TML PASCAL II

Translated into French by B Capslock
Re-Translated into English by Michael A. Stephens

Revision Date: 4 July 2007

INTRODUCTION

Welcome to TML Pascal II, the second generation of the famous Pascal language development system for the Apple IIgs. TML Pascal II implements certain news functions making the programming easier and ensuring a total compatibility with operating system 5.0. In particular, TML Pascal II now allows you to edit and create resources. This version is intended to be used on only one computer; for use on a AppleShare network, you must get a special version of TML Pascal II for network functionality. The TML Pascal II programming language is intended to satisfy the greatest number of programmers on the Apple IIgs. It is based on the Pascal language which has been used for years, with many extensions for programmers familiarized with other versions of Pascal.

TML Pascal II recognises the usage of modules, direct I/O access to disk, and standard subroutines like Moveleft, Fillchar, etc which one finds in the extensions of UCSD Pascal from Apple. And of course, many of the features of TML Pascal for the Macintosh like type casting operations on a bit, the CYCLE and LEAVE declarations, the usage of modules and more still, are all available in TML Pascal II for the Apple IIgs.

TML Pascal II for the Apple IIgs was designed to take full advantage of the new features of the machine. It runs in true 16 bits native mode GS/OS. It is possible to access each routine of the Apple IIgs ToolBox in the same way that you can access Prodos 16, GS/OS, and the resources.

With TML Pascal II, you will be able to develop autonomous applications for GS/OS, NDA and CDA desk accessories, text mode or graphical applications. What’s more, with the development of applications taking advantage of the IIgs ToolBox, TML Pascal II allows you to develop what we call "plain vanilla" or "textbox" applications. This possibility enables you to directly enter programs, starting from the text examples, and to compile them. As for the development environment, TML Pascal II provides a console window of 20 lines and 80 columns in 640 super-high resolution mode.

The user guide and reference manual of TML Pascal II was written to guide you in the usage of TML Pascal II; however, they are not training manuals.

In order to use TML Pascal II, you will need an Apple IIgs with monitor, a 3.5” 800 K floppy drive, and a memory expansion card with at least 512 KB for a total of 768 K read-write memory. For the development of important applications, two 3.5” 800 K floppy drives or a hard disk are recommended. A printer to print out your source listings will also be a welcome addition.

If you have any questions relating to TML Pascal II or Pascal programming, you can contact customer service Monday to Friday between 8:30am and 5:30pm (local time) on 904 636 8592. In regards to more technical problems, we ask you to call our technical service line between 2pm and 5pm on 904 636 0118. Additionally, if you prefer to send us a letter, you can write to us at:

TML SYSTEMS,
Inc 8837-B Goodbys Executive Drive Jacksonville,
Florida 32217

ABOUT THE MANUAL

This manual was designed to help you get going quickly with TML Pascal II and also to be used as reference manual when your level of programming increases. This manual is divided into four parts:

1. The User Guide
2. Programming
3. A Language Reference
4. The Appendices

The user guide introduces you to TML Pascal II and will teach you how to configure your system for use, and how to obtain the maximum out of using TML Pascal II. The purpose of the user guide is to provide you with an overview of the TML Pascal II development environment.

The Programming section introduces you to the basics of TML Pascal II by teaching you how to write each of the five various types of programs using TML Pascal II.

The Language Reference is a complete reference of the possibilities provided by TML Pascal II. Note that this section is a reference and is not a guide for using the Pascal language. If you are not familiar with Pascal language programming, you will need an additional manual to teach you how to use the Pascal language.

The Appendices summarise the error messages and the results of TML Pascal II inputs/outputs, the compilation directives and the functions of the Apple IIgs ToolBox. In addition, a look at the more advanced features of TML Pascal II is given, with a comparison between TML Pascal II and the original TML Pascal.

CHAPTER 1: FIRST CONTACT

Before using TML Pascal II you will have to take precautions to protect your software. As it is not possible to use the TML Pascal II diskette provided by itself (it does not contain GS/OS), it can be sufficient to write protect the disk when you use it to install TML Pascal II on your hard disk. However, it is preferable to make a backup diskette and to arrange the original one in a safe place.

Please remember that the philosophy of TML Systems is to sell quality products at a reasonable price without copy protection. This system only works if you play the game. By Purchasing TML Pascal II, you are licensed to make backups of the software for your personal needs, but any backups made should not be given to or used by anyone else.

To format a new diskette and to make a backup, you can use your usual copy program (allowing for the duplication of the resource files) or better yet, by using the copy functionality within the GS/OS Finder. If you are not accustomed to these tools then consult their respective manuals.

SYSTEM CONFIGURATION

The TML Pascal II distribution diskette contains all the files necessary for the installation of TML Pascal II. A development system on the Apple IIgs using TML Pascal II can be configured in many ways, however, in the following sections we indicate the most common configurations available to you:

- A Single 800k Floppy Drive
- Dual 800k Floppy Drives
- A Hard Disk

To build a TML Pascal II development environment of you will need two different sets of software:

- The TML Pascal II Software
- The Apple IIgs System Software (GS/OS)

A Single 800k Floppy Drive

The minimum configuration is to have only one 800k floppy drive. Although this configuration reduces the space available to you on the diskette, it still provides a complete and powerful development system using TML Pascal II. A development system running on only one 800K floppy contains the minimum of the software referred to above. With this configuration you have approximately 100 KB to develop your applications. However, it will be necessary to do a little ' house cleaning' on your copy of the TML Pascal II diskette by removing, in particular, the SAMPLES files. This will give you more space to save your sources.

Dual 800k Floppy Drives

With two 800k floppy drives it becomes possible to use the second drive for storage of the source code for any applications that you develop. Nevertheless, the system files must reside on the TML Pascal II diskette.

A Hard Disk

A hard disk offers speed as well as the greatest flexibility for your development system. It is recommended that you install both the Apple IIgs System Software diskettes and that of TML Pascal II on the hard disk. However, please ensure that you are using the latest version of the Apple IIgs System Software and not an older version. Version 5.0 of GS/OS is the minimum. Neither the TML Pascal II software, nor any applications developed by it, will function correctly under GS/OS prior to version 5.0.

INTERPRETED vs. COMPILED LANGUAGES

TML Pascal II is a compiled language. In this respect, it differs notably from the interpreted languages such as Applesoft Basic. A programming language consists of a collection of statements, expressions and operators, commonly referred to as syntax or language structure. Whereby the languages written in machine language are generally incomprehensible for humans, they are completely understood by the microprocessor of the computer. In our case, the 65816 microprocessor.

Before a statement written in a high level language can be executed, it must initially be translated into language understood by the computer: machine language. Machine language is made up of a continuation of binary numbers (0's and 1's) which is understood by the computer as being a series of YES-NO states. These YES-NO states characterise the operations that the machine is able to execute. This succession of 0's and 1's is not easily comprehensible by man.

The greatest part of each programming language is made up during the translation of programs into machine language. In an interpreted language, this translation is made when the program is being executed. This method is sometimes called 'just in time' compilation. If a statement of a program is executed 100 times for example, the translation into machine language is executed 100 times also by the interpreter. Generally, interpreted languages are executed more slowly than the compiled languages because the translation occurs during program execution.

On the other hand, a compiled language program is translated into machine language before its execution. Thus, each line of the program is translated only once (at the time of compilation). What's more, compilation allows for any syntax errors to be detected before the program is executed. Of course, the compiler cannot detect any logical errors (for example infinite loops).

Generally, the compiled languages are executed much more quickly than the interpreted languages. Moreover, the compiled languages can be executed independently of any language processor. Thus, compiled TML Pascal II programs can be executed under GS/OS without TML Pascal II being used. On the other hand, Applesoft Basic programs can only be executed if the Basic language is loaded in memory beforehand.

AND NOW?

If you are new to programming, read this manual in its entirety (TML Pascal II) and get a good book that introduces you to the Pascal programming language (for example the

online Pascal introduction found at http://www.taoyue.com/tutorials/pascal/contents.html). If you know Apple IIgs programming with a language other than Pascal, it is enough for you to read chapters 4 to 10 in order to learn the specifics of TML Pascal II and the programs that you can develop in such an environment. If you already know Pascal well, then appendix E will be your main point of reference to learn the various functions working with system tools.

CHAPTER 2: USING TML PASCAL II

LAUNCHING TML PASCAL II

Put a copy of your TML Pascal II diskette in a 3.5” floppy drive and turn on your computer. When the booting has completed, the finder desktop is displayed along with an icon of the TML Pascal II diskette drawn in the right part of the screen. Click on this icon, and then select the OPEN option from the FILE menu (or double-click on the TML Pascal II icon). A window opens: double-click on icon *TMLPascal.II* in order to load TML Pascal II into memory.

THE PROGRAMMING ENVIRONMENT

TML Pascal II is designed to take full advantage of the possibilities of the GS: mouse interface, pull-down menus, multi-window, etc. With such a machine, the programming is simplified: the editor and the compiler function in the same working environment.

The main menu of TML Pascal II only allows the user to edit a source file or resource. In this chapter and the following, we will see how to edit a source file; chapter 4 we will specifically deal with resources files.

EDITING SOURCE FILES

The edit windows of TML Pascal II are tools allowing for the easy creation/modification your source code. TML Pascal II allows you to have as many source files opened simultaneously that the memory available of your system allows. Obviously, the more RAM you have available in your GS, the more space you will have to create programs and to open edit windows.

As you open new source files, TML Pascal II will place them in new edit windows independent of any of the others. But only one edit window can be active at a time. The active window is always the one located in the foreground, all other edit windows will be inactive.

TML Pascal II also uses dialogs to communicate with the user. These dialogs are generally used to inform the programmer when he must provide certain information in order to continue his work (or to correct certain erroneous conditions). The dialogs also use buttons OK/CANCEL or YES/NO to confirm or cancel your decisions.

Using the Menu to Edit Source Files

Choose the OPEN option from the FILE menu. A standard dialog for loading files appears on the screen. Click on the SOURCE TEXT FILE radio button then double-click on file HELLOWORLD in file list. Almost instantaneously, the listing is displayed in a new edit window. You will notice that then the bar of menus includes/understands new options: these are editing tools. These tools will be examined in detail within chapter 5.

Naming Conventions

When you opened the HelloWorld file, you probably noticed the presence of various files with very similar names. These similarities are the result of following the conventions

allocated to each of the various file types. The reason for this is to differentiate the source files, Resource and Application. TML Pascal II uses the following naming convention:

Source files: are to use suffix P at the end of the file names

Resource files: are to use suffix R at the end of the names

Application: TML Pascal II will automatically append suffixes when creating the name of an application

NOTE: TML Pascal II does not automatically create file names complying with these rules: it is up to you to follow these rules when creating new files.

Basic Rules for the Editor
The TML Pascal II Editor is very similar to traditional word processing. Thus the experienced GS user will not be out of place editing his source files with TML Pascal II. You have the following capabilities:

- to cut, copy, paste and delete blocks of text
- search and replacement of text
- indentations (tabulations)
- choice of the font and its size
- printing to an Imagewriter or Laserwriter

In addition to the above, TML Pascal II has specific options:

- to check the syntax of a source file
- to execute a program
- to create an application
- to use certain functions of GS/OS (to rename, copy)
- to indicate any programming errors

Using the HelloWorld source file, familiarise yourself with the various formatting controls. For example, copy-paste a selection of text, change the font and its size in the current edit window, cancel your modifications, etc. Within chapters 3 and 5 we will study in detail the different options available within TML Pascal II.

CHAPTER 3: PROGRAM CREATION

COMPILATION

TML Pascal II has three options for compilation: TO MEMORY, TO DISK and CHECK SYNTAX. These options for compilation can be found in the COMPILE menu.

Compilation TO MEMORY is almost certainly the option you will use the most. This option carries out compilation of the source found in the active edit window. In case of success, compilation continues with the execution of the program compiled in memory.

The compile TO DISK option is similar to the preceding one, but the compiled program is not executed, it is saved to disk. This option is used when your program is finished and tested: this option then produces the final application which is startable from the launcher.

Lastly, CHECK SYNTAX makes it possible to quickly check the syntax of the source found in the active window. There is neither execution of the compiled program nor saving to disk in this case.

When one of the compilation options is selected, a window is displayed indicating the progress of the process. When the 'thermometer' is completely coloured in, compilation is finished.

TESTING SOURCE CODE

The option Check Syntax is the fastest option of compilation: there is no code generated, but instead checks for the correct spelling of key words, statements, functions, etc. However, it does not test the logic of the program. For example, an infinite loop could not be detected by Check Syntax. Open the file *TextBook.p* example and launch the option Check Syntax.

When the 'thermometer' is completely coloured in, the syntax checking is finished. This took only a few seconds, thanks to the performances of the TML Pascal II compiler and with the compactness of the source code tested.

If the test indicates that no error was detected, it does not mean that the program is perfect. The guarantee is that there are no errors in syntax. It is important to carry out a syntax check to detect any errors which would cause a program to 'crash' when it is executed after a *Compile To Memory*.

If an error is detected, compilation is stopped and TML Pascal II displays a window indicating the type of error. In the active edit window, the code responsible for generating the error is displayed in inverse text.

PROGRAM EXECUTION

When you are certain that your program does not contain any syntax errors, it is time to execute it. To achieve this, select the option *Compile To Memory*. During compilation, TML Pascal II generates code which will be executed within the memory of the GS. The programming environment will be closed (windows, menus) and your application launched.

When you leave your program, you will return to TML Pascal II automatically, with the desktop in the state you had left it in (opened windows, sources displayed, etc.).

As it is possible that your source contains logical errors which would cause a crash during execution, TML Pascal II makes it possible to save the desktop right before launching your compiled code. This behaviour is called AUTOSAVE. If this option is selected, TML Pascal II carries out a backup of all the open files modified since last compilation. With this option enabled, you will be unlikely to lose any of your work if the IIgs is locked up by an error during program execution. Option AUTOSAVE is commented on in chapter 5.

CREATE AN APPLICATION

As mentioned previously, compilation to memory is very fast and very practical. However, using this method to execute the program, you are required to run TML Pascal II and compile the source prior to running the program. Fortunately, there is an easier means to launch an application. For that, it is necessary to perform a *Compile To Disk*. This will create a program object that is able to be executed directly from the GS/OS Finder. This means that you will no longer be required to run TML Pascal II in order to launch your application: simply double click on your applications icon from within the GS/OS Finder to launch it.

Compiling to disk is takes slightly longer than the other methods of compilation due to the writing of the compiled code to disk. The name of the compiled program will be the same as that of the source without the *'.p'* suffix.

COMPILING UNITS

In the same manner that TML Pascal II compiles a program, it can compile a Unit. This compilation cannot generate a program object, but it is used to produce code containing the information related to your application. Units are used to split a very bulky program into logical units. A Unit can include constants, types, variables, procedures and functions.

As a Unit cannot be executed, the TML Pascal II compiler acts differently with such a source. If one compiles a Unit to memory, TML Pascal II carries out the compilation then returns control to the editor instead of executing the code. Once the compilation is completed, the Unit's code is saved in memory and can be used later on by a program.

If one compiles a Unit to disk, TML Pascal II does not create an executable program, but instead creates an object file containing the table of symbols used and also any compiled libraries. The code produced will be saved with the suffix *'.p.o'*. If another program wishes to link to this compiled Units code, TML Pascal II expects to find the files with the extension *'.p.o'*. The file name of the compiled Unit is always the source name plus an additional suffix *'.o'*. Thus, the file name for object code that one links to usually ends in *'.p.o'*.

The option to Check Syntax behaves in the same manner as for a normal program. That is, *Check Syntax* will only check that the current Unit does not contain any syntax errors in its statements.

ERROR DETECTION

Until now, we have spoken about performing compilation that occurs without any errors. Let us now take a quick look at how TML Pascal II detects and manages any errors. TML Pascal II consists of three parts: the editor, the compiler and the linker. These three parts function and work together to give the illusion of a single entity. But by including/understanding how these three parts function, you will better understand how TML Pascal II detects any errors within your programs source code.

The editor, of course, is what you use most of the time. It makes it possible to open the windows containing your programs and the majority of the available menu options are selectable via the editor. The compiler is used when you choose one of the three compilation options. It makes it possible to detect syntax errors and to produce the object code. Lastly, the linker is employed only when compiling to disk or memory. It will combine the various programs objects (units) and will allocate the internal storage necessary for your application.

The editor only will indicate errors occurring when working in the editing environment. For example, it will indicate an error if you try to save a file on a protected diskette, or if you try to open a file without having sufficient memory. The compiler indicates the errors by using an error code. Finally, the linker detects any errors occurring at the time the file is created using GS/OS: for example if the diskette is locked, or if the directory is full, etc.

When an error is detected, TML Pascal II will implement a series of actions in order not to lose your data and in order to display a dialog window indicating the suspected cause of the error. In addition to the message, an icon is displayed that indicates the type of the error. Lastly, if the error relates to a part of your source code, the part concerned is displayed in inverse text within the edit window.

Editing errors are indicated by a triangle surrounding an exclamation mark; Compilation errors are indicated by a green ladybird and Linkage errors are indicated by a chain. If the message describing the error includes an error number, this is a standard GS/OS error code as described in GS/OS manuals. Appendix A includes/describes the list of error messages displayed by TML Pascal II.

CHAPTER 4: RESOURCES

INTRODUCTION

One of the more interesting additions to System 5.0 is the Resource Manager (Resource Administrator). The Resource Manager is a special tool which handles the resources stored in the Resource Component of GS/OS files. System 5.0 highlights that a file stored under GS/OS can have two components: the given component and the resources of the component. A file has a single name, but each part can be opened and treated as a separate file by GS/OS.

The data component is typically handled by using the system calls of GS/OS: to open, read, write and close the file. The file is simply treated like a collection of bytes on a disk or any other storage peripheral. The organisation of data in the data component of a file is not well defined and is typically very different for each type of file. Consult the GS/OS Reference manual for documentation concerning the routines provided by GS/OS to handle data components. A resource component, on the other hand, is handled by the Resource Manager. The Resource Manager defines the precise structure for the resource component and provides several routines to read and utilise the information stored in a resource component. TML Pascal II adds a Resource editor to create and edit resources graphically, and it is the subject of this chapter. For comprehensive documentation on the resource components managed by the Resource Manager, see the *Apple //gs ToolBox Reference* Update manual.

RESOURCES

A resource is a collection of formatted, organised data of a certain type, which represents a menu or a menu bar, a window, an alarm chime, or any other type of data as defined by the system or the user. The exact structure of each resource type is not defined by the Resource Manager. The Resource Manager only defines how the resources are stored on the disk, not their contents. A program identifies a resource by its Resource Type and its Resource ID. The Resource Type defines a class or a group of resources which share a common format. The Resource ID identifies only a single case (example) of a resource of any given standard resource. When combined, the Resource Type and the Resource ID completely identify the resource and its format. Note that the Resource ID is only a single case for the given Resource Type. Two resources of different Resources Types can have the same Resource ID.

The Resource Type is a whole number of two bytes. The following table shows the fields of the types of resources:

Types of resources defined by the Apple Resource Type	
$0000	invalid number for a standard resource (do not use)
$0001 - $7FFF	valid for use
$8000 - $FFFF	reserved for use by system applications

Among the types of resources reserved for the use of the system, Apple has preset several types of resources. These preset types of resources are used to store the representations of the elements of limps with tools of the IIgs. For example, a resource

p]ut to be used to define the structure and the contents of a menu or a window. These preset resources are shown in the following table:

TML Pascal II	Name of Resource Type	Number of Resource Type (hex)
	rIcon	$8001
	rPicture	$8002
YES	rControlList	$8003
YES	rControlTemplate	$8004
YES	rPString	$8006
	rStringList	$8007
YES	rMenuBar	$8008
YES	rMenu	$8009
YES	rMenuItem	$800A
	rTextForLETextBox2	$800B
	rCtlDefProc	$800C
YES	rWindParam1	$800E
	rWindParam2	$800F
	rWindColor	$8010
YES	rTextBlock	$8011
	rStyleBlock	$8012
YES	rToolStartup	$8013
	rResName	$8014
YES	rAlertString	$8015
YES	rText	$8016
	rCodeResource	$8017
	rCDEVCode	$8018
	rCDEVFlags	$8019
	rTwoRects	$801A
	rListRef	$801C
YES	rCString	$801D
YES	rErrorString	$8020

The name of Resource Type in the table above is given for descriptive purposes. Moreover, the unit of Resources.p that interfaces with the Apple IIgs toolbox uses the same names as the constants indicated in the above table. The TML Pascal II column indicates if the TML Pascal II resource editor allows you to edit the given resource.
As mentioned above, each resource has Resource Type and Resource ID. A Resource ID is comprised of four bytes. The table below shows the Resource ID ranges:

Range	Resource ID
$00000000	number of invalid Resource ID (do not use)
$00000001 - $07FEFFFF	for the use of applications
$07FF0000 - $07FFFFFF	reserved for system use
$08000000 - $FFFFFFFF	values not to be used

When new resources are created, a unique Resource ID must be obtained for the Resource Type to which the resource belongs. The Resource Manager provides the UniqueResourceID routine for this purpose. The Resource IDs are important to retain, because they are used as parameters for several toolbox routines which load and create menus, windows, etc.

THE RESOURCE EDITOR

Introduction

As mentioned previously, TML Pascal II contains a resource editor. The TML Pascal II resource editor is used to create graphics and edit several of the predefined resource types created by Apple. By using the resource editor, you will be able to create menus, windows, alert boxes, strings etc. with only a few mouse clicks. The resources which you create can be incorporated into a program with the aim of creating complex desktop applications quickly and easily.

The Resource Window: When TML Pascal II opens a resource for editing, it opens a new window. This window contains two drop-down lists. The list on the left displays each Resource Type for which a resource exists in the resource file. The display shows the number of the Resource Type and optionally the name of Resource Type. Only the resources which the resource editor knows about are shown with the name of the Resource Type. That makes it possible to distinguish which resources can be edited or not.

The list on the right-hand side displays the resources available for the type selected in the left list. The list is displayed showing each Resource ID number. For example, the Resource Type window is selected and the file contains two resources which are displayed in the window (1001 and 1002). To edit a particular resource, simply double click on its Resource ID.

In addition to these two lists, the Resource Window contains pop-up menu and a button. The New Resource button is used to create new resources for the Resource Type currently selected. The pop-up menu is used to create new Resource Types that do not already appear in the file.

Each resource has its set of attributes which defines how the resource can be used. The attributes are stored by the Resource Manager for each resource in the attribute Word flag. In addition to this, the Resource Manager provides two routines: GetResourceAttr and SetResourceAttr for reading and writing a resource’s attributes.

The TML Pascal II resource editor provides a button "Attr..." in each edit window of resource. Clicking this button will display an Attributes dialog window which displays the current attributes associated with this resource. Clicking the OK button in the dialog will save the new attribute configuration for the resource.

Here is the meaning of each resource attribute:

Resource Attribute	Description
Locked	If this attribute is set, the Resource Manager will call NewHandle to create a locked handle when it allocates memory for the resource.
Fixed	If this attribute is set, the Resource Manager will call NewHandle to create a fixed handle when it allocates memory for the resource.
Resource Converter	This attribute indicates if a stored resource representation in a resource component is required to be converted into a different representation when it is read into memory. If the attribute is set,

	then the resource must be converted.
Write-protection	If this attribute is set, the resource is write protected. This means that an application cannot modify the contents of a resource within the resource component file.
Preload	If this attribute is set, the Resource Manager will automatically load the resource into memory when the resource file is opened. If a resource is not configured to be preloaded, then it must be loaded explicitly into memory by using the LoadResource routine within the Resource Manager.
Do not cross bank	If this attribute is set, the Resource Manager will require that when NewHandle creates a handle that it does not cross into a different memory bank when allocating memory for a resource.
Do not use special	If this attribute is set, the Resource Manager will require that when NewHandle creates a handle that it will not occupy special memory when allocating memory for a resource.
Page aligned	Iif this attribute is set, the Resource Manager will require that when NewHandle creates a handle that it will be page aligned when allocating memory for a resource.
Purge level	If this attribute is set, the Resource Manager will pass the purge level value into NewHandle when it allocates memory for a resource.

NOTE: the default value for all resource attributes is NOT (not set) and each level of purging is set to zero (0).

PASCAL STRING RESOURCE

The Pascal String (rPString) can store up to 255 ASCII characters. The string starts with a whole byte that is a numeric value indicating the number of characters following in the resource. The Pascal String resource is used heavily by the other types of resources. For example, the resource types of rMenuItem, rMenu, rWindParam1 and several of the different rControlTemplate use a rPString resource to store their titles.

C STRING RESOURCE

The C string (rCString) stores any number of characters ending in a byte set to 0 (NULL). There is no restriction on the number of characters contained in a C string. The rCString resources are not often used in TML Pascal programs, because Pascal does not provide the native functions to use these string types. However the TML Pascal II Resource Editor does support this Resource Type.

ALERT STRING RESOURCE

The Alert String (rAlertString) stores character strings which is terminated by a byte set to zero (NULL). This string is used with the AlertWindow function of the Window Manager to display simple alert windows. The alert string resource stores the message to be displayed in a alert window along with special codes that define the size of the window, if an icon is to be displayed, and any buttons required.

