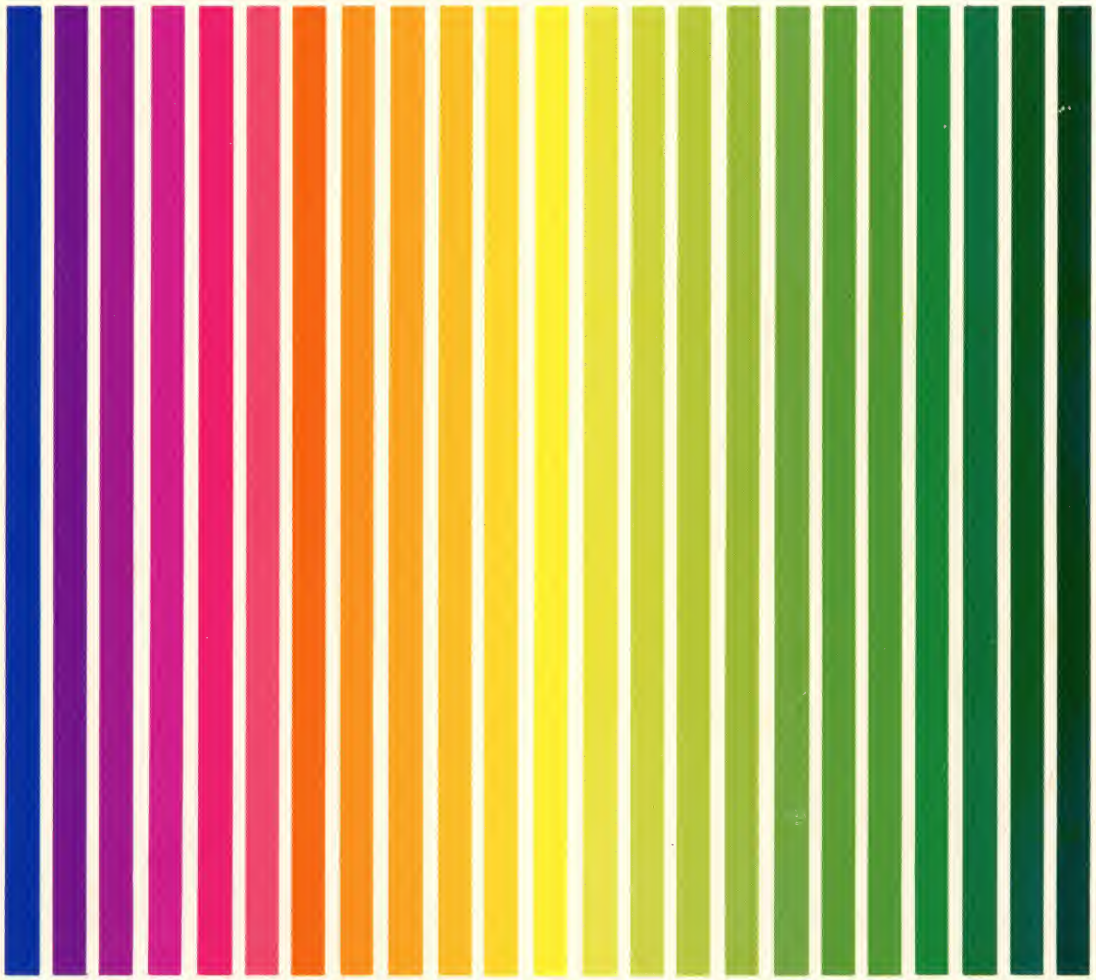
APX ATARI® PROGRAM EXCHANGE



ATARI® PASCAL LANGUAGE SYSTEM

APX-20102

March 1982

User-Written Software for ATARI Home Computers

ATARI®

PASCAL LANGUAGE SYSTEM

REFERENCE & OPERATIONS MANUAL

COPYRIGHT 1982 ATARI, INC. © ALL RIGHTS RESERVED

Copyright and right to make backup copies. On receipt of this computer program and associated documentation (the software), ATARI grants to you a nonexclusive license to execute the enclosed software and to make backup copies of the computer program for your personal use only, and only on the condition that all copies are conspicuously marked with the same copyright notices as appear on the original. This software is copyrighted. You are prohibited from reproducing, translating, or distributing this software in any unauthorized manner.

Notice: The names and addresses used in this manual are fictitious and are included for demonstration purposes only.

TRADEMARKS OF ATARI

The following are trademarks of Atari, Inc.

