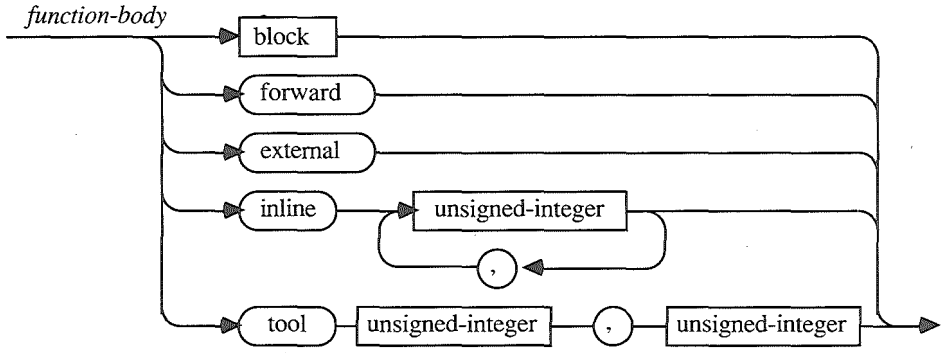
Example of a tool declaration:

```
procedure MoveTo(h,v: integer);      Tool 4,58;
```

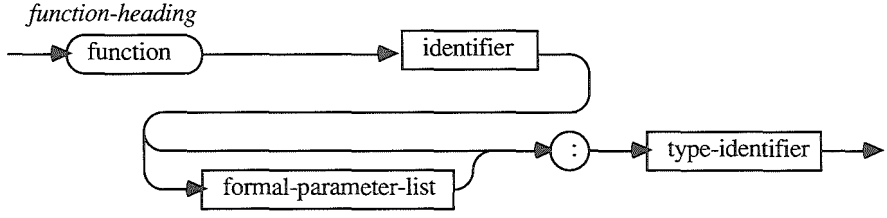
Function Declarations

A function declaration associates an identifier with a block as a function so that it can be activated by a function call to compute and return a value of some type.





The function heading specifies the identifier for the function, the formal parameters (if any), and the type of the function result. The function result type may be any simple or structured type.



A function is activated by the evaluation of a function call, which gives the function's identifier and any actual parameters required by the function. The function call appears as an operand in an expression. The expression is evaluated by executing the function and, in effect, replacing the function call with the value returned by the function.

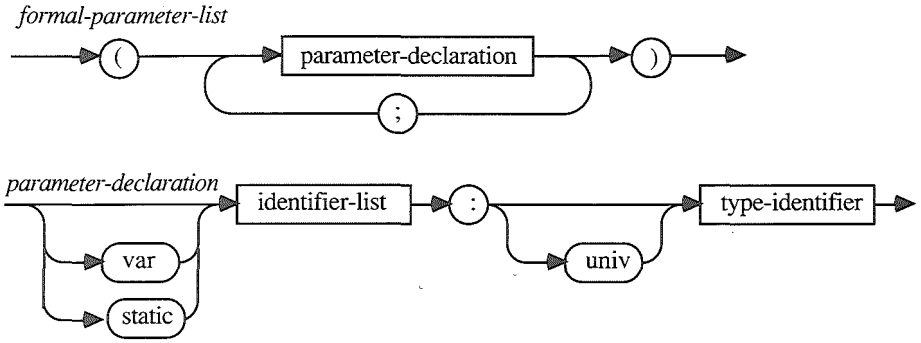
The statements to be executed upon activation of the function are specified by the statement part of the function's block. The block should normally contain at least one assignment statement that assigns a value to the function identifier. The result of the function is the last value assigned. If no such assignment statement exists, or if it exists but is not executed, the value returned by the function is undefined.

If the function's identifier is used in a function call within the function's block, the function is executed *recursively.*

A function can be declared **forward**, **external**, **inline**, or **tool** in same manner as a procedure as described above.

Parameters

The declaration of a procedure or function specifies a formal parameter list. Each parameter declared in a formal parameter list is local to the procedure or function being declared, and can be referred to by its identifier in the block associated with the procedure or function.



There are three kinds or parameters: *value parameters, variable parameters,* and *static parameters.* They are distinguished as follows:

- A parameter group without a preceding **var** or **static** is a list of value parameters.
- A parameter group preceded by **var** is a list of variable parameters.
- A parameter group preceded by **static** is a list of static parameters.

Note that the type of a formal parameter must be a type identifer or the reserved word **string**, a new anonymous type declaration is not allowed.

Value Parameters

A formal value parameter acts like a variable local to the procedure or function, except that it gets its initial value from the corresponding actual parameter upon activation of the procedure or function. Changes made to a value parameter do not affect the value of the actual parameter.

A value parameter's corresponding actual parameter in a procedure statement or function call must be an expression, and its value must not be of a file type or of any structured type containing a file type.

The actual parameter must be assignment compatible with the type of the formal value parameter. If the parameter type is **string**, then the formal parameter is given a static size attribute of 255.

IMPLEMENTATION NOTE

If the size of the formal parameter (in bytes of storage) is greater than 4 bytes, then the actual parameter is passed by address and then the value of the actual parameter is copied into local storage for the formal parameter so that assignments to the formal parameter do not affect the value of the actual parameter. See Static Parameters below and the Appendix "Inside TML Pascal" for more information.

Variable Parameters

A variable parameter is used when the value of a parameter must be passed back from a procedure or function to the caller. The corresponding actual parameter in a procedure statement or function call must be a variable reference. The formal variable parameter represents the actual variable during the activation

of the procedure or function, so any changes to the value of the formal parameter are immediately reflected in the actual parameter.

Within the procedure or function, any reference to the formal variable parameter accesses the actual parameter itself. The type of the actual parameter must be identical to the type of the formal variable parameter (this can be bypassed by using UNIV described below). If the formal parameter is **string**, it is given the length attribute 255, and the actual variable parameter must be a string type with a length of 255.

File types can only be passed as variable parameters.

Components of variables of any packed structured type cannot be used as actual variable parameters.

If the reference to an actual variable parameter involves indexing an array or finding the object of a pointer, these actions are executed before the activation of the procedure or function.

Static Parameters

Static parameters are a special extension to TML Pascal for the Apple IIGS for the specific purpose of obtaining improved code generation. Static parameters are treated *exactly* like value parameters described above except for the restriction that a static formal parameter should not be assigned a new value within the procedure or function.

Value parameters whose formal type requires more than 4 bytes of storage are passed by address and then copied into local storage for the formal parameter so that assigning new values to the formal value parameter does not affect the actual parameter. However, there are cases when the formal parameter is only *read* from and never written to. In these cases, it is not necessary to copy the actual parameter value into local storage for the formal parameter, the formal parameter may access the actual parameter directly.

Static parameters reduce the amount of stack space required by an application, and reduce execution time by not having to copy the value of an actual parameter into local storage for the formal parameter.

WARNING

TML Pascal DOES NOT check that a static parameter is never written to, it is the responsibility of the programmer to ensure the correct usage of static parameters.

UNIV Parameter Types

When the word UNIV appears before the type identifier in the formal parameter list, the restriction that the actual parameter and formal parameter must be assignment compatible in the case of value and static parameters, and identical in the case of variable parameters is not enforced. When UNIV is used the actual parameter may be of any type so long as the number of bytes required to store a value of the actual parameter's type is the same as that of the formal parameter.

Chapter 8

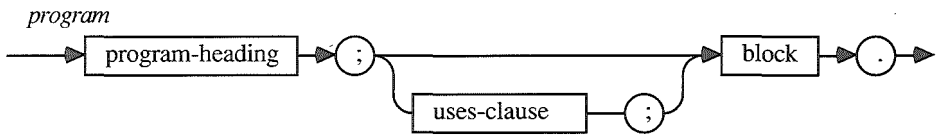
Programs and Units

TML Pascal provides four basic constructs which are the fundamental units of a piece of Pascal source code. These are *programs, units, unit specifications,* and *unit bodies.* The main difference between a program and any of the units, is that a program represents a complete application which can be compiled and executed. A unit, however, can not be executed by itself, it is merely a construct in which *parts* of a program can be defined and compiled independently of a program.

Each of the four basic constructs are compiled separately to produce *object code.* The object code from a program or unit is then pieced together by the Linker inorder to create a standalone ProDOS16 application or Desk Accessory. You should consult the *TML Pascal User's Guide* for the exact details of this process.

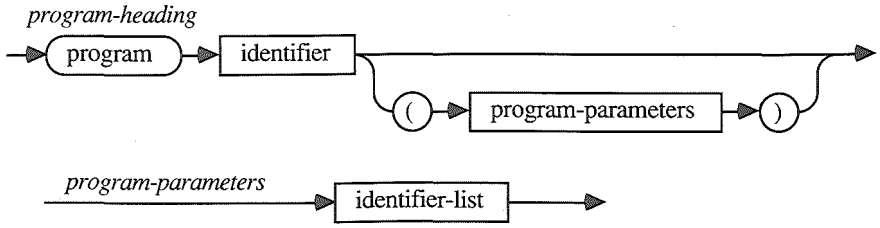
Programs

A Pascal program has the form of a procedure declaration except for its heading and an optional **uses** clause.



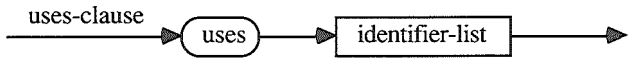
The occurrence of an identifier immediately after the word **program** in the program heading declares it as the program's identifier. The program parameters, if present, are used to designate the program as a *Standard Pascal Application* instead of an *Apple IIGS Application.* In particular, if either of the two standard file variables *Input* or *Output* appear in the program parameters, the program is designated as a *Standard Pascal Application,* otherwise it is an *Apple IIGS Application.* The occurrence of any other names in the program parameter list are ignored, and have no affect on the program.

For more information regarding *Standard Pascal Applications* and *Apple IIGS Applications* consult the *TML Pascal User's Guide.*



Uses Clause

The uses clause is used to identify those units which are required by a program or unit in order to compile sucessfully.



When the name of a unit appears in a uses clause, the declarations of that unit's interface part or unit specification (described below) are considered to have been declared in place of the uses clause. In order to use a unit in a uses clause, the unit must have already been compiled. When TML Pascal encounters a unit's name in a uses clause, it searches for a file having the name of the unit with the suffix ".USYM". This file contains the declarations found in the unit's interface part or unit specification encoded in the compiler's internal symbol table format.