For a complete description of the format and structure of the alert string, see the Window Manager chapter of Apple IIgs ToolBox Reference Update.

The TML Pascal II resource editor allows you to edit and create alert stringa to use with the AlertWindow procedure of the Window Manager. However, the alert strings have a very special formatting:

- Alert strings (AlertString) must start with the character of separation (the '/' character is recommended)

- the character of separation is used to separate the text alert strings (AlertTextString) from the alert string buttons (AlertStringButtons). To create a button within an alert, simply enter the character of separation followed by a caret (circumflex accent), a pound symbol (#) and finally a number between 0 and 6 corresponding with the following button titles:

```
#0 OK
#1 Cancel
#2 Yes
#3 No
#4 Try again
#5 Quit
#6 Continue
```

- the formatting codes for LETextBox2 are allowed inside alert string text. These codes may be entered by clicking the button "Insert LETextBox2 Code" inside the AlertString Resource dialog of TML Pascal II. The formatting options must be entered after all AlertStringText text, because the codes are invisible characters that make editing quite difficult. The codes of LETextBox2 are well documented in the LineEdit chapter of the ToolBox Reference Manual for the Apple IIgs.

TOOL STARTUP RESOURCE

The rToolStartup resource is used by an application to specify to the IIgs ToolBox which tools are required by the application and if the application uses the 320 or 640 display mode. The rToolStartup resource is used with a new StartupTools function and the ShutDownTools function provided by the tools within the ToolLocator. These two routines combined with rToolStartup provide the means to start using the tools within the IIgs ToolBox. The operations included/understood:

- to start the Resource Manager
- to open a resource fork of application
- to allocate a suitable amount of direct page memory for the tools the application uses
- to start each tool used by the application

MENU BAR RESOURCE

The menu bar resource (rMenuBar) is an ordered list of Menu resources which defines a menu bar. The resource is used by the NewMenuBar2 procedure of the Resource Manager to create a menu bar for the application.

The long rectangle at the top of the dialog represents the menu bar currently defined by the resource. The pull-down list in the bottom left corner is a complete list of each menu resource in the open resource file. The buttons Insert Menu and Delete Menu are used to add menus or to delete a menu bar from it. To add a new menu bar to the menu, first select the menu in the menu bar that you wish the new menu bar to appear after. Then select the menu to be added from the list of valid menus, and finally click the button Insert Menu. To erase a menu from the menu bar, select the menu to be erased in the menu bar and then click on the Delete Item button.

MENU RESOURCE

The resource Menu (rMenu) is an ordered list of *MenuItem* resources which defines a menu. The Menu resources are typically accessed via the resources Menu Bar, but they can be used directly by the *NewMenu2* routine of the Menu Manager tool. The Menu resource is one of two super resources that the TML Pascal II Resources editor allows (the other is the Window resource). A Menu resource, as defined by Apple, simply stores references pointing to other resources. In particular, a Pascal string resource for the title of the menu, and then an ordered list of references for each *MenuItem* (rMenuItem) resources. Each *MenuItem* resource has its own references to other Pascal strings for its title. For example, a menu with only 6 menus items (menu elements) actually consists of 14 different resources. 1 for the menu resource, 1 for the Pascal string resource containing the title of the menu, and 6 resource items referring to 6 Pascal string resources containing each menu items title. It becomes clear, that to create the many menus which an application requires would be quite tedious if each resource were to be created and referred to individually. By using simple dialogs, the TML Pascal II resource editor allows these resources to be created and edited easily via the Resources Menu. The TML Pascal II Resource Editor does not provide direct modification of each Menu resource. The edit element in the top left of the dialog is the title of the menu. The pull-down list in the left part of the dialog is the current list of menu elements (Menu Items) contained in the menu.

To edit an element of a particular menu, simply click on its name in the list. When a menu element is selected, its name and properties are displayed in several elements to the right of the dialog window. The Resource Editor allows you to specify one of five style types to be applied to menu elements, and if the menu element is to have a boundary line and/or is validated. Furthermore, you can specify an equivalent keyboard command (short-cut) for the menu element and a marking character. The Item ID is the value returned by *TaskMaster* or *MenuSelect* in an application when the user selects an element of the menu. The Resource Editor also uses this number as the resource ID of the Menu Item and the Pascal string resource for the title.

WINDOW RESOURCE

The window resource (rWindParam1) stores information necessary to create a window on the Apple IIgs desktop by using the *NewWindow2* function of the Window Manager. The window resource defines the location of the window, the size, the title, the frame definition and other attributes. In addition, the resource can refer to a list of integrated controls. The integrated controls can be buttons, check boxes, radio buttons, text-edit boxes, pop-up menus, etc. which appear in the contents of the window. The window resource is the second of the two super resources which the TML Pascal II Resource Editor supports (the other is the Menu resource). In the same way as the menu resource, the window resource

can refer to many other resources. A window can refer to a Pascal string resource for its title, and if the window has integrated controls, it will reference a resource Control List (rControlList). A resource *ControlList* then references several Control Template resources (rControlTemplate) for each button, check box, radio button, text-edit box, pop-up menu, etc., which appears in the window. And further, each Control Template can refer to a Pascal string resource for its title. It is clear that creating a window resource would be very tedious if each one of these elements were to be created individually and then referred to manually. The TML Pascal II Resource Editor does not provide any direct means of editing Control List or Template Control resources.

The large blue area in the center of the dialog is used to represent a 50% scaled view of the GS desktop with a window. The window represents the location and the size of the window as it is defined by the window resource. To change the location of the window, simply click within the contents area of the window and move it. To change the window size, click in the resize box in the lower right corner of the window and move it. The Center button can be used to quickly center the window on the desktop. The radio buttons for 640 and 320 modes are used to tell the Resource Editor which graphic screen to use when the window is created. The Resource Editor uses this information to correctly calculate the scaled size of the window in the dialog. The Frame button is used to display a dialog containing the structure definition of the window and its attributes. The Controls button is used to display the Content Controls dialog (integrated controls) which allows you to add any required integrated controls to the window.

Window Frame Definition

The window definition dialog is used to define the attributes of the window structure, zoom box, the size of the contained data and the height of the information bar. The meaning of each resource attribute is explained in the following table:

Attribute	Meaning
Title bar	If the window is to have a title bar.
Close box	If the window is to have a standard closing box.
Alert frame type	Will cause the window to be drawn in an alert style instead of the standard window style. An alert frame type is typically used for modal dialogs.
Vertical scroll bar	If the window is to have a vertical scroll bar.
Horizontal scroll bar	If the window is to have a horizontal scroll bar.
Grow box	If the window is to have a resizing box.
Zoom box	If the window is to have a zoom box.
Moveable	If the window is allowed to be moved on the desktop.
Quick in content	Set if clicking in the window area should select the window as if the contents had been clicked.
Visible	Set if the window is to be visible when it is created.
Information bar	Set if the window is to have an information bar.
Zoomed	Set if the window is to be zoomed when it is created.
Zoom rect	Defines the co-ordinates for top, left, bottom and right extents of the window when it is zoomed.
Data height/width	Defines the pixel height and width of the window.
Info height	Defines the height of the information bar if the window has one.

Window Controls Definition

The window control dialog is used to define which controls are to be used in the window. The Integrated controls are a simple button, a check box, a radio button, an edit line, an edit text, a grow icon, a static text box, an image, an icon button, a scroll bar, a pop-up menu, a list. To create a new control, to click simply on the pallet of control in the left part of the box of dialog for the type of desired control then to click inside the window at the place o| control will have to be placed.

CHAPTER 5: TML PASCAL II MENUS

THE APPLE MENU

The menu item *About TML Pascal…* contains the references and copyright information for TML Pascal II. Below this are the NDA's available on your currently booted system disk. Below these, OPEN NDA is found if one has just compiled a desk accessory into memory. This makes it possible to test the NDA program which you have just compiled. If a desk accessory has just been compiled, the option REMOVENDA is also present. It makes it possible to release the memory used by the compiled desk accessory code. Note however, that after the use of the OPEN option, REMOVE NDA disappear from this menu.

THE FILE MENU

This menu usually contains the options:

NEW: to create a new source or resource edit window. One can open as many windows as the available memory allows.
OPEN: to open a source file or existing resource.
CLOSE: to close the window of active edition. If the file were not saved, an alarm requires of you to confirm.
SAVE: to save the active window with its current name
SAVE AS: to save the active window by re-naming it
REVERT: to replace all of the modifications made to the active edit window, by the last saved version.
PRINT OPTIONS: allows you to configure the printed page header: optional presence of the name, the date, and the page number.
PAGE SETUP: allows you to define the printer commands. Whether to use continuous paper or sheet feeder (cut sheet). With the Continuous option, header text is only printed on the first page and no extra line feeds are inserted at the foot of each page. With Cut Sheet, header text is printed on top of each page, with an appropriate number of line feeds inserted at the foot of each page and it is necessary to specify the number of lines to print on each page (default: 8.5" X 11"). The Printer Commands box makes it possible to send a command BEFORE printing (changing the font, style, etc.). The control characters are represented by * (Note: this * is a circumflex accent) followed by a letter representing the control character to be sent. For example, * [will send ASCII 27 (Escape character). (Note: this * is a circumflex accent)
PRINT: print the file in the active window. By pressing the OPTION key when selecting this option, only the selected text will be printed (highlighted text).
QUIT: will exit TML Pascal II and return to the Launcher. An alert dialog may be displayed to ensure that any files being edited are saved prior to exiting.

THE EDIT MENU

The standard options:

UNDO: cancel the last edit operation carried out.
CUT: transfers the selected area of text into the clipboard.
COPY: duplicates the selected area of text into the clipboard.

PASTE: sticks the contents of the clipboard into the edit window, starting at the current cursor position.
CLEAR: to erase the selected area.
SELECT ALL: selects the entire contents of the current edit window.
SET FONT-SIZE: allows you to define the style and size of the font used in the edit window, as well as the tabulation width.

THE SEARCH MENU

FIND: displays a dialog window in which you specify the string to be found, with the search starting at the current cursor position (not from the beginning of the file).
FIND NEXT: continue searching for the string specified in the ‘Find’ dialog, starting at the current cursor position.
FIND SELECTION: search for the string selected in the edit window, starting at the current cursor position.
REPLACE: displays the Replacement menu which allows you to specify the string to be found (starting at the current cursor position) and the string which will replace it.
REPLACE SAME: the replacement of strings continues as indicated in the 'Replacement' dialog.
GOTO SELECTION: scrolls the current edit window so that the cursor becomes visible.

THE WINDOW MENU

NEXT WINDOW: moves the active window behind all the other windows. In doing so, the next highest window is placed on top and becomes active.
GET INFORMATION: displays the active files prefix, its size in bytes, and the number of lines it contains.
LAST ERROR: displays an alert window containing the last error message displayed by TML Pascal II.

THE COMPILE MENU

TO MEMORY: compile the source code of the active window into memory. If the source file is a program, it is executed after compilation. If the source file is an NDA, then the OPEN and REMOVE NDA options are activated in Apple menu. If the source is a single unit, then it is simply compiled into memory.
TO DISK: compile the source code of the active window onto disk. If the source file is a program, TML Pascal II generates a compiled file of type $B3. If the file is single unit, the compiled object code will be created with a filename suffix of 'p.o'.
CHECK SYNTAX: check if the active windows source contains any syntax errors.
ADD RESOURCES: binds the main file with a resource file present on disk. This connection makes it possible to link the resources with the program at the time of compilation.
PREFERENCES: allows you to set a number of standard settings.

K-byte symbol table: memory allocated for the symbol table. This table is a storage area used by the compiler to store label declarations, variables, tables, procedures and functions. In the majority of the cases, the standard size of 12k is sufficient. However, if an

error of the type ERROR SYMBOL TABLE SPACE EXHAUSTED occurs, it will be necessary to increase this parameter (maximum 32K, minimum 2K).

K-byte Stack: TML Pascal II programs need a stack for execution. Default value: 8K; minimum value: 1K and maximum value: 32K. This option can also be changed by directive $StackSize.

Keyboard break: allows CTRL-C detection. If this option is selected, TML Pascal II generates code between each instruction in order to detect if CTRL-C were typed. One can also use the directive $KeyboardBreak to activate the option.

Unit search path: allows you indicate which prefix TML Pascal II is to use to access UNITS as indicated in a USES clause. The default value is 1:TOOLINTERFACES: which indicates the ToolInterfaces file located in the folder containing the Pascal compiler.

Auto save text: indicates if you want to save open source files automatically when compiling to memory. This makes it possible to save any open source files to disk in case of any 'incidents' produced when executing the compiled code.

Total system memory: indicates the maximum RAM size
Free memory: indicates the size of available RAM
Largest memory block: size of the largest block available in memory

OK: one accepts the menu preference settings (or you can type RETURN).

CANCEL: any modifications done are erased and the old settings are recovered.

RELEASE MEMORY: purge memory occupied by compiled code.

THE GS/OS MENU

RENAME: allows you to rename a file residing on disk.
DELETE: allows you to completely erase a file present on disk.
TRANSFER: allows you to launch an application without returning to the Launcher.

CHAPTER 6: TEXTBOOK APPLICATIONS

The programming of an Apple IIgs can sometimes appear daunting and intimidating. For this reason, the TML Pascal II compiler was produced to not only develop applications on an Apple IIgs with desk accessories which use the "desktop" interface, but to also provide a programming environment similar to the more traditional Pascal. i.e. the possibility of writing useful applications without having to use windows, menus, controls, etc.

This type of application is called a "textbook" application. These "plain vanilla" applications are still autonomous GS/OS applications, but they represent a standard Pascal program most representative of ones found in a work or traditional data-processing environment. These programs make little or no use of specialised IIgs Tools and thus ensure the portability of these Apple IIgs programs.

A "textbook" program is executed on an Apple IIgs in super-high resolution 640 mode. The compiler generates code which initialises the Apple IIgs, and creates a single window on the screen allowing for a standard display of 20 lines and 80 columns. The content area of this window allows usage of the standard Pascal procedures READLN and WRITELN, and because it acts as an Apple IIgs window, you will also be able to use QUICKDRAW graphics.

"Textbook" programming is particularly useful for programmers new to the Pascal language who subsequently discover the attractive features of the Apple IIgs. Experienced programmers will also appreciate this TML Pascal II feature, as it allows for rapid construction of tools or to test a new portion of code without having to worry about the details using the Apple IIgs Tools.

The structure of a "textbook" program is very simple and straightforward.

The first example program FIRSTPROG is a "textbook" program:

```
PROGRAM FirstProg;
BEGIN
      Writeln('Hello World');
      Readln;
End.
```

CHAPTER 7: GRAPHIC TEXTBOOK APPLICATIONS

Because 'textbook' programs are already running in Apple IIgs super-high-resolution mode, by using QUICKDRAW it is possible use graphics functions within your 'textbook' programs. Such a program is defined by the graphics procedure which initialises the IIgs desktop using Quickdraw, and specifies 320 or 640 resolution mode. The newly created desktop has neither menu bars nor windows. For example:

```
PROCEDURE Graphics(screenMode: Integer);
```

So, transforming our previous ‘Hello World’ example into a graphic application:

```
PROGRAM HelloWorld;
BEGIN
      Graphics(640);
      WriteLn('Hello world');
      ReadLn;
END.
```

To be able to use QuickDraw it is necessary to include the name of this module in the USES clause of your program. When your program is executed, QuickDraw will already be initialised and ready for use. In addition, the Event Manager is also available to monitor actions performed by the activate mouse.

For example:

```
PROGRAM GraphicHelloWorld;

USES  Types,
      QuickDraw;

VAR   aRect: Rect;

BEGIN
      Graphics(320);
      SetRect(aRect, 10, 10, 30, 40);
      FrameRect(aRect);

      OffsetRect(aRect, 25, 30);
      FrameOval(aRect);

      OffsetRect(aRect, 25, 30);
      FrameRRect(aRect, 20, 20);

      MoveTo(50,30);
      DrawString('Hello world');

      ReadLn;
END.
```

You will find an example graphic textbook application named Graphics.p on the TML Pascal II disk.

CHAPTER 8: DESKTOP APPLICATIONS

This chapter gives a broad outline of using TML Pascal II and usage of the Apple IIgs 'ToolBox'. The toolbox naturally becomes the basis for event management, any applications based on the clipboard, and desk accessory applications. The toolbox is an important collection of software organised into several subsets according to their function called 'Tool Sets' or 'Managers'. Within each tool set lies the complete set of routines necessary for the particular function of the tool. Each of the tools is assigned a unique tool number and each of its routines is allotted a particular function number.

TML Pascal II gives access to the Apple IIgs toolbox thanks to a collection of Pascal modules. Within each module one finds Constants, Types, Procedures and Functions which correspond to one or more of the Apple IIgs ToolBox routines. Therefore, each time an application requires access to a ToolBox routine, it must specify the name of the Pascal module in the USES clause to define a link to the particular ToolBox routine. These Pascal modules are in the TOOLINTF/ folder of the Pascal distribution diskette. For example, the USES clause that makes 'QuickDraw II' routines available:

```
USES QuickDraw;
```

THE TOOLS OF AN APPLE IIgs

Starting with a minimum of IIgs System Software version 5.0, 30 different tools are defined as subsets of the IIgs toolbox. Each of them is listed in the table below along with the Pascal module which defines its interface. It also specifies if each tool resides in the IIgs ROM, or if the tool resides on disk as a file tool that must be loaded into RAM before being used. If a tool must reside in RAM, then its corresponding file tool must be available in the SYSTEM/TOOLS/ folder of the current boot disk. The name of a file tool is TOOLxxx, where xxx represents a three digit number corresponding to the number allocated to the tool. For example, the file tool of the screen manager has the name: TOOL014.

Tool Number	Tool Name	Pascal Module	RAM	ROM
1	Tool locator	Locator.p		X
2	Memory Manager	Memory.p		X
3	Miscellaneous Tools	MiscTool.p		X
4	QuickDraw II	QuickDraw.p		X
5	Desk Manager	Desk.p		X
6	Event Manager	Event.p		X
7	Scheduler	Scheduler.p		X
8	Sound Manager	Sound.p		X
9	Apple Desktop Bus	ADB.p		X
10	SANE	SANE.p		X
11	Integer Math	IntMath.p		X
12	Text Tools	TextTool.p		X
13	Reserved for the System			X
14	Window Manager	Windows.p	X	
15	Menu Manager	Menus.p	X	
16	Control Manager	Controls.p	X	
17	System Loader	Loader.p	X	

18	QuickDraw Aux Routines	QDAux.p	X	
19	Print Manager	Print.p	X	
20	Line Edit	LineEdit.p	X	
21	Dialog Manager	Dialogs.p	X	
22	Scrap Manager	Scrap.p	X	
23	Standard File	StdFile.p	X	
24	Disk Utilities	n/a	X	
25	Note Synthesizer	NoteSyn.p	X	
26	Note Sequencer	NoteSeq.p	X	
27	Font Manager	Fonts.p	X	
28	List Manager	Lists.p	X	
29	Audio Comp-Expan.	ACE.p	X	
30	Resource Manager	Resources.p	X	
32	MIDI	MIDI.p	X	
34	TextEdit Manager	TextEdit.p	X	

WHAT DO THESE TOOLS DO?

The following paragraph briefly explains the functionality of each Apple IIgs tool. This is intended to be a short introduction to each tool and you will have to consult volumes 1,2 and 3 of the Apple IIgs ToolBox Reference to learn more.

The 7 base tools

TOOL LOCATOR
The tool locator is the most important. Without this tool it would be impossible to access the other tools. The tool locator enables you to load the tools that reside on disk into RAM and to make use of them, as we saw in our previous example, without knowing where in memory they are stored.

MEMORY MANAGER
It is the second most important after the Tool Locator. This tool is entirely responsible for the allocation, de-allocation and movement of memory blocks on the Apple IIgs. It keeps a record of the memory capacity available and which part is allocated and to what.

MISCELLANEOUS TOOLS
It primarily consists of system routines which must be available for most of the other tools to function.

QUICKDRAW
It is the tool which controls the graphical environment of the Apple IIgs and allows the drawing of simple objects and text. All other tools which create graphic objects, like the window manager, make use of this tool.

AUXILIARY QUICKDRAW
This tool contains additional routines which supplement the QuickDraw tool found in ROM.

EVENT MANAGER
The event manager allows applications to control and react to a user action like the actions of a mouse or keyboard.

RESOURCE MANAGER
The resource manager is responsible for the handling of data resources of GS/OS files.

The tools for interfacing with the Desktop

CONTROL MANAGER
The control manager contains all the routines necessary for the handling of controls. Controls being things such as a button, scroll-bars, check boxes, etc.

DESK MANAGER
This tool makes an application be able to support desk accessories – both Classic Desk Accessories (CDA's) and New Desk Accessories (NDA's).

DIALOG MANAGER
This tool provides routines which make it possible for an application to create and use dialog windows and alarms that facilitate communication between a user and your program.

FONT MANAGER
This tool makes it possible for your application to use various different character types and styles.

LINE EDIT
This tool makes it possible for a program to present text on the screen that can be edited by the user.

LIST MANAGER
This tool is used to create lists which are used when displaying the results of a selection by a data amount or type.

MENU MANAGER
This tool controls and allows the use of the pull-down menus, and presents the available choices within an application.

SCRAP MANAGER
This tool provides the desktop 'clipboard' supplements allowing an application to cut, copy, and paste within an application.

TEXTEDIT MANAGER
This tool implements a text editor, accepting changes of font and style.

WINDOW MANAGER
This tool creates the Office environment and is responsible for the creation and manipulation of windows.

The peripheral management tools

APPLE DESKTOP BUS
This tool is a method and a protocol for the connection of input devices, such as keyboards and mice, with the Apple IIgs. The routines of this tool are used to send commands and data between the Apple Desktop Bus microcontroller and the remainder of the system.

PRINT MANAGER
This tool makes it possible for an application to use the QuickDraw routines to print text and graphics on an ImageWriter or LaserWriter printer.

STANDARD FILE
The standard user interface allowing a file to be opened or saved.

TEXT TOOLS
This tool ensures that there is a link between drivers for character based hardware which must be executed in emulation mode, and the applications running in native mode.

Tools for managing the system:

SCHEDULER
The scheduler delays the activation of a desk accessory or any other task until the necessary resources become available.

SYSTEM LOADER
It is responsible for the loading and the repositioning of code, such as applications or desk accessories, into memory.

The sound tools

SOUND MANAGER
This tool makes it possible to access the sound generation electronics of the Apple IIgs to create basic sounds.

NOTE SYNTHESIZER
This tool is used to create complex musical sounds by using the musical electronics of the Apple IIgs.

NOTE SEQUENCER
This tool is used to bind notes of the Note Synthesizer into sequences, motifs and phrases constituting a sound.

ACE
This tool compresses/decompresses digitised sounds allowing their file sizes to be reduced when saved to disk.

MIDI

This tool allows you to connect a standard MIDI instrument to the IIgs via one of the serial ports.

The mathematical tools

INTEGER MATH
This tool consists of all the mathematical operations for integers. This includes multiplication, division, conversions, etc.

SANE
This tool completes the standard numerical environment of Apple. It provides a high precision IEEE 754 and 854 floating point arithmetic component.

HOW TO PERFORM A CALL TO A TOOLBOX ROUTINE

This section is intended for the experienced programmers who want to include/understand how a toolbox routine is called from within TML Pascal II. If you are satisfied with the toolbox functions built-in to TML Pascal II, and do not need to use the additional toolbox routines, you can skip over this section.

As specified above, TML Pascal II allows access to the Apple IIgs toolbox thanks to the built-in Pascal modules which define the interface necessary for each routine of a particular tool. Each tool routine is defined as a procedure or function according to whether it returns a value or not, and can have zero or more parameters. Finally, the declaration of a procedure or function is added using the *Tool* statement (the Tool Directive can be found in chapter 7 of the reference manual). The Tool statement is an extension particular to TML Pascal II for the Apple IIgs, with the aim of defining an interface to the ToolBox. The declaration of the following procedure is extracted from module QUICKDRAW.P and it defines the link with the MoveTo procedure of the QuickDraw tool.

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

As you can see, the declaration of the procedure is supplemented with the statement Tool 4,58. The first entry of the Tool statement specifies the tool (toolset) that the routine belongs to. In this case, it is tool number 4 which happens to be the Quickdraw tool. The second entry is the number of the routine within the tool. Each routine within a tool has a unique function number. The MoveTo routine has been allocated function number 58. Together, these two entries make it possible to uniquely identify a procedure within the ToolBox. The Apple IIgs has a defined process for the activation of a ToolBox routine. To call a routine, space for the results of the function must be reserved on the stack, followed by the list of the parameter values. The X register of the 65816 must then be loaded with the function number (of the tool routine) and the number of the tool so that the X register contains: 256 * function number + tool number. Finally, a jump to the subroutine at the address $E10000 is made, which in turn contains a jump to the 'tool locator’ which finds the code associated with the desired tool routine and passes control to it. When the routine returns, all of the parameters have been removed from the stack, leaving the result on top of the stack. Additionally, if an error occurs during the execution of the tool routine, the carry flag of the 65816 will be set to true and the accumulator of the 65816 will contain an error code.