ATARI®
ATARI 400™ Home Computer
ATARI 800™ Home Computer
ATARI 410™ Program Recorder
ATARI 810™ Disk Drive
ATARI 820™ 40-Column Printer
ATARI 822™ Thermal Printer
ATARI 825™ 80-Column Printer
ATARI 830™ Acoustic Modem
ATARI 850™ Interface Module
ATARI Program-Text Editor
ATARI Disk Operating System (DOS II)
ATARI BASIC

****

Distributed by

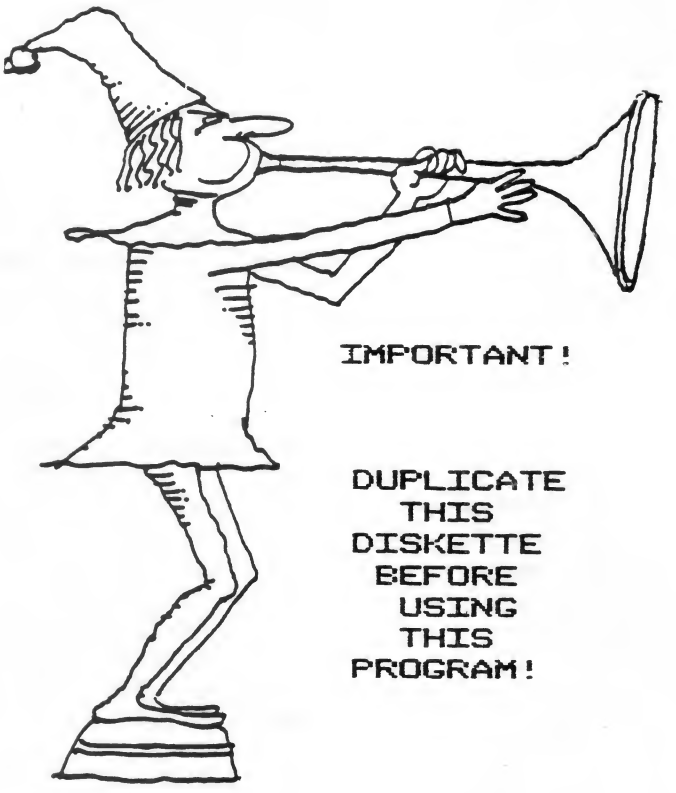
The ATARI Program Exchange
P. O. Box 427
155 Moffett Park Drive, B-1
Sunnyvale, CA 94086

To request an APX Software Catalog, write to the address above, or call toll-free:

800/538-1862 (outside California)
800/672-1850 (within California)

Or call our Sales number, 408/745-5535.

****



This APX diskette is unnotched to protect the software against accidental erasure. However, this protection also prevents a program from storing information on the diskette. The program you've purchased involves storing information. Therefore, before you can use the program, you must duplicate the contents of the diskette onto a notched diskette that doesn't have a write-protect tab covering the notch.

To duplicate the diskette, call the Disk Operating System (DOS) menu and select option J, Duplicate Disk. You can use this option with a single disk drive by manually swapping source (the APX diskette) and destination (a notched diskette) until the duplication process is complete. You can also use this option with multiple disk drive systems by inserting source and destination diskettes in two separate drives and letting the duplication process proceed automatically. (Note. This option copies sector by sector. Therefore, when the duplication is complete, any files previously stored on the destination diskette will have been destroyed.)

V. Additional Terms and Conditions

A. Licensee understands and agrees that:

(1) The Run-Time System is distributed on an "as is" basis without warranty of any kind by Atari.

(2) The entire risk as to the performance and quality of the Run-Time System is with Licensee.

(3) Should the Run-Time System as incorporated into Licensee's products prove defective following its purchase, Licensee and not Atari, Atari's distributors, or retailers, assumes all costs associated with or resulting from use of Licensee's products including all necessary repair or servicing.

(4) Atari shall have no liability to Licensee or to customers of Licensee for loss or damage, including incidental and/or consequential damage, caused or alleged to be caused, directly or indirectly, by the Run-Time System. This includes, but is not limited to, any interruption in service or loss of business or anticipatory profits resulting from the use or operation of the Run-Time System.

B. Licensee shall indemnify and hold Atari harmless from any claim, loss, or liability allegedly arising out of or relating to the operation of the Run-Time System as used by Licensee or customers of Licensee pursuant to this Agreement.

C. Licensee shall not suggest, imply or indicate in any manner that any of his/her software products which incorporate or use the licensed Run-Time System are approved or endorsed by Atari.

D. Licensee acknowledges that a failure to conform to the provisions of Subsection C of Section V will cause Atari irreparable harm and Atari's remedies at law will be inadequate. Licensee acknowledges and agrees that Atari shall have the right, in addition to any other remedies, to obtain an immediate injunction enjoining any breach of Licensee's obligations set forth in Section V.C above.

E. No waiver or modification of any provisions of this Agreement shall be effective unless in writing and signed by the party against whom such waiver or modification is sought to be enforced. No failure or delay by either party in exercising any right, power or remedy under this Agreement shall operate as a waiver of any such right, power or remedy.

F. This Agreement shall bind and work to the benefit of the successors and assigns of the parties hereto. Licensee may not assign rights or delegate obligations which arise under this Agreement to any third party without the express written consent of Atari. Any such assignment or delegation, without written consent of Atari, shall be void.

G. The validity, construction and performance of this Agreement shall be governed by the substantive law of the State of California and of the United States of America excluding that body of law related to choice of law. Any action or proceeding brought to enforce the terms of this Agreement shall be brought in the County of Santa Clara, State of California (if under State law) or the Northern District of California (if under Federal law).

H. In the event of any legal proceeding between the parties arising from this Agreement, the prevailing party shall be entitled to recover, in addition to any other relief awarded or granted, its reasonable costs and expenses, including attorneys' fees, incurred in the proceeding.