Example of a Uses Clause:

***uses** QDIntf, GSIntf;*

When a unit named in a uses clause uses other units itself, the names of those units must also appear in the uses clause, and they must appear *before* the unit is named. Consider the following example:

```
Unit UnitA;
interface
  const a = 1;
implementation
end.
```

```
Unit UnitB;
uses UnitA;
interface
  const b = a;
implementation
end.
```

```
Program MyProg;
uses UnitA, UnitB;
 const Myconst = b;
begin
end.
```

In this example, the program *MyProg* declares a constant *MyConst* be have the value *b* ,which is declared in unit *UnitB.* Therefore, a uses clause is used to name *UnitB.* However, *UnitB* has a uses clause which names *UnitA.* Thus, the uses clause in the program *MyProg* must name *UnitA* in its uses clause, and further it must appear before *UnitB.*

When a unit or unit specification is compiled, TML Pascal creates a special *Unit Symbol* file which contains the declarations from the unit's interface part or unit specification in TML Pascal's symbol table format. This file has the same name as the unit with the suffix ".USYM". Also encoded in the file is the unit's uses clause. This information lets TML Pascal check to see if all the units a particular unit requires have already been named in a uses clause and that they have not been recompiled since the last compilation of this unit.

If a unit or unit specification has been recompiled, then all units which use it must also be recompiled. For instance, in the example above, if *UnitB* is recompiled, then the *MyProg* must also be recompiled, but *UnitA* need not be recompiled. And if *UnitA* is recompiled then both *UnitB* and *MyProg* must be recompiled.

Code Segmentation

An Apple IIGS application may consist of one or more *code segments.* Small programs are usually made up of only a single code segment, but larger programs are divided into several code segments because the Apple IIGS limits the size of an individual code segment to 64K bytes. The reason for the size restriction is that a code segment must not cross the boundries of a *bank* of memory. On the Apple IIGS, a bank of

memory is 64K bytes.

Code segments are named so that the Linker can oraganize the different pieces of code together based on their code segment names. The default code segment name is *main.* In order to change the name of the current code segment, the TML Pascal *{$CSeg segname }* compiler directive is used. When a *{$CSeg segname }* directive appears in a program or unit, the code for all subsequent procedures and functions is placed in the new code segment. To restore code segmentation back to the default segment, merely place the *{$CSeg main }* directive in your program.

For more information regarding the use of the *{$CSeg segname }* directive see Appendicies B and C.

Data Segmentation

An Apple IIGS application may consist of one or more *data segments* for allocating storage for global variables. Programs are usually made up of only a single data segment, but programs which require a large amount of global storage are divided into several code segments because the Apple IIGS limits the size of an individual data segment to 64K bytes. The reason for the size restriction is that a data segment must not cross the boundries of a *bank* of memory. On the Apple IIGS, a bank of memory is 64K bytes.

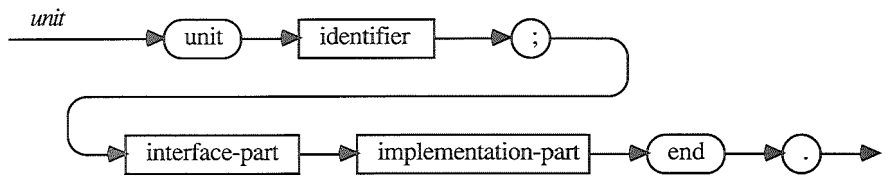
Data segments are named just as code segments are so that the Linker can oraganize the different pieces of data together based on their data segment names. The default code segment name is *~global.* In order to change the name of the current data segment, the TML Pascal *{$DSeg segname }* compiler directive is used. When a *{$DSeg segname }* directive appears in a program or unit, the data for all subsequent global variable declarations is placed in the new data segment. To restore data segmentation back to the default segment, merely place the *{$DSeg ~global }* directive in your program.

Unless a program absolutely requires a large amount of global storage, the *{$DSeg segname }* should not be used. The reason for this is that all global storage allocated outside of the *~global* data segment is addressed using less efficient addressing modes than data allocated in the *~global* data segment.

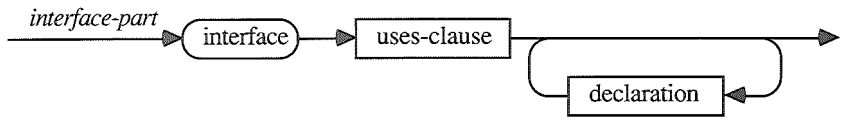
For more information regarding the use of the *{$DSeg segname }* directive see Appendicies B and C.

Units

Units are the basis for modular programming in TML Pascal. Units are compiled separately from one another and should be used to organize large programs into logically related parts. Dividing a program into several units also reduces the amount of time necessary to recompile a piece of code.



The identifier following the reserved word **unit** is the unit's identifier. This name is used to create the compiler's unit symbol file that will be used when this unit's identifier appears in a uses clause.

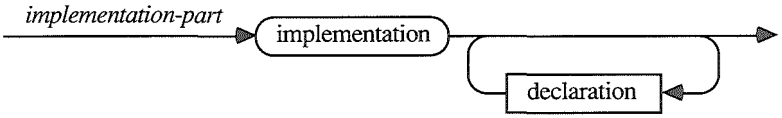


The interface part of a unit declares constants, types, variables, procedures, and functions that are *public*, that is, available to the unit or program that names the unit in its uses clause. In other words, the scope of the public declarations is the entire program or unit that uses the unit. The program or unit that uses a unit can access the public declarations just as if they had been declared in its own block.

Label declarations are not permitted in the interface part. Procedures and functions in the interface part are declared by giving only the procedure or function name, the formal parameters (if any), and the result type (if a function). In other words, you give only the part that defines how the procedure or function is called. If a procedure or function is **external**, **inline**, or **tool** then this directive must be given in the interface part, otherwise the declaration is treated as if **forward** had been specified.

Variables, procedures and functions which appear in the interface part are global. The entire unit is within the scope of the block in which the uses clause that references the unit appears.

The interface part may contain a uses clause, so any unit can use another unit.



The implementation part, which follows the last declaration of the interface part, declares any constants, types, variables, procedures, or functions that are *private*, that is, not available to the program or unit which uses it. Private procedures and functions are declared like procedures and functions in programs, with a procedure or function heading and a body.

All public procedures and functions declared in the interface part are redeclared in the implementation part. Formal parameters and result types may be omitted, but if they appear, they must be identical to the previous declaration.

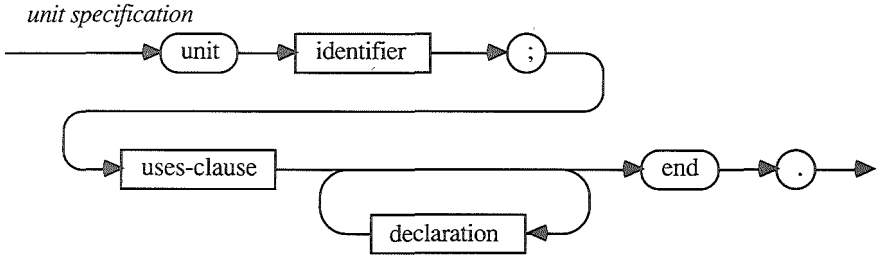
Separate Unit Specifications and Bodies

The unit structure defined in the previous section is the traditional structure of a unit found in many Pascal compilers. That is, the interface and implementation parts are in the same file. While this style unit greatly assists in dividing an application into logically related parts, it has two major limitations: two units can not include each other in their uses clause, and the need for recompilation of dependant units if only the private implementation part is changed.

TML Pascal supports a new style unit in addition to the traditional style unit to address these issues. The new style unit splits the unit interface and implementation parts into two separate files which are the *unit specification* and *unit body* respectively.

The Unit Specification

The unit specification syntax is nearly identical to that of a standard unit except that it does not have the reserved words **interface** or **implementation.** The unit specification contains declarations for constants, types, variables, procedures and functions that are public. Again, the procedures and functions only have their headings declared, and are completed in the unit body.

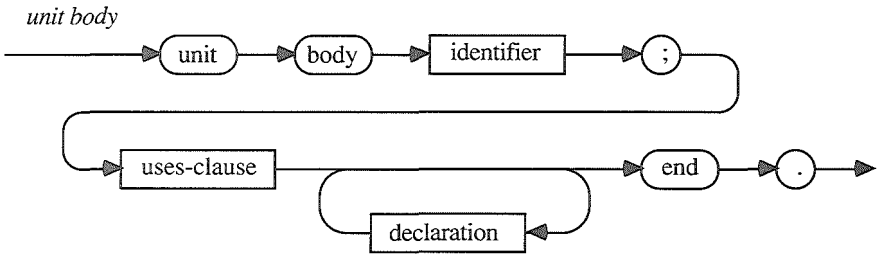


The Unit Body

The unit body corresponds the the unit implementation part of the standard unit. It completes the bodies of the procedure and function declarations made in the corresponding unit specification, as well as declaring additional constants, types, variables, procedures and functions which are private.

The unit body has an *implicit* uses clause immediately before the unit body's own uses clause which specifies all of the units given in the unit specification's uses clause as well as the unit specification itself. Then the unit body may have its own uses clause which specifies additional units required by the unit body.

The unit body may be recompiled without affecting the recompilation of units which use the unit specification.



Note that by having unit bodies it is possible for two units to use each other in their unit bodies. For example:

```
Unit UnitA;
  type atype = ^integer;
end.

Unit Body UnitA;
  uses UnitB;
  var a: atype;
      b: btype;
end.
```

```
Unit UnitB;
  type btype = ^longint;
end.

Unit Body UnitB;
  uses UnitA;
  var a: atype;
      b: btype;
end.
```

Example of a separate unit specification and unit body:

```
Unit IntegerStack;

  procedure Push(elt: integer);
  procedure Pop(var elt: integer);
  function isEmpty: boolean;

end.

Unit Body IntegerStack;

  var
      stack: array[0..20] of integer;
      top: integer;

  procedure Push(elt: integer);
  begin
    ...
  end;

  procedure Pop(var elt: integer);
  begin
    ...
  end;

  function isEmpty: Boolean;
  begin
    ...
  end;

end.
```

Chapter 9

Input / Output

This chapter describes the standard input and output procedures and functions provided by the TML Pascal compiler for the manipulation of files. The standard I/O procedures and functions are predeclared. Since predeclared entities act as if they were declared in a block surrounding the program or unit, no conflict arises from a declaration that redeclares the same identifier within the program except that it hides the predeclared procedure or function .

Introduction to I/O in TML Pascal

The following paragraphs outline how input and output are implemented in TML Pascal and how to use the standard I/O routines provided by TML Pascal to manipulate files and devices.