By using the Tool statement with a procedure or function declaration, and if the preceding conventions are followed, TML Pascal II will generate an instruction to store the accumulator value in the global variable ToolErrorNum (see chapter 10 of the reference manual) so that any error codes returned by a tool routine can be examined. To illustrate this, the statement MoveTo(16,20) will generate the following 65816 instructions:

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

In order to allow programs written in TML Pascal II to perform error handling for calls made on tool routines, TML Pascal II defines a special function ISTOOLERROR which examines the state of the carry flag of the microprocessor. The IsToolError function should only be used IMMEDIATELY after a call to a tool routine to ensure that the state of the carry flag has not been altered by another operation. Thus, a program written in a TML Pascal II could use the following code to detect an error occurring in the MoveTo routine:

```
MoveTo(16,20);
if IsToolError then
      Temp:= _ToolErr;
      Writeln('The following error occurred in MoveTo, #',Temp);
end;
```

Notice that the value of _ToolErr was copied into the temporary variable *Temp* before the call to Writeln. This is because Writeln itself calls upon the ToolBox and this could alter the value of the _ToolErr associated with the error condition as returned by MoveTo.

There are at least three cases where the compiler does not need to perform *STA _ToolErr*. They are as follows:

1. A lot of tool routines do not return an error (this is the case within the example above).
2. An application has guaranteed that all the possible error conditions can not exist.
3. An application which would not be affected if an error occurred.

If an application has toolbox calls that predominantly fit any of the 3 points above, then the generation of STA _ToolErr can increase the size of an application unnecessarily. To avoid this, TML Pascal II has an statement $ToolErrorChk that will to deactivate/reactivate the generation of the STA (see appendix B of the reference manual).

For example, the following call to the *MoveTo* routine will not generate the instruction *STA ToolErrorNum*:

```
(*$ToolErrorChk-*)
MoveTo(16,20);
```

Even though the use of the $ToolErrorChk statement can save a great quantity of code, the programmer must be very careful in its usage to avoid testing the value of *_ToolErr* when no error code has been stored.

EVENT MANAGEMENT

The concept of event management is probably a little different from what you have practiced whilst programming up to date. The usual and conventional strategy of application programming consists of working out a sequence of actions that are executed one after the other. Action1 is executed first, followed by action2, then action3 etc.

Event management uses the opposite approach. All actions are possible at a given time and the action to execute depends on an event, generally in response to user interaction with the program.

By using this strategy, the basic structure of all applications on the Apple IIgs is almost always identical. The main program generally consists of the following declarations:

```
BEGIN
      StartUpGSTools;
       (* routines for initialising menus, windows, etc. *)
      MainEventLoop;
      ShutDownGSTools;
END.
```

The procedures StartUpGSTools and ShutDownGSTools are responsible for the loading and initialisation of the ToolBox tools to be used in the application, and then for the unloading and ‘cleanup’ before the program ends (see Basic Utilisation of the Apple IIgs ToolBox). The MainEventLoop procedure is responsible for the detection of events and then the response to the events. The typical structure of this procedure is as follows:

```
procedure MainEventLoop;
var Event: EventRecord;
code: integer;
begin
      gMainEvent.wmTaskMask := $001FFFFF; (*permit a task to handle
      everything*)
      gDone := false;

      repeat
      code:=TaskMaster($FFFF, gMainEvent);
      case code of
            wInGoAway: DoClose;

            wInSpecial,
            wInMenuBar: HandleMenu;

            wInControl: DoControlHit;
      end;
until gDone;

end;
```

The possible GetNextEvent events:

Event	Description
nullEvent	When no event took place
mouseDownEvt	Generated when the mouse button is pressed
mouseUpEvt	Generated when the mouse button is released
keydownEvt	Generated when a key is pressed
autoKeyEvt	Generated when a key is pressed and held down
updateEvt	Specifies that the window contents must be redrawn (refreshed)
activateEvt	Indicates when a window becomes active or inactive
switchEvt	Generated when a control switch is pressed
deskAccEvt	Generated at the time of a call to a CDA
driverEvt	When PostEvent is sent by a peripheral (usually beginning or end of transmission)
app1Evt-app4Evt	Up to 4 events can be defined by the user and passed into the event queue thanks to PostEvent.

The possible TaskMaster events:

Event	Description
wInDesk	mouse event on the desktop, outside of any window
wInMenuBar wInSpecial	mouse event in the menu bar with the mouse button released on a menu item other than an NDA
wInContent	mouse event in the Content area of a window
wInDrag	mouse event in the Drag area of a window
wInGrow	mouse event in the Grow box of a window
wInGoAway	mouse event in the box of closing of a window
wInZoom	mouse event in the Zoom box of a window
wInInfo	mouse event in the information bar of a window
wInFrame	mouse event in the body of a window
wInactMenu	selection of an inactive menu item
wClosedNDA	closing of an NDA
wCalledSysEdit	call to System Edit
wTrackZoom	
wHitFrame	mouse event in the body of the active window
wInControl	mouse event inside a control within a window

PROGRAM STRUCTURE

The source code of a program can, for the most part, be organised according to the desires of the programmer. However, Desktop applications must respect certain constraints. Indeed, any Desktop application must start by initialising its tools, then the

global variables, create the menus and windows, manage the events, and finally to close the tools before quiting the application.

Here is an extract of the SKELETON.p program, where it is setting up its resources. In this case, the StartupTools function refers to the resource StartStop which defines the tools used by the application. See chapter 4 for more information on the resources.

```
gMyMemoryID := MMStartUp;
gStartStopRef := StartupTools(gMyMemoryID, refIsResource,
ref(kStartStopRefID));
if _ToolErr = NoError then
begin
      InitialiseGlobals;
      SetUpMenus;
      SetUpWindows;
      InitCursor;
      MainEventLoop;
end;
ShutDownTools(refIsHandle,gStartStopRef);
```

Adding resources

As one can see in the example above, TML Pascal II programs can use resources defined by a suitable editor (see chapter 4). When a program is compiled, the resources are required to be copied along with the GS/OS application. The linker takes care of this by copying the resource file specified in Add Resources menu, into the compiled application.

Definition Procedures (DefProcs)

Often, the ToolBox routines of the Apple IIgs must call a procedure which is part of your program. These types of procedures, and sometimes functions, are given the name of *Definition Procedures* which is abbreviated to *DefProcs*. The reason for this name is because these routines are generally used to allow an application to define the procedures used for certain operations. For example, there are the procedures for menu definition which make it possible for an application to provide layout procedures to represent menus, perhaps a menu which would contain a palette of colours in place of the usual heading. As you might expect, the ToolBox also allows definition of procedures for windows, controls, lists, etc.

Another component of the ToolBox in which an application must use a definition procedure is the tool routine 'NewWindow' of the Window Manager. The routine 'NewWindow' has only one parameter record which is: 'NewWindowParamBlk'. This record contains all the information the window manager needs to trace and maintain the new window. Three fields of the record require definition procedures. The following Pascal record shows the fields of the record 'NewWindowParamBlk' which require definition procedures.

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

The routine wContDefProc, for example, is called by the window manager if it detects that the displayed window contents must be updated due to the fact that a part of the window, which was previously hidden, becomes visible.

When one uses the ‘NewWindow2’ tool routine, it is also necessary to call upon definition procedures since one of the parameters of the function is called the WINDOW'S CONTENT DEFINITION PROCEDURE. The Content defProc routine is called by the Window Manager when it determines that the contents of a window must be updated (redrawn) because a previously hidden area of a window becomes visible.

As you might guess, the conventions of a call to a ToolBox procedure for a definition procedure are not the same as for the normal Pascal procedures. Consequently, it is necessary for an application to declare to the TML Pascal II compiler that a particular procedure is in fact a definition procedure and that it must use the call conventions of the ToolBox routines. To achieve this, the following compiler directive is used: $DefProc. This directive must appear immediately before each procedure that is a definition procedure.

There is an observation that must be made concerning definition procedures: the addressing of global variables. Typically, global variables are addressed using the absolute addressing mode of the 65816 rather than the less effective long absolute addressing mode. This is due to the fact that TML Pascal II makes sure that the data bank register points to the bank of memory containing the programs global variables. However, in the case of definition procedures, the normal convention used by TML Pascal II can not be applied as you cannot guarantee that ToolBox routines will behave in the same manner - for example, the ToolBox routine may change the value of the data bank register. Because of this, it is necessary to force TML Pascal II to use the long absolute addressing mode for global variables in a definition procedure to ensure that they are referenced correctly.

Example:

```
(*$DefProc*)
PROCEDURE WindowContentDraw;
BEGIN
      ....
END.
```

Large programs and segmentation

The Apple IIgs limits the size of program code and data segments to 64 KB. The code segments contain the implementation code, while the data segments contain the space required for the global variables of the application. The reason for this 64 KB restriction is that a segment should not cross the limit of a memory bank. On the Apple IIgs, a memory bank is 64 KB. Therefore, to develop applications having more than 64 KB of code or data, the program must be segmented. Normally, TML Pascal II automatically creates a code segment and a data segment for a given application. To obtain more than one segment, the compiler directive $CSeg and $DSeg must be used.

Segmentation of code

The code segments are given names so that the linker can organise the various parts of the assembled code based upon the name of each segment. The name of a code segment

by default is MAIN. To change the name of the code segment in use, the TML Pascal II compiler directive (* $CSeg segname *) is used. When this directive appears in a program or in a module, the code of all the procedures and functions which follow is placed in the new code segment. To return to using the default code segment, use the directive (* $CSeg Main *).
For more details concerning the use of the compiler directive (* $CSeg segname *), see appendices B and C of the reference manual.

Segmentation of data

The data segments are given names in the same way as the code segments so that the linker can organise the data areas based upon their name. The name of a data segment by default is - global. To change the name one uses the compiler directive (* $DSeg segname *). Unless a program absolutely needs additional memory for its global variables, the above compiler directive should not be used because access is less efficient.
For more details concerning the use of the compiler directive (* $DSeg segname *), see appendices B and C of the reference manual.

CHAPTER 9: NEW DESK ACCESSORIES

The New Desk Accessories (NDA) are 'mini-applications' which can run inside Apple IIgs applications. There are currently two kinds of desk accessories: Classic Desk Accessories (CDA) and New Desk Accessories (NDA). NDA's are intended to be executed in the desktop environment with event management. An NDA runs in a window and takes control when its window is the uppermost on the desktop. The new desk accessories are accessible, within desktop applications, via the Apple menu. TML Pascal II provides tools to aid in the development of new desk accessories in Pascal, and this is the subject of this chapter.

START

Since NDA's operate in the desktop environment you must make sure that the following tools are loaded and initialised:

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

Additional tools may also be available, but you cannot determine if they are loaded and initialised. If a new desk accessory needs additional tools, it must ensure that they are available by loading and initialising them itself.

THE SOURCE FILE

The source code of a new desk accessory is completely different from that of a normal program. In particular, an NDA does not have a main section but instead contains four special procedures:

```
DAOpen, DAClose, DAAction and DAInit.
```

In addition to these four necessary procedures, three elements containing additional information are required: the periodicity, the event mask and its menu name. This information is specified in a TML Pascal II compiler directive $DeskAcc.

```
(* $NDA period eventmask menuName *)
```

The periodicity specifies the frequency that DARun will call the NDA to make it functional. A period of 1 accounts for 1/60 of a second, a period of 2 accounts for 1/30 of a second, etc. A period of $FFFF means never. For example, if an NDA displays the time then it is necessary for it to have a period of 60 so that it updates every second.

The event mask specifies which events are handled by the desk accessory. The values available to an NDA are a subset of the values available to Apple IIgs applications using

GetNextEvent or TaskMaster. They have been listed below by referring to the Events.p module. From the six events listed below, the update and activation events are always transmitted to the desk accessory regardless of the event mask, however, the four types of events remaining must be specified explicitly. If all events are to be handled by the desk accessory then an event mask of -1 (or $FFFF>) must be specified.

```
CONST mDownMask = 2;
      mUpMask = 4;
      keyDownMask = 8;
      autoKeyMask = 32;
      updateMask = 64;
      activeMask =256;

EveryEvent = -1; (* $FFFF*)
```

Finally, the menu name is the name of the accessory that will appear in the Apple menu of an application working with desk accessories.

As we mentioned above, this information is specified with the compiler directive $NDA. This directive must be the very first line of the program, appearing even before the word PROGRAM. For example, the following directive specifies a period of 1 second, that all events will be handled by the desk accessory, and that the menu name of this accessory is "Clock".

```
(* $NDA 60 -1 Clock *)
```

The desk accessories function differently from normal applications with regard to the addressing of global variables. TML Pascal II allocates the memory location necessary for the global variables in a data segment. A data segment is loaded into memory just like the code of a desk accessory is loaded into memory. However, the data bank register of the 65816 is not able to point to the memory bank containing the desk accessories global variables data segment. As the compiler cannot be sure which area the global variables will be stored in, it must always use the long absolute addressing mode of the 65816 when referring to global variables. As this is not the norm for TML Pascal II programs, it should be specified to the compiler with the directive $LongGlobals+

Now let us present the basic structure of a new desk accessory written in TML Pascal II.

```
(*$NDA 60 -1 Clock *)

UNIT MyClockNDA;

FUNCTION DAOpen:WindowPtr;
PROCEDURE DAClose;
PROCEDURE DAAction(Code: Integer; Param: LongInt);
PROCEDURE DAInit(Code: Integer);

IMPLEMENTATION

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;

END.
```

The DAInit procedure

The DAInit procedure is called when the DeskStartUp and DeskShutDown routines of the Desk Manager tool are called by an application to initialise or close a new desk accessory. The value of the *Code* parameter indicates under which circumstance the routine was called. If code=0, DAInit was called for closing, otherwise for opening. In any case, the routine is required to contain the necessary code for initialisation and closing of the desk accessory.

```
PROCEDURE DAInit(code:integer);
(* the variable myWindOpen is global *)
BEGIN
      if code=0 then begin
            (* ask to close, verify that the DA window is closed *)
      end
      else begin
            (* initialisation *)
            myWindOpen:=false
      end
END;
```

The majority of NDA's can be open at the same time that an application window is being displayed. An NDA must check to see if the window is open or not. The best way of doing this is by using a global variable *windOpen* which will be TRUE when the window is open and FALSE otherwise. Therefore, when DAInit is called during DeskStartUp, it will have to set the global variable *windOpen* to FALSE. It is possible that DAInit will be called to close the window because of a DeskShutDown, when its window is still open. This occurs when a user leaves an application with an NDA still open on the desktop. It is important that the DAInit procedure checks that the window is closed before allowing the NDA to end.

The DAOpen function

This function is called as a result of an application calling the OpenNDA routine of the Desk Manager tool. This routine must check if the desk accessory has already been opened, and if so, leave without doing anything. If the desk accessory is not open, then the function will have to create the desk accessory window, to make it a system window, and to return a pointer to the window back to the creating window as the result of the function. Here is a code snippet which shows the basic structure of DAOpen.

```
FUNCTION DAOpen: windowPtr;
(* the variables myWindOpen, myWindPtr, and myWind are global *)
BEGIN
      if myWindOpen then SelectWindow(mywindPtr)
```

```
        else begin
              myWindOpen:=true;
              myWindPtr := NewWindow2(...);
              SetSysWindow(myWindPtr);
        end;
        DAOpen:=myWindPtr;
END;
```

The DAClose procedure

The DAClose procedure will have to close the desk accessory if it is found to be open. It must also gracefully handle the situation without raising an error if it is called when the desk accessory is not open.

```
PROCEDURE DAClose
(* the variables myWindPtr and myWindOpen are global *)
BEGIN
        if myWindOpen then begin
              CloseWindow(myWindPtr);
              myWindOpen:=false;
        end;
END;
```

The DAAction procedure

The DAAction procedure is the routine that does all the work associated with the desk accessory after it has been opened and until it is closed. The DAAction procedure has two parameters: a *code* that indicates which kind of action to take and a parameter *param* whose meaning depends on the code parameter. There are nine possible values for the code, and each of them must be handled by the DAAction procedure. These actions are enumerated in the following table with the meaning of the *param* parameter in each case.

Action	Description
DAEvent	An event concerning the desk accessory occurred, *Param* points to the record describing the event.
DARun	The specified duration of the NDA's period has elapsed. *Param* not used.
DACursor	This code is sent to an NDA when it is the uppermost window on the desktop, each time SystemTask is called. The goal is to make it possible for the desk accessory to change the cursor when it is over the NDA window. *Param* not used.
DAMenu	This is sent to a desk accessory if a menu item is selected. LoWrd(param) is the identifier of the menu and HiWrd(param) is the identifier of the menu item.
DAUndo	These 5 codes are transmitted to a desk accessory if the application determines that the user has selected one of these edit operations in the Edit menu. The procedure DAAction will set the *Param* code to a value of 1 if an action took place, or a value of 0 otherwise.
DACut	
DACopy	
DAPaste	
DAClear	

The following code snippet shows the basic structure of DAAction:

```
PROCEDURE DAAction(Code : integer; Param: Longint);
(* myWindPtr is a global variable *)
VAR currPort: GrafPtr;
BEGIN
      case code of

            DAEvent:begin
                  case EventRecordPtr(param)^.what of
                        mousedownEvt:;
                        mousUpEvt: ;
                        keyDownEvt: ;
                        autoKeyEvt: ;
                        updateEvt: ;
                        activateEvt: ;
                  end;
            end;

            DARun: begin
                  currPort := GetPort;
                  SetPort(myWindPtr);
                  SetPort(currPort);
            end;

            DACursor: begin
                  (* code for modifying the cursor *)
            end;

            DAMenu : begin
                  (* code to respond to a menu selection *)
            end;

            DAUndo : begin
                  (* code to 'undo' the last action within the DA *)
                  code := 1;
            end;

            DACut : begin
                  (* code to perform a 'cut' operation in the DA *)
                  code := 1;
            end;

            DACopy : begin
                  (* code to perform a 'copy' operation in the DA *)
                  code := 1;
            end;

            DAPaste : begin
                  (* code to perform a 'paste' operation in the DA *)
                  code := 1;
            end;

            DAClear: begin
                  (* code to perform a 'delete' operation in the DA *)
                  code := 1;
            end;
      end;
END;
```

COMPILING AN NDA

During development of an NDA, compiling to memory allows you to artificially install the NDA into the Apple menu for testing purposes. When the NDA is has been successfully tested, you should compile to disk. When you have successfully compiled an NDA to disk, it must be installed in the SYSTEM/DESK.ACCS/ folder on the boot disk so that other applications can access it. The installation of a new desk accessory is done according to the following three steps outlined below:

1. New desk accessories are programs (files to be loaded into GS/OS) having the file type *$B8*.
2. The Desk Manager tool of the Apple IIgs requires that all desk accessories be placed in SYSTEM/DESK.ACCS/ folder. Consequently if you wish the desk accessory to be loaded, it is necessary to copy it into this folder.
3. Finally the Apple IIgs must be restarted. During the restart, the SYSTEM/DESK.ACCS/ folder is examined to see which desk accessories are installed. As this examination only occurs at the time of startup, it is necessary to reboot to have any new accessories recognised.

CHAPTER 10: CLASSIC DESK ACCESSORIES

The Classic Desk Accessories (CDA) are 'mini-applications' which can run inside Apple IIgs applications. CDA's are intended to be executed outside of the desktop interface and without event management. A CDA is activated from within the Control Panel (Apple-Ctrl-Esc).

PROGRAM STRUCTURE

The structure of a CDA is rather similar to a normal text application, except that there is no MAIN program and that there are two special routines: starUpCDA and ShutDownCDA which are called directly by the DeskManager at the appropriate time. Additionally, the source code must start with the $CDA directive which indicates to the compiler that the source implements a CDA. Following this directive, one places the name of the CDA as it will appear in the Control Panel menu.

```
(*$CDA menuName*)
```

This directive must appear BEFORE the reserved word UNIT.

Here is the complete strucutre of a CDA:

```
(*$CDA SHRDump*)
UNIT MySHRDump;

INTERFACE

PROCEDURE StartUpCDA;
PROCEDURE ShutDownCDA;

IMPLEMENTATION

PROCEDURE StratUpCDA;
BEGIN
      (* code for the routine *)
END;

PROCEDURE ShutDownCDA;
BEGIN
      (* code for the routine *)
END;

END.
```

The StartUpCDA procedure

This procedure does not have any parameters nor does it have any specified task to accomplish. The DeskManager calls this procedure when the CDA's name is selected in the Control Panel menu. Contrary to NDA's, CDA's do not generate events and it is entirely up to you to manage your CDA.

The ShutDownCDA function

This procedure does not have any parameters nor does it have any specified task to accomplish. The DeskManager calls this procedure when DeskShutDown is called by an application, or if one passes from ProDOS8 into GS/OS. This function makes it possible to

finalise any actions started by the StartUpCDA procedure; but if CDA's are to be left open so that you can return to it from within the running application, you will not have to call this function.

COMPILING A CDA

During development of a CDA, compiling to disk is the only possible way to compile a CDA. This is contrary to NDA's, as one cannot compile to memory for testing. After you have successfully compiled a common desk accessory, it must be installed in the SYSTEM/DESK.ACCS/ folder of the boot disk so that other applications can access it. The installation of a traditional desk accessory is done according to the following three steps:

1. Traditional desk accessories are programs (files to be loaded into GS/OS) having the file type *$B9*.
2. The Desk Manager tool of the Apple IIgs requires that all desk accessories be placed in the SYSTEM/DESK.ACCS/ folder. Consequently if you wish the desk accessory to be loaded, it is necessary to copy it into this folder.
3. Finally the Apple IIgs must be restarted. During the restart, the SYSTEM/DESK.ACCS/ folder is examined to see which desk accessories are installed. As this examination only occurs at the time of startup, it is necessary to reboot to have any new accessories recognised.

CHAPTER 11: RESERVED WORDS

BASIC ELEMENTS

We will see that basic elements can be classified as "special symbols", "identifiers", "statements", "non signed numbers", "labels" and "character strings". Apart from character strings, characters are generally not case sensitive within a program.

The text of a Pascal program consists of basic elements and separators, where separators are made up of whitespace (space character or tabs) or a comment. Two adjacent basic elements must be separated by one or more separators, and if each element is an identifier then the separator must be a number or a word-symbol.

SPECIAL SYMBOLES

Special symbols are elements which have a particular significance and which are used to delimit the syntactic modules of the language.

The following single characters are special symbols:

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

The following pairs of characters are special symbols:

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

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

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

IDENTIFIERS

Identifiers are used to denote constants, types, variables, procedures, functions, programs, modules, and fields in records. An identifier can be any length so long as it can fit on one line, however, only the first 255 characters are significant. Identifiers are case insensitive – ie. upper and lower case letters are equivalent. An identifier can not have the same name as a reserved word.

Examples of standard TML Pascal II identifiers:

```
Exit Maxint Writeln _DataInit A_very_long_identifier
```

DIRECTIVES

Directives are identifiers which have a particular significance in the context of a declaration of a procedure or function. They are able to be used as identifiers in all the other contexts. These are:

```
EXTERNAL FORWARD INLINE TOOL
```

(See chapter 17)

NUMBERS

Unsigned integers can be decimal or hexadecimal (hexadecimal numbers are prefixed with the $ character) – these represent constants of the *integer* or *long integer* data types. Unsigned real numbers using decimal notation represent constants of the *extended* data type. The letter "E" or "e" preceding a factor represents the powers of ten.

Examples of numbers:

```
1 +100 -0.1 $A05D 5.329E4 NaN(1) Inf
```

LABELS

A label is a set of characters whose value entirety between 0 and 9999. The removal of the leading zeros of a label does not modify its significance. For example, 1 and 0001 are equivalent. These labels are used with the GOTO statement which will be described in chapter 16.

CHARACTER STRINGS

Character strings are a succession of zero or more printable characters, all on the same line of a program and enclosed within apostrophes (single quotes). The maximum number of characters in a string is 255. A character string without any characters between the apostrophes is called a null string.

Character strings represent a value of the *string* type. As a string type, character strings are not only compatible with other string types, but also with character types and packed strings. All values of a string type have a *length* attribute. In the case of character strings, the length is fixed - it is equal to the current number of characters inside the apostrophes. A pair of adjacent apostrophes inside a character string is interpreted as a single apostrophe, and is therefore counted as a single character when determining the length of the string.

Examples of chains characters:

```
'A' ';' 'Pascal' 'This is a string' 'don''t' '''' ''
```

DECLARATIONS OF CONSTANTS

A declaration of a constant defines an identifier to represent a constant value inside the block containing the declaration.

A signed number can be an integer or a real.

COMMENTS AND COMPILATION DIRECTIVES

Constructs such as:

```
{ any text }
(* any text *)
```

are called comments.

The replacement of whitespace by a comment or a comment by whitespace does not effect the operation of a program. Because of this, a comment is like a separator, and it can appear in a program anywhere where whitespace can appear.

Comments of the form {...} can be embodied within comments of the form (*...*) and vice versa, however no other form of comment is available. The appearance of a } in a {…} comment or a *) in a (*...*) comment, always finishes the comment.

A compilation directive is a comment containing the $ sign immediately after the { or (* which begins the comment. The $ character is then followed by one or more letters which represents a precise compiler directive. The compiler directives are used to modify the behaviour of the compiler. Each of the compiler directives and their effects are described in appendix B.