VI. Specific Disk Operating System Exclusion

The license granted herein does not relate in any way to the ATARI® Disk Operating System, DOS II. Inquiries relating to such a license should be sent to:

Atari, Inc.
Home Computer Division
60 East Plumeria Drive
San Jose, CA 95134
Attn: Software Acquisition Group

For Atari:

Atari, Inc.

1265 Borregas Avenue

P.O. Box 427

Sunnyvale, CA 94086

By: Bruce W. Irvine

Name: Bruce W. Irvine

Title: V.P., HCD Software

Date: 2-25-82

#59(A3) 2/23/82

TABLE OF CONTENTS

CHAPTER 1:	ATARI PASCAL INTRODUCTION AND OVERVIEW	1
1.1	Manual Overview	2
1.2	System Overview	3
1.3	System Requirements	3
1.4	Run-Time Requirements	4
1.5	ATARI Pascal Distribution Diskette Information	4
CHAPTER 2:	HOW TO OPERATE THE PASCAL LANGUAGE SYSTEM	6
2.1	Compile, Link and Run a Sample Program	7
2.1.1	Compile Sample Program	7
2.1.2	Link Sample Program	9
2.1.3	Run Sample Program	11
2.2	Compiler Operation	12
2.2.1	Invocation and Filenames	12
2.2.1.1	DOS and QUIT Options	12
2.2.1.2	Compile	12
2.2.2	Compilation Data	13
2.2.3	Compiler Toggles	14
2.2.3.1	Entry Point Record Generation (E)	14
2.2.3.2	Include Files (I)	14
2.2.3.3	Strict Type and Portability Checking (T,W)	14
2.2.3.4	Run-Time Range Checking (R)	15
2.2.3.5	Run-Time Exception Checking (X)	15
2.2.3.6	Listing Controls (L,P)	15
2.2.3.7	Summary of Compiler Toggles	16
2.2.4	Built-in Routines and Include Files	17
2.2.5	Error Messages	18
2.2.6	Line Numbers	18
2.3	Linker Operation	19
2.3.1	Invocation and Commands	19
2.3.2	Linker Option Switches	19
2.3.2.1	Run-Time Library Search (/S)	19
2.3.2.2	Memory Map (/M)	19
2.3.2.3	Load Map (/L) and Extended Load Map (/E)	19
2.3.2.4	Program (/P) and Data (/D) Origin	20
2.3.2.5	Continuation Lines (/C)	20
2.3.2.6	Linker Input Command File (/F)	20
2.3.2.7	Linker Switch Summary	21
2.3.2.8	Relocatable File Requirements	21
2.3.2.9	Linker Error Messages	21
2.3.2.10	Attributes of Linkable Modules	22
2.4	Object Program Execution	23
2.5	ATARI Program-Text Editor (MEDIT)	24
2.5.1	Running the ATARI Program-Text Editor	24
CHAPTER 3:	ATARI PASCAL LANGUAGE SYSTEM EXTENSIONS	25

7.7	Declaration and Denotation of Variables	91
7.7.1	Entire Variables	92
7.7.2	Component Variables	92
7.7.2.1	Indexed Variables	92
7.7.2.2	Field Designators	92
7.7.2.3	File Buffers	92
7.7.3	Referenced Variables	92
7.8	Expressions	93
7.8.1	Operators	94
7.8.1.1	The Operator NOT	94
7.8.1.2	Multiplying Operators	94
7.8.1.3	Adding Operators	94
7.8.1.4	Relational Operators	94
7.8.2	Function Designators	94
7.9	Statements	95
7.9.1	Simple Statements	95
7.9.1.1	Assignment Statements	95
7.9.1.2	Procedure Statements	96
7.9.1.3	GOTO Statements	96
7.9.2	Structured Statements	96
7.9.2.1	Compound Statements	96
7.9.2.2	Conditional Statements	96
7.9.2.2.1	If Statements	96
7.9.2.2.2	Case Statements	96
7.9.2.3	Repetitive Statements	97
7.9.2.3.1	While Statements	97
7.9.2.3.2	Repeat Statements	97
7.9.2.3.3	FOR Statements	97
7.9.2.4	With Statements	97
7.10	Procedure Declarations	98
7.10.1	Standard Procedures	100
7.10.1.1	File Handling Procedures	100
7.10.1.2	Dynamic Allocation Procedures	101
7.10.1.3	Data Transfer Procedures	101
7.10.2	FORWARD	101
7.10.3	CONFORMANT ARRAYS	102
7.11	Function Declarations	104
7.11.1	Standard Functions	104
7.11.1.1	Arithmetic Functions	104
7.11.1.2	Predicates	104
7.11.1.3	Transfer Functions	104
7.11.1.4	Further Standard Functions	104
7.12	INPUT AND OUTPUT	106
7.12.1	The Procedure READ	106
7.12.2	The Procedure READLN	106
7.12.3	The Procedure WRITE	106
7.12.4	The Procedure WRITELN	106
7.12.5	Additional Procedures	106
7.13	Programs	108