Using the Standard I/O Routines

In Pascal, a file variable is any variable whose type has been declared as a file type. TML Pascal distinguishes between two classes of file types: *textfiles* and *typed-files.* A textfile file variable is a file variable which has been declared of type *Text. Text* is a special predeclared TML Pascal type identifier. A Textfile is essentially a packed file of characters organized into lines. Each line is terminated by a special end of line character (the end of line character on the Apple IIGS is the carriage return – *Chr(13)*). A typed-file file variable is a file variable which has been declared of a file type using the **file of** construct to define the file component type.

Before a file variable is used, it must be *opened* in order to associate the file variable with an external file or device. External files are disk files which are managed using ProDOS16. Devices are special I/O mechanisms such as the keyboard or display. An existing file is opened using the *Reset* procedure, while a new file is created and opened using the *Rewrite* procedure.

The standard files *Input* and *Output* are automatically opened when a program begins execution. *Input* is a read-only file which is associated with the Apple IIGS keyboard, and *Output* is a write-only file which is associated with the Apple IIGS display in either of the two super-hires graphics modes. When reading from *Input,* the characters are echoed to *Output.* Standand output is not available in any of the other Apple IIGS display modes.

A file is a linear sequence of components, all of the same type as the component type of the associated file variable. Each component has a component number. The first component number of a file is component zero.

Textfiles are accessed sequentially, in either read-only or write-only mode depending upon whether they were opened with *Reset* or *Rewrite.* Typed-files can be accessed either sequentially or randomly regardless of which procedure was used to open the file. To access a file sequentially, the *Read* and *Write* procedures are used, while the *Seek* procedure is used to access a file randomly.

Once a program has completed using a file, it must be closed using the standard procedure *Close.* Closing a file completely updates a file and breaks the association between the Pascal file variable and the external disk file or device. Once a file variable has been closed, it can be opened again with the same or another file. Note that it is not necessary to close the standard file variables *Input* and *Output.*

TML Pascal does not support the file variable buffer together with the *Get* and *Put* procedures as defined in Standard Pascal. Instead, the *Read* and *Write* procedures are used to achieve the same results.

For example:

```
f^:=i;          or      Get(f);
Put(f);                 i:=f^;
```

can be written as

```
Write(f,i);             Read(f,i);
```

Disk Files

When specifing an external file to any of the standard TML Pascal procedures, the file's ProDOS16 *pathname* must be given. A *pathname* consists of a file name optionally preceded by the file's volume name and zero or more directory names. The volume name, directory names, and file name are separated by slashes (/). For example,

```
MyVolume/MyDir1/ ... /MyDirN/MyFile
```

If a pathname which is passed to a standard TML Pascal I/O procedure does not include a volume name, then it is assumed to reside in the default directory (also known as the default prefix). In order to change the default directory, the ProDOS16 procedure *P16SetPrefix* must be used before calling a standard TML Pascal procedure. The *P16SetPrefix* procedure is declared in the *P16Calls* Unit.

TML Pascal creates textfiles with ProDOS16 file type $04 (text file) and typed-files with ProDOS16 file type $00 (binary file).

For more information regarding files and ProDOS16, see the Apple documentation *ProDOS16 Reference.*

Devices in TML Pascal

In addition to external disk files, TML Pascal supports a set of devices for input and output. These devices are the keyboard, the display, and the printer. The keyboard and display devices are automatically available to all programs using the Apple IIGS super-hires display modes when the program begins execution with the standard file variables *Input* and *Output* respectively.

The printer is also available as a text device, but must be explicitly opened using the *Rewrite* procedure with the filename 'Printer:'. For example,

```
var Printer: Text;
begin
   Rewrite(Printer,'printer:');
   Writeln(Printer,'Hello world');
   Close(Printer);
end;
```

Standard Procedures and Functions for All Files

The Reset Procedure

Syntax: Reset (f [, title])

Reset opens an existing file for input or "rewinds" an open file by repositioning the current file position to the zero component. *f* is a file variable of any file type. *title* is an optional string type expression.

If *title* is provided in the parameter list, then *Reset* attempts to open an already existing file with the name *title* and then associates the file variable *f* with the external file. If the file can not be opened, then an error is returned in *IOResult.*

If *title* is not provided in the parameter list, then *f* must already be associated with an open file. In this case, *Reset* repositions the file position to the zero component of the file.

The Rewrite Procedure

Syntax: Rewrite (f [, title])

Rewrite creates and opens a new file or erases the contents of an already open file. *f* is a file variable of any file type. *title* is an optional string type expression.

If *title* is provided in the parameter list, then *Rewrite* creates and opens a new external file with the name *title* and then associates the file variable *f* with the external file. If the file already exists, it is opened and its entire contents erased.

If *title* is not provided in the parameter list, then *f* must already be associated with an open file. In this case, *Rewrite* repositions the file position to the zero component of the file and erases the entire contents of the file.

The Close Procedure

Syntax: Close (f)

Close closes the open file. *f* is a file variable of any file type. The association between *f* and its external file is broken and the file system marks the external file "closed".

The Rename Procedure

Syntax: Rename (oldtitle, newtitle)

Renames an exiting file external file. *oldtitle* and *newtitle* are string type expressions. The external file named *oldtitle* is renamed to *newtitle*. If a file with the name *oldtitle* can not be found then an error is returned in IOResult.

The Erase Procedure

Syntax: Erase (title)

Erases an external file. *title* is a string type expression. The external file with the name *title* is deleted from

its external storage device.

The IOResult Function

Syntax: IOResult
Result type: Integer

Returns an integer value that is the status of the last I/O operation performed. A value of zero indicates successful completion of the last I/O operation, while a non-zero value indicates an error. The error codes are summarized in Appendix A.

Note that IOResult returns the status of the *last* I/O operation performed. Thus, the following two statements do not provide the anticipated results.

```
Reset(f,'myfile');
Writeln('IOresult for Reset = ',IOResult);
```

The call to the IOResult function in the Writeln parameter list actually returns the status of the write operation for the string 'IOresult for Reset = ' since that was the most recent I/O operation, and not the call to Reset. Instead the previous two statements should be rewritten as:

```
Reset(f,'myfile');
tmpint := IOResult;
Writeln('IOresult for Reset = ',tmpint);
```

Standard Procedures and Functions for Typed-Files

The Read Procedure

Syntax: Read (f, v_1 [, v_2 , ... , v_n])

Reads a file component into a variable. *f* is a file variable, and each parameter *v* is a variable of the same type as the component type of the file *f*. For each parameter *v*, the file component at the current file position is read into *v* and the file position advanced to the next file component. If an attempt is made to read past the end of file, then an error is returned by IOResult.

The Write Procedure

Syntax: Write (f, v_1 [, v_2 , ... , v_n])

Writes a variable into a file component. *f* is a file variable, and each parameter *v* is a variable of the same type as the component type of the file *f*. For each parameter *v* the vale of *v* is written to the file component at the current file position and the file position is advanced to the next file component. If the current file position is at the end of the file, then the file is expanded to include the new file component.

The Seek Procedure

Syntax: Seek (f, n)

Changes the current file position to the file component *n*. *f* is a file variable, and *n* is an expression of type LongInt. The number of the first file component is zero. If the value of *n* is greater than the number of compenents in the file, then the current file position is moved to the end of the file, and *Eof(f)* is true.

The FilePos Function

Syntax: Filepos (f)
Result Type: LongInt

Returns the number of the file components at the current position of a file. *f* is a file variable.

The Eof Function

Syntax: Eof (f)
Result Type: Boolean

Returns the end of file status of a file. *f* is a file variable. *Eof(f)* returns *true* if the current file position is beyond the last component of the file, otherwise it returns *false*.

Standard Procedures and Functions for Textfiles

The Read Procedure

Syntax: Read ([f,] v_1 [, v_2 , ... , v_n])

Reads one or more values from a textfile into the corresponding parameters v_i. *f*, if specified, is a textfile variable. If *f* is omitted, the standard file *Input* is assumed which is associated with the Apple IIGS keyboard. Each *v* is a variable of an integer, longint, real, char, or string type.

Read a Char type variable. With a char type variable, *Read* reads one character from the file and assigns that character to the variable. If *Eof(f)* was true before the read was performed, then the value Chr(0) is returned. If *Eoln(f)* was true before the read was performed, then the value Chr(13) is returned. The next read will start with the next character in the file.

Read an Integer or LongInt type variable. With an integer or longint type varaible, *Read* expects a sequence of characters which form a signed whole number. All spaces, tabs, and end of lines are skipped until the beginning of the numeric string is found. Then all characters which are not a space, tab or end of line are assumed to be part of the numeric string. The string is then interpretted as a numeric value. If any characters in the string do not represent a signed whole number, then an error is returned by IOResult. The next read will start with the character which terminated the numeric string.

Read a Real type variable. With a real type variable, *Read* expects a sequence of characters which form a signed floating point number. All spaces, tabs, and end of lines are skipped until the beginning of the numeric string is found. Then all characters which are not a space, tab or end of line are assumed to be part of the numeric string. The string is then interpretted as a floating point value. If any characters in the string do not represent a real number, then an error is returned by IOResult. The next read will start with the character which terminated the numeric string.

Read a String type variable. With a string type variable, *Read* reads all characters into the string variable up to, *but not including,* the next end of line character. The next read will start with the end of line character which terminated the read. Note that successive reads of a string type will not read successive lines from the file since a read of a string type variable *never* advances past an end of line character.

The ReadIn Procedure

Syntax: ReadIn ([f,] v_1 [, v_2 , ... , v_n])

This procedure is an extension to the *read* procedure. After doing the same as *Read* for the parameter list, it skips to the beginning of the next line of the input file by skipping all characters in the input file until an end of line character is found and then reading that end of line character. Again, if *f* is omitted, then the standard file *Input* is assumed.

The Write Procedure

Syntax: Write ([f,] v_1 [, v_2 , ... , v_n])

Writes one or more values to a textfile. *f* if specified, is a textfile variable. If *f* is omitted, the standard file *Output* is assumed which is associated with the TML Pascal standard Pascal "Plain Vanilla" environment. Each *v* is an expression of an integer, longint, real, char, boolean or string type.

Each *v* is known as a *write-parameter*. Each write-parameter has the form

OutExpr [: MinWidth [: DecPlaces]]

where *OutExpr* is an output expression of an allowable type. *MinWidth* and *DecPlaces* are expressions with integer-type values.

MinWidth specifies and *minimum* field width. *MinWidth* must be greater than zero. Exactly *MinWidth* characters are written (using leading spaces if necessary), except when *OutExpr* has a value that must be represented in more than *MinWidth* characters; in this case, enough characters are written to represent the value of *OutExpr*. Likewise, if *MinWidth* is omitted, then enough characters as necessary are written to represent the value of *OutExpr*.