Examples of compiler directives:

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

CHAPTER 12: BLOCKS, VISIBILITY, AND ACTIVATION

DEFINITION OF A BLOCK

A BLOCK is the basic unit of Pascal source code. It consists of a declaration part and a statement part. The declaration part contains zero or more declarations which can appear in any order. The statement part is a sequence of statements and it always follows the declaration part.

Each block is a part of a procedure, function, program or module (unit). All identifiers and labels declared in the declaration part of a block are known as locals. The program block contains all the other blocks; consequently, the declarations in the program block are known as globals.

The label declaration part defines the labels that are used to mark statements in the corresponding statement part. Each label must mark exactly one statement in the statement part.

The constants declaration part contains the local constant declarations for this block. (See “Declarations of Constants” in chapter 11).

The type declaration part contains the local type declarations for this block. (See chapter 13).

The variable declaration part contains the local variable declarations for this block. (See chapter 10).

The procedures and functions declaration part contains the local procedure and function declarations for this block. (See chapter 17).

RULES OF VISIBILITY

Visibility of a declaration

The appearance of an identifier or a label within a define declaration means that the identifier or label is associated some significance at the point of the declaration. All other occurrences of this identifier or label will only appear within the "visibility" of this declaration. A declaration is visible only to the block that contains it and all the blocks which are contained within it, with the exceptions being described in the following paragraphs.

Redeclaration in an inner block

Let us suppose that EXTERIOR is a block and that INTERIOR is another block declared inside EXTERIOR. If an identifier declared in block EXTERIOR has the same name as an identifier declared in the INTERIOR block, then the INTERIOR block and all its inner blocks are excluded from the visibility of declarations made within block EXTERIOR.

Position of the declarations in a block

The declaration of an identifier or a label must precede the occurrences of this identifier or labels within the text of a program. i.e. identifiers and labels cannot be used without being declared. There is one exception to this rule: in a type declaration, the field type of a pointer can be an identifier that has not been declared yet. In this case, the identifier must be declared somewhere in the same declaration as the pointer type.

Redeclaration inside a block

An identifier or label cannot be declared more than one once in a block, unless it is declared inside an inner block or if it appears in the list of fields declared for a record.

A records field type is declared within a record type. It has significance only in reference to a variable of this record type. Consequently, a field identifier can be declared inside the same block containing another identifier of the same name as long as it was not previously declared in the same list of fields. In a similar manner, an identifier that has been declared as a record field identifier can also be re-used within the same block.

Identifiers of standard objects

TML Pascal II provides a library of predeclared types, procedures and functions which act as if they had been declared in a block containing the whole program. They are visible throughout the whole program or module (see chapters 9 and 10, where you will find each of the standard identifiers within TML Pascal II).

Visibility of the interface modules and module identifiers

Programs, modules, module specifications, and module bodies containing a USES clause, see the additional identifiers contained within each module listed in the USES clause. These identifiers act as if they had been declared in the same block that the USES clause appeared in.

ACTIVATION

The execution of a block means that it is in activation. At a given time, a block can have zero or more activations. If a block is not the target of execution it has zero activation. If a block is the target of execution, it has at least one activation. When a block has more than one activation, it is said to be recursive.

CHAPTER 13: VARIABLE TYPES

When you declare a variable you must indicate its type. The type of a variable determines the range of values that the variable can be assigned and also which operations can be performed on it. A type declaration introduces an identifier to specify a new type.

When an identifier appears to the left of a declared type, it is declared as an identifier of the (declared) type for the block in which the type declaration takes place. The visibility of a type identifier is not included the same, except for the pointer types (?? TODO: figure out what this means).

SIMPLE TYPES

All the simple types define a collection of ordered values.

An integer type identifier is one of the standard identifiers - Integer or LongInt. A real type identifier is another of the standard identifiers - Real, Single, Double, Comp or Extended. See “Numbers” in chapter 11 on how to define integer constants and real type values.

Ordinal types

The ordinal types are a subset of the simple types which have the following particular characteristics:

- ordinal types are ordered groups with each value having an ordinal value which is a whole number. Except for integer types, the first ordinal value of an ordinal type is zero, the next a 1, etc…. For integer types, a value’s ordinal number is the value itself. Each value of an ordinal type, except for the first, has a unique predecessor based on the ordinal number of the type. Moreover, each value of an ordinal type, except for the last, has a successor based on the ordinal number of the type.
- the standard functions ORD and ORD4 can be applied to any value of an ordinal type and the ordinal number of the value will be returned.
- the standard function PRED can be applied to any value of an ordinal type and it will return the predecessor value.
- the standard function SUCC can be applied to any value of the ordinal type and it will return the successor value.

TML Pascal II has four preset ordinal types: Integer, LongInt, Boolean, and Char. Additionally, there are two classes available for the user to define ordinal types: Enumerated types and Interval types.

The standard ordinal types

Integer

The values of the integer type are a subset of all numbers. A variable of the integer type can have a value within the range `(-maxint-1 .. maxint)`, i.e. -32768 to 32767. The standard integer constant *maxint* is defined as 32767.

LongInt
The values of the long integer type are also a subset of all numbers. A variable of the long integer type can have a value within the range `(-maxlongint-1 .. maxlongint)`, i.e. -21474836478 to 2147483647. The standard constant *maxlongint* is defined as 2147483647. Arithmetic operations with integer type operands use integer precision (16 bits) or long integer precision (32 bits) according to following rules:

- integer constants that fall within the range of the Integer type are considered an *Integer* type. Any other integer constants are considered to be of *Longint* type.
- if operator and operand are of *Integer* type the result is of *Integer* type (possibly truncated to 16 bits). In the same manner, if two operands are of *Longint* type a precision of 32 bits will be used and the result will be of type *Longint*.
- when one operand is of type *Longint* and the other is of type *Integer*, the *Integer* operand is initially converted into a *Longint* and the result is of type *Longint*.
- the expression located to the right of an assignment statement is evaluated independently of the left part.

An *Integer* value can be explicitly converted into a *Longint* by using the standard function ORD4 which is described in chapter 20.

Boolean
The values of a Boolean type are specified by using the predeclared constant identifiers FALSE and TRUE, where `ord(false)=0` and `ord(true)=1`. Boolean values are used by the Pascal statements IF, REPEAT and WHILE.

Char
The character type is a collection of values made up of ASCII characters. By calling the function ord (CH), where CH is a character value, returns the ordinal number of CH. Constant strings of length 1 can be used to assign the value of a character type constant, and any value of the character type can be generated via *CHR* function.

Enumerated types
An enumerated type defines an ordered collection of values which are referred to by identifiers. The order of these values is determined by the order in which the identifiers are listed. Consequently for two enumerated identifiers X and Y, if X precedes Y then the ordinal number of X is lower than that of Y.

When an identifier appears inside a list of identifiers for 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 ordinal number of an enumerated constant is its position in the identifier list, taking into account that the ordinal number for the first enumerated constant is always 0.

Examples of enumerated types:

```
work_days = (Monday, Tuesday, Wednesday, Thursday, Friday)

colour = (red, yellow, green, blue)
```

Here, yellow is an enumerated constant of the *colour* type with a sequence number of 1. Friday is an enumerated constant of the *work_days* type with a sequence number of 4, and so on… and therefore we can state the following:

```
ORD (Monday) < ORD (Tuesday)
ORD (Thursday) > ORD (Tuesday)
PRED (green) = yellow
SUCC (red) = yellow
```

Interval types (also known as 'Subrange types')

A defined interval type is a subset of ordinal type values called the host type. The definition of an interval type specifies the minimum and maximum values of the interval.

The two constants of an interval type must be of the same ordinal type. Intervals types of the form has A. .B require that A be less than or equal to B. An interval type variable has all the properties of the host type variables, with the added restriction that its value must always be within the range defined by the interval type.

Examples of the intervals types:

```
1..100
-128..127
Monday..Thursday
```

Real types

The real types are collections of values that are subsets of real numbers, which can be represented in floating point notation using a given number of digits. In general, the floating point notation of a value *n* can be broken down into three components *m*, *b*, and *e* such that n = m * (b^e), where *b* is always 2 and *m* and *e* are whole values falling within the range of the real type. The values *m* and *e* will therefore determine the range and precision of real types.

There are four standard real types in TML Pascal II: single, double, comp and extended. Additionally, the standard identifier for reals is defined as being of the extended type. Real types are different in range and precision from the actual values that they represent.

The real types:

Type identifier	Memory occupied	Range
Single	4 bytes	1.4^{-45} to 3.4^{38}
Double	8 bytes	5.0^{-324} to 1.7^{308}
Real / Extended	10 bytes	1.9^{-4951} to 1.1^{4932}
Comp	8 bytes	-9.2^{18} to 9.2^{18}

The possible real values are:

- Finite values (a subset of all real numbers). The value 0 has an associated sign (it can be negative or positive)
- Infinite values, +INF and - INF resulting from capacity overflow or a divide by zero.
- NaN (Not a Number) represents the result of an operation that can not be represented numerically (multiplying by ±0 for example). NaN is represented by NaN(x) where X is an Integer defining the source of NaN.

Reals are implemented in TML Pascal II by the SANE tool. Operations on reals are also made possible by using the interface SANE.P.

Real type values are converted into *extended* type before an operation is performed on them, and the result of these operations is always in extended type. An extended value can always be used where a *single*, *double* or *comp* is required, provided that the value falls within the corresponding range.

The real type values are converted into extended type by the compiler before calculations are performed so that the maximum precision can be obtained. Consequently, calculations performed on variables already in extended type will be performed more quickly and provide more compact code than calculations on data stored in other representations due to the fact that no conversion is required. The smaller representations should be used when data storage space is more critical than the speed of execution.

STRUCTURE TYPES

A structure type is characterised by its structure and the type of its components. The type of a component is allowed to be of the same type as structure itself. There is no inherent limit to the number of levels within a structure.

The use of the word PACKED in a structure type declaration indicates that the component variables for this type will be compression to save memory, even if doing so makes access to component variables less effective. Note that you cannot use packed variable components as parameters to procedures or functions. The compaction is only performed at the byte level, not at the bit level. For more regarding this, see Appendix D.

Array types

An array type defines a structure that has a set number of components, and all of the components are of the same type.

```
ARRAY[ <INDEX TYPE> ] OF <COMPONENT TYPE>
```

The type which follows the word OF defines the type of the arrays components. The number of elements is specified by one or more index types. Each index type adds a dimension to the array. The index type must be an ordinal type.

There is no limit to the number of dimensions of a array; however, TML Pascal II limits arrays to being no greater than 32,767 bytes.

An array of the form:

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

is regarded as a packed string type. A packed string type has certain properties that are not applicable to other array types [see the identical and compatible arrays later in this chapter].

Examples of array types:

```
array[1. . 100] of real
```

```
packed array [colour] of Boolean
array [Boolean] of integer
array[low..high] of Boolean
```

If the component type of an array is also an array, the resulting type is either an array of arrays or a multi-dimension array. For example:

```
array[boolean] of array[0..maxsize] of real
```

is equivalent to:

```
array[boolean, 0..maxsize] of real
```

A component of an array can be accessed by using indexes to the component inside brackets immediately following the array identifier. One can use several sets of brackets. For example:

```
var anArray: array[1..maxlength,1..maxwidth] of real
```

permits the following:

`anAray[1,1]` or `anArray[1][1]` will access the first element of the first sub-array.

Notice that `anArray[2]` will return the complete second sub-array.

Record types

A record type consists of a specified collection of components called fields, each one capable of being a different type. Each field of a record type must specify its type, and the name of its identifier.

The fixed part of a record type specifies the list of fields by giving an identifier and a type for each field. Each field and its associated data is accessible by using a variable of the record type in question.

Example of the recording type:

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

The variable part of a record is made up of lists of "alternate" fields which are allocated in the same memory space as a variable record, thus the data stored at this located can be accessed in various ways. Each of the lists is called a variant. The variants are replaced in memory and the variant fields are accessible at all times.

Each variants value is determined by one or more constants. The constants must be distinct and be of an ordinal type compatible with the label field of the variant. The variant part allows an optional identifier which announces a field label. If a field label is present, it is regarded as a field within the preceding fixed part.

Examples of record types with variants:

```
record
      surname, first_name: string(80);
      age : 0 . . 99;
      case married: boolean of
            true : (maiden_name: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 of an ordinal type known as the base type. The possible values of a set type are a subset of the possible values of the base type.

TML Pascal II limits the base type to 256 possible values. If the base type is a subset of Integer, it will be limited to a range of 0..255. For more information regarding memory allocation and the representation of data, see Appendix C.

Each set type can have the value *()* which is called the empty set.

Examples of sets:

```
set of Char
set of 0..31
set of (red, green, blue)
```

File types

A file type is a structure type comprising a linear succession of components of the same type. The type of a component can be any type which is not a file type or a structured type containing a file type component. The number of components is not specified in a file type declaration.

The standard “text” file type is a particular packed file of characters, organised in lines. These character files are handled by specific I/O procedures explained in chapter 19.

Because of the representation of types in TML Pascal II, accessing the elements of a character file (text) uses 16 bit words, while in a packed character file one accesses elements using 8 bits words. For more detail, see Appendix D.

Examples of file types:

```
IntFile = file of integers
```

TML Pascal II allows the passing of file type variables into procedures or functions only if they are parameters.

STRING TYPES

A string type is a succession of characters having a dynamic length attribute and a constant dimension attribute of 1 to 255. The constant dimension is a maximum limit applied to the length of values of this type. If the dimension attribute is not specified, then it will be given a default value of 255. The current value of the length attribute for a string type is returned by the standard function *length*. A null string is a string type value that has a dynamic length of zero.

The sort order between two string values is determined by the comparison of characters in the corresponding positions. When the two strings are of different lengths, each character of the longest string will not correspond to a character of the shorter string, and in these cases where the common parts are equal, the longer string will be considered of higher value. For example, 'attribute' is larger than 'at'. Two strings must therefore have the same length to be considered equal.

Strings are stored in the format of a byte indicating the length of the string, followed by all the characters of the string. One can directly modify the length of strings by modifying their first byte:

```
aString(0) := chr(5);
```

The operators applicable to strings are examined in chapter 15; and the standard procedures and functions for string handling are described in chapter 20.

String examples:

```
string(50)
string(255)
string
```

POINTER TYPES

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

If the identifier of the base type is not yet declared, it must be declared in the same type declaration section 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 storage area on the stack for the dynamic variable, and sets the pointer value to be the memory address allocated. The operator '^' directs the pointer variable to the memory area containing a referenced variable. The 'POINTER' function points the pointer variable to a particular memory address.

The predeclared identifier constant 'NIL' represents a constant pointer value which is a possible value for all pointer types. By design, the Nil pointer does not point to anything.

Examples of the pointer types:

```
^LongInt
^Char
^String(32)
```

IDENTICAL AND COMPATIBLE TYPES

Two types *are,* or *are not* "identical". In certain contexts it is necessary that two types be identical. At other times, if the types are not identical it may be that they must be compatible, and at other times again, assignment compatibility is required.

Identical types

Identical types are necessary in the following contexts:

- between formal parameters and the current variable
- between formal result types and the current function parameters
- between values of formal parameter variables and the current list of function or procedure parameters
- when a one dimensional array PACKED ARRAY OF CHAR is compared with another using an operator

Two types, T1 and T2 are identical if one of the following statements is true:

- T1 and T2 have the same type identifier
- T1 is declared as being equal to a type identical to T2

Compatible types

Assignment compatibility is required when a value is assigned to anything, either explicitly (as in an assignment statement) or implicitly (like the passing of parameter values).

A value of type T2 is deemed compatible with type T1 if one of the following statements is true:

- T1 and T2 are identical types and they are neither of file type nor of a structured type containing an element of file type
- T1 is a real type and T2 is an integer type
- T1 and T2 are ordinal types and the value of T2 falls within the range of possible values for T1
- T1 and T2 are set types with a compatible base type and the value of all the elements of type T2 are within the range of possible values of type T1
- T1 is a string type or a character type, and T2 is a string type or character constant within apostrophes
- T1 is a packed string type of N elements and type T2 is a string type of constant characters within apostrophes also having a length of N.

There is an assignment error if compatibility is required, but none of the above statements are true.

CHAPTER 14: VARIABLES

DECLARATION OF VARIABLES

A variable declaration is used to allocate and associate a block of memory with a particular type. A variable is an entity in which a value can be stored. Each identifier in the list of identifiers within a variable declaration shows that each variable has its type specified in the declaration.

The presence of an identifier inside the list of the identifiers of a variable declaration, announces that there is a variable identifier for the block in which the declaration appears. One can therefore refer to the variable throughout the block unless the identifier is not redeclared in a sub-block. A redeclaration creates a new variable using the same identifier without affecting the value of the original variable.

Examples of variable declaration:

```
x,y,z: real;
c: color;
p1,p2: nobody;
today: date;
operator: [plus,minus];
digit: 0..9;
coord: polar;
fact,error: boolean;
```

REFERENCE VARIABLES

A reference variable may be returned to a complete variable, a component of a structure type or string, a dynamic variable pointed to by a variable of the pointer type, or to a variable that one accesses by way of a function call.

Qualifiers

The reference to a variable is made up of a variable identifier followed by zero or more elements which modify the meaning of the variable thus referenced.

For example with:

```
var aMultiDimArray: array[1..100] of array[1..100] of integer
```

One can write:

`aMultDimArray` - to access the whole array
`aMultDimArray[1]` - to access the first sub-array
`aMultDimArray[1,1]` to access the first element of the first sub-array

Tables, strings and indexes

A single element of an array variable can be accessed by using a reference to an array type variable followed by an index value that specifies the element concerned. In a similar

way, a particular character in a string variable is specified by using the reference variable followed by an index defining the position of the character.

Examples of indexed arrays:

```
m(i,j)
a(i+j)
```

Each index value selects an element within the array at the corresponding dimension. The number of index values should not exceed the number of indexed types in the table declaration. The index value must be assignment compatible with the corresponding index type.

When one indexes a multidimensional array, you can either multi-index or use multiple expressions to access an element. These methods of access are interchangeable.

For example:

```
MaMatrice(i)(j) (* has the same meaning as *)
MaMatrice(i, j)
```

A string variable can be indexed with a simple expression, whose value must be in the range *0. .n*, where *n* is declared as the length of the string. Indexing a string gives access to a particular character of the string. The first character of a string variable (index 0) contains the dynamic length of the string. In general, one cannot assign a value to a position within the string unless the particular character position is already occupied. i.e. if the dynamic length of the string is less than the position of the indexed character being assigned, the operation will leave the string unchanged. For additional information, see chapter 20.

Records and fields designators

A given field of a record variable is specified by using a reference to the record variable followed by the appropriate field designator.

Examples of field designators:

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

It is an error to access a variant component from an uninitialised record. See chapter 13.

Inside a WITH statement, a field designator does not have has to be preceded by a reference to the record variable.

Dynamic pointers and variables

The value of a pointer variable is either NULL or a value pointing to a dynamic variable.

The dynamic variable pointed to by the variable pointer is referred to by writing the variable pointer immediately followed by a caret character (also known as a circumflex).

Dynamic variables and pointer values that are not null, are created using the standard procedure NEW. Additionally, the operator '@' (at) and the standard procedure POINTER can be used to create pointer values which in fact do not point to dynamic variables but are treated as if they do.

The NULL constant does not point to any variable. An error will be raised if you try to access a dynamic variable when the value of the pointer is NULL or undefined.

Examples of references to dynamic variables:

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

Variable type modification (Variable type casts)

The reference to a variable of a given type can be changed into a reference to a variable of another type thanks to a mechanism called 'variable type casts'.

When a type modification is applied to a reference variable, the variable is temporarily treated as being of the type specified in the given type identifier. The variable's dimensions (i.e. the number of bytes that it occupies in memory) must be the same as the dimensions of the type specified by the type identifier. A variable type cast can be followed by one or more qualifiers in the same manner as a reference to a traditional variable.

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 15: EXPRESSIONS

Expressions indicate values. The simplest expression for example, is the reference to a variable. However, most expressions are made up operators and operands. The majority of Pascal operators are binary, meaning that they have two operands. The other operators are unary and have only one operand. When more than one operator appears in an expression, priority rules are applied to determine which operands are associated which operators. For example, the expression:

```
a + b * c
```

can either be interpreted as `(a+b)*c` or as `a+(b*c)`. Priority rules remove any ambiguity in the interpretation:

- when an operand appears between two operators of different priority, it is associated with the operator having the greatest priority
- when an operand is written between two operators of the same priority, it is associated with the left operator
- an expression between brackets is always evaluated before applying any operator to it

Operator Priority:

Priority	Operator Type	Operators
First (highest)	Unary operators	@, NOT
Second	Multiplying operators	*, /, DIV, MOD, AND
Third	Addition operators	+, -, OR
Fourth (lowest)	Relational operators	=, <>, <, >, <=, >=, IN

Therefore, the expression `a+b*c` is interpreted as `a+(b*c)` because `*` has a higher priority than +. Note that `a+b-c` is interpreted as `(a+b)-c` because `+` and `-` have the same priority. The priority rules follow the syntax of expressions, which are constructed from factors, terms and simple expressions.

The result of an expression is obtained by applying relational operators to simple expressions:

```
Expression --> SimpleExpression [RelationalOperator SimpleExpression]
RelationalOperator --> '=' | '<>' | '<' | '>' | '<=' | '>=' | 'IN'
```

Example of expressions:

```
x=1.5
C in tint1
result <> error
p<=q
```

The simple expression syntax is the result of applying adding operators and signs to terms:

```
SimpleExpression --> [SignOperator] Term | SimpleExpression AddingOperator Term
```

Examples of simple expressions:

```
x+y
-x
tint1 + tint2
b or c
```

The syntax of a term is the result of applying multiplication operators to factors:

```
Term --> Factor | Term ('*' | '/' | '**' | 'div' | 'mod' | 'and’) Factor
```

Examples of terms:

```
x*y
e/(1-e)
result and error
```

The syntax of a factor is the result of the following constructions:

```
Factor -->
      @ |
      Numeral |
      VariableAccess |
      '(' Expression ')' |
      Not Factor
VariableAccess -->
      VariableNameUse |
      VariableAccess '[' Expression ']' |
      VariableAccess '.' FieldNameUse
```

Examples of factors:

```
x (variable reference)
@x (pointer to a variable)
15 (unsigned constant)
'Hello' (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

Operators are classified as either arithmetic operators, Boolean operators, set operators, relational operators, or '@' operators.

Arithmetic operators

The following two tables show the types of operands and the results returned using binary and unary operators respectively.

Table: Binary arithmetic operations (on two elements)

Operator	Operation	Operand type	Result type
+	Addition	Integer or Real	Integer or Real (1)
-	Subtraction	Integer or Real	Integer or Real (1)
*	Multiplication	Integer or Real	Integer or Real (1)
/	Division	Integer or Real	Real (2)
DIV	Whole Division	Integer	Integer
MOD	Modulus	Integer	Integer

(1) if a mixture of integer and real operands are used then the integer operand is converted into a real and the result is of type real.
(2) the integer operands are always converted into real values even if the two operands are of integer type.

Table: Unary arithmetic operations (on a single element)

Operator	Operation	Operand type	Result type
+	Sign Identity	Integer or Real	Integer or Real
-	Sign Negation	Integer or Real	Integer or Real

If the operators +, -, *, div, or MOD are applied to two operands of the same integer type, (*Integer* or *Longint*) the result will always be in the same integer type as the operands. If one of the operands is of *Longint* type and the other of *Integer* type, then the *Integer* operand is initially converted into a *Longint* and the result is of *Longint* type. In all other cases, the value of the result is determined by the normal rules of integer mathematics. An error will be raised if the value of the result is outside the range:

`(-maxint-1)..maxint` for *Integers*
`(-maxlongint-1)..maxlongint` for *Longints*

If one of the operands of the operators '+', '-', or '*' is of a real type, the result is always of *Extended* type and its value is an approximation of the true mathematical result. The result of any operation using the '/' operator is always of *Extended* type.

If the operand of a sign identity or a sign negation is of an integer type, the result will always be of the same type of integer, and the absolute value of the result will always be equal to the absolute value of the operand.

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

Boolean operators

The operand types and the corresponding results for Boolean operations are shown in the following table:

Table: Boolean operations

Operator	Operation	Operand type	Result type
OR	Logical OR	Boolean	Boolean
AND	Logical AND	Boolean	Boolean
NOT	Logical Negation (Unary)	Boolean	Boolean

The result of a Boolean operation is determined by the rules of Boolean algebra, for example `(A and B)` is evaluated as being true if and only if A is true and B is true.