APPENDIX A: LANGUAGE SYNTAX DESCRIPTION 109

APPENDIX B: RESERVED WORDS 117

APPENDIX C: ERROR MESSAGES 118

APPENDIX D: ATARI PASCAL FILE I/O 126

APPENDIX E: BIBLIOGRAPHY 142

APPEMDIX F: PLAYER/MISSILE DEMO PROGRAM 143

APPENDIX G: HELPFUL HINTS 152

INDEX 153

TABLE OF FIGURES

Figure 1-1 Schematic Diagram of ATARI Pascal Operation 3

Figure D-1 File Input and Output 130

Figure D-2 Text Files 138

Figure D-3 Writing to a printer and number Formatting 140

PREFACE

PASCAL - WHAT IS IT?

Pascal was created by Niklaus Wirth to facilitate teaching a systematic approach to computer programming and problem solving. This high-level structured programming language is suited for professional software developers, making it an excellent tool for developing and maintaining programs.

PURPOSE OF THIS MANUAL

This reference and operations manual defines the language features of ATARI Pascal and can help you to understand how to use these features. This manual assumes familiarity with the Jensen and Wirth's "Pascal User Manual and Report" and/or International Standards Organzation (ISO) draft standard (DPS/7185). The standard Pascal features that differ in ATARI Pascal from those in the standard and in Jensen and Wirth's "Report" are described here. This manual also contains information on how to operate the compiler and linker; a description of the implementation of ATARI Pascal data types; and a summary of built-in features and examples of their usage.

AUDIENCE

This manual is specifically designed for advanced programmers who are familiar with Pascal and with the features of the ATARI 800 Home Computer System. This manual is not suited for learning Pascal or the ATARI 800 Home Computer.

HOW TO USE THIS MANUAL

We recommend starting with the Introduction and Overview (Chapter 1) and then proceed through Chapter 2, which describes how to operate the system, recommendations for backup and a sample program to get you started. The rest of the manual is technical and should be referred to as needed.

PRODUCT CONSIDERATIONS

The ATARI Pascal Language System was designed for use by experienced software developers. The steps required to compile an ATARI Pascal program are time consuming. Memory limitations, diskette capacity and access time will affect product performance. As with other APX programs, ATARI does not support this product after the sale.

REPORTING PROBLEMS

All documented problems submitted to The ATARI Program Exchange will be studied and considered in future revisions of this product.

CHAPTER 1: ATARI PASCAL INTRODUCTION AND OVERVIEW

This manual describes the ATARI Pascal Language System being offered through the ATARI Program Exchange as a software development tool for professional developers. ATARI Pascal is a pseudo-code compiler which supports the International Standards Organization (ISO) draft standard (DPS/7185 as of 10/1/80), including variant records, sets, typed and text files, passing procedures and functions as parameters, GOTO out of a procedure, conformant arrays and program parameters. Additions to the standard available in ATARI Pascal include:

Additional predefined scalars: BYTE, WORD, STRING.
Operators on integers & (and), !, / (or) !, ? (NOT)
Else on CASE statement
Null Strings
Absolute Variables
External procedures
Additional built-in procedures and functions:
    graphic, sound, and controller definitions
    real and trancendental definitions
    move and fill procedures
    bit and byte manipulation
    file manipulation procedures
    heap management aids
    string manipulation
    address and sizeof functions
Modular compilation facilities

In addition, run-time error handling provides for divide by zero check, heap overflow check, string overflow check, range check and user-supplied error routines.

ATARI Pascal has been designed for data processing applications consisting of compilers, editors, linkers, business, and entertainment packages. It is designed to operate with the ATARI Disk Operating System 2.0S and is compatible with the ATARI Program Text-Editor [TM].

This chapter presents an overview of this manual, the system and compilation and run-time system requirements, and it describes the files on the distribution diskettes.

Because of the availability of many text books on the Pascal programming language, this document is not a tutorial but rather a reference manual and a detailed description of the extensions and additions that make ATARI Pascal unique. Refer to the bibliography for additional reference materials.

1.1 Manual Overview

The following provides a brief overview of each chapter contained in this manual.

Chapter 1: This chapter introduces and outlines the features of ATARI Pascal, provides an overview of the system and identifies the system requirements.

Chapter 2: This chapter gets you started. It describes the options of the compiler and linker and it presents step-by-step instructions to compile, link, and run a sample program.

Chapter 3: This chapter describes the extensions to ATARI Pascal. It presents such features as modular compilation, built-in procedures, graphics and sound extensions.

Chapter 4: This chapter briefly summarizes of the run-time error handling routines.

Chapter 5: This chapter describes the structure of a program generated by the compiler. Data storage is also discussed in this chapter.

Chapter 6: This chapter briefly compares ATARI Pascal and UCSD Pascal.

Chapter 7: This chapter defines the language features of ATARI Pascal.