DecPlaces specifies the number of decimal places in a fixed-point representation of a *real* value. It can be specified only if *OutExpr* has a real-type value, and if *MinWidth* is also specified. If specified, it must be greater than zero. If *DecPlaces* is not specified, a floating-point representation is written.

The WriteIn Procedure

Syntax: WriteIn ([f,] v_1 [, v_2 , ... , v_n])

This procedure is an extension to the *Write* procedure. After doing the same as *Write* for the parameter list, it writes the end of line character to the file. Again, if *f* is omitted, then the standard file *Output* is assumed.

The Eof Function

Syntax: Eof [(f)]
Result Type: Boolean

Returns the end of file status of a file. *f* is a textfile variable. *Eof(f)* returns *true* if the current file position is beyond the last component of the file, otherwise it returns *false*. If *f* is omitted then the standard file *Input* is assumed.

The Eoln Function

Syntax: Eoln [(f)]
Result Type: Boolean

Returns the end of line status of a file. *f* is a textfile variable. *Eoln(f)* returns *true* if the character at the current file position is the end of line character or if *Eof(f)* is true, otherwise it returns *false*.

The Page Procedure

Syntax: Page [(f)]

Writes the form feed character to a textfile. *f* is a textfile variable. If *f* is omitted then the standard file *Output* is assumed.

Chapter 10

Standard Procedures and Functions

This chapter describes all the standard, predeclared procedures and functions, and the single predeclared variable, ToolErrorNum, provided in TML Pascal, except for the standard Input/Output procedures and functions which are documented in Chapter 9.

Standard procedures and functions are predeclared. Since predeclared entities act as if they were declared in a block surrounding the program or unit, no conflict arises from a declaration that redeclares the same identifier within the program except that it hides the predeclared procedure or function .

The Flow of Control Procedures

The Exit Procedure

Syntax: Exit [(id)]

The Exit procedure causes execution of a particular block to terminate immediately. Essentially, it is equivelant to a **goto** statement to a label at the very end of the block identified by *id*. If the *id* parameter is omitted then the current block is terminated.

The Halt Procedure

Syntax: Halt

The Halt procedure causes execution of a program to terminate immediately.

The Cycle Procedure

Syntax: Cycle

The Cycle procedure causes the execution of the body of a loop to skip to the end of the loop and continue execution of the next iteration of the loop. The Cycle procedure is only meaningful in a **for** loop, a **while** loop, and a **repeat** loop. If it appears outside of the context of these statements, it has no affect.

The Leave Procedure

Syntax: Leave

The Leave procedure causes the execution of the body of the loop in which it occurs to terminate and continue execution with the first statement after the loop. The Leave procedure is only meaningful in a **for** loop, a **while** loop, and a **repeat** loop. If it appears outside of the context of these statements, it has no affect.

Dynamic Allocation Procedures

These procedures are used to manage the heap, a memory area that is unallocated when a program begins execution. The heap used by the dynamic allocation procedures is the Apple IIGS Application heap, and the routines are implemented using the Apple IIGS Memory Manager.

The New Procedure

Syntax: New(p)

New(p) creates a new variable of the base type of *p*, and makes *p* point to it. The variable can be referenced as *p^*. It is an error if the heap does not contain enough free space to create the new variable. *New* actually calls the Memory Manager routine *NewHandle* to allocate a region of memory which is *locked* and *fixed bank*, and then returns a pointer to the region of memory.

The Dispose Procedure

Syntax: Dispose(p)

Dispose(p) destroys the dynamic variable referenced by p and returns its memory region to the heap. It must be a variable that was previously assigned by the *new* procedure or was assigned a meaningful value by an assignment statement. The value of *p* then becomes undefined and it is an error to subsequently make reference to *p^*.

Transfer Procedures and Functions

Note that the standard procedures Pack and Unpack as defined by the Pascal Standard are not implemented in TML Pascal.

The Trunc Function

Syntax: Trunc(x)
Result Type: LongInt

Trunc(x) returns a *LongInt* result that is the value of the real type variable *x* truncated to the nearest whole number that is between 0 and x inclusive. It is an error if the result of this rounding is outside the range *-maxlongint-1..maxlongint.*

The Round Function

Syntax: Round(x)
Result Type: LongInt

Round(x) returns a *LongInt* result that is the value of the real type variable *x* rounded to the nearest whole number. If x is exactly halfway between two whole numbers, the result is the whole number with the greatest absolute magnitude. It is an error if the result of this rounding is outside the range *-maxlongint-1...maxlongint.*

The Ord4 Function

Syntax: Ord4(x)
Result Type: LongInt

Ord4(x) returns the ordinal number of an ordinal type or pointer type value. *Ord4* corresponds to *Ord*, except that the type of the result is always *LongInt.*

The Pointer Function

Syntax: Pointer(x)
Result Type: the *anonymous* pointer type

Returns a pointer value that points to whatever is at the address *x* as though it were a dynamic variable created at that address. This pointer is of the same type as *nil* in that it is assignment compatible with any pointer type.

Arithmetic Procedures and Functions

The Inc Procedure

Syntax: Inc(x)

Increments the *Integer* type variable *x* by 1. Note that *Inc* is not available for *LongInt.*

The Dec Procedure

Syntax: Dec(x)

Decrements the *Integer* type variable *x* by 1. Note that *Dec* is not available for *LongInt.*

The Abs Function

Syntax: Abs(x)
Result Type: same type as parameter.

Returns the absolute value of *x*; i.e. if *x* is negative, -x is returned; otherwise *x* is returned. *X* is an integer or real type argument.

The Sqrt Function

Syntax: Sqrt(x)
Result Type: Real

Returns the positive square root of *x*, i.e. the positive value y such that y*y=x. It is an error if the result is a value too small to be represented by the real type *extended*. *X* is an integer or real type argument.

The Sin Function

Syntax: Sin(x)
Result Type: real

Returns the trigonometric sine of *x* in radians. *X* is an expression of a real type.

The Cos Function

Syntax: Cos(x)
Result Type: real

Returns the trigonometric cosine of *x* in radians. *X* is an expression of a real type.

The Exp Function

Syntax: Exp(x)
Result Type: real

Returns the value of e^X, where *e* is the base of the natural logarithms. It is an error if the result cannot be represented with the real type *extended*. *X* is an expression of a real type.

The Ln Function

Syntax: Ln(x)
Result Type: real

Ln(x) returns the natural logarithm ($\log_e$) of x. *X* is an expression of a real type.

The Arctan Function

Syntax: Arctan(x)
Result Type: real

Returns the principle value, in radians, of the arctangent of x. *X* is an expression of a real type.

Ordinal Functions

The Odd Function

Syntax: Odd(x)
Result Type: Boolean

Returns *True* if *x* is odd, i.e. not divisible by 2 without a remainder. If *x* is even, it returns *False*. *X* is an expression of an ordinal type.

The Ord Function

Syntax: Ord(x)
Result Type: Integer or LongInt

If *x* is of type *Integer* or *LongInt*, the result type is the same as *x*. If *x* is a pointer type, the result is the corresponding address of the dynamic variable pointed to by *x*, of type *LongInt*. If *x* is of an ordinal type, the result is of type *Integer* and the value is the ordinality of *x*. The standard procedure Ord4 should be used if the result type *LongInt* is desired, regardless of the type of *x*.

The Chr Function

Syntax: Chr(x)
Result Type: Char

Returns the *Char* value whose ordinal number is *x*. For any *Char* value *ch*, the following is always true: chr(ord(ch)) = ch.

The Succ Function

Syntax: Succ(x)
Result Type: same as parameter

Returns the successor of *x*. It is an error if *x* is the last value in the type of *x*, i.e. it has no successor.

The Pred Function

Syntax: Pred(x)
Result Type: same as parameter

Pred(x) returns the successor of *x*. It is an error if *x* is the first value in the type of *x*, i.e. it has no predecessor.

String Procedures and Functions

The string procedures and functions do not accept as parameters *packed string types*, but rather only *string types*.

The Length Function

Syntax: Length(str)
Result Type: Integer

Returns the dynamic length of a string.

The Pos Function

Syntax: Pos(substr, str)
Result Type: Integer

Pos(substr, str) searches for *substr* within *str*, and returns an *Integer* value that is the index of the first character of *substr* within *str*. If *substr* is not found, *pos(substr, str)* returns zero.

Concat Function

Syntax: Concat(str_1 [, str_2, ...str_n])
Result Type: anonymous string type

Concat(str_1, ..., str_n) concatenates all the parameters in the order in which they are written, and returns the concatenated string. Note that the number of characters in the result cannot exceed 255.

The Copy Function

Syntax: Copy(source, index, count)
Result Type: string type

Copy(source, index, count) returns a string containing *count* characters from the string *source*, beginning at *source[index]*.

The Delete Procedure

Syntax: Delete(dest, index, count)

Delete(dest, index, count) removes count characters from the value of the string *dest*, beginning at *dest[index]*.

The Insert Procedure

Syntax: Insert(source, dest, index)

Insert(source, dest, index) inserts the string *source* into the string *dest*. The first character of *source* becomes *dest[index]*.

Logical Bit Functions and Procedures

This section describes a set of procedures and functions for bit manipulations. These routines correspond to a set of essentially identical machine instructions of the 65816.

The BitAnd Function

Syntax: BitAnd(arg1,arg2)
Result Type: Integer or LongInt depending on types of arg1 and arg2

BitAnd returns the logical AND of its two arguments. *arg1* and *arg2* are both expressions of an ordinal type.

The BitOr Function

Syntax: BitOr(arg1,arg2)
Result Type: Integer or LongInt depending on types of arg1 and arg2

BitOr returns the logical OR of its two arguments. *arg1* and *arg2* are both expressions of an ordinal type.

The BitXor Function

Syntax: BitXor(arg1,arg2)
Result Type: Integer or LongInt depending on types of arg1 and arg2

BitXor returns the logical exclusive OR of its two arguments. *arg1* and *arg2* are both expressions of an ordinal type.

The BitNot Function

Syntax: BitNot(arg1)
Result Type: Integer or LongInt depending on types of arg1 and arg2

BitNot returns the logical negation (one's complement) of its argument. *arg1* is an expression of an ordinal type.

The BitSL Function

Syntax: BitSL(arg)
Result Type: Integer or LongInt depending on types of arg1 and arg2

BitSL left shifts the bits of *arg* by one bit. *arg* is an expression of an ordinal type.

The BitSR Function

Syntax: BitSR(arg)
Result Type: Integer or LongInt depending on type of arg

BitSR right shifts the bits of *arg* by one bit. *arg* is an expression of an ordinal type.