Set operators

The operand types and the corresponding results for Set operations are shown in the following table:

Table: Set operations

Operator	Operation	Operand type	Result type
+	Union	Set Type	If the set types of the operands are the same then the result will be the same set type, otherwise it will be a compatible set type.
-	Difference	Set Type	
*	Intersection	Set Type	

The results of set operations are determined by the normal rules for sets:

- an ordinal value C is in the set A+B if and only if C is in A or 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 B

Relational operators

The operand types and the corresponding results for Relational operations are shown in the following table:

Table: Boolean operations

Operator	Operand type	Result type
= <>	Compatible simple, pointer, set, string, or packed string type	Boolean
< >	Compatible simple, string, or packed string type	Boolean
<= =>	Compatible simple, set, string, or packed string type	Boolean
IN	Left operand is an ordinal of type T, and the right operand is a set of type T	Boolean

Comparison between ordinal types

When the operands of the operators '=', '<>', '<', '>', '>=' or '<=' are ordinal types they must be compatible types. The only exception to this rule is where one of the operands is a real type and the other is an integer type. The result is the mathematical relation of ordinal types. When real types are compared, the results can be different from that of ordinal types due to the fact that the representation of a real value is only an approximation.

Comparison between strings

When the relational operators '=', '<>', '<', '>', '<=', and '>=' are used to compare strings, they are compared according to their lexicographic order. The length of a string does not matter when it is the subject of a comparison because all string values are compatible. What's more, a *Char* type value is also compatible with a *String* type value, and when a comparison is made, the *Char* value is treated like a *String* of length 1. When a *Packed String* value which has N elements is compared with a *String* type value, it is treated as a *String* type value with a length of N.

Comparison between packed strings

The relational operators '=', '<>', '<', '>', '<=' and '>=' can also be used to compare two packed string values if both have the same number of elements. If the number of elements is N, then the result is the same as if the two strings were of *String* type and of length N. See also chapter 13 on Packed Array of Char.

Comparison of sets

If A and B are the 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 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 respectively indicate equivalence and non-equivalence of the sets A and B, and A<=B and A>=B respectively indicate the inclusion of A in B and the inclusion of B in A.

Comparison of pointers

The relational operators = and <> can be applied to operands of compatible pointer types. Two pointers are equal if and only if they point to the same object (ie. they point to the same memory address).

Testing for set membership

The operator *in* returns a value of true if the ordinal type value operand (LHS) is a member of the set type operand (RHS); if not, it returns a value of false. The type to the left of the operator must be compatible with the base type of the right-hand side of the operator.

The @ operator

A pointer value which points to a variable, procedure or function can by created using the @ operator. The types of the operands and results are shown in the following table.
@ is a unary operator that takes the reference to an identifier of a variable, procedure, or function to use as its operand and calculates the value of its pointer. If the operand of the @ operator is a reference to a variable, then the value of the pointer is the memory address of the variable. If the operand of the @ operator is a procedure or function identifier, then the value of the pointer is the entry point of the procedure or the function. The type given to the calculated pointer is equivalent to the type of the NIL pointer. Because of this, it can be assigned to any pointer variable.

Table: @ operation

Operator	Operation	Operand type	Result type
@	Pointer Creation	Reference to a variable, procedure, or function.	Pointer Type

The @ operator can only be used with procedures and functions declared in the declaration part of the program or module (global declarations) when the resulting pointer value is passed to an Apple IIgs ROM routine. The procedures and functions declared in the declaration part of another procedure or function (embedded declarations) have a different calling convention which is not compatible with the Apple IIgs ROM routines. Additionally, the TML Pascal II compilation option 'DefProc' will have to be used when the resulting pointer value is passed to an Apple IIgs ROM routine. See appendix "Inside TML Pascal II" for details on Pascal call conventions and the use of the "DefProc" compilation option.

@ with variables

The use of the @ operator with ordinary variables (not parameters) is straight forward. For example, if you have:

```
type twochar = packed array (0..1) of Char;
var int: Integer;
twocharptr: ^twochar;
```

then you can write:

```
twocharptr: = @int;
```

this forces *twocharptr* to point to the variable *int*. Since the *Integer* and *twochar* types have the same storage type, the value of *int* accessed via *twocharptr* is interpreted as being of *twochar* type.

@ *and parameter values*

If @ is applied to a formal parameter value, the result is a pointer to the stack address containing the actual parameters value. If *aParam* is a formal variable for a procedure, and *actParam* is the variable passed into the procedure as the actual variable *aParam*, and *aPtr* is a pointer; and if one performs the following:

```
aPtr := @aParam;
```

then *aPtr* is a pointer to *actParam* on the stack, and *aPtr^* will give the value of *actParam*.

@ *with a parameter variable*

If @ is applied to a formal parameter variable, the result is a pointer to the actual parameter. In this case, the pointer is taken from the stack. If *aParam* is a formal variable parameter for a procedure and *actParam* is the variable passed into the procedure as the actual parameter *aParam*, and *aPtr* is a pointer; and if one performs the following:

```
aPtr := @aParam;
```

then *aPtr* is a pointer to *actParam* and *aPtr^* will give the contents of *actParam*.

@ *with a procedure or function*

When @ is used with a procedure or a function, the result will be a pointer to its entry point. It is not possible to use such a pointer within Pascal itself. The only possible use of a pointer to a procedure or function is for passing it into a ToolBox procedure so that the IIgs ToolBox can call the function indicated within a JSL type ASM instruction. Pointers to procedures are generally used to implement the definition procedures and filter procedures.

The @ operator must be only used with procedures or functions declared in the declaration part of a program or unit (global declarations) when the resulting pointer value is to be passed to a ToolBox routine. Procedures and functions declared within a procedure or function (overlapping declarations) have different calling conventions making them incompatible with the IIgs ToolBox routines.

FUNCTION CALLS

A call to a function specifies that the block associated with the function identifier is to be activated. The result returned by an activated function is used like an expression value. If the function has formal parameters, then the function declaration must contain the corresponding list of parameters to be used. Each actual parameter is substituted for the corresponding formal parameter. Examples of function calls:

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

See chapter 17 for a description of procedure calls.

SET CREATION

The creation of a set specifies the values of a set type and it is realised by writing expressions surrounded by square brackets. Each expression specifies a value of the set.

If there is nothing within the pair of square brackets, then the set is empty. It is possible to assign the empty set to all *set* types. An elements such as `x..y` means that all the values between `x` and `y` are included within the set. If the value of `x` is greater than the value of `y`, then `x..y` means no element and `(x..y)` is an empty set.

The values used in an element range expression (e.g. `x..y`) for creating a set, must be ordinal type compatible. If `a` is the smallest ordinal value of the resulting set, and if `b` is the largest ordinal value of the set, then the base type of the resulting set is `a..b`.

Examples of sets:

```
(red, C, green)
(1, 5, 10..k MOD 12, 23)
('A'..'Z', 'a'..'z', CHR(13))
```

VALUE TYPE MODIFIERS (Value Type Casts)

The type of an expression can be changed into another type by using a value type modifier.

The argument of the expression must be of an ordinal or pointer type. The result of a value type modifier is of the specified type with its ordinal value calculated by converting the expression. The syntax of a value type modifier is almost identical to that of a variable type modifier. However, the value type modifier operates on values and not on variables and therefore cannot handle references to variables. Because of this, a value type modifier can not have a qualifier on left hand side of an assignment declaration, nor can it have an actual parameter where the formal parameter is declared as `VAR parameter`.

Examples of value type modifiers:

```
integer('c')
ptr($89F2)
boolean(0)
```

CHAPTER 16: STATEMENTS

Statements describe the algorithmic actions able to be executed. There are two classes of statements: simple statements and structured statements. Statements can be prefixed by a label which can be referred to by a GOTO statement. A label is a positive integer constant, which must first be declared in the label declaration section (see chapter 11).

SIMPLE STATEMENTS

A simple statement is a single statement that does not contain any other statement. The empty statement is a simple statement not containing any symbols and not specifying any action.

Assignment statements

The assignment statement makes it possible to carry out one of two actions:

1. to replace the current value of a variable by a new value specified in an expression
2. to specify an expression whose value will be returned by a function

The symbol := can be read as 'give the value'. The expression must be compatible, in terms of assignment, with the type of the variable or the result type of the function. The function identifier must be that of the current function being executed.

The reference to a variable on the left can identify a variable of any type except FILE. With most variables, the reference to the variable is just an identifier name, but in certain cases the variable may be followed by a qualification:

- **strings**: the name of the variable is followed by an index value between brackets
- **arrays**: the variable is identified by the name of the array followed by an index between brackets for each dimension of the array
- **record fields**: the name of the field variable must be preceded by the structure name and a point. The only exception to this is when a field variable is referenced within a WITH statement loop for the corresponding record
- **dynamic variables**: the reference to a variable is denoted by the pointer name followed by a '^'

Moreover, a reference to a variable located on the left hand side of an assignment can be a type definition. See chapter 13 for more information on the compatibility of types and the syntax of these variables.

Examples of assignment:

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

It is not specified if the evaluation of the reference to the variable is carried out before or after the evaluation of the expression. Nevertheless, when the reference to the variable is established, it is not impacted by any flow on effects due to the execution of the rest of the assignment statement. Therefore:

```
A[i] := A_Function(i);
```

is related to whether *i* is modified by the function, and if this is the case, according to whether the function is evaluated before or after *A[i]*.

Procedure statements

A procedure statement makes a call to a code block specified by the procedure identifier (ie. performs a call to a subroutine). If the procedure has formal parameters, then the procedure statement must contain the corresponding list of actual parameters. Each actual parameter is replaced by the corresponding formal parameter in the list, at the time of the procedure call.

The identifier used to define a procedure must be identical to that used in the declaration. Parameters in the declaration of a function or procedure are called “formal parameters”; those in the call statement are called “actual parameters”. The values in the list of actual parameters will be passed into the formal parameters at the time of the procedure or function call. The number of formal parameters listed must be equal and they must be compatible with the actual parameters. There are 3 exceptions to this compatibility:

- the sub-types are equivalent to their base types
- a formal parameter of *LongInt* type accepts a actual parameter of Integer type
- the formal parameters preceded by UNIV accept any actual parameter occupying the same memory space. See chapter 19 for more about UNIV.

The actual parameters specified in a call to a function or procedure must comply with the following rules:

- the actual parameter variables, unlike parameter values, must be variables. Actual parameter variables cannot be constants, expressions, packed elements or variables.
- the value of a string parameter variable can be passed into any string parameter variable, without concern length of the string.
- if the value of an actual parameter exceeds the defined range of a formal parameter, an execution error will be produced.

STRUCTURED STATEMENTS

Structured statements consist of a number of different statements that are carried out conditionally, repeatedly, or sequentially.

Sequential statements

This type of statement defines a sequence of statements that will be executed in the order that they are written.

In Pascal the body of a procedure, function, or MAIN program is simply a succession of statements. Each sequence starts with a BEGIN and is concluded by an END. Each individual statement located between these keywords ends in a ‘;’. A sequence of statements can overlap with other sequences (nested blocks). In these cases, each END is paired with the nearest preceding BEGIN.

Example:

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

Conditional statements

A conditional statement determines which of its components (if any) will be executed.

The IF statement

The IF statement carries out one or more statements (can be a sequence of statements) if a Boolean expression is TRUE. An optional ELSE clause can be added so that a different statement will be executed when the Boolean expression is FALSE. The statement associated with ELSE clause, can also be a sequence of statements.

IF statement

```
if boolean_expression then
        statement1
else
        statement2
```

The expression between the IF and THEN is usually made up of logical and relational operators and it must always be of *boolean* type. It is possible to have overlapping IF statements. In these cases, each ELSE is always associated with the nearest IF that it is not already associated with another ELSE.

Examples:

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

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

The CASE statement

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

CASE statement

```
case ordinal_expression of
  case_constant: statement;
  ...
  case_constant: statement;
otherwise
  statement;
  ...
  statement;
end
```

The CASE statement executes the statement(s) prefixed by the CASE constant that is equal to the selector value. If no CASE constant meets this criterion, and the OTHERWISE clause is present, then the statement(s) following the OTHERWISE clause are executed; if no OTHERWISE clause is not present, then execution continues with the statement following the CASE statement. Each statement in the CASE or OTHERWISE clause can be a sequence of statements.

Examples:

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

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

Repetition statements

These indicate that a sequence of statements must be repeated a certain number of times.

If the number of repetitions is known beforehand, the FOR statement is most suitable; if not, then REPEAT and WHILE statements are used.

The REPEAT statement

The REPEAT statement contains an expression that controls the number of repetitions that occur for the sequence of statements that fall between the REPEAT and UNTIL keywords.

Repeat statement

```
repeat
  statement;
  ...
  statement;
until boolean_expression
```

The expression must generate a result of boolean type. The statements between the keywords REPEAT and UNTIL are executed repetitively until the expression is evaluated as being TRUE. The statement sequence is executed at least once, because the expression is evaluated after the execution of the sequence. REPEAT and UNTIL do not require the use of BEGIN and END keywords to delimit the sequence.

Examples:

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

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

The WHILE statement

The WHILE statement contains an expression that controls the number of repetitions that occur for a statement.

WHILE statement

```
while boolean_expression do
  statement
```

The expression must generate a result of boolean type and is evaluated before each loop is executed. The statement is executed repetitively as long as the expression returns a value of TRUE. If the expression returns FALSE then the loop will be ended and the statement no longer executed. The statement controlled by a WHILE statement can be either a simple statement, or a sequence of statements.

Examples:

```
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 := x*x
end
```

NOTE: Ensure that the boolean expression changes to an appropriate value within the REPEAT and WHILE statements, otherwise an endless loop will result. You can also leave a loop by using GOTO, EXIT or LEAVE (see later).

The FOR statement

The FOR statement causes repetitive execution of a statement whilst the index value remains within a defined range. This index value progresses sequentially for each execution of the loop.

FOR statement

```
for variable := initial_value ( to | downto ) final_value do
  statement
```

The first expression following the := is called the INITIAL VALUE. The second expression following the := is called the FINAL VALUE. The control variable, initial value, and final value must be compatible. See chapter 13 regarding the compatibility of types.

A control variable is of scalar type (Integer, Char, boolean, subrange, or a type defined by the user). However, it cannot be an array, a string, a record field, or a dynamic variable. The control variable must be declared in the block containing the FOR statement. A new value is assigned to it by the FOR statement before each loop execution. The value of the control variable is accessible within the statement managed by the FOR loop.

If the reserved word after the first expression is TO, the control variable will be incremented before each loop execution. The execution of a FOR statement continues until the control variable has a larger value than the final value allows.

If the reserved word after the first expression is DOWNTO, the control variable will be decremented before the start of each loop, and process execution continues until the control variable has a lower value than the final value allows.

Here are some rules to respect when defining a FOR statement:

- The control variable must be a simple variable declared in the local scope.
- If the control variable is of subrange type, or a user defined scalar, it must accept all values ranging between the initial value and the final value.
- The control variable should not be modified within the loop managed by the FOR statement.

- The control variable can not be included in the expressions defining the initial or final values.
- The control variable can not be specified at the end of a FOR statement.
- The initial and final values are evaluated only one time, before the execution of the first loop. Following this, any modification made to either of these values will have no effect on the execution of the loop.
- If initial and final values are the same, the loop is executed only once.

Examples:

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

for c := red to blue do check(c);
```

Control statements

The repetition, conditional, and assignment statements, as well as procedures or functions calls, are used in the majority of cases within Pascal. But in certain cases, you can have situations asking for the immediate stop to a program. For these cases, TML Pascal II has the following tools:

- The GOTO statement will pass control to another part of the program located in the same block.
- The CYCLE statement forces a repetition statement to immediately execute the next iteration of a loop.
- The LEAVE statement forces the immediate exit from a repetition statement loop.
- The HALT statement will stop the execution of the program immediately.

The GOTO statement

The GOTO statement passes program control to the statement corresponding to the given label.

GOTO statement

```
goto label
```

Here are the rules to respect when using this statement:
- the label referred to by a GOTO statement must be in the same block as the GOTO statement, or in a block containing the GOTO statement.
- a jump to a structured statement originating from an external location will produce unpredictable effects. However, TML Pascal II does not detect this kind of situation.

See also chapter 13.

The CYCLE statement

The CYCLE statement passes control of the program to the end of the statements managed by the WHILE, REPEAT or FOR ready for the next iteration. The use of CYCLE outside of a WHILE, REPEAT or FOR loop generates an error.

Example:

```
for i:=1 to 100 do
begin
      if a[i] <=0 then cycle;
      f(a[i]);
end;
```

CYCLE is not a reserved word, and it can therefore be redefined. Under these circumstances, it cannot be used in the block where it has been redefined.

The LEAVE statement

The LEAVE statement immediately exits a WHILE, REPEAT or FOR loop and passes control to the statement following the loop. The use of LEAVE outside of a loop generates an error.

Example:

```
while i<63 do
begin
      if a[i] = x then leave;
      i := i+1;
end;
```

LEAVE is not a reserved word, and it can therefore be redefined. Under these circumstances, it cannot be used in the block where it has been redefined.

The WITH statement

The WITH statement is a simple method of indicating fields within a record. Using a WITH statement, the fields of one or more record variables can be accessed by using only the field identifiers.

Here the syntax of the WITH statement:

```
with-statement = 'with' record-variable-list 'do' statement

record-variable-list = record-variable { ';' record-variable }
```

The use of a variable in a WITH statement characterises the use of a record. In a WITH statement, each reference to a variable is first interpreted as a record field. Similarly, if a variable of the same name is accessible, it is always the reference to the record field which is accessed within a WITH statement.

Here are the rules to be respected when using WITH:

- When one accesses a field of another record, the record should either be listed in the record variable list (directly after the WITH) or alternatively in its explicit form (e.g. `record.field`).
- WITH statements can overlap. In such a case, the WITH containing the overlapping WITH remains equally valid in the overlapping block.
- When several records have fields with the same name, the WITH will refer to the last record variable listed directly following the WITH.
- When the field identifier of a record is the same as a variable or other identifier declared outside of the record, the WITH will access the field.

For example:

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

is equivalent to:

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

When more than one record variable is listed in a WITH statement such as:

```
with var1, var2, ... varN do statement
```

then it is treated like an overlapping construct:

```
with var1 do
with var2 do
...
with varN do
      statement
```

If varN is a field of var1 and var2, it is interpreted in the form var2.varN and not as var1.varN.

NULL statements

NULL statements are statements that do not contain anything. Nonessential semicolons are treated as null statements by TML Pascal II. The result of having a null statement is that you end up with two statements instead of one. Generally, this is not a problem, but under certain circumstances it can produce errors.

CHAPTER 17: PROCEDURES AND FUNCTIONS

Procedures and functions enable you to overlap blocks in the main block or to define UNITS. Procedures and functions are also referred to as “subroutines”. Each procedure and function has a header followed by a block or a special directive. A *procedure* is activated by a specific procedure statement, and a *function* is activated whilst evaluating an expression containing a call to a function.

PROCEDURE DECLARATION

A procedure declaration associates an identifier with a block. Such a block is then able to be activated by a procedure statement.

Syntax

```
procedure-declaration = procedure-heading ';' procedure-block

procedure-block =
      block |
      forward |
      external |
      inline <unsigned_integer> |
      tool <unsigned_integer>, <unsigned_integer> …

block = declarative-part statement-part
```

The procedure heading specifies the procedure identifier and the formal parameters allowed.

```
procedure-heading = 'procedure' identifier [ '(' formal-parameter-list ')' ]
```

The syntax of formal parameters is described further later in this chapter.

A procedure is activated by a procedure statement that indicates the procedure identifier and the current parameters required by it. The statements to be executed by the procedure are specified by the *statement* part of the procedure block. If the same procedure identifier is used in a procedure statement within the block of the procedure, the procedure is carried out recursively. It is basically saying that the procedure is calls itself during the course of its execution.

Example of a procedure declaration:

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

Sometimes, instead of a block following a procedure or function declaration, you might find a directive FORWARD, EXTERNAL, INLINE or TOOL. See the paragraphs dealing with these directives, later in this chapter.

FUNCTION DECLARATION

A function declaration associates an identifier with a block, in the form of a function able to be activated by a function call in order to calculate and return a value of the specified type.

Syntax

```
function-declaration = function-heading ';' function-block

function-block =
      block |
      forward |
      external |
      inline <unsigned_integer> |
      tool <unsigned_integer> [ ',' <unsigned_integer> ]

block = declarative-part statement-part
```

The function heading specifies the function identifier, the formal parameters allowed, and the type of the result.

```
function-heading =
      'function' identifier [ '(' formal-parameter-list ')' ] ':' result-type
```

A function is activated by evaluating a function call, which provides the function identifier along with the required formal parameters. The call to a function is in the form of an operand of an expression. The expression is evaluated by executing the function, which will replace the function call with the value returned by the function.

The statements to be executed by the function are specified within the *statement* part of the function block. The block should normally contain at least an assignment statement which will assign a value to the function identifier. The result of a function is the last value assigned. If no assignment statement exists, or if it exists but is not executed, the value returned by the function is undefined.

If the function identifier is used as a function call inside the function block, the function is carried out recursively.

Example of a function declaration:

```
function Num2String(N: integer;): string;
var V: integer;
S: string;
begin
      V := Abs(N):
      S := '';
      repeat
            S:= concat(Chr(V mod 10 + ord('0')),S);
            V := V div 10;
      until V = 0;
      if V<0 then S:= concat('-',S);
      Num2String:=S;
end;
```

A function can be declared FORWARD, EXTERNAL, INLINE or TOOL in the same manner as for procedures. If the return value of a function is a record type or a pointer to a record type, it cannot be used in the record list of a WITH statement in order to assign values to fields of this record. The compiler will interpret the use of the function identifier in a WITH statement as being a call to the function (i.e. not as a record).

FUNCTION AND PROCEDURE DIRECTIVES

Following a procedure or function, the block of statements can be replaced by one of these four directives:

- FORWARD: allows the procedure or the function to be immediately declared while not declaring its statements block until later on.
- EXTERNAL: allows procedures and functions written in another language than TML Pascal II to be linked with your program.
- INLINE: allows you to replace Pascal statements by 65816 machine language code.
- TOOL: indicates that a procedure or function implements an Apple IIgs ToolBox routine.

FORWARD directive

A procedure declaration having a FORWARD directive instead of a statement block, is called a *forward declaration*. At some point after the FORWARD declaration, the procedure will be defined by a definition declaration (a procedure using the same procedure identifier and this time containing a statement block). A definition declaration can contain the formal parameters list again, and in this case, the list must be identical to that in the FORWARD declaration. The FORWARD declaration and the declaration of definition must be in the same declaration block, but they do not need to be contiguous. Therefore, other procedures, functions, types or variables can be declared between them and can make calls to the procedure in question. A FORWARD declaration also allows for recursive usage.

The FORWARD declaration in combination with the definition declaration, constitute the complete procedure declaration. The procedure is considered as being declared as opposed to being a FORWARD declaration. Procedures and functions cannot be FORWARD declared in the INTERFACE part of a UNIT.

Examples:

```
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 directive

A procedure declaration whose body consists of only the EXTERNAL directive, defines the Pascal interface for a routine assembled or compiled in a language other than TML Pascal II. The external code for the routine must be available for the linker.

Example:

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

In this example, GotoXY is an external procedure that has to be linked with the main program before execution. It is up to the programmer to make sure that the external functions and procedures are compatible with their declarations in the Pascal program. The linker does not carry out any test for compatibility.

INLINE directive

INLINE directives make it possible to write code in machine language in place of a statement block. The machine code can only be made up of a sequence of integer constants that each represent one byte of machine language. When the procedure is called, the compiler generates the computer code specified by the INLINE directive. If a procedure has parameters, they are pushed on top of the stack where the code is generated. The purpose of the INLINE directive is to write small routines. For example, the following procedure will erase the interrupt inhibit flag by generating a CLI instruction:

```
procedure GenCli; inline $58;
```

TOOL directive

The TOOL directive is used to define the body of a procedure as being one of the Apple IIgs ToolBox routines. The IIgs ToolBox is divided into several tool sets, each one containing a number of routines. Each tool is identified by a unique number, and each routine is indicated by a unique function number. This method of designation of the ToolBox routines makes it possible for TML Pascal II to generate the appropriate code to call upon a ToolBox function. For example, the MoveTo procedure of the QuickDraw tool (tool number 4) is identified by function number 58. Therefore, the TOOL directive used to achieve this is:

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

PARAMETERS

The declaration of a procedure or a function specifies a formal parameter list. Parameters declared in a formal parameters list are local to the declared procedure or function, and can be addressed by their identifiers within the block associated with the procedure or function.

```
formal-parameter-list =
      '(' parameter-declaration { ';' parameter-declaration } ')'

parameter-declaration =
      identifier-list ':' type-identifier |
      'var' identifier-list ':' type-identifier |
      'static' identifier-list ':' type-identifier |
```

Procedure and function declarations of can have up to three kinds of formal parameters:

1. value: parameters not preceded by the any keyword
2. variable: parameters preceded by the VAR keyword
3. static: parameters preceded by the STATIC keyword

VALUE parameters

A formal value parameter behaves like a local variable within the procedure or function, except that its initial value is assigned by the corresponding current parameter during the activation of the procedure or function. Any modifications made to a value parameter do not affect the value of the current parameter. A value parameter corresponding to a current parameter in a procedure or function call must be an expression, and its value cannot be a File type or any other structured type containing a File type.

The current parameter must be assignment compatible with the type of the formal value parameter. If the parameter is a String type, then the formal parameter must have a static size of 255. If the size (in bytes) of a formal parameter is greater than 4 bytes, then the current parameter is passed by address and its current value is copied into the local variables area. However, any assignments made to the formal parameter do not affect the value of the current parameter.