Appendix A: A complete description of the language syntax

Appendix B: The reserved words list

Appendix C: A complete description of each compilation error message

Appendix D: ATARI Pascal File I/O

Appendix E: A bibliography of additional reading suggestions

Appendix F: Player/Missile Demo Program

Appendix G: Helpful Hints

1.2 System Overview

The ATARI Pascal Language System contains the Pascal monitor, compiler, linker, run-time subroutine library and interpreter. Figure 1-1 shows a diagram of the relationship among these products. Reference to the ATARI Program-Text Editor (APX-20075) has been included to show its relationship to ATARI Pascal.

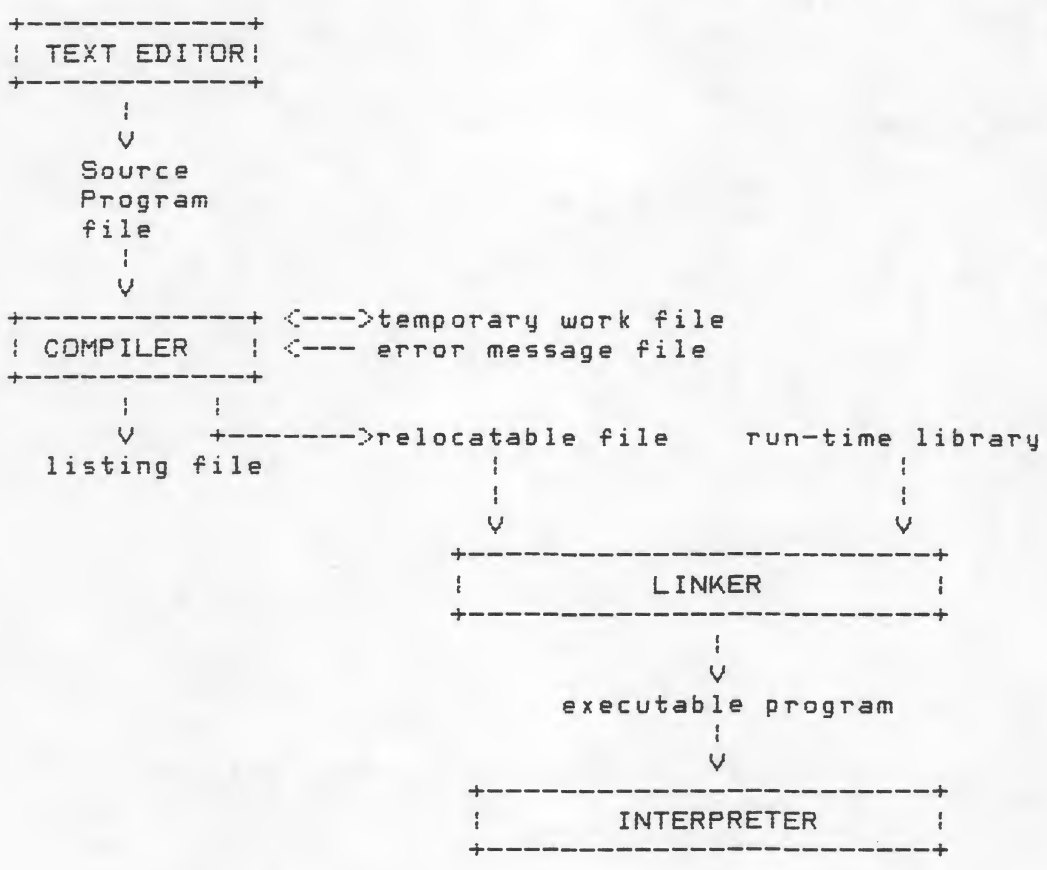


Figure 1-1 Schematic Diagram of ATARI Pascal Operation

The ATARI Program-Text Editor may be used to create and modify the Pascal source program. The compiler is used to translate the source program into relocatable machine code. The user then links this machine code with the run-time subroutine library to produce an executable object program.

1.3 System Requirements

The ATARI Pascal Language System requires the ATARI 800 with 48K of RAM and two ATARI 810 Disk Drives. The ATARI 825 80-Column Printer and the ATARI 850 Interface Module are optional. ATARI Pascal also

requires the ATARI Program-Text Editor. When using ATARI Pascal, no cartridge should be inserted in the cartridge slot.

1.4 Run-Time Requirements

The ATARI Pascal Language System generates programs that use a variety of run-time support subroutines that are extracted from PASLIB, the run-time library, and other relocatable modules. These run-time routines handle such needs as "multiply" and "divide" and file input and output interface to the Operating System.

1.5 ATARI Pascal Distribution Diskette Information

The ATARI Pascal Language System is distributed on diskettes compatible with the ATARI 810 Disk Drive. The system consists of two diskettes containing object, source and relocatable files. Listed below are the names of each file and a brief description of their contents.

Diskette 1 PASCAL/LINKER

File Contents

DOS.SYS	ATARI Disk Operating System
DUP.SYS	ATARI Disk Operating System
PASCAL	Interpreter used to execute all Pascal object programs.
MON	Pascal monitor loaded by the PASCAL file, providing the menu to specify the desired operation: compile, link, edit or run.
LINK	Pascal linker used to take relocatable files (.ERL) and run-time library files as input to create object files (.COM).
LINK.OVL	Pascal linker part two.
PASLIB.ERL	Run-time subroutine library in relocatable form. Should always be linked last.
FPLIB.ERL	Run-time support routines for floating point arithmetic and transcendental functions.
GRSND.ERL	Run-time support routines for graphic, sound and controller functions.
CALC.PAS	This is the source file for the Pascal demo program.