The BitRotL Function

Syntax: BitRotL(arg)
Result Type: Integer or LongInt depending on type of arg

BitRotL left rotates the bits of *arg* by one bit. *arg* is an expression of an ordinal type.

The BitRotR Function

Syntax: BitRotR(arg)
Result Type: Integer or LongInt depending on type of arg

BitRotR right rotates the bits of *arg* by one bit. *arg* is an expression of an ordinal type.

The HiWord Function

Syntax: HiWord(arg)
Result Type: Integer

HiWord returns the high order word of the ordinal value *arg*, that is, bits 31-24 of a LongInt. If *arg* is not a LongInt, then HiWord returns zero. When the argument is a simple variable or array access, no code is

generated by this function because the argument is simply addressed and used as an integer.

The LoWord Function

Syntax: LoWord(arg)
Result Type: Integer

LoWord returns the low order word of the ordinal value *arg*, that is, 23-0 of a LongInt. When the argument is a simple variable or array access, no code is generated by this function because the argument is simply addressed and used as an integer.

Miscellaneous Functions

The SizeOf Function

Syntax: SizeOf(id)
Result Type: Integer

Returns the number of bytes occupied by the variable or type *id.*

The Card Function

Syntax: Card(s)
Result Type: Integer

Counts the number of elements in the set *s* and returns an integer value which is the cardinality of the set, that is, the number of members in the set.

Apple IIGS ROM Tool Error Handling

The Apple IIGS ROM tools define a convention for reporting errors that may have occured in the execution of a ROM routine. If an error is detected during the execution of a ROM tool, then upon exiting the tool call and returning to the application, the 65816 carry flag is set and the accumulator contains an error code describing the error that was detected. TML Pascal provides a mechanism to obtain this information in a Pascal program.

The IsToolError Function

Syntax: IsToolError
Result Type: Boolean

Returns *True* if the last Apple IIGS ROM tool call detected an error during its execution, otherwise it returns *False*. *IsToolError* tests the carry flag of the 65816 processor to determine if an error exists. The function must be called *immediately* after a tool call, before any other operation is performed that might affect the 65816 carry flag. In the case that the tool call is a function and the function appears in an expression, the result of *IsToolError* may be incorrect since evaluation of the expression may have affected the carry flag.

The ToolErrorNum Variable

Syntax: ToolErrorNum
Type: Integer

ToolErrorNum contains the error code returned by the last call to an Apple IIGS ROM tool. A non-zero value indicates an error. The compiler generates code which stores the value of the accumulator into the variable *ToolErrorNum* immediately after the tool call returns, before any other operation is performed that might destroy the value.

Example usage of IsToolError and ToolErrorNum:

```
LoadTools(ToolRec);                { An Apple IIGS ROM Tool call }
if IsToolError then begin
  tmpToolErrorNum := ToolErrorNum;
  WriteIn('Error occurred in LoadTools: ',tmpToolErrorNum);
  end;
```

Note that *ToolErrorNum* was saved to a temporary variable before calling the standard procedure *WriteIn.* This is necessary since the implementation of *WriteIn* calls Apple IIGS ROM tools that would corrupt the value of *ToolErrorNum* with respect to the *LoadTools* call.

Syntax Errors

Error in simple type.
Identifier expected.
'PROGRAM' expected.
')' expected.
':' expected.
Unexpected symbol.
Error in parameter list.
'OF' expected.
'(' expected.
Error in type.
'[' expected.
']' expected.
'END' expected.
';' expected.
Integer constant expected.
'=' expected.
'BEGIN' expected.
Error in declaration part.
Error in field list.
',' expected.
'..' expected.
'.' expected.
'INTERFACE' expected.
'IMPLEMENTATION' expected.
Error in constant.
':=' expected.
'THEN' expected.
'UNTIL' expected.
'DO' expected.
'TO' or 'DOWNTO' expected.
Error in factor.
Error in variable.

Semantic Errors

Duplicate identifier.
Low bound exceeds highbound.
Identifier is not of appropriate class.
Identifier not declared.
Incompatible subrange types.
File not allowed here.
Type must not be real.
Tagfield type must be scalar or subrange.
Index type must not be real.
Index type must be scalar or subrange.
Base type must not be real.
Base type must be scalar or subrange.
Error in type of standard subprogram parameter.
Unsatisfied forward reference.
Repitition of parameter list is not identical to previous declaration.
File value parameter not allowed.
Missing result type in function declaration.

Appendix A

Compiler Error Messages and IOResult Codes

TML Pascal Compiler Errors

Error Reporting

Whenever the Compiler detects an error in the Pascal source an error message is generated showing the source line in which the error was detected with line number, where in the source line the error was detected, and a sequence of one or more error descriptions. Note that the Compiler displays where the error was detected, and not necessarily where the error occurred, although the two are usually coincident. The following are example error messages.

```
   123 Type  NewArr = array[1,10] of UndeclTyp;
                                ^1      ^2
(1)  '..' expected.
(2)  Identifier not declared.

  1055 MyStr := 'Hello World;
                ^1
(1)  String constant must not exceed source line.
```

Errors for a single line are accumulated (up to a maximum of 10) and reported after the line has been successfully scanned. The position at which each error was detected is indicated by the ^ character followed by a digit. The digit indicates which of the subsequent numbered error messages applies to the error.

Error Messages

The following is a complete list of the error messages generated by the TML Pascal compiler. In some messages there is the special character '^' which is substituted by the compiler with an identifier, label, or some other value to help make the error message as meaningful as possible.

Lexical Errors

String constant must not exceed source line.
Error in numeric literal.
Illegal character in input.
Source line exceeds 255 characters.
End of input encountered before end of program.
End of file encountered while reading a comment.

Fixed point formatting allowed only for real types.
Number of parameters does not agree with declaration.
Actual parameter may not be PACKED for VAR formal parameter.
Operands are not assignment compatible.
Tests on equality allowed only.
Strict inclusion not allowed.
File comparison not allowed.
Illegal type of operand(s).
Type of operand must be Boolean.
Set element type must be scalar or subrange.
Set element types not compatible.
Type of variable is not array.
Index type is not compatible with declaration.
Type of variable is not record.
Type of variable must be pointer.
Illegal parameter substitution.
Illegal type of loop control variable.
Illegal type of expression.
Assignment of files not allowed.
Label type incompatible with selecting expression.
Subrange bounds must be scalar.
No such field in this record.
Actual parameter must be a variable.
Control variable must not be declared on intermediate level.
Multidefined case label.
Again forward declared.
Multidefined label.
Multideclared label.
Undeclared label.
Error in base set.
Control variable must not be formal.
Assignment to control variable is not allowed.
Forward referenced type "^" not completed in previous block.
Forward declared subprogram "^" not completed in previous block.
Label ^ was declared but not defined in previous block.
Size of string must be between 1 and 255.
@ is not allowed for expressions or INLINE subprograms.
Type cast to a different size is not allowed.
Too many nested scopes of identifiers.
Too many nested procedures and/or functions.
Too many errors in this source line.
Index expression out of bounds.
Implementation restriction.

Unit Errors

Unit "^" requires unit "^".
Repitition of unit not allowed.
Dependant unit "^" has been recompiled since the last compilation of "^".
Unable to open USES file: ^.

Nested include directives are not allowed.
Unable to open INCLUDE file: ^.

ProDOS16 Error Codes

This section lists the possible result codes of the standard function *IOResult* which reports the success of an I/O operation. The codes correspond to those returned by ProDOS16, except for result codes -1, -2, and -3, which are generated by the TML Pascal runtime routines for Pascal specific errors. The ProDOS16 error codes are provided here for reference, for full documentation regarding these error codes consult Apple Computer's *ProDOS/16 Reference* manual.

General Errors

0 No error.
1 Invalid call number.
5 Call pointer out of range.
6 ProDOS is busy.

Device Call Errors

16 Device not found.
17 Invalid device reference number.
37 Interrupt vector table full.
39 I/O error.
40 No device connected.
43 Write protected.
46 Disk switched.

File Call Errors

64 Invalid pathname syntax.
66 FCB table full.
67 Invalid file reference number.
68 Path not found.
69 Volume directory not found.
70 File not found.
71 Duplicate pathname.
72 Volume full.
73 Volume directory full.
74 Version error (incompatible file format).
75 Unsupported (or incorrect) storage type.
76 End of file encountered.
77 Position out of range.
78 Access not allowed.
79 File is open.
81 Directory structure damaged.
82 Unsupported volume type.
83 Parameter out of range.
85 VCB table full.
87 Duplicate volume.
88 Not a block device.

89 Invalid level.
90 Block number out of range.
91 Illegal pathname change.
92 Not an executable system file.

TML Pascal Specific Errors

-1 Textfile is not open for reading.
-2 Textfile is not open for writing.
-3 Numeric string conversion error in textfile.

Appendix B
Compiler Directives

TML Pascal provides for several directives (or options) which affect the operation of the compiler and/or the code generated by the compiler. These compiler directives are written within the Pascal comment delimiters {...} or (*...*). A directive always begins with the symbol '$' and must appear immediately inside the opening comment delimiter and is followed by a letter (case insensitive) which designates the particular directive.

There are two types of directives: a switch directive and a parameter directive. A switch directive turns on or off a particular compiler feature by specifying'+' or '-' ,respectively, immediately after the directive letter. A parameter directive has one or more string arguments such as filenames or segment names. A string argument is terminated by a blank, an asterisk, or a right brace. If a string argument must contain one of these characters, then the string should be enclosed in single quotes.

Examples of compiler direcitves:

```
(*$A+ *)

{$I 'Filename with blanks' }

{$CSeg NewSeg }
```

The following sections describe each of the compiler directives available in TML Pascal.

Write Source to .Asm File

{$A+} or {$A-}

Default: {$A-}

Turn on (+) or off (-) the writing of Pascal source code lines as assembler comments to the .Asm file. This option only has affect if the compiler has been invoked to generate 65816 assembler source output rather than object code. The option is very useful when attempting to read the 65816 code generated by the compiler.

Set Code Segment

{$CSeg *segname* }

Default: {$CSeg main }

The *CSeg* option directs the compiler as to which code segment all subsequent subprograms should be allocated. The default code segment has the special reserved name *main*, for other code segment names, any string of characters is allowable so long as it does not contain a space. See Appendix C for more information regarding the use of code segments.

DefProc Subprogram

{$DefProc }

Default: *DefProc not active*

The DefProc directive directs TML Pascal to generate an alternate form of subprogram entry and exit code which is compatible with the Apple IIGS ROM tools. Often times, it is necessary to pass the address of a procedure of function in your program to an Apple IIGS ROM tool so that it may be called directly from ROM by a tool routine. For example, the TaskMaster routine in the Window Manager toolset, must call a *DrawContent* procedure in your program in order to have the content region of a window drawn or redrawn.