The formal and current parameters must assignment compatible. This restriction can be circumvented by declaring the parameters in UNIV form (see the second last paragraph of this chapter).

VARIABLE parameters

A variable parameter is used when the value of the parameter is to be returned from the procedure or function. The corresponding current parameter in a procedure or function call statement must be a reference variable. The formal variable parameter represents the current variable during the execution of the procedure or function, and any modification made to the formal parameter value is immediately transferred to the current parameter. Within a procedure or function, any reference to a formal variable parameter relates to the current parameter. The current and formal parameters must be assignment compatible. This restriction can be circumvented by declaring the parameters in UNIV form (see the second last paragraph of this chapter).

If the parameter is a String type, then the formal parameter must have a static size of 255, and the current variable parameter must be of String type with a length of 255.

File types cannot be passed in variable parameters. Components of packed structure type variables cannot be used as current variable parameters. If you access a current variable via an index into a table, or the variable is reached via a pointer identifier, or reached via a field of a record, the action of accessing the variable is carried out prior to the execution of the procedure or function. It is necessary to pay attention when accessing variables that have been relocated in the main segment. The compaction of this segment can cause the source to be moved which would produce surprising results.

STATIC parameters

Static parameters are extensions specific to TML Pascal II for the Apple IIgs. They were added with the aim of allowing the production of a powerful object code. The static parameters are treated in the same manner as value parameters with the following added restriction: a new value cannot be assigned to a formal static parameter within the procedure or function.

A parameter value whose formal type requires more than 4 bytes of storage is passed by address and then copied into the local variable storage area. Thus the assignment of new values to the formal value parameter does not modify the current parameter (see appendix D). However, in certain cases if the formal parameter is only read (and not written to), it does not need to copy the current parameter into the local variable storage area for use by the formal parameter since it can be addressed directly. Static parameters reduce the size of the stack required by an application and also reduce the execution time because it does not need to copy the current parameter value into the local area.

NOTE: TML Pascal II does not check to see if a static parameter will be written to or not. It is up to the programmer to ensure the correct use of static parameters.

UNIV parameters

When the keyword UNIV appears before a type identifier in the formal parameter list, the restriction stating that current and formal parameters must be assignment compatible for value parameters, or identical if they are variable parameters, no longer applies. When UNIV is used, the current parameter can be of any type provided that the number of bytes necessary for storing the current parameter value is the same as that of the formal parameter.

Here is an example of a UNIV parameter:

```
TYPE Ptr = ^Char;
VAR aLong: LongInt;
aPtr : Ptr;

procedure aProc(p: univ Ptr);
begin
end;

begin
      aProc(aLong);
```

```
        aProc(aPtr);
end.
```

Parameter list compatibility

Parameter list compatibility is essential for agreement between formal and current parameter lists. Two formal parameter lists are compatible if they contain the same number of parameters and if the parameters are in agreement within the list. Two parameters agree if one of these conditions is true:

- They are value parameters of identical type.
- They are variable parameters of identical type.
- The formal parameter has UNIV before its type, and its current parameter is a value or a variable of the same size. The parameters must both be values or variables.

CHAPTER 18: PROGRAMS AND UNITS

INTRODUCTION

TML Pascal II offers two kinds of source definition for your applications: Programs and Units. The difference between the two is that a program corresponds to a complete application and can therefore be compiled and executed. A Unit cannot be executed; rather, it is a separate module in which parts of a program can be defined and compiled independently of the main program. Programs and Units are compiled separately. Their object files are required by the TML Pascal II Linker so that it can combine them into one executable file.

PROGRAMS

A Pascal Program declaration resembles a procedure declaration that only has a header and the optional presence of the USES clause.

```
program = 'program' program-header ';' [ USES clause ]
```

The identifier immediately following the word PROGRAM in the program header is the program identifier. The program parameters described by Jensen and Wirth and the ANSI standard have nothing to do with TML Pascal II.

NOTE: versions 1.x of TML Pascal used the reserved words Input and Output in the header. These words were used to indicate to the compiler that it was to create a Plain Vanilla environment. With TML Pascal II, it no longer needs to employ these reserved words. In order to create a Plain Vanilla environment (from now on called textbook Graphic), all that is required is to call on the Graphics procedure.

```
program-header = identifier [ '(' program-parameter-list ')' ]
```

THE USES CLAUSE

The USES clause is used to identify Units required by a program or a Unit in order for it to be compilable.

When TML Pascal II encounters an identifier in a USES clause (the name of a UNIT), it must be able to find the corresponding compiled code containing the symbols table and object code of that unit. To assist the linker, the string “.p.o” is added to the end of Unit names. For example:

```
uses Globals, fileStuff;
```

will make use of the files Globals.p.o and fileStuff.p.o.

The USES clause in the main program indexes the necessary Units for the program. These units are comprised of those used directly by the main program and those used by the Units themselves. It is possible that the name of the object code file of a Unit does not correspond to the name of the Unit. In such a case, the compiler directive $U makes it

possible to specify a particular Units filename. The directive $U must appear immediately before the Unit name in the USES clause. For example, one could have written:

```
uses globals,
{$U :MyDisk:MyFolder:fileStuff.p.o} fileStuff;
```

As illustrated in this example, the directive $U is used to indicate the complete prefix to access the unit not located in the same directory as the source being compiled. The directive $U is documented further in appendix B.

When a Unit named in a USES clause uses other Units, the names of these units must also appear in the USES clause and their names must precede the Unit in question. For example:

```
UNIT UnitA;
INTERFACE
      type colors=(red,white,blue);
IMPLEMENTATION
END.
```

```
UNIT UnitB;
USES UnitA;
INTERFACE
      type Rec = record
              i : Integer;
              c : colors;
            end;
IMPLEMENTATION
END.
```

```
PROGRAM MyProgram;
USES UnitA, UnitB;
VAR aRec: Rec;
BEGIN
END.
```

In this example, the *MyProgram* program declares a variable *aRec* of the Rec type, which is declared in *UnitB*. However, a USES clause is used to name *UnitB* and this Unit also uses a USES clause in order to name *UnitA*. Therefore, the USES clause of the *MyProgram* program must name *UnitA* before *UnitB*.

If a Unit is recompiled, all Units which use it must also be recompiled. This so that interdependent units do not try to refer to declarations that have removed or modified. For example, if *UnitB* is recompiled in our preceding example, then *MyProgram* must be recompiled but not *UnitA*. However, if one modifies *UnitA*, a recompile of *UnitB* and *MyProgram* will be needed.

UNITS

Units are the corner stone of modular programming. Units are compiled separately from each other and are used to organise important programs in logical sections. By dividing a program into several parts, one also reduces the compile time of each part. Here are some good reasons to use Units:

- They allow you to make important programs modular
- They allow you to define declarations and statement blocks that are usable by various programs
- They can be used to make parts non-listable to everyone

The identifier following the reserved word UNIT is the unit identifier. It is the name which others units and the main program will have to use in their USES clauses.

The syntax of Units:

```
Unit
    +
    !
    +->| interface | --> | implementation | -> | END | -> | . |->
       | part      |     | part           |
```

The identifier following the reserved word UNIT is the unit identifier. It is the name that others units and the main program will have to use in their USES clauses.

```
Interface part
              !-+
                +->| USES   |-+ +-| declaration |<-+
                   | clause |
```

The *Interface part* of a Unit declares the public constants, types, variables, procedures and functions. That is, statements made in the *Interface part* are available for other Units and Programs that list the units name in their USES clause. In other words, the valid scope of these public statements is the entire program or unit making use of the Unit. Programs and units using a Unit can access public statements as if they had been declared in a block of its own.

Label declarations are not allowed in the *interface part* of a Unit. Procedures and functions within the *interface part* are only declared by indicating procedure and function names, the possible formal parameters, and the result type (for functions). You cannot put a statement block for procedures and functions in the *interface part*. Instead of this, the procedure or function header is repeated within the *implementation part*, and it is there that the statement blocks are declared. Declarations of procedures and functions in the *Interface part* behave as if a FORWARD directive had been specified. However, procedure and function declarations can also include the EXTERNAL, INLINE and TOOL directives within the interface part, and in these cases they will not have a block declared within the implementation part. Variables, procedures and functions appearing in the *interface part* are known as *global*. An entire Unit is under the management of the block in which a USES clause references it. The *interface part* can contain a USES clause, and thus a Unit can use another Unit.

```
Implementation  part
                    +-| declaration |>-+
```

The implementation part which follows the last declaration of the interface part, declares all private constants, types, variables, procedures and functions. That is, these are not available for other units or programs that list the Unit in their USES clause. Private procedures and functions are declared in the same way as procedures and functions in programs (with a header and a body).

Public procedures and functions declared in the interface part are redeclared within the implementation part. The only exception relates to procedures and functions declared using the EXTERNAL, INLINE and TOOL directives. Formal parameters and result types

can be omitted, but if they are indicated, then they must be listed identically to the preceding declaration.

CHAPTER 19: INPUT/OUTPUT

FILE ACCESS

TML Pascal II manages the file access in two manners:

1. I/O routines of the Apple IIgs ToolBox.
2. Specific TML procedures and functions.

QuickDraw and the Event Manager remain the most direct methods of managing the screen, keyboard and mouse. Calls to the GS/OS 5.0 operating system also make it possible to easily handle files. Appendix C contains the list of file interfaces of the ToolBox.

The TML Pascal II internal routines also make it possible to easily handle file access. This chapter exclusively deals with this method.

FILES IN PASCAL

Within TML Pascal II, files are managed using “file variables”. A file variable is quite simply a variable that has been declared as a file type (see Chapter 13). TML Pascal II accepts two types of files:
- text files.
- structured files.

Text files are declared with a preset TEXT type. Text files store data in the form of character sequences organized in lines. TML Pascal II has a number of special functions and procedures for managing lines of text. TML Pascal II predefines two text files for each program: *Input* and *Output*. The *Input* file is predefined to start reading from the keyboard only, and the Output file is predefined to write to the screen (text or graphics screen).

Structured files consist of a sequence of components. In a text file, a component is a character; in a structured file, a component can be any type other than a file type or a structure type containing a file type. A single component is referred to as a “logical record”.

Examples of files:

```
var aFile: text; { example of a text file }
var aFile: file of integer; { example of a structured file }
```

File variables refer to files that are made up of a sequence of components. For text files, the components are always of type *Char*. For the structured files with, a component can be any Pascal type, except a *File* type or a structure containing a *File* type. In each case, the component is regarded as being a logical record. Files can to have any number of logical records, but you can only access one at a time with a file variable. The position where a logical record is accessed, in relation to the beginning of the file, is called the “current file position”. Before using a file variable, it should be associated with a file: this process is called “to open” the file. There are three procedures for opening a file: Reset, Rewrite and Open.

Note: the file variables Input and Output are predefined, and automatically open when a program is launched.

To open a file, one must specify its external name. This name can be a GS/OS access path or the name of a peripheral.

STANDARD PROCEDURES AND FUNCTIONS FOR ALL FILES

The RESET procedure

syntax: `Reset(F [, title ])`

Reset opens an existing file for reading or *repositions* an open file by setting the current file position to component 0. The file is opened only for sequential reading. When an already open file is *repositioned*, its contents are not erased. 'F' is a file variable of any *File* type; 'title' is an optional string.

If ‘title' is specified in the parameter list, then RESET will open an existing file named ‘title' and this will associate the external file with the 'F' file variable. If ‘title' is not a GS/OS compatible filename, nor a name of peripheral, or if the file cannot be opened, an error will be returned in IOResult.

If ‘title' is not specified in the parameter list, then 'F' must have already been associated with an open file. In this case, RESET sets the current file position to component 0 of the file. If 'F' is not already associated with a file, an error will be returned in IOResult.

The REWRITE procedure

syntax: `Rewrite(F [, title ])`

Rewrite opens an existing file for writing, or creates a new file, or *repositions* an open file by setting the current file position to component 0. The file is opened only for sequential writing. When an already open file is *repositioned* with REWRITE, its contents are not erased. 'F' is a file variable of any file type; ‘title' is an optional string.

If ‘title' is specified in the parameter list, then REWRITE will create and open a new external file named ‘title' and the File variable 'F' will be associated with this external file. If the file already exists, it is opened and its contents are not erased.

If ‘title' is not specified in the parameter list, then 'F' will must have already been associated with an open file. In this case, REWRITE sets the current file position to component 0 of the file. If 'F' is not already associated with a file, then an error will be returned in IOResult.

The OPEN procedure

syntax: `Open(F, title)`

OPEN will open an existing file or create a new file. The file is opened for read/write with random access. When an existing file is opened, its contents are not erased. 'F' is a file variable of any File type; 'title' is an optional string.

If 'title' is not a GS/OS compatible filename, nor a name of a peripheral, or if the file cannot be opened, an error will be returned in IOResult.

The CLOSE procedure

syntax: `Close(F)`

CLOSE will close an open file. 'F' is a File variable of any File type. The association between 'F' and the external file are removed, and the system marks the file *closed*.

The EOF procedure

syntax: `EOF(F)`
result: `Boolean`

The EOF procedure returns the end of file state. 'F' is a file variable. EOF(f) returns TRUE if the current file position is after the last position within the file; otherwise it returns FALSE.

The SEEK procedure

syntax: `Seek(F, n)`

Change the current file position to the component 'n' but do not read the new logical record. 'F' is a File variable, 'n' is an expression of *LongInt* type. For text files, the size of a logical record is 1. The number of the first logical record is 0. If the value of 'n' is greater than the component count of the file, then the current position is positioned at the end of the file and EOF(F) returns TRUE.

The ERASE procedure

syntax: `Erase(title)`

The ERASE procedure erases an external file. 'title' is an expression of the string type. The external file with name 'title' is erased from the external peripheral on which it was written.

The IORESULT procedure

syntax: `IOResult`
result: `Integer`

The IORESULT procedure returns an integer value corresponding to the error state of the last I/O operation carried out. A value of 0 indicates that the operation was successfully executed; a non-zero value indicates that an error occurred.

You will notice that IOResult returns the state of the last I/O operation. Also, the following two statements do not give the results that you might expect:

```
Reset(F, 'myFile');
Writeln('IOResult for Reset = ', IOResult);
```

The call to the IOResult function in the *Writeln* parameter list returns the error state of the *Writeln* operation of the string 'IOResult for Reset = ' since that is the last I/O operation carried out. If you had wanted to return the error state of *Reset*, you would have to write:

```
Reset(F, 'myFile');
svIOResult: = IOresult;
Writeln('IOResult for Reset = ', svIOResult);
```

The FILEPOS function

syntax: `FilePos(F)`
result: `LongInt`

The FILEPOS function returns the current file position of the open file 'F'. The first logical record within a file has position 0. With structured files, a logical record is an occurrence of the component type. For text files, the component type is a byte. 'F' is a file variable associated with an open file.

The RENAME procedure

syntax: `Rename(oldName, newName)`

The RENAME procedure allows you to rename a file. 'oldName' and 'newName' are string expressions. The external file named 'oldName' will be renamed to 'newName'. If no file exists with the name 'oldName', an error will be returned in IOResult.

STANDARD PROCEDURES FOR STRUCTURED FILES

The procedures identified in this paragraph are used for random access to logical records within structured files. The component type of these files can be any type other than a File type or a structure type containing a File type.

The READ procedure for structured files

syntax: `Read(F, v1 [, v2..., vn ])`

The READ procedure allows you to read a component of a file variable. 'F' is a file variable, and each 'v' parameter is a variable of the same type as the component type of

the file 'F'. For each 'v' parameter, the file component at the current file position is read into 'v' and the current file position is advanced to the next component. If one tries to read after the end of the file, an error is returned in IOResult.

Procedure READ is also used with text files (see later). With text files, the file variable 'F' is an optional parameter, because if it is omitted, the READ statement will carry out reading from the standard input defined for text type (i.e. the keyboard). For structured files, the file variable is mandatory.

The WRITE procedure for structured files

syntax: `Write(F, v1 [, v2..., vn ])`

The WRITE procedure makes it possible to write each 'v' component variable to a file. 'F' is a file variable, and each 'v' parameter is a variable of the same type as the component type of the file 'F'. For each 'v' parameter, the value of 'v' is written to the component file at the current file position, and the current file position is advanced to the next component. If one tries to write after the end of the file, a new component is added to the end of the file.

The WRITE procedure is also used for text files (see later). With text files, the file variable 'F' is an optional parameter, because if it is omitted, the WRITE statement will carry out writing to the standard output defined for text type (i.e. the screen). For structured files, the file variable is mandatory.

THE STANDARD PROCEDURES AND FUNCTIONS FOR TEXT FILES

Text files are distinguished from the other file types due to the fact that they are organised into collections of lines all ending in a carriage return. Text files are different from files defined as 'file of char' because the former is always organized in lines, while the latter is not forced to do this. None of the procedures and functions defined in this paragraph requires the explicit use of a file variable as a parameter. If you do not specify a file variable parameter, the predefined file Input or Output is used by default (if reading, INPUT which will be used by default; otherwise OUTPUT will be used).

The READ procedure for text files

syntax: `Read([ F, ] v1 [, v2..., vn ])`

The READ procedure reads one or more values from a text file into the corresponding parameters $v_1...v_n$. If the variable 'F' is included, it must be a File of text type. If ‘F’ is not specified, the standard file Input is used by default, which happens to be the keyboard. Each 'v' variable is an Integer, Longint, Real, Char or String type.

To read a Char variable

For variables of Char type, the Read procedure reads a character from the file and assigns it to the variable. If Eof(F) is TRUE before reading, the value Chr(0) will be returned. If Eoln(F) is TRUE before reading, the value Chr(13) will be returned. The next read will take place starting from the next character in the file.

To read a LongInt or Integer variable

For Integer or LongInt variables, one reads a sequence of characters that together form a signed number. All spaces, tabulations and end of lines are skipped until the beginning of a numerical string is found. Consequently, all characters other than 'space', ' tabulation' or 'end of line' are regarded as forming part of the numerical string. The string is then interpreted as being in the form of a numerical value. If no character of the string represents a signed number, then an error is returned by IOResult. The next read will be done starting from the next character after the numerical string.

To read a Real variable

With Real variables, one reads a sequence of characters that together form a signed floating point number. All spaces, tabulations and end of lines are skipped until the beginning of a numerical string is found. Consequently, all characters other than 'space', ' tabulation' or 'end of line' are regarded as forming part of the numerical string. The string is then interpreted as being in the form of a numerical value. If no character of the string represents a real number, then an error is returned by IOResult. The next read will be done starting from the next character after the numerical string.

To read a String variable

With a String variable, one reads all the characters into the string variable until the next 'end of line' character (this one not being read). The next read will start from this 'end of line' character. Notice that successive reads of String type, will not carry out successive reads of lines since a read of String type never continues beyond the ‘end of line’ character.

The READLN procedure

syntax: `Readln([F, ] v1 [,v2...,vn ])`

This procedure is an extension of the READ procedure. After having carried out the same job as READ, READLN will continue jumping to the start of the next line of the file while reading all the characters into the corresponding variables until an 'end of file’ is reached. If there is no next line, EOF(F) will become TRUE. Naturally, if one does not specify 'F', the standard file *Input* is used by default.

The WRITE procedure with text files

syntax: `Write([ F, ] v1 [,v2...,vn ])`

This procedure will write one or more values into a file of text type. If ‘F’ is not specified, the standard file *Output* is used, which is generally the monitor screen of the Apple IIgs. Each ‘v’ value can be an expression of Integer, long integer, real, character, boolean, or string type. The ‘v’ parameters are called write parameters. Each write parameter is of the form:

```
OutExpr [: MinWidth [: DecPlaces ] ]
```

OutExpr is an expression of an authorized type.
MinWidth and *DecPlaces* are expressions with values of the Integer type. *MinWidth* indicates the minimum width of the field; it must be greater than or equal to 0. There will be exactly *MinWidth* characters written (with leading spaces if necessary), unless OutExpr has a value that is only able to be represented by using more characters than *MinWidth*; in this case, the number of characters necessary to represent *OutExpr* is written. In the same manner, if *MinWidth* is not specified, each *OutExpr* will be written using the number of characters necessary to represent it. *DecPlaces* indicates the number of decimal places used in the representation of real numbers. One can only use it if *OutExpr* is of real type, and if *MinWidth* is also indicated. If *DecPlaces* is used, it must be greater than 0. However, if *DecPlaces* is not specified, *OutExpr* is written using exponential representation.

The WRITELN procedure

syntax: `Writeln([ F, ] v1 [,v2...,vn ])`

This procedure is an extension of the Write procedure. After having carried out the same operations as Write, the Writeln procedure sends an end-of-line character (carriage return) to the file.

The EOLN function

syntax: `Eoln [ (F) ] result: boolean`

This function returns the end-of-line indicator for a file. ‘F’ must be declared as a file of Text type. `Eoln(F)` will return TRUE if the character currently read is a end-of-line character, or if `Eof(f)` is TRUE; otherwise the result is FALSE.

The PAGE procedure

syntax: `Page [ (F) ]`

This procedure writes a form feed character to a text file. ‘F’ must be declared as a file of Text type. If ‘F’ is not specified, the standard file *Output* is used.

DISK FILES AND TML PASCAL II

When one specifies an external file within a TML Pascal II procedure, the complete access name must be used as the file name. A complete access name consists of the file name, possibly preceded by the volume name and one or more folder names. The names of volumes, folders, and file names are separated by a ‘:’.

For example:

```
MyVolume:MyDir1: ...:MyDirN:MyFile
```

However, one can also use the old ProDos16 syntax and put a ‘/’ in place of a ‘:’.

For example:

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

(See the GS/OS reference handbook for further information).

PERIPHERALS AND TML PASCAL II

In addition to external disk files, TML Pascal II accepts peripherals as sources/destinations for input/output. These peripherals must be recognized by GS/OS – for example the keyboard, screen, printer, etc. At the time of launching a program, the Keyboard and Screen are the standard files *Input* and *Output* respectively. The printer is also available as a text peripheral; however, it must be explicitly opened using the REWRITE procedure using ".PRINTER" as the name of device.

For example:

```
PROGRAM TestPrinter;
var f: Text;
begin
      Rewrite(f,'.PRINTER');
      for i:=1 to 10 do
            Writeln(f,'Hello to the printer!');
      Close(f);
end;
```

CHAPTER 20: STANDARD PROCEDURES AND FUNCTIONS

This chapter describes the predefined standard procedures and functions of TML Pascal II, with the exception of the input/output procedures and functions already described in chapter 19. The standard procedures and functions are predefined. This means that they function as if they were declared in an appended Unit. There is no declaration conflict caused by redefining the same identifiers within a program, but in these cases the predefined procedure or function is hidden. It should be noted that predefined procedures and functions cannot be used as parameters to procedures or functions.

THE GRAPHICS PROCEDURE

syntax: `Graphics(screenMode: integer);`

The Graphics procedure is used to initialize the graphical environment of the Apple IIgs. This procedure must be called at the beginning of the body of the main program. The procedure initializes the *QuickDraw* and *EventManager* tools and sets the SHGR screen mode to 640 or 320 according to the *ScreenMode* value. The screen can be used for standard I/O operations for the *Readln*, *Writeln* procedures and any other I/O routines. QuickDraw graphics can also be drawn to this screen. The Graphics procedure is provided in TML Pascal II in order to simplify access to drawing. Programming in graphics mode is detailed in chapter 7.

THE EXECUTION CONTROL PROCEDURES

The procedures described in this chapter allow an immediate branching to another part of the program.

The EXIT procedure

syntax: `Exit(id)`

The exit procedure immediately stops execution of a program block: a block being a function, procedure, or the whole program. In a general way, it is equivalent to a GOTO jump to a label located at the end of the block identified by ID.

The HALT procedure

syntax: `Halt`

The Halt procedure immediately stops execution of the program.

The CYCLE procedure

syntax: `Cycle`

The Cycle procedure immediately passes control to the end of the loop in progress and will continue executing with the next loop value. This procedure only has meaning within WHILE, FOR, or REPEAT loops; used outside of these loops, it has no effect. See chapter 16 for more information.

The LEAVE procedure

syntax: `Leave`

The Leave procedure makes it possible to immediately pass control to the statement directly after the loop in progress. This procedure only has meaning within WHILE, FOR, or REPEAT loops; used outside of these loops, it has no effect. See chapter 16 for more information.

PROCEDURES FOR DYNAMIC MEMORY ALLOCATION

These procedures are used to manage memory areas not allocated at the time of program execution. These procedures use the functionality of the *MemoryManager* in order to recover free memory.

The NEW procedure

syntax: `New(P)`

New(p) creates a variable of type ‘P’ and makes ‘P’ refer to it. ‘P’ can be a pointer of any type. The value of ‘P’ is referred to by using p^. New uses the *NewHandle* function to find an area of free memory and it returns a pointer to the allocated storage block. An error code is returned if the reserved area is insufficient to contain the new variable. In this case, ‘P’ will be set to the Nil pointer and the *HeapResult* function will return an error code.

The DISPOSE procedure

syntax: `Dispose(P)`

Dispose(p) destroys the dynamic variable referenced by ‘P’ and releases the memory area which it occupied. ‘P’ must be a variable previously created by New(P) or previously assigned a value by an assignment statement. After the call to Dispose(P), the value of ‘P’ becomes undefined and an error will be generated if one references the value previously associated with ‘P’.