Diskette 2 Pascal Compiler

File Contents

PH0	Phase 0 of the Pascal compiler used for syntax scan and creation of token file.
PH1	Phase 1 of the Pascal compiler used to create the permanent symbol tables and build the user symbols.
PH2	Phase 2 of the Pascal compiler containing code generation initialization.
PH3	Phase 3 of the Pascal compiler used to create the relocatable object code file.
PH4	Phase 4 of the Pascal compiler used to complete the object code generation.
ERRORS.TXT	File containing ATASCII text for error messages.
GSPROCS	This file is the include file containing graphic, sound and controller definitions.
FLTPROCS	This file is the include file containing real number and transcendental function declarations.
MOVES	This file is the include file containing declarations for character arrays.
BITPROCS	This file is the include file containing declarations for bit manipulation routines.
HEAPSTUF	This file is the include file containing declarations for heap procedures.
DSKPROCS	This file is the include file containing file manipulation procedures.
STDPROCS	This file is the include file containing standard Pascal routines including the floating point routines.
ISOPROCS	This file is the include file containing ISO standard Pascal routines excluding floating point routines.
STRPROCS	This file is the include file containing string processing procedures and functions.

ISOPROCS + FLTPROCS = STDPROCS

CHAPTER 2: HOW TO OPERATE THE PASCAL LANGUAGE SYSTEM

This chapter describes how to use the ATARI Pascal Language System contained on the PASCAL/LINKER and Pascal Compiler diskettes. It covers the following information:

Section 1 provides step-by-step instructions on how to compile, link and run a sample program.

Section 2 describes the compiler and its options.

Section 3 describes the linker and its options.

Section 4 describes how to run an object program.

Section 5 describes the ATARI Program Text-Editor.

2.1 Compile, Link and Run a Sample Program

Before compiling and running the sample program described in this section, make a backup copy of all diskettes included in this package.

2.1.1 Compile Sample Program

Step One

Place the PASCAL/LINKER diskette into disk drive 1 and boot the Disk Operating System 2.0S. Then use option C to copy the sample calculation program "CALC.PAS" to a blank diskette on disk drive 2. At this time use the L option to load the file named "PASCAL" from disk drive 1. The Pascal menu will then appear.

```
      ATARI Pascal
Version 1.0 : 1-Mar-82
  (c) 1982 by ATARI

E)dit              C)ompile
L)ink              R)un
D)os               Q)uit

Enter letter and [RETURN]:
```

Step Two

Respond to the Pascal menu displayed on the screen with the command "C" [RETURN] to begin compilation.

When prompted for your source filename, type "D2:CALC.PAS" [RETURN].

The monitor will then prompt you for a token and code file name. Respond with [RETURN] for each.

A message will then be displayed "Change D1 to compiler disk." At this time place the Pascal Compiler (diskette 2) into disk drive 1 make sure the sample program "CALC.PAS" is in disk drive 2 and then press [RETURN].

The compiler will be loaded into memory and prompt you to choose a listing device. Respond "P:" (printer), "E:" (screen), or [RETURN] (no listing).

The compiler will proceed to display the following compilation statistics.

```
Loading Compiler
------------------------------------
      ATARI Pascal
Version 1.0 - 1-Mar-82
  (c) 1982 by ATARI
------------------------------------
```

```
Syntax Scan

Creating: D2:CALC.TOK
Listing file, P: or E:
<return> for none

File does not contain line numbers
<    0> ..............
Including Text from File: D1:STDPROCS
<   14> ...................
<   32> ...................................
<   64> ...................................
<   96> ...................................
<  128> ....
End of Phase 0                    (syntax / token file generation)
Source lines processed:   132
Loading Phase I
Open as input: D2:CALC.TOK
Open as output: D2:CALC.ERL
Available Memory:     4387        (total symbol table space)
User Table Space:     3264        (after predefined symbols)
Version 1.0, Phase 1
#####                             (one # for each routine body)
Remaining Memory  2100            (after user symbols)
Version 1.0, Phase 2
SUBREAL             18
ADDREAL             43
TF                  64            (decimal offset from beginning)
CALC               119
MENU               915
CALCULAT
External:          TRUNC
External:          SQRT
External:          SIN
External:          ROUND
External:          OUTPUT
External:          LN
External:          INPUT
External:          EXP
External:          COS
External:          ARCTAN

Lines :            130
Errors:              0
Code  :           1737
Data  :             64
REPLACE D1 THEN                   (place diskette 1 PASCAL/LINKER)
Type <return> to continue         (in disk drive 1, then press [RETURN]


Minutes later.........
```

The system will prompt you to "REPLACE D1 THEN Type [RETURN] to continue." At this time remove the Pascal Compiler from disk drive 1 and insert the PASCAL/LINKER in disk drive 1 then press [RETURN].

The compilation process will then be completed and the Pascal menu will display.