These types of procedures and functions are called *Definition Procedures* or *DefProcs*. The subprogram calling conventions required for definition procedures is different than the normal conventions used by TML Pascal. Thus, it is necessary to inform the compiler if subprogram is to be called directly by an Apple IIGS Tool.

Because this option changes the calling conventions of a subprogram, the compiler imposes a special restriction on its usage. This option can not be switched on and then off like most other options, but instead MUST appear before every subprogram which is a definition procedure. Consider the following program fragment:

```
{$DefProc }
Procedure DrawContent;
begin
   ...
end;

{$DefProc }
Procedure DrawInfoBar;
begin
   ...
end;
```

Note that it is still possible to call a procedure or function which has been designated as a Definition Procedure from within your own program since TML Pascal keeps track of how each procedure or function is defined.

Desk Accessory

{$DeskAcc *period eventMask menuName* }

The *DeskAcc* directive is used to inform the compiler that a program is actually a *Desk Accessory* rather than a ProDOS16 application. The structure of a desk accessory program is somewhat different than an application. In particular, TML Pascal must generate a special header which contains the *period* in 60ths of a second in which the desk accessory needs periodic servicing, an *event mask* which describes what kinds of events the desk accessory must act on, and the *name* for the desk accessory that should appear in an application's *Apple Menu.*

For complete information about writing desk accessories in TML Pascal see *"Writing Desk Accessories"* in the *TML Pascal User's Guide.*

Because the compiler must generate special code for desk accessories before any code in a program, the option MUST appear before the keyword **program** in your source code for it to have any affect. Consider the following source code fragment:

```
{$DeskAcc 60 -1 TMLClock }
program TMLClock;
begin
   ...
end.
```

Set Data Segment

{$DSeg *segname* }

Default: {$DSeg ~global }

The *DSeg* option directs the compiler as to which data segment all subsequent global variable declarations should be allocated. The default data segment has the special reserved name *~global*, for other data segment names, any string of characters is allowable so long as it does not contain a space, although conventions usually have the name begin with the tilde (~) character. Remeber that the *~global* data segment is the special segment in which the compiler uses the more efficient absolute addressing rather than absolute long addressing. See Appendix C for more information regarding the use of data segments.

Include File

{$I *filename* }

This directive causes the compiler to temporarily suspend compiling code from the the current input file and begin compiling from the source file specified by *filename.* When the compiler reaches the end of this file it resumes compilation from the previous file at the line after the directive. Include file directives are not allowed to appear in files which are include files themselves. That is, nesting of include files is not allowed.

Long Globals

{$LongGlobals+ } or {$LongGlobals- }

Default: {$LongGlobals- }

This option directs the Compiler to either turn on (+) or off (-) the generation of absolute long addresses for global varaibles in the *~global* data segment. Normally, the compiler generates code which sets the 65816 Data Bank register to the memory bank containing the global variables allocated in the *~global* data segment. However, there are several occasions where a program can not rely on this assumption. Two of these are Desk Accessories and subprograms which are definition procedures. In order to guarantee that the compiler generates code which correctly addresses a program's global variables in the *~global* data segment under the conditions stated above this option should be turned on, thus forcing absolute long addressing for all global variables. See Appendix C for more information about addressing global variables.

Stacksize

{$StackSize *numbytes* }

Default: {$StackSize 8096 }

The *StackSize* directive is used to direct TML Pascal as to how much space (in bytes of memory) should be allocated for the application's runtime stack. The runtime stack is used to store the return addresses of subprogram calls made during execution of a program and for a subprogram's local variables. Thus, the use of local variables in your program directly affects the size runtime stack your program requires.

The default size is 8K or 8096 bytes. If a program requires more or less storage, then this option should be used, however, at least 1K or 1024 bytes and no more than 40K or 40960 bytes may be requested, however, TML Pascal does not check the value specified in the directive. See Appendix C for more information regarding the runtime stack.

Note that this option MUST appear before the keyword **program** in your source code for it to have any affect. Consider the following source code fragment:

```
{StackSize 10240 }
program myProg;
begin
   ...
end.
```

External Referenced Variable

{$XrefVar+} or {$XrefVar-}

Default: {$XrefVar-}

The *External Referenced Variable* directive informs TML Pascal that subsequent global variable declarations should not have storage allocated. Rather, the global variable declaration is treated as an external reference to a global variable declared elsewhere.

Typically, this directive is used for Pascal to access global storage declared in 65816 assembly language. However, it may be used with any language compatible with TML Pascal linking conventions.

Consider the following source code fragment:

```
var  GlobVar1: integer;

     {$XrefVar+ }
     GlobVar2: integer;
     {$XrefVar- }

     GlobVar3: integer;
```

Appendix C
Inside TML Pascal

TML Pascal Memory Model

The environment in which an Apple IIGS application runs may be divided into 4 basic components: the *Application Code,* the *Application Globals,* the *Runtime Stack,* and the *Application Heap.* All of these components of an application coexist in the Apple IIGS's RAM memory. Memory in the Apple IIGS is partitioned into 64K byte *banks* which are managed by the Apple IIGS Memory Manager. A standard Apple IIGS comes with 4 banks of 64K byte RAM memory numbered $00, $01, $E0, and $E1. RAM expansion cards can be added to the Apple IIGS beginning at bank $02 and may extend to bank $7F, other bank numbers are reserved or not available.

For a thorough introduction to the architecture to the Apple IIGS see Apple Computer's *Technical Introduction to the Apple IIGS.*

The Application Code

An Apple IIGS application may consist of one or more *code segments.* Small programs are usually made up of only a single code segment, but larger programs are divided into several code segments because the Apple IIGS limits the size of an individual code segment to 64K bytes. The reason for the size restriction is that a code segment must not cross the boundries of a *bank* of memory.

TML Pascal generates a separate *code module* for each Pascal procedure and function declared in a program. Each of these code modules is associated with a *load segment name* which is used to organize separate code segments together by the linker. The default segment name is *main.* When an applicaton has grown large enough to require more than one code segment, the *{$CSeg segname }* directive is used to change the segment name assigned to subsequent code modules. The compiler can be restored to use the default code segment name by specifying *{$CSeg main }.*

The Application Globals

TML Pascal allocates storage for global variables in *data segments.* By default, the data segments are given the *load segment name* "~global". The Linker uses the load segment names associated with each data segment to group them together into *load segments.* Programs are usually made up of only a single data segment, but programs which require a large amount of global stroage are divided into several data segments because the Apple IIGS limits the size of an individual data segment to 64K bytes. The reason for the size restriction is that a data segment must not cross the boundries of a *bank* of memory.

When an application requires a large amount of global storage, it should use the *{$DSeg segname }* directive to instruct the compiler to allocate subsequent global variable declarations into a new data segment. The compiler can be restored to the default data segment by specifying *{$DSeg ~global }.*

During the execution of program initialization code generated by TML Pascal, the 65816 *Data Bank Register* is set to point to the bank of memory which contains the global variables declared in the *~global* data segment. Because of this, references to global variables in the *~global* data segment can use the *Absolute Addressing Mode.* Global variables in all other data segments are addressed using the less efficient *Absolute Long Addressing Mode.*

In some cases, it is necessary to force the compiler to use *Absolute Long Addressing* for global variables regardless of which data segment they are declared in. These two cases are *Definition Procedures* and *Desk Accessories.* In the case of Definition procedures, the procedure or function is called directly by an Apple IIGS ROM tool. In this situation, it can not be guaranteed that the 65816 Data Bank Register still references the memory bank containing the *~global* data segment since the ROM tools may have temporarily changed it. Thus, absolute long addressing must always be used since it does not rely upon the data bank register. To force TML Pascal to use absolute long addressing for a definition procedure, use the *{$LongGlobals+ }* directive. After the code for a definition procedure merely return the compiler to its normal state by specifying *{$LongGlobals- }.*

In addition to definition procedures, desk accessories require absolute long addressing. Desk accessories execute within the environment of other applications. Thus, there is no initialization code that sets the data bank register to the memory bank containing the desk accessory's *~global* data segment. Desk accessories should ALWAYS specify *{$LongGlobals+ }* at the beginning of the program.

The Runtime Stack

The runtime stack is a special block of memory that the application uses to maintain the return addresses of procedures and functions, and to store parameters and local variables. During the execution of application initialization code, a block of memory is allocated in *bank $00* of the Apple IIGS. The block is allocated in bank $00 because this is the only bank of memory in which the 65816 *Stack Register* is able to operate.

By default, TML Pascal allocates a stack containing 8096 bytes (8K) for an application. If an application requires additional or less stack space, then you should specify the *{$StackSize numbytes }* directive in order to change the amount of space allocated for the stack.

The *{$StackSize numbytes }* directive must appear before the keyword **program** for it to have any affect. For example, the following code fragment would cause a 10K stack to be allocated.

```
{$StackSize 10240 }
program MyApp;
...
```

Desk Accessories do not have initialization code which allocates and initializes a runtime stack since they run within the environment of other applications. Thus, when writing an application be sure to leave a reasonable amount of stack space for use by desk accessories, and of course when writing a desk accessory be sure to use as little stack space as possible (Remember the default, and typical, runtime stack is only 8K bytes).

WARNING

Neither the Apple IIGS nor TML Pascal has any way detect the amount of stack space actually used by an application. If insufficient space has been reserved for the runtime stack, then execution of the application will destroy the contents of memory.

The Application Heap

The *application heap* is the memory still available after the application's code, global data, and runtime stack have been allocated. This memory is available to an application via the Memory Manager routines defined in the GSIntf.Pas unit provided with TML Pascal. Memory may also be allocated and deallocated in the application heap using TML Pascal's *New* and *Dispose* procedures.

Nearly every application will allocate at least one memory block in bank $00 for initializing the Apple IIGS tools used in the application. Many of the Apple IIGS toolsets require one or more *pages* allocated in bank $00 to function. These pages are sometimes called *zero pages*.

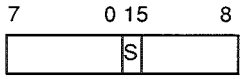
Data Representation

This section shows how each of the Pascal data types is represented in memory. Note that the 65816 stores bytes of word size data in memory backwards from its representation in the accumulator. That is, the least significant bits are in low memory while the most significant bits are in high memory. Consider the following Pascal declaration:

```
type  trick = packed record
                 case integer of
                    0: (int: integer);
                    1: (highbyte: 0..255;
                        lowbyte:  0..255);
              end;
```

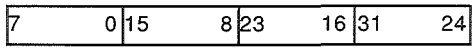
This record type does not give the results one might expect. On the 65816, referencing *highbyte* would actually return the low order byte of the integer *int,* and not the high order byte. The following paragraphs should clarify the storage layout of the Pascal data types.