THE TRANSLATION FUNCTIONS

These functions are used to translate a value from one type into another. Note that the standard procedures *Pack* and *Unpack* of Pascal are not implemented in the TML Pascal II.

The TRUNC function

syntax: `Trunc(X)`
result: `LongInt`

Trunc(X) returns a LongInt value representing the real variable ‘X’ truncated to the nearest integer in the range 0 to X inclusive. There is an error if the result of truncating is outside the range ((-maxlongint-1) .. maxlongint).

The ROUND function

syntax: `Round(X)`
result: `LongInt`

Round(X) returns a LongInt value representing the real variable ‘X’ rounded to the nearest integer. If ‘X’ is exactly halfway between two integers, the result is the integer having the greatest absolute value. There is an error if the result of rounding is outside the range ((-maxlongint-1) .. maxlongint).

The ORD4 function

syntax: `Ord4(X)`
result: `LongInt`

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

The POINTER function

syntax: `Pointer(X)`
result: `pointer of generic type`

Pointer converts an Integer or LongInt value into a pointer type. The value returned by Pointer(X) is a pointer to the physical address containing X’s value. This pointer is of the same type as the Nil Pointer in the sense that it is compatible with any pointer type. The value of Pointer(0) is the Nil pointer.

THE ARITHMETIC FUNCTIONS AND PROCEDURES

The arithmetic functions and procedures carry out operations on values of integer or real type. The implementation of these routines can be performed using the TML Pascal II compiler or by using the functions with the SANE tool.

The INC procedure

syntax: `Inc(X)`

Increment the variable X (which is of type Integer or LongInt) by one.

The DEC procedure

syntax: `Dec(X)`

Decrement the variable X (which is of type Integer or LongInt) by one.

The ABS function

syntax: `Abs(X)`
result: `same type as the parameter`

Return the absolute value of ‘X’. That is, if ‘X' is negative, the function will return ‘X’. ‘X’ is an argument of type Integer or Real.

The SQRT function

syntax: `Sqrt(X)`
result: `extended`

If ‘X’ is not negative, the value returned is of type Extended and corresponds to the square root of ‘X’. If ‘X’ is negative, a NaN (Not a Number) diagnosis is made and an illegal operation indicator is set (see Apple Numeric Manual).

The ODD function

syntax: `Odd(X)`
result: `boolean`

The function returns TRUE if ‘X’ is odd (non-divisible by 2 without remainder). If X is even, the function returns FALSE. ‘X’ is an expression of ordinal type.

The SIN function

syntax: `Sin(X)`
result: `extended`

This function will return the trigonometric sine of ‘X’. ‘X’ is an expression of real type and must represent an angle in radians. If ‘X’ is infinite, a NaN (Not a Number) diagnosis is made and the invalid operation indicator is set.

The COS function

syntax: `Cos(X)`
result: `extended`

This function will return the trigonometric cosine of ‘X’. ‘X’ is an expression of real type and must represent an angle in radians. If ‘X’ is infinite, a NaN (Not a Number) diagnosis is made and the invalid operation indicator is set.

The EXP function

syntax: `Exp(X)`
result: `extended`

This function will return the value of *e* to the power of ‘X’ (*e* being the base of natural logarithms). If a floating point overflow occurs, the result is +inf. ‘X’ is an expression of real type.

The LN function

syntax: `Ln(X)`
result: `extended`

This function will return the natural logarithm of ‘X’. ‘X’ is an expression of real type. If ‘X’ is negative, a NaN diagnosis is made and the invalid operation indicator is set.

The ARCTAN function

syntax: `Arctan(X)`
result: `extended`

This function will return the Arctangent value of ‘X’. ‘X’ is an expression of real type. All numeric values between +inf and -inf are valid.

THE ORDINAL FUNCTIONS

The ordinal functions described in this chapter refer to ordinal values of scalar or pointer types. See chapter 13 for more information on scalar and pointer types.

The ORD function

syntax: `Ord(X)`
result: `Integer or LongInt`

Ord returns the ordinal value of a scalar or pointer. If ‘X’ is of type Integer or LongInt, the result type will be identical to that of ‘X’. If ‘X’ is of pointer type, the result will be the corresponding address of the dynamic variable pointed to by ‘X’, in type LongInt. If ‘X’ is of ordinal type, the result is of type Integer and the value is the ordinal of X. The standard

procedure Ord4 must be used instead if you want the result to be of type LongInt, regardless of the type of X.

The CHR function

syntax: `Chr(X)`
result: `Char`

This function will return the Char value whose ordinal value is ‘X’. For any Char value CH, the following is always true: `chr(ord(CH)) = CH`.

The SUCC function

syntax: `Succ(X)`
result: `same type as the parameter`

This function will return the successor of ‘X’. An error occurs if ‘X’ is the last value authorized for the type (i.e. it does not have a successor).

The PRED function

syntax: `Pred(X)`
result: `same type as the parameter`

This function will return the predecessor of ‘X’. An error occurs if ‘X’ is the first value authorized for the type (i.e. it does not have a predecessor).

THE FUNCTIONS AND PROCEDURES FOR STRINGS

The functions and procedures for strings do not accept parameters of type *Packed Array of Char*. Instead only those of type String are accepted.

The LENGTH function

syntax: `Length(str)`
result: `Integer`

This function will return the dynamic length of a string.

The POS function

syntax: `Pos(substr, str)`
result: `Integer`

Pos(substr, str) will search for the *substr* substring within the *Str* string, and if found it will return the integer value corresponding to the index of the first character of *Substr* in *Str*. If *Substr* is not found in *Str*, the function returns 0.

The CONCAT function

syntax: `Concat(str1 [, str2... strn ])`
result: `generic String type`

The *Concat* function concatenates the supplied parameters in the order given and returns the result in a single String. The resultant String cannot exceed 255 characters in length.

The Copy function

syntax: `Copy(source, index, count)`
result: `String type`

The Copy function returns a String of *count* characters from the *source* string starting from `source[index]`.

The DELETE procedure

syntax: `Delete(dest, index, count)`

The procedure Delete removes *count* characters from the *dest* string starting from `Dest[index]`.

The INSERT procedure

syntax: `Insert(source, dest, index)`

The Insert procedure inserts the *source* string into the *Dest* string, with the first character of *source* being inserted into `dest[index]`.

LOGICAL FUNCTIONS AND PROCEDURES

This chapter describes the bit handling routines. These routines correspond to equivalent instructions of the 65816.

The BAND function

syntax: `BAnd(arg1, arg2)`
result: `Integer or LongInt`

This function returns the logic AND of its two arguments. Where arg_1 and arg_2 are two expressions of scalar type.

The BOR function

syntax: `BOr(arg1, arg2)`
result: `Integer or LongInt`

This function returns the logic OR of its two arguments. Where arg_1 and arg_2 are two expressions of scalar type.

The BXOR function

syntax: `BXor(arg1, arg2)`
result: `Integer or LongInt`

This function returns the logic EXCLUSIVE OR of its two arguments. Where arg_1 and arg_2 are two expressions of scalar type.

The BNOT function

syntax: `BNot(arg)`
result: `Integer or LongInt`

This function returns the logical negation of its argument (its 1's complement). Where *arg* is an expression of scalar type.

The BSL function

syntax: `BSL(arg)`
result: `Integer or LongInt`

BSL left shifts the bits of *arg*. Where *arg* is an expresion of scalar type. A zero is introduced into the least significant bit.

The BSR function

syntax: `BSR(arg)`
result: `Integer or LongInt`

BSR right shifts the bits of *arg*. Where *arg* is expresion of scalar type. A zero is introduced into the most significant bit.

The BROTL function

syntax: `BRotL(arg)`
result: `Integer or LongInt`

BRotL carries out a left rotation on the bits of *arg*. Where *arg* is expresion of scalar type. The most significant bit is reintroduced into the least significant bit.

The BROTR function

syntax: `BRotR(arg)`
result: `Integer or LongInt`

BRotR carries out a right rotation on the bits of *arg*. Where *arg* is expresion of scalar type. The least significant bit is reintroduced into the most significant bit.

The HIWRD function

syntax: `HiWrd(arg)`
result: `Integer`

HiWrd returns the most significant word of *arg* (a scalar or pointer). That is, bits 31-16 of a LongInt. If *arg* is not a 32 bit value, HiWrd returns 0. When the argument is a simple variable or array, no code is generated by the function because the argument is only addressed and used like an Integer.

The LOWRD function

syntax: `LoWrd(arg)`
result: `Integer`

LoWrd returns the least significant word of *arg* (a scalar or pointer). That is, bits 15-0 of a LongInt. When the argument is a simple variable or array, no code is generated by the function because the argument is only addressed and used like an Integer.

MISCELLANEOUS FUNCTIONS AND PROCEDURES

This chapter describes the byte handling functions and procedures as well as the routines for use on *Packed Array of Char*. The byte handling routines make it possible for a program to regard a variable as a sequence of bytes, without worrying about the data type. The byte handling routines are: *MoveLeft*, *MoveRight* and *SizeOf*.

The routines for working with *Packed Array of Char* are: *ScanEq*, *ScanNE* and *FillChar*. The parameters used for these routines cannot be indexed variables as the routines always start with the first character of an array.

The SIZEOF function

syntax: `SizeOf(id)`
result: `LongInt`

This function returns the number of bytes occupied by the variable or type ID. The value of *SizeOf* is determined by the Pascal compiler which treats it as a constant during compilation.

The CARD function

syntax: `Card(s)`
result: `Integer`

This function will determine the number of elements in *s* and return the resulting Integer value representing the cardinality of *s*. That is, the numbers of elements in *s*.

The MOVELEFT procedure

syntax: `MoveLeft(source, dest, count)`

MoveLeft copies a block of *count* consecutive bytes from *source* to *dest*, starting with the lowest address (i.e. the first byte of *source* and *dest*). Where *source* and *dest* are variables of any type other than a File type or a structure containing a File type and *count* is an Integer expression whose value is not checked. If *source* and *dest* overlap, you can only use this procedure if *source* has the highest address.

The MOVERIGHT procedure

syntax: `MoveRight(source, dest, count)`

MoveRight copies a block of *count* consecutive bytes from *source* to *dest*, starting with the highest address (i.e. the last byte of *source* and *dest*). Where *source* and *dest* are variables of any type other than a File type or a structure containing a File type and *count* is an Integer expression whose value is not checked. If *source* and *dest* overlap, you can only use this procedure if *source* has the lowest address.

The FILLCHAR procedure

syntax: `FillChar(dest, count, CH)`

FillChar fills a block of *count* consecutive characters with the *CH* character, starting with the start address of *dest*. Where *dest* is a variable of type *Packed Array of Char* and *count* is an Integer expression whose value is not checked. *CH* is a value of the Char type.

The SCANEQ function

syntax: `ScanEq(limit, CH, source)`
result: `Integer`

ScanEq scans a memory block starting at the address of *source* and search for the first occurrence of *CH*. The function is active until *CH* is found, or if it has analysed *Limit* bytes.

If *CH* is not found within the limit indicated, the value returned is equal to *Limit*. Otherwise, the value returned corresponds to the number of bytes analysed before *CH* was found. *Limit* is an Integer expression locked to 16 bits and is not checked. *CH* is of type Char; *source* is a variable with a value of type *Packed Array of Char*.

The SCNANE function

syntax: `ScanNe(limit, CH, source)`
result: `Integer`

ScanNe functions in the same manner as ScanEq, with the difference that it seeks for the first character different to *CH*.

THE MANAGEMENT OF TOOLBOX ERRORS CALLS

The Apple IIgs ToolBox obeys an error handling convention dealing with errors occurring during execution of a ToolBox function. If an error is detected during execution of a ToolBox routine, the 65816 will have the carry bit set to 1 and the accumulator will contain the corresponding error code. TML Pascal II allows you to recover this information within your programs.

The ISTOOLERROR function

syntax: `IsToolError`
result: `Boolean`

This function returns TRUE if the last ToolBox function called raised an error; if not, it returns FALSE. This function tests the 65816 carry bit to determine if an error occurred. The function must be called immediately after a function call and before another operation can affect the carry bit of the microprocessor. If the ToolBox function call is within an expression, the result of *IsToolError* may be incorrect since the evaluation of the expression may modify the carry bit. In such a case, a program must test the variable _ToolErr.

The _TOOLERR variable

syntax: `_ToolErr`
type: `Integer`

The *_ToolErr* variable contains the error code returned by the last ToolBox function call. A value other than 0 indicates that an error occurred. The compiler generates code that stores the contents of the accumulator into the *_ToolErr* variable immediately after a call to the ToolBox, and before another operation can modify the value.

Examples of using *IsToolError* and *_ToolErr*:

```
h := NewHandle(100,myMemoryID,0,Ptr(0));
if IsToolError then begin
      theErr:=_ToolErr;
```

```
    Writeln('Memory allocation error:',theErr);
    end;
```

Notice that *_ToolErr* was saved to a temporary variable before the call to the Writeln procedure. This is necessary because the Writeln procedure calls upon several ToolBox functions that would modify the contents of *_ToolErr* and would thus give an incorrect result.

Several ToolBox functions are designed to not generate an error, but TML Pascal II does not know this and therefore generates the corresponding instructions to safeguard the error code. If an application does not want the error management code generated, the compiler directive {$ToolErrorCheck-} should be used. See appendix B for more information.

Note: the 1.x versions of TML Pascal used the *ToolErrorNum* variable to return errors produced during ToolBox function calls. This variable is also available in TML Pascal II, but it is preferable to use *_ToolErr* for the purposes of standardisation.

APPENDIX A: ERROR MESSAGES

This appendix lists the possible errors produced within TML Pascal II. These errors may be produced within the editor, the compiler or the linker; take for example GS/OS and I/O errors. Some explanations are provided in order to better explain why and how the error message is generated; and sometimes it is indicated how to correct the error. Certain messages contain the "*" character which basically means that the message you will see on screen will include an identifier, label, or value as determined by TML Pascal II.

ERRORS WITHIN THE EDITOR

Memory is getting low. Close a document window.

TML Pascal II detects that memory is full and advises you to release some memory by closing a document before any of your data is lost. You can also select the RELEASE MEMORY option in the *Preferences* menu.

Can't open that file. The file is already open in another window.

You cannot have the same document open twice.

Error reading file.

An error occurred when reading a document from disk. This can occur if the file is damaged or if the diskette were removed from the drive.

Error saving file.

This error occurs if TML Pascal II cannot save a document to disk. This can occur if a diskette is locked, if the diskette were removed from the drive, or if the diskette is full.

Error deleting file.

This error occurs if you choose DELETE in the GS/OS menu and the diskette was removed from the drive, or if the diskette is locked.

Error renaming file

This error occurs if you chose to rename a file with RENAME command in the GS/OS menu and you either gave an illegal name, the diskette was removed from the drive, or the diskette is locked.

Insufficient memory to complete that operation

This error occurs when an operation could not be performed due to a lack of free memory.

ERRORS OF COMPILATION

Lexical errors

String constant must not exceed source line

This error is generated when a String constant does not end within closing quotation marks.

Error in numeric literal

Indicates that the incorrect syntax was used for a numeric literal value.

Illegal character in input

An illegal character was identified in the source file.

Incomplete program

The end of the file is reached before the Program or Unit is finished by a point (full stop).

End of file encountered while reading a comment

The TML Pascal II compiler reached the end of the file without finding the end of a comment. All comments must start with { or (* and ended by } or *). See chapter 11 for more information.

Syntax errors

These error messages indicate that your program is using an invalid syntax. Although an error message may suggest that a symbol is missing from an illegal expression, it is possible that the error is caused by some other error occurring beforehand in the source code. These errors can sometimes be illusive. To assist in tracking down such errors, see chapters 11 to 20 to familiarise yourself with the Pascal syntax.

- ***Identifier expected***: an identifier is required.
- ***Unexpected symbol***: a symbol not placed correctly.
- ***Integer constant expected***: should be using an Integer constant.
- ***Error in statement***: incorrect instruction.
- ***Error in expression***: incorrect expression.
- ***BEGIN expected***: a missing BEGIN statement is detected.
- ***DO expected***: a missing DO statement is detected.
- ***END expected***: a missing END statement is detected.
- ***IMPLEMENTATION expected***: a missing IMPLEMENTATION directive in a Unit is detected.
- ***INTERFACE expected***: a missing INTERFACE directive in a Unit is detected.
- ***OF expected***: a missing OF statement is detected.

- ***PROGRAM or UNIT expected***: a missing PROGRAM or UNIT directive at the beginning of document is detected.
- ***THEN expected***: a missing THEN statement within an IF condition is detected.
- ***TO or DOWNTO expected***: a missing TO or DOWNTO statement within a FOR loop is detected.
- ***UNTIL expected***: a missing UNTIL statement within a REPEAT loop is detected.
- ***`) : [ ] ; = , .. . :` = expected***

Semantic errors

Duplicate identifier

This error occurs when the same identifier is declared more than once in the same block.

Low bound exceeds high bound

In a table declaration, the lower index value is declared as being higher than the upper index value.

- Identifier is not of appropriate class:
- Identifier not declared:
- Sign not allowed:
- Incompatible subrange types:
- File not allowed here:
- Tagfield must be scalar or subrange:
- Index type must be scalar or subrange:
- Base type must be scalar or subrange:

The specified type must be a scalar or a type such as Integer, Char, Boolean, etc. The type cannot be Real.

Error in type of standard subprogram parameter:

This error occurs when the type of an expression passed into a parameter of a procedure or TML Pascal II predefined function, doesn't match the corresponding formal parameter type. See chapters 19 and 20.

Repetition of parameter list is not identical to previous declaration

This error occurs when the parameters list for a FORWARD declared function or procedure, or within a Unit interface, is repeated, it does not correspond with the declaration.

File value parameter not allowed

File type parameters must always be specified using VAR.

LongInt case/control variable/index expressions are not implemented

TML Pascal II does not allow the use of *LongInt* type expressions, loop control variables, or array indexes within Case expressions.

Missing result type in function declaration

This error occurs if the result type of a function is not declared.

Fixed point formatting allowed only for real types

This error occurs if a decimal point is used on a numeric type other than a Real.

Number of parameters does not agree with declaration

This error occurs if the number of parameters given in a procedure or function call does not match the declaration.

Actual parameter may not be PACKED for VAR formal parameter

This error occurs if a packed parameter value is declared with VAR.

Operands are not assignment compatible

This error occurs if the operands of an expression are not compatible.

Tests on equality allowed only

Means that you can only test two operands for equality.

Strict inclusion not allowed

Strict inclusion is not allowed.

File comparison not allowed

You cannot compare File types.

Illegal type of operand(s)

An operand has a prohibited type.

Type of operand must be boolean

The operand must be have type Boolean.

Set element type must be scalar or subrange

The type of an enumeration must be a scalar or subrange.

Set element types not compatible

The enumeration types are not compatible.

Type of variable is not array
The variable given is not an array (where it should be an array).

Index type is not compatible with declaration
The type of the index does not correspond to its declaration.

Type of variable is not record
The variable given is not a record.

Type of variable must be pointer
The variable must be a pointer type.

Illegal parameter substitution
Type of parameter does not exactly match the corresponding formal parameter.

Illegal type of loop control variable
The type used for the loop control variable is prohibited.

Boolean expression expected
The expression is required to be type Boolean.

Assignment of files not allowed
It is not permitted to assign values to variables of File type.

Label type incompatible with selecting expression
The label type is incompatible with a given expression.

Subrange bounds must be scalar
The upper and lower bound values for a subrange must be scalar.

No such field in this record
An attempt to access a field not existing in the specified record has been made.

Actual parameter must be variable
The parameter given to a procedure or function call must be a variable.

Control variable must not be declared on intermediate level

You cannot declare a control variable on an intermediate level.

Multidefined case label
The CASE statement was declared more than once.

Again forward declared
A FORWARD declaration has been used more than once.

Multidefined label
A label has been defined more than once.

Multideclared label
A label has been declared more than once.

Undeclared label
A label has been used but not declared.

Error in base set
There is an error in the enumeration.

Illegal function result assignment
Incorrect assignment of the function result.

Must EXIT to an enclosing subprogram
You must EXIT to an outer subroutine.

Control variable must not be format
You should not format the control variable.

Assignment to control variable is not allowed
You cannot assign a value to a control variable.

Forward referenced type "*" not completed in previous
The type of a FORWARD reference for "*" was not specified.

Forward declared subprogram "*" not completed in previous
The subroutine “*” that was FORWARD declared was never defined.

Label "*" was declared but not defined in previous block
The label "*" was declared but not initially defined.

Size of string must be between 1 and 255
The size of a string must be between 1 and 255 characters.

@ is not allowed for expressions in INLINE and TOOL subprograms
You cannot use the @ assignment in INLINE or TOOL subroutines.

Type cast to a different size is not allowed
You cannot modify the size of a type by using a type cast.

Too many nested scopes of identifiers
Too many overlapping identifiers.

Too many nested procedures and/or functions
Too many overlapping procedures or functions.

Index expression out of bounds
An index value was used outside of the range declared.

Implementation restriction
Limit of the implementation.

Unit errors

The Unit "*" is required to USE this unit
This error occurs when a Unit is named within a USES declaration of another Unit but not specified within this declaration.

Repetition of unit not allowed
One cannot repeat a Unit within a USES declaration.

This unit must be recompiled
This error occurs when a Unit named in the USES declaration needs to be recompiled. To correct the error, recompile the Unit indicated within the Uses declaration.

Unable to find/open unit's symbol file

This error occurs when a Unit file with extension "p.o" is not in the current prefix, nor in the Unit access path specified in the Preferences menu.

Unable to write unit symbol file for this unit

This error occurs when the compiler cannot create the "p.o" file. The diskette may be locked, absent, or full.

Unit must be recompiled with current version of compiler

When you use a new version TML Pascal, you need to recompile your Units.

Symbol table space exhausted

This error occurs when the number of declarations in a Unit fills the available allocated memory for the symbol table of the Unit. You should modify the size allocated for the symbol table in the Preferences menu.

Linker errors

Out of Memory

Segment "" specified as both CODE and DATA*

These errors occur when you specify the same segment name within a compiler directive {$DSeg segName} and {$CSeg segName}.

Segment "" too large*

A CODE or DATA segment is larger than 64K. You must re-segment your program so that no segment exceeds this limit. See chapter 16.

Unresolved linker reference to symbol ""*

An external label definition cannot be found by the linker. You must check the spelling of the symbol to ensure that it is correct.

Unable to create/open application file

After compiling to disk, the linker tries to write the executable file. This error occurs when the file cannot be created and/or opened. Possible causes could be that the disk is locked or absent.

Error in writing to application file

This error occurs when TML Pascal II is able to create and/or open the application file but an error occurred when trying to write to it. This occurs with a locked or full disk.

GS/OS error codes

This chapter lists the GS/OS errors produced during I/O operations, except for the error codes -1, -2, and -3 generated for specific errors within TML Pascal II routines. For further information on this list of errors, consult the GS/OS Reference handbook.

General Errors

```
$00   no error.
$01   invalid GS/OS call numbers.
$04   numbers of parameters out limit.
$07   active GS/OS.
```

Errors for peripheral I/O

```
$10   device not found.
$11   invalid device request.
$20   invalid request.
$21   invalid state or control code.
$22   bad call parameter.
$23   character peripheral not open.
$24   character peripheral already open.
$25   interrupt vector table full.
$26   non available resources.
$27   I/O error.
$28   no device connected.
$29   active drive.
$2B   Disk is write protected.
$2C   calculation of bytes invalid.
$2D   bad block address.
$2E   the disk was corrupted.
$2F   peripheral removed or absent support.
```

Errors for file I/O

```
$40   invalid pathname.
$43   incorrect reference.
$44   path not found.
$45   volume not found
$46   file not found.
$47   duplicate pathname.
$48   volume full.
$49   volume directory full.
$4A   version error.
$4B   unsupported storage type.
$4C   end-of-file met.
$4D   position out of range.
$4E   non authorised access.
$4F   insufficient buffer.
$50   file already open.
$51   directory errors.
$52   volume type unknown.
$53   invalid parameter.
$54   out of memory.
```

```
$57   duplicate volume.
$58   not a block device.
$59   invalid file level.
$5A   block number out of range.
$5B   invalid access path.
$5C   file not executable.
$5D   system not accepted.
$5F   too many applications in memory.
$60   data not available.
$61   end of directory reached.
$62   invalid FST call.
$63   the file does not contain the specified resource.
```

Errors specific to TML Pascal II

```
-1    Txt file not open for reading.
-2    Txt file not open for writing.
-3    string conversion errors within a Txt file.
```

APPENDIX B: COMPILER DIRECTIVES

TML Pascal II provides a number of directives (or options) that affect compilation and/or how code is generated by the compiler. These compiler directives are to be written between the comment delimiters – that is, between *{..}* or *(*..*)*. A directive will always start with the *$* symbol, it must appear directly after the opening comment delimiter, and it must be followed by one or more characters to indicate which directive is being used.