NOTE: If the compiler fails to complete compilation, check to see if the diskettes are in the proper drives. If they are try <SYSTEM RESET>. If both of these attempts fail, the only recourse is to turn off your computer and turn it on again.

2.1.2 Link Sample Program

Step One

To create the relocatable object file, respond to the Pascal menu with the command "L" [RETURN] to begin the linking process. At this time the following will be displayed.

```
Loading Linker
when LINKER prompts with "*" enter
your .ERL file names separated by
commas ending with PASLIB/S

Then type [RETURN]

LINKER V1.0
```

When prompted for your filename by an asterisk (*), you don't need to use an extension (.ERL) but you must use the device prefix "D2:".

The Pascal library routines must then be linked along with your program.

At this time respond to the filename prompt with the following:

```
D2:CALC,FPLIB,PASLIB/S [RETURN]
```

NOTE: This program may be used as an example of using the Floating Point Library (FPLIB) routines.

The linker will then display the following statistics and print "LINK COMPLETE TYPE [RETURN]".

```
D2:CALC.ERL          <48A7H>
D1:FPLIB.ERL         <2FFAH>
D1:PASLIB.ERL        <1F50H>

Undefined Symbols

-- No Undefined Symbols --
```

```
11405 bytes written to D2:CALC.COM

Total Data   :    00BEH bytes
Total Code   :    2BCEH bytes
Remaining    :    1442H bytes

Link complete type [RETURN]
```

At this time press [RETURN] and the PASCAL menu will display.

2.1.3 Run Sample Program

To run the sample program respond to the Pascal menu with the command "R" then [RETURN] to run the object program.

You will then be prompted for the filename and should respond with the following:

D2:CALC.COM

The calculation program will begin execution displaying the message "ENTER FIRST OPERAND?" Try this example for adding 5.5 to 99.256. First respond with "5.5" then [RETURN]. The message "R1 = 5.500E+00" should be displayed followed by "ENTER SECOND OPERAND?". Respond with "99.256" then [RETURN]. The message "R2 = 9.92560E+1" should be displayed followed by "ENTER OPERATOR:" followed by a list of operators. Respond with the operator "+" then [RETURN]. The result "104.756" should then be displayed. You should now press the [ESCAPE] key to return to the DOS menu.

You have now completed the compilation, linking and running of your first ATARI Pascal program!

2.2 Compiler Operation

2.2.1 Invocation and Filenames

The ATARI Pascal Language System is executed under the ATARI Disk Operating System (DOS 2.0S). To execute the compiler, place the PASCAL/LINKER (diskette 1) in disk drive 1 and LOAD the file called PASCAL from the DOS menu. This file is the Pascal interpreter and will automatically call the Pascal monitor with a filename of MON. The monitor then displays the following menu:

```
     ATARI Pascal
  Version 1.0 : 1-Mar-82
    (c) 1982 by ATARI

  E)dit              C)ompile
  L)ink              R)un
  D)os               Q)uit

  Enter letter and [RETURN]:
```

Select the first character of the desired function and enter this character followed by a [RETURN].

2.2.1.1 DOS and QUIT Options

The "DOS" and "QUIT" operation allows you to exit the Pascal menu and return to the ATARI Disk Operating System.

2.2.1.2 Compile

When you select "C" for "Compile," the monitor will request you to enter three file names and then load the compiler. The first request is for the source file name. You may then respond with the filename prefix (D2:) to identify the device, the input filename, and the extension .PAS. The Compile function then requests the name for the token and code files. If there is sufficient room on the diskette containing the source file you may respond by simply depressing [RETURN] in response to these requests. If there is not sufficient room you may specify that these files be placed on separate diskettes by specifying the FULL file name as desired. NOTE: None of the Compiler files may be cassette based.

A message will then be displayed "Change D1 to compiler disk." At this time place the Pascal Compiler (diskette 2) in disk drive 1, place the diskette containing your source program in disk drive 2 then press [RETURN]. ATARI Pascal then creates a relocatable file <name>.ERL which must be linked with the Pascal linker to the routines in the run-time library (PASLIB).

2.2.2 Compilation Data

The ATARI Pascal compiler will periodically display characters during the first two phases of the compilation (Phase 0 and Phase 1).

A period (.) will be displayed on the console for every source code line syntax scanned during Phase 0. At the beginning of Phase 1, the available memory space is displayed. This is the number of bytes (in decimal) of memory before generation of the symbol table. Approximately 1K of the symbol table space is consumed by pre-defined identifiers. When a procedure or function is found, a pound sign (#) will be displayed on the console. At the completion of Phase 1, the number of bytes remaining in memory is displayed in decimal.

Phase 2 generates object code. When the body of each procedure is encountered the name of the procedure is displayed so that you can see where the compiler is in the compilation of the program. The linker /M (Map) option will list the absolute addresses of the procedures in each module. Upon completion the following lines display:

```
Lines :   lines of source code compiled (in decimal).
Errors:   number of errors detected.
Code  :   bytes of code generated (in decimal).
Data  :   bytes of data reserved (in decimal).
```

2.2.3 Compiler Toggles

A compiler toggle may be included in the source program to signal the compiler that you wish to enable or disable certain options. The format of this toggle is (*$_ _ _ _ *) where the blanks are filled in with the toggle. The compiler does not accept blanks before the key letter or trailing or imbedded blanks in names but will skip over leading blanks; e.g., (*$E +*) is the same as (*$E+*), but the (*$ E +*) will be ignored.