Integer A signed two's complement integer in the range -32,768 to 32,767 requiring 2 bytes of storage. Bit 15 is the sign bit.



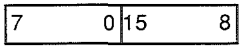
(Note: S denotes the sign bit)

LongInt A singed two's complement integer in the range -2,147,483,648 to 2,147,483,647 requiring 4 bytes of storage. Bit 31 is the sign bit.

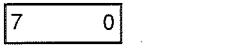


Boolean An enumerated type of *(False,True)* requiring one byte of storage, where the boolean value is in bit 0.

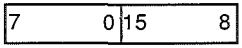
Char An enumerated type of the ASCII character set having 256 values. The character value requires two bytes of storage where the value is in the lower order byte (bits 7-0).



Enumeration An unsigned byte or word size integer. If the enumeration type consists of 128 or fewer enumeration constants then the a value of the type occupies a single byte of storage otherwise it occupies a word of storage.

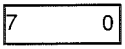


<= 128 enumerations

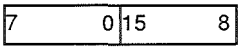


> 128 enumerations

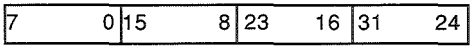
Subrange A signed byte, word, or longword. If the range is within -128..127 a byte is used to represent the subrange, if the range is within -32768..32767 a word is used, otherwise a longword is used to represent the subrange. For example, the subrange types SignedByte and Byte from the QDIntf.Pas interface are represented as a byte and a word respectively.



-128..127

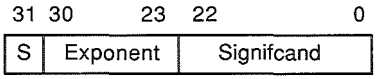


-32768..32767

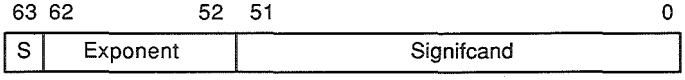


all others

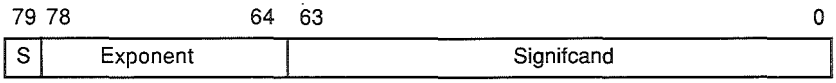
Single A 32-bit real number represented in IEEE standard single precision format implemented as the SANE Single type.



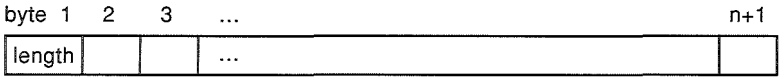
Double A 64-bit real number represented in IEEE standard double precision format implemented as the SANE Double type.



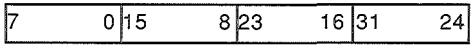
Real / Extended An 80-bit real number represented in IEEE standard extended format. Both are implemented as the SANE Extended type.



String[n] An n+1 byte size *Pascal String* consisting of a byte containing the current string length (not counting the length byte itself) followed by bytes containing ASCII characters.



Pointers A 24-bit physical memory address occupying 4 bytes of storage. Only 3 bytes are needed to store the 24-bit address value so bits 31-24 are always zero. The *nil* pointer is represented as the 32-bit value zero.



Sets A sequence of bytes up to a maximum of 32 bytes or 256 bits representing the powerset of the base-type. The number of bytes used is the minimum number required to represent the powerset. An ordinal value of the base-type is represented by a single bit whose bit number is the ordinal value. If an ordinal value is a member of a set then its bit is set to 1 otherwise it is set to 0. If the ordinal values of the base-type are in the range 0..15 then two bytes are used to represent the set. If the ordinal values of the base-type are in the range 0..31 then four bytes are used to represent the set, etc.

Files A 22 byte data structure used internally by the TML Pascal runtime routines. In addition to the file variable itself, an open file associated with an external disk file has a 512 byte *access buffer* allocated in the application heap for use by ProDOS16; additionally, a textfile has a 256 byte line buffer.

Arrays / Records Components of unpacked arrays and records are allocated contiguously as defined above. Arrays are stored in row order. That is, the last index varies fastest. Record components are allocated as they textually appear in their declaration.

The implementation of TML Pascal performs data packing to byte boundries only, no bit packing is available. A data type is represented as a byte in a packed structure if and only if the number of bits required to store all values of the type is less than or exactly eight bits. For example, the standard type Char or the type Byte (declared in QDIntf.Pas) require exactly 8 bits to represent all of their values, therefore in a packed structure Char and Byte are allocated a byte of storage, but otherwise require a word of storage.

Calling Conventions

This section outlines the TML Pascal compiler conventions for calling procedures and functions, Definition procedures, parameter passing, and function result values.

Calling a Subprogram

TML Pascal uses a stack-based parameter passing convention for calling subprograms. Before calling a procedure or function, the parameters are pushed onto the stack in the same order as their declaration. If a function is being called, the storage for the function result is allocated on the stack before pushing any parameters. If the procedure or function is a normal subprogram (not a Definition Procedure), then after returning from a subprogram call, all paramters are removed from the stack by the calling environment. In the case of a Definition Procedure the parameters are removed by the procedure or function itself before returning to the calling environment, but leaves the function result (if any) on the stack.

The following is skeleton code for a normal procedure call.

```
lda    pppp                    ; Push first parameter
pha
...
lda    pppp                    ; Push last parameter
pha
jsl    >Aproc                  ; Call the procedure
tsc                            ; Remove parameters from stack
clc
adc    #nn
tcs
```

The following is skeleton code for a normal function call.

```
pha                            ; Reserve space for result value
lda    pppp                    ; Push first parameter
pha
...
lda    pppp                    ; Push last parameter
pha
jsl    >Afunc                  ; Call the function
tsc                            ; Remove parameters from stack
clc
adc    #nn
tcs
pla                            ; Remove function result from stack
                               ;  and place in accumulator
```

Subprograms are always called in full 65816 native mode (ie. 16-bit accumulator and index registers). There fore, if the processor is not in full native mode before the call it is forced to full native mode before the call is made. For example, assume that the accumulator is in 8-bit mode and the index registers are in 16-bit mode, then the following code is generated.

```
                               ; Accumlator currently in 8-bit mode
rep    #$20                    ; Change accumlator to 16-bit mode
LONGA  ON
jsl    >ASubprog
```

If the subprogram being called is declared at a level other than the global level (not in the program or unit block), then a *static link* is pushed on the stack after all the parameters have been pushed. The static link serves as a mechanism to address local variables in nested stackframes. Because of this static link, the address of a nested subrogram should never be passed to an Apple IIGS ROM tool as a definition procedure since this is not the calling convention it expects.

Variable Parameters

Actual variable parameters (**var** parameters) are always passed by reference to the formal parameter, that is, as a pointer that points to the storage occupied by the actual parameter. The pointer is passed as a 32-bit value. The high order word is pushed first followed by the low order word.

Consider the following example for a passing the global variable `GlobVar` as a variable parameter using the absolue addressing mode.

```
pea     GlobVar|-16             ; Push high order word first
pea     GlobVar                 ; Push low order word second
```

Value Parameters

Actual value parameters are passed with their value on the stack or by reference depending on the size of the value. If the size of the value parameter occupies 4 bytes or less, then its value is passed on the stack and is the formal paramter. If the size of the value parameter occupies more than 4 bytes, then a 32-bit pointer to the value is passed on the stack. The called procedure or function then copies the value into local storage for the formal parameter so that changing the value of the formal parameter does not affect the value of the actual parameter.

Static Parameters

Actual static parameters are passed on the stack exactly like actual value parameters. The difference between static and value parameters is that if the size of the actual parameter is greater than 4 bytes the called procedure or function DOES NOT copy the value into local storage for the formal parameter. Thus, it is illegal to give the formal static parameter a new value since it would change the value of the actual parameter.

Static parameters have been introduced in TML Pascal to conserve the amount of space used by the runtime stack for storing formal parameters as well as improve the execution speed of programs since no unnecessary copying is performed.

IMPLEMENTATION NOTE

TML Pascal does not check that a new value is never assigned into a static parameter. It is the responsibility of the programmer to ensure that static parameters are used correctly.

Function Results

Storage for function results is reseved on the stack by the calling subprogram before any parameters are pushed onto the stack. If the function result is of type *Integer, LongInt, Char, Boolean,* or any subrange, enumerated or pointer type, or the real type *Single*, 2 or 4 bytes of storage are allocated. If the result type requires only 1 byte of storage, 2 bytes are allocated and the low memory address byte contains the value.

If the result type is of type *Double, Comp, Extended*, or any array, string, or record type, then the calling subprogram allocates temporary space within its stackframe for the result value, and pushes a 4 byte pointer to the temporary storage. The calling subprogram removes the pointer from the stack when the function returns, and the temporary storage is deallocated when no references to the value exist.

Entry/Exit Code (Normal Subprograms)

Each Pascal procedure and function begins and ends with standard entry and exit code which creates and removes its activation.

The standard entry code is as follows:

```
phd                                 ; Save previous frame pointer
tsc
sec
sbc    #xx
tcd                                 ; Establish new frame pointer
clc
adc    #yy
tcs                                 ; Allocate local storage
```

First, the direct page register from the previous activation is saved. The direct page register is used as a *frame pointer* for an activation. The saved frame pointer is called the *dynamic link*, and is required to restore the state of the previous activation.

After saving the previous frame pointer, *xx* bytes are subtracted from the current stack pointer to establish the frame pointer for this activation. *xx* is computed so that the first *word* of storage in the stack activation (ie. the function result or first parameter) is at direct page offset 254. Choosing this offset allows all parameters and as many local variables as possible to be addressed using the very efficient direct page addressing mode.

Once the frame pointer for the activation is established, *yy* bytes are added to that value to allocate the storage needed for local variables, value parameters copied local, and compiler temporaries.

Note that no registers are saved, and it is assumed that the processor is in full native mode.

The standard exit code is as follows:

```
tdc
clc
adc    #xx
tcs                                 ; Deallocate local storage
pld                                 ; Restore previous frame pointer
rtl
```

Local storage is first removed by adding the value *xx* to the frame pointer. Then the frame pointer from the previous activation is restored and the RTL instrution is executed to return to the calling subprogram.

Entry/Exit Code (DefProcs)

Definition procedures (and functions as well) are special subprograms which are typically called by the Apple IIGS ROM. These subprograms have entry code which is identical to normal subprograms, but whose exit code removes the subprogram's parameters, but not any function result value in order to conform with the ROM calling conventions.