There are two types of directives: switch-type directives and parameter directives. A switch-type directive includes or removes a possibility by specifying + or - immediately following the directive. A parameter directive specifies one or more string arguments (e.g. file names, names of segments, etc). String arguments end in a space, followed by an asterisk and a closing parenthesis, or a single closing brace. If a string argument contains one of these characters, the string must be put within quotation marks.

Examples:

```
(*$LongGlobals+ *)
{$CSeg NewSeg}
```

THE CDA DIRECTIVE

```
(*$CDA menuName*)
```

The CDA directive is used to inform the compiler that this program implements a CDA application as opposed to a classic GS/OS application. The reason for informing the compiler of this is because the structure of a CDA is quite different from a classic application. In particular, TML Pascal II must generate a special header that contains the name of the CDA as it is to appear in the Apple menu. See chapter 10 for more information on writing your own CDA.

As the compiler is required to generate special code for the CDA before anything else, the compiler directive must appear before the UNIT keyword within your source.

Example:

```
(*$CDA SHRDump*)
UNIT MySHRDump;
...
end.
```

CODE SEGMENT

```
                (*$CSeg segname*)
Default value:  (*CSeg main*)
```

The CSeg directive informs the compiler which segment the following program will be allocated in. The default segment has the reserved name MAIN. For other segments

names, any character strings not containing a space are allowed. See chapter 8 for more information on the use of segments.

DEFINITION PROCEDURE

```
(*$DefProc*)
```

The (*$DefProc*) directive informs the compiler that the next procedure or function in the source code implements a ToolBox definition procedure. A definition procedure is implemented in the same manner as any other procedure or function, except that in this case, the compiler generates some additional code. More precisely, the compiler generates code that sets the DBR (data bank register) equal to the memory bank that contains the global variables of the Pascal program. When leaving the procedure, specific code is generated to restore the original DBR value.

DATA SEGMENT

```
                (*$DSeg segname*)
Default value:  (*$DSeg "global*)
```

The option (*$DSeg*) informs the compiler that the global variables following will be allocated in the specified segment. The default segment has the reserved name “GLOBAL. For other segment names, you can use any character string not containing a space, and preceded by the “ (double quote) character. The *“global* data segment is the particular segment in which the compiler can use absolute addressing; which is more powerful than long absolute addressing. See chapter 8 for more information regarding the use of data segments.

EXTERNAL VARIABLE REFERENCES

```
                (*$J+*) or (*$J-*)
Default value:  (*$J-*)
```

The external variable reference directive of TML Pascal II indicates which of the following global variables will have an area of memory allocated. After the (*$J+*) directive, any global variable declarations are treated as external references to global variables declared elsewhere. This directive is used within TML Pascal II so that you can use global variables defined in external ASM or C modules.

Example:

```
VAR GlobVar1: integer;

(*+J*)
GlobVar2: integer;
(*-J*)

GlobVar3: integer;
```

LONG GLOBALS

```
                (*$LongGlobals+*) or (*$LongGlobals-*)
Default value:  (*$LongGlobals-*)
```

This option informs the compiler to switch on (+) or switch off (-) the generation of long absolute addressing for global variables within the *"global* data segment. Normally, the compiler generates code which forces the DBR (data bank register) to be the same as the memory bank which contains the global variables allocated within the *"global* data segment. However, on some occasions, the program will require that these banks be different. In these cases, it is necessary to indicate to the compiler that you want to access global variables of the *"global* segment by using the long absolute addressing method.

THE NDA DIRECTIVE

```
(*$NDA period event Mask menuName*)
```

The NDA directive informs the compiler that the program is an NDA and not a standard GS/OS application. The reason for doing this is because the structure of an NDA is quite different from that of a normal application. In particular, TML Pascal II must generate special header code indicating the period (in 1/60ths of a second), which is the periodic access interval of the NDA; the EventMask which describes the event types managed by the NDA; and the Name which will appear in the list of NDA's (in the Apple menu). See chapter 9 for further information on the structure of an NDA.

As the compiler must generate special header code for the NDA, this option must appear in your source before the reserved word UNIT.

Example:

```
(*$NDA 60 -1 TMLClock*)
UNIT TMLClock;
...
END.
```

STACK SIZE

```
                (*$StackSize numBytes*)
Default value:  (*$StackSize 8096*)
```

The *StackSize* directive informs TML Pascal II the size (in bytes) to allocate for the stack during execution of the application. This stack is used during execution to preserve the return address of subroutines as well as local variable values. Intensive use of local variables will therefore require a larger stack size. The default stack size value is 8K (8096 bytes) (Note: the 1.1 version of TML Pascal II that I have has a default stack size of 4K). If a program requires more or less memory, the required size should then be specified. However, the stack size must be at least 1K (1024 bytes) and no more than 40K (40960) is allowed (although TML Pascal II does not check the size specified in the directive). See appendix D for more information regarding the use of the stack during program execution.

Note: this option MUST appear within your code before the reserved word PROGRAM.

Example:

```
(*StackSize 10240*)
PROGRAM MyProgram;
begin
...
end.
```

UNIT SEARCH PREFIX

```
                (*$U GSOS_prefix*)
Default value:  (*$U :0*)
```

This option makes it possible to search the specified prefix for Units with the ‘P.O’ suffix. TML Pascal II does not recompile the interface Units indicated in a USES statement, but instead loads a table of precompiled declarations stored in a ' P.O' file. In order to be able to find these files, TML Pascal II uses an access prefix to the units; this prefix is *:0* by default (the current prefix). One can therefore specify a particular prefix to search with the following example directive:

```
USES TYPES,
(*$U :TML:MYSTUFF:*) HandyRoutines;
```

Note that if the 'P.O' files cannot be found within the prefix specified in the compiler directive ($U*), TML Pascal II will try to find them using the search prefix specified in the Preferences menu (see chapter 5). If a 'P.O' file cannot be found after searching all of the specified prefixes, an error will be returned.

TOOLBOX FUNCTION ERRORS

```
                (*$ToolErrorChk+*) or (*$ToolErrorChk-*)
Default value:  (*$ToolErrorChk+*)
```

This directive makes it possible for an application to handle errors generated within calls to ToolBox functions. As we saw in chapter 20, TML Pascal II generates a *STA _ToolErr* after each ToolBox function call. Therefore, the global variable _ToolErr will always contain the error code returned by the last function called. A value of *_ToolErr* other than 0 indicates that an error occurred during execution of the last ToolBox function, and the value of *_ToolErr* is used to determine which type of error occurred. In the majority of cases, an application does not need to retrieve the error after each ToolBox function call. Additionally, you can disable this option in order to produce less bulky code.

APPENDIX C: TOOLBOX UNITS

The ToolBox consists of an important collection of routines that facilitate programming the Apple IIgs. The ToolBox implements the graphics management via QuickDraw as well as desktop management (including windows, menus, dialogs, controls, etc.). As one saw in chapter 8, TML Pascal II allows access to the ToolBox by using a number of different Units. This appendix has a comprehensive list of Units at your disposal.

Due to the extent of these files, they will not be listed in the present documentation. Instead we ask that the reader return to your favourite word processor to edit and to print the '. P' files located in the LIBRARIES folder of the SOURCE CODE LIBRARY diskette.

APPENDIX D: THE HEART OF TML PASCAL II

TML PASCAL II MEMORY MANAGEMENT

The environment in which an application functions can be divided into 4 parts:

1. application code
2. global variables
3. execution stack
4. available memory

These 4 components of an application must coexist in the RAM of the Apple IIgs. The memory of the Apple IIgs is partitioned into 64K banks, and is managed by the *MemoryManager*. The standard Apple IIgs (ROM0 or ROM01) includes 4 RAM banks of 64K numbered $00, $01, $E0 and $E1. RAM expansion cards can be added to provide additional memory between banks $02 and $7F. Tthe other banks are reserved and are not available for use.

Application code

An application can consist of one or more segments. Smaller programs often consist of only one segment, but very large applications are divided into several segments because each segment cannot exceed 64K. In fact, a segment cannot overlap two memory banks. TML Pascal II generates independent modules for each procedure and declared function within the program. Each one of these loaded modules is associated with a segment name used to organise the modules into the correct segments during linking. The default segment name is MAIN. When an application becomes sufficiently large to more than one segment, one must use the directive *(*$CSeg segname*)* in order to associate the given segment name with the procedures following. One can restore any following procedures to the default segment by using the compiler directive *(*$CSeg main*)*.

Global variables

TML Pascal II stores global variables in a data segment. By default, the data segment has the name *“global*.

The linker uses the loaded segment names associated with every data segment in order to group them into the correct segment. Most programs are in theory made up of only one data segment, but applications requiring a large number of global variables have to be split into several data segments. In fact, each data segment cannot exceed the size of a memory bank - that is, no greater than 64K.

When an application requires several data banks, one must use the directive *(*$DSeg segname*)* in order to indicate to the compiler that you wish to allocate another segment for the variables following. You can restore future global variable declarations to the main segment by issuing the compiler directive *($DSeg “global*)*.

During execution of the initialisation code generated by TML Pascal II, the data bank register (the DBR) of the 65816 is set to point to the memory bank containing the global variables declared in the *“global* segment. Because of this, references to global variables

within the *"global* segment can use *absolute addressing*. The global variables in other data segments are addressed in *long absolute* mode (slower and consumes more memory).

Execution stack

The execution stack is a special storage block that the application uses to store procedure and function return addresses and to preserve parameter and local variable values. During execution of the applications initialisation code, stack memory is allocated in bank $0 (because it is the only memory bank in which the stack register of the 65816 can function). By default, TML Pascal II defines a stack of 8K (8096 bytes) (Note: the version of TML Pascal II that the translator has, defaults to a 4K stack). If an application requires more or less space for its stack, you must specify it by using the compiler directive *(*$StackSize numbytes*)*.

The directive *(*$StackSize numbytes*)* must appear before the reserved word PROGRAM. For example, the following code will generate a 10K stack for the application:

```
(*StackSize 10240*)
PROGRAM MyApp;
...
```

Desktop accessories do not have initialisation code for allocating and initialising the execution stack since they function within the environment of an application. Therefore, when writing your applications, you must ensure that you leave sufficient free memory in the stack for using NDA's or CDA's. On the other hand, when writing a desktop accessory, it is best to use the least possible stack space. Always remember that the default stack space size is 8K.

Neither the Apple IIgs nor TML Pascal II can determine the stack size used by an application. If you reserve an insufficient amount of space within the execution stack, the surrounding memories areas will be destroyed during execution of your application.

Available memory

The *available memory* is memory not used by an applications implementation code, its data segment, and its execution stack. This free memory is put at the disposal of the *MemoryManager* of an application, using the routines within the Unit *GSIntf.Pas*. One can also reserve and release memory from the available memory by using the procedures New and Dispose. Each application will need to reserve at least one storage block in bank 0 for tool initialisation. The majority of applications will need more than one page reserved in bank 0. These pages are known as ZERO PAGES.

DATA REPRESENTATION

This chapter will show you how each data type is stored within memory of the Apple IIgs. You will note that the 65816 stores each byte of a word using the little-endian representation; this means that the most significant bits are in the highest memory area. For example:

```
type experiment = packed record
      case integer of
            0: (int: integer);
            1: (highbyte: 0..255;
```

```
                lowbyte : 0..255);
        end;
```

This record does not produce the result which you may think. With the 65816, by referring to *highbyte*, one returns the least significant byte of the *int* Integer, and not the most significant byte. (Note: anyone familiar with Intel microprocessors may already be familiar with these concepts, as Intel microprocessors are also little-endian). The following paragraphs will explain to you how Pascal stores its data types.

Integer: the two's complement of a signed integer in the range -32768/32767 and requiring 2 bytes of storage. Bit 15 is the sign bit.

```
7 0 15 8

I ISI I (where S is the sign bit)
```

LongInt: the twos complement of an integer in the range -2147483648/2147483647 and requiring 4 bytes of storage. Bit 31 is the sign bit.

```
!7 0!15 8!23 16! 24!
```

Boolean: an enumerated type of (False, True) requiring a single byte of storage, with the actual boolean value found in bit 0. A byte of storage is used within a packed array or a record.

```
!7 0!15 8!
```

Char: enumerated type of ASCII characters having 256 possible values. Each character value requires two bytes of storage, with the actual value found in the most significant byte (bits 7-0). A byte of storage is used within a packed array or a record.

```
!7 0!15 8!
```

Enumeration: an unsigned byte or a twp byte integer. If the enumeration type contains less than 128 constants, the first value of the type occupies only one byte of storage if it is used in a packed array or a record; otherwise it occupies two bytes.

```
!7 0!       <= 128 enumerations

!7 0!15 8!  > 128 enumerations
```

Subrange: a signed byte, a word, or a long word. If the range is within -128/127, a word is used for unpacked structures or simple variables; but in a packed array or a record, only one byte is used to represent the subrange. If the range is within -32768/32767, a word is used; otherwise a long word is used for the representation.

```
!7 0!                   -128..127

!7 0!15 8!              -32768..32767

!7 0!15 8!23 16!31 24!  Others
```

Single: a real number using 32 bits represented in IEEE single precision format, implemented in SANE as type Single.

```
31 30 23!22 0

!S! exponent ! significand !
```

Double: a real number using 64 bits represented in IEEE double precision format, implemented in type SANE Double.

```
63 62 52 51 0 !S! exponent ! significand !
```

Real/Extended: a real number using 80 bits represented in IEEE standard wide format. Both are implemented in SANE as type Extended.

```
79 78 64 63 0 !S! exponent ! significand !
```

String[n]: Pascal strings are of *n+1* bytes length. Each string consists of a length byte (the length of the String, not part of the string itself) followed by the bytes containing the ASCII characters making up the string.

```
byte 1 2 3 ... n+1

!long ! ! ! ... ! !
```

Pointers: a 24 bit memory address occupying 4 bytes of storage. Only 3 bytes are used to store a 24 bit address, and bits 31-24 are always set to 0. The NIL pointer is represented by a 32 bit zero value.

```
!7 0!15 8!23 16!31 24!
```

Sets: A sequence of bytes up to a maximum of 32 bytes, or 256 bits that represent the size of the base type. The number of bytes used is the minimum number required to represent this size. The ordinal value of the base type is represented by only one bit. If the ordinal value forms part of a set, then its bit is to 1, otherwise it is set to 0. If the ordinal values of the basic 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 4 bytes will be used to represent it, etc.

Files: a 22 byte data structure used internally by TML Pascal II for handling files. In addition to the file variable, the associated open file on an external disk has an access buffer allocated in free memory for management by GS/OS; a text file has a buffer of 256 bytes.

Arrays / Records: the components of an unpacked array or record are allocated consecutively as more are defined. Arrays are stored line by line. That is, the last index is higher than the first. The components of records are allocated in the order that they appear within a declaration. The TML Pascal II implementation can only pack data at the byte level, with bit level packing not possible. A data type is represented by a byte for packed records, if and only if, the number of bits necessary to store all the values of the type is less than or equal to 8 bits. For example, the standard *Char* and *Boolean* types require less than 8 bits to represent all their possible values; therefore, in a packed record, a *Char* or *Boolean* type can be represented by a byte; otherwise a word would have been required.

CALLING CONVENTIONS

Calling a subroutine

TML Pascal II passes parameters by using the stack during a call to a subroutine. Before calling a procedure or function, the parameters are pushed onto the stack in the same order as they were declared. If a function is called, storage for the result is allocated on the stack prior to pushing the parameters. When a call is completed, control again passes to the main program with the resulting parameters popped from the stack, but the result of the function (if it is a function) is left on the stack. The program must pop the result from the stack when it uses it.

Here is an example of a procedure call:

```
lda pppp          ; push the 1st parameter onto the stack
pha
...
lda pppp          ; push the last parameter onto the stack
pha
jsl >Aproc        ; call the procedure
                  ; with the parameters on the stack
```

Here is an example of a function call:

```
pha               ; reserve space for the result
lda pppp          ; push the 1st parameter onto the stack
pha
...
lda pppp          ; push the last parameter onto the stack
pha
jsl >Afunc        ; call the procedure
                  ; with the parameters on the stack
                  ;
pla               ; pull the result from the stack
                  ; and place it into the accumulator
```

Subroutines are always called in 65816 native mode (accumulator and index registers using 16 bits). However, if the processor is not in native mode before a call, it is forced into native mode prior to the call being executed. For example, if the accumulator is using 8 bits and the index registers are using 16 bits, the following code is generated:

```
                  ; accumulator using 8 bits
rep #$20          ; accumulator using 16 bits
LONGA ON
jsl >ASubprog
```

If the called subroutine is declared at a level other than the global level (i.e. somewhere other than the main block or a Unit), then a static link is pushed onto the stack directly after the parameters. This static link is used to address local variables stored within the overlapping stack. Because of this static link, the address of an overlapping subroutine should never be passed to a ToolBox definition procedure routine since this is not the convention used by the ToolBox tools.

Variable parameters

Variables parameters (VAR) are always passed by reference to the formal parameter. That is, like a pointer to the storage area occupied by the current parameter. The pointer is passed in the form of a 32 bit (4 bytes) value. The most significant word is pushed onto the stack before the least significant word. In the following example, the global variable GlobVar is passed as a VAR parameter by using the absolute addressing:

```
pea GlobVar|-16   ; push the MSB onto the stack
pea GlobVar       ; push the LSB onto the stack
```

Value parameters

Value parameters are either passed with their values on the stack, or by reference, depending upon the size of the value. If the size of the value parameter occupies 4 bytes or less, the value is passed in via the stack. If the size of the value is greater than 4 bytes, a 32 bit pointer to the value is pushed onto the stack. A called procedure or function then copies the value into local storage in such a way that any modification of the formal parameter value does not affect the actual parameter value.

Static parameters

Static parameters are pushed onto the stack in the same way as value parameters. The difference between value parameters and static parameters is that if the size of the actual parameter is larger than 4 bytes, the called procedure or function does not copy the value into local storage for the formal parameter. Therefore, it is illegal to give a new value to the static formal parameter since it will change the value of the current parameter. Static parameters were introduced into TML Pascal II in order to preserve the size of the execution stack for the storage of formal parameters for the purpose of accelerating data processing.

Note: TML Pascal II does not check if a new value is assigned to a static parameter. It is up to the programmer to ensure the correct usage of static parameters.

Functions results

The storage for function results is reserved on the stack by the calling subroutine before the pushing the parameters onto the stack. If the result of the function is of type Integer, LongInt, Char, Boolean, Subrange, Enumeration, pointer, or Single real, then 2 or 4 bytes of storage are allocated. If the result type only requires one byte, 2 bytes are allocated and the value is stored in the lowest address.

If the type of the result is Double, Comp, Extended, Array, String, or record, then the calling subroutine will allocate temporary space within its stack for the result, and it will push a pointer of 4 bytes onto the stack pointing to this temporary space. The calling subroutine pulls the pointer from the stack when the function returns the result and the temporary storage area is released if there is no reference to the value.

I/O code

Each Pascal procedure and function starts and finishes using standard entry and exit code.

Here is the standard entry code:

```
phd               ; save the previous stack pointer
tsc
sec
sbc #xx
tcd               ; create a new stack pointer
clc
adc #yy
tcs               ; allocate local storage
```

Firstly, the current direct page register is saved. The direct page register is used as a pointer. The saved pointer is called the dynamic link and is used to return things just as they are.

After having saved the stack pointer, *xx* bytes are subtracted from the current stack pointer in order to create the new pointer. *xx* is calculated so that the first word of stack storage (i.e. the function result or the first parameter) is a multiple of 254 within the direct page. Thanks to this choice, the parameters and the majority of local variables can be addressed using direct page addressing.

When the stack pointer is defined, *yy* bytes are added to this value in order to allocate the storage space necessary for local variables, value parameters copied locally, and for the needs of the compiler. No register is saved and it is assumed that the processor is in native mode. Here is the standard exit code:

```
tdc
clc
adc #xx
tcs               ; deallocate local storage
pld               ; recover the stack pointer

lda 2,S
sta mm,S
lda 1,S           ; the return address is before
sta mm-1,S        ; the parameters
tsc
clc
adc #mm-2         ; deallocate the parameters
tcs
rtl
```

Local storage is initially deallocated by adding the value *xx* to the stack pointer. One then recovers the previous stack pointer thanks to the *PLD* instruction. Next, the parameters are “removed" by moving the return address above the first parameter and by repositioning the stack pointer to this new address. Finally, *RTL* is executed to return to the calling subroutine. Note that if a procedure or function does not have parameters, the exit code is then:

```
tdc
clc
adc #xx
tcs               ; deallocate local storage
pld               ; recover the stack pointer
rtl
```

APPENDIX E: COMPARISON OF TML PASCAL II AND TML PASCAL I

TML Pascal II implements new functions as compared to the old version TML Pascal I. In this chapter we will highlight the differences between the two versions so that former users of TML Pascal I can easily adapt to this new product. In what follows, the differences between the two versions are presented chapter by chapter.

CHAPTER 1: DISCOVER TML

The most important difference between TML I and TML II is that TML II is now specified for use with GS/OS 5.0. The applications developed with TML II do not work with system software prior to version 5.0, nor will they work with versions of ProDOS 16. This limitation is imposed mainly due to the implementation of the Resource Manager in TML II, and tools not being available in system versions prior to 5.0.

CHAPTER 2: USING THE DESKTOP ENVIRONMENT

The TML II editor allows an unlimited number of windows to be open on the screen. Additionally, within each window, you can use a font with a particular style as well as personalised tabulation. The editor now also implements an UNDO command. Lastly, the editing environment makes it possible to create an edit window for the resource files.

CHAPITRE 3: CREATE A PROGRAM

The file naming convention denoting files has changed with TML II. Source files must now end with a suffix *'.p'* instead of *'.pas'*.

The name of compiled Unit files is also changed: instead of *'.usym'*, the suffix added by the compiler is from now on *'.o'*, which gives a total suffix of *'.p.o'*. These conventions make it possible for the compiler to easily find the modules which it needs. The menu item RESOURCES was added to the COMPILE menu in order to be able to specify which resources are to be compiled with the final application.

CHAPITRE 4: RESOURCES

The resources are a new function of TML II. Programs written with TML I should be converted without too much trouble into TML II by allowing them to draw upon the functionality of the Resource Manager.

CHAPTER 7: GRAPHIC APPLICATIONS

TML I implemented 'plain vanilla' applications. This has been replaced by 'textbook graphics' applications. With the original, a program could specify the parameters *(Input, Output)* within itself to indicate that it wanted to work in the Plain Vanilla environment. Example:

```
Program Test(Input, Output);
```

Plain Vanilla connects to the graphic screen in 640 mode and creates a window entitled 'TML PASCAL'. This makes it possible to develop graphic applications easily.

TML Pascal II has a new procedure called *'Graphics'*. This procedure accepts a parameter that specifies the graphic mode: 320 or 640; and instead of just opening a window on the screen, it makes it possible to work with the whole screen.

CHAPTER 8: DESKTOP APPLICATIONS

Applications can, from now on, use defined resources to create menus, windows, dialogues, etc. The resources are initially created from within the TML Resource Editor. One specifies the resources to be compiled with the main program by using the Resources menu item found within the Compile menu.

CHAPTER 9: NDA

The source code of an NDA can from now on be written in the form of a Unit. An NDA does not have a MAIN program, and it is not necessary to use the same structure as used for standard programs.

CHAPTER 10: CDA

The source code of a CDA can from now on be written in the form of a Unit. An CDA does not have a MAIN program, and it is not necessary to use the same structure as used for standard programs.

CHAPTER 11: RESERVED WORDS

TML II defines 3 new real constants in order to aid in the writing of numerical applications. These numbers are *Inf* (infinite), *NaN* (Not A Number) and *pi*.

CHAPTER 19: I/O

TML II from now on implements the OPEN procedure to open a file for random read/write access.

TML II is GS/OS compatible. Because of this, file names must respect Prodos 16 or GS/OS syntax. Moreover, the names of devices can be used. For example, the name of the printing device is now *'.PRINTER'* instead of *'PRINTER'*. It should be noted that the name *'.PRINTER'* must be written using capital letters.

CHAPTER 20: STANDARD PROCEDURES AND FUNCTIONS

The GRAPHICS procedure is new in TML II. The following standard functions and procedures were renamed in TML II in order to conform to Apple's specifications:

Name in TML I	Name in TML II
BitAnd	BAND
BitOr	BOR
BitXor	BXOR
BitNot	BNOT
BitSL	BSL
BitSR	BSR
BitRotl	BROTL
BitRotr	BROTR
HiWord	HiWrd
LoWord	LoWrd

In TML II, HiWord and LoWord are reserved for the functions in IntMath.p. The preset global variable returning an error code is from now called *'_ToolErr'*, but the old name *'_ToolErrorNum'* is also accepted.

ANNEXE B: COMPILER DIRECTIVES

The following compiler directives were renamed:

```
$DeskAcces renommé en $NDA
$P $U
$XrefVar $J
```

The following directives have been added since version 1 (and were thus already available in version 1.5):

```
$CDA
$DefProc
```

ANNEXE C: INTERFACES

The ToolBox interfaces were largely modified in order to be compatible with system 5.0. Certain interfaces were added.

ANNEXE D: THE HEART OF TML PASCAL II

The types Boolean, Enumeration and the small sets of Integers are represented using 2 bytes instead of one. However, when these types are a *Packed Record* or *Array*, they do not use any more than one byte of memory.