Examples:

```
(*$E+*)
(*$P*)
(*$I D:USERFILE.LIB*)
```

2.2.3.1 Entry Point Record Generation (E)

$E+ and $E- control the generation of entry point records in the relocatable file. $E+ causes the global variables and all procedures and functions to be available as entry points (i.e., available to be referenced by EXTERNAL declarations in other modules). $E- supresses the generation of these records thus causing the variables, procedures, and functions to be logically private. The default state is $E+ and the toggle may be turned on and off at will.

2.2.3.2 Include Files (I)

$I<filename> causes the compiler to include the named file in the sequence of Pascal source statements. Filename specification includes drive name and extension in standard format.

The format is as follows:

```
(*$IDn:XXXXXXX*)
      or
(*$IDn:XXXXXXX.PAS*)
```

where n is the disk drive number
where XXXXXXX is the Include file name

Using these standard Include file procedures as examples, you may create Include files to be used during the compilation process.

2.2.3.3 Strict Type and Portability Checking (T,W)

$T+, $T-, $W+, and $W- control the strict type checking / non-portable warning facility. These features are tightly coupled (i.e. strict type checking implies warning non-portable usage and vice versa). The default state is $T- ($W-) in which type checking is relaxed and warning messages are not generated. This may be turned on and off throughout the source code as desired. A use of non-standard logic and/or built-in routines will cause error 500 to be generated. This error is not fatal but serves as a warning to the programmer. Code

generated with error 500 during the compilation will still execute properly.

2.2.3.4 Run-time Range Checking (R)

$R+ and $R- control the compiler's generation of run-time code which will perform range checking on array subscripting and storing into subrange variables. The default state is $R- (off) and this toggle may be turned on and off throughout the source code as desired.

2.2.3.5 Run-time Exception Checking (X)

$X+ and $X- control the compiler's generation of run-time code, which will perform run-time error checking and error handling for what is termed exceptions. Exceptions are:

Zero divide
String overflow/truncation
Heap overflow

The system philosophy under which ATARI Pascal operates states that zero divide and string overflow are treated in a "reasonable" manner when exception checking is disabled. Zero divide returns the maximum value for the data type and string overflow results in truncation of the string rather than modification of adjacent memory areas. The default state is $X- and may be changed throughout the source code as desired. See chapter 4 for more discussion of run-time error handling and options.

2.2.3.6 Listing Controls (L,P)

The $P and $L+, $L- toggles control the listing generated by the first pass of the compiler. $P will cause a formfeed character (CHR(12)) to be inserted into the .PRN file. $L+ and $L- are used to switch the listing on and off throughout the source program and may be placed wherever desired.

2.2.3.7 Summary of Compiler Toggles

Listed below is a summary of available compiler toggles:

Compiler Toggles		Default
$E +/-	Controls entry point generation	$E+
$I <name>	Includes another source file into the input stream (e.g. (*$I XXX.LIB*)	
$R +/-	Controls range checking code	$R-
$T +/-		$T-
$W +/-	Controls strict type checking and generation of warning messages	$W-
$X +/-	Controls exception checking code	$X-
$P	Enter a formfeed in the .PRN file	
$L +/-	Controls the listing of source code	$L+

2.2.4 Built-in Routines and Include Files

The ATARI Pascal compiler contains only the logic necessary for defining "magic" pre-defined procedures, functions and variables. These are such routines as READ, WRITE, ADDR, SIZEOF, etc. which require in-line code generation by the compiler or require support for a variable number of parameters.

All other routines are defined using a special keyword "PREDEFINED" and two special types ANYTYPE and ANYFILE. You must include in the source program declarations for these routines. This is normally done using the $I toggle to include STDPROCS and other similar files. STDPROCS contains declarations for procedures and functions defined by the ISO standard for Pascal. Additional files contain declarations for procedures and functions which are extensions to the ISO standard such as string routines, ASSIGN, IORESULT etc. You may edit STDPROCS and these files to contain only the routines necessary for a given program.

This method of defining built-in routines is present because the ATARI 800 Home Computer has limited memory for all the declarations and user symbols used in compiling large programs.

2.2.5 Error Messages

Compilation errors are numbered in the same sequence and meaning as those in Jensen and Wirth's "User Manual and Report". The error messages, brief explanations, and some causes of the error are found in Appendix C.

Error 407, Symbol Table Overflow: Occurs in Phase 1 when not enough symbol table space remains for the current symbol. This may be alleviated by breaking the program into modules.

2.2.6 Line Numbers

ATARI Pascal allows line numbers. When line numbers are desired, the first line of the program source file must contain a numeric value. It then assumes all lines contain line numbers and the line number must start in column one. Line numbers may be of any length and it should be noted that they are ignored by the compiler.

2.3 Linker Operation