The following is the exit code for definition procedures:

```
tdc
clc
adc    #xx
tcs                                ; Deallocate local storage
pld                                ; Restore previous frame pointer

lda    2,S
sta    mm,S
lda    1,S                         ; Move the return address down over
sta    mm-1,S                      ;  the parameters
tsc
clc
adc    #mm-2                       ; Deallocate the parameters
tcs
rtl
```

The first part of the exit code is identical to that of normal subprograms – local storage is deallocated and the previous frame pointer is restored. After that, however, the parameters are "removed" from the stack by moving the return address down overtop of the first parameter(s) and then positioning the stack pointer to the new location of the return address. And finally, the RTL is executed to return to the calling subprogram (typically in the Apple IIGS ROM.

Note that a definition procedure having no parameters has exit code exactly like normal subprograms since is no need to move the return address over the parameters.

Linking with Assembly Code

In TML Pascal it is possible to integrate code developed with Apple Computer's *Apple Programmer's Workshop (APW)* with Pascal code. This section describes the conventions that should be followed when developing assembly code to be integrated with TML Pascal.

Subprograms

Procedures and functions written in assembly language must be declared as **external** in a Pascal program or unit. For example:

```
function MyAsmFunc(i: integer): boolean;    external;
```

In the corresponding assembly language source file the procedure or function name must appear as a label with the START directive and the argument to the START directive is the procedure's or function's code segment name. For example:

```
MyAsmFunc     START    main
              ...
              END
```

In this example, the function will be linked in the code segment having the name *main*. Any name can be used here, but it should generally match the name of a code segment being used in the Pascal program. Recall that *main* is the default code segment name.

It is up to the programmer to ensure that the implementation of the assembly language routine abides by the TML Pascal calling conventions.

In addtion, to Pascal programs calling assembly language routines, it is possible for assembly language routines to call Pascal procedures and functions. When calling a Pascal procedure or function it is up to the programmer to ensure that parameters are passed to the Pascal routine as it expects them.

Identifiers in TML Pascal are significant to 255 characters, so any length name for the procedure or function will be correctly recognized.

Variables

Global variables declared in Pascal programs and units may be accessed in assembly language routines. To reference a Pascal global variable merely use the identifier's name in your assembly language routine. The variable reference is resolved by the linker to reference the Pascal declared variable.

To address the global variable, 65816 absolute addressing may be used if the Pascal global is declared in the data segment *~global*, otherwise absolute long addressing should be used.

Global variables declared in assembly language may also be accessed by Pascal procedures and functions by using the compiler's *{$XRefVar}* directive when declaring a global variable. For example,

```
var    {$XRefVar+}
       AsmGlobal: integer;
       {$XRefVar-}
```

In assembly language, the variable *AsmGlobal* should appear in a public data segment where the label appears with the assembly language ENTRY directive. Thus,

```
Globals      DATA   ~global
AsmGlobal    ENTRY
             ds     2
             END
```

Processor Mode

All Pascal procedures and functions rely upon and force the 65816 processor to be in full 16-bit native mode before calling another procedure or function, and upon returning from a procedure or function. In addtion, the processor should not have the decimal mode flag set.

Register Saving Conventions

An assembly language routine may assume that the contents of the A, X and Y registers have no meaning to the calling environment and thus need not be saved and restored upon subprogram entry and exit. However, the Direct Page register must be saved and restored upon subprogram entry and exit. In addition, if the assembly language routine changes the 65816 Data Bank Register it MUST restore it to its origional value before returning to the Pascal program. If the Data Bank Register is corrupted by the assembly language routine, the Pascal program will not correctly address global variables.

Appendix D

Comparing TML Pascal with ANS Pascal

This appendix compares TML Pascal with the American National Standard (ANS) Pascal as defined by ANSI/IEEE770X3.97-1983 in the book *American National Standard Pascal Computer Programming Language* (ISBN 0-471-88944-X, published by The Institute of Electrical and Electronics Engineers in New York).

Exceptions to ANS Pascal Requirements

- In ANS Pascal, an identifer may be of any length and all characters are significant. In TML Pascal, an identifier may be of any length, buy only the first 255 characters are significant. Note that most editors restrict line lengths to at least this length.

- In ANS Pascal, the @ symbol is an alternative for the ^ symbol. In TML Pascal, the @ symbol is an operator.

- In ANS Pascal, a comment may begin with { and end with *), or begin with (* and end with }. In TML Pascal, comments must begin and end with the same set of symbols.

- In ANS Pascal, a file variable has an associated buffer variable, which is referenced by writing the ^ symbol after the file variable. In TML Pascal, a file variable does not have an associated buffer variable, and writing the ^ symbol after a file variable is an error.

- In ANS Pascal, the statement part of a function must contain at least one assignment to the function identifier. In TML Pascal, this requirement is not enforced.

- In ANS Pascal, a field that is the selector of a variant part may not be an actual variable parameter. In TML Pascal, this requirement is not enforced.

- In ANS Pascal, procedures and functions allow procedural and functional parameters; these parameters are not implemented in TML Pascal.

- In ANS Pascal, the standard procedures *Reset* and *Rewrite* take only one parameter, a file variable. In TML Pascal, *Reset* and *Rewrite* allow an optional second parameter, a string type expression, which names an external file.

- ANS Pascal defines the standard procedures *Get* and *Put*, which are used to read from and write to files. These procedures are not defined in TML Pascal.

- In ANS Pascal, the standard procedures *Read* and *Write* are defined in terms of *Get* and *Put* and references to buffer variables. In TML Pascal, *Read* and *Write* function as in ANS Pascal, but they are automatic operations.

- In ANS Pascal, the syntax *New(p,cl,...,cn)* creates a dynamic variable with a specific active variant. In TML Pascal, this variation of the *New* procedure is not allowed.

- In ANS Pascal, the syntax *Dispose(q,kl,...km)* removes a dynamic variable with a specific active variant. In TML Pascal, this variation of the *Dispose* procedure is not allowed.

- ANS Pascal defines the standard procedures *Pack* and *Unpack*, which are used to "pack" and "unpack" packed variables. These procedures are not defined in TML Pascal.

- In ANS Pascal, a **goto** statement within a block may refer to a label in an enclosing block. In TML Pascal, this is an error.

- In ANS Pascal, it is an error if the value of the selector in a **case** statement is not equal to any of the case consonants. In TML Pascal, this in not an error; instead the **case** statement is ignored unless it contains an **otherwise** clause.

- In ANS Pascal, a *Read* from a text file with a char type variable assigns a blank to the variable if *Eoln* was *True* before the *Read*. In TML Pascal, a carriage return character (*Chr(13)*) is assigned to the variable in this situation.

- In ANS Pascal, a *Read* from a text file with an integer type or a real type variable ceases as soon as the next character in the file is not part of a signed-integer or a signed-number. In TML Pascal, reading ceases when the next character in the file is a blank, a tab, or an end of line character.

- In ANS Pascal, a *Write* to a text file with a packed string type value causes the string to be truncated if the specified field width is less than the length of the string. In TML Pascal, the string is always written in full, even if it is longer than the specified field width.

Extensions to ANS Pascal

The following TML Pascal features are extensions to Pascal as specified by ANSI/IEEE770X.97-1983.

- The following are reserved words in TML Pascal:

body	**interface**	**string**	**uses**
implemention	**otherwise**	**unit**	

- An identifier may contain underscore characters after the first character.

- Integer constants may be written in hexadecimal notation. Such constants are prefixed by a $.

- String constants are compatible with the TML Pascal string types.

- Label, constant, type, variable, procedure, and function declarations may occur any number of times in any order in a block.

- A signed constant identifier may denote a value of type *Integer*, *LongInt*, or *Extended*.

- TML Pascal implements the additional integer type *LongInt*, and the additional real types *Single, Double, Comp*, and *Extended*.

- Arithmetic operations on *Integer* operands produce *Integer* results. Arithmetic on *LongInt* operands or mixed *Integer* and *LongInt* operands produce *LongInt* results. *LongInt* values are compatible with the *Integer* type provided they are in the *Integer* range.

- Arithmetic operations on real type operands or mixed integer type and real type operands produce *Extended* values. *Extended* values are compatible with the *Single, Real, Double*, and *Comp* types, provided they are in the range of those types.

- TML Pascal implements string types, which differ from the packed string types defined by ANS Pascal in that they include a dynamic length attribute that may vary during execution.

- The type compatibility rules are extended to make char types and packed string types compatible with string types.

- String type variables can be indexed as arrays to access individual characters in a string.

- The type of a variable reference can be changed to another type through a variable type cast.

- The relational operators can be used to compare strings.

- TML Pascal implements the @ operator, which is used for obtaining the address of a variable or a procedure or function.

- The type of an expression can be changed to another type through a value type cast.

- The **case** statement allows an optional **otherwise** part.

- Procedures and functions can be declared as **external** (assembly language subroutines), **inline** (inline machine code), and **tool** (Apple IIGS ROM tools).

- TML Pascal implements *static* formal parameters in addition to *value* and *variable* formal parameters defined in ANS Pascal.

- TML Pascal implements the *UNIV* formal parameter type.

- TML Pascal implements *Units, Unit Specifications,* and *Unit Bodies* to facilitate modular programming and separate compilation.

- TML Pascal implements the following file handling procedures and functions, which are not available in ANS Pascal:

Close	Rename	Seek
Erase	IOResult	FilePos

- String type values may be input and output with the *Read*, *ReadLn*, *Write*, and *WriteLn* standard procedures.

- TML Pascal implements the following standard procedures and functions, which are not found in ANS Pascal:

Exit	Length	Delete	MoveRight	HiWord
Halt	Pos	Insert	FillChar	LoWord
Ord4	Concat	SizeOf	ScanEQ	Inc
Pointer	Copy	MoveLeft	ScanNE	Dec
Cycle	Leave	IsToolError		

Implementation Dependent Features

The effect of using an implementation dependent feature of Pascal, as defined by ANSI/IEEE770X3.97-1983, is unspecified. Programs should not depend on any specific path being taken in cases where an implementation dependent feature is being used. Implementation dependent features include:

- The order of evaluation of index expressions in a variable reference.
- The order of evaluation of expressions in a set constructor.
- The order of evaluation of operands of binary operator.
- The order of evaluation of actual parameters in a function call.
- The order of evaluation of the left and right sides of an assignment.
- The order of evaluation of actual parameters in a procedure statement.
- The effect of reading a text file to which the procedure *Page* was applied during its creation.
- The binding of variables denoted by the program parameters to entities external to the program.

M

N

O

P, Q

R

S

T

U

V

W

4241 Baymeadows Road • Suite 23 • Jacksonville, Florida 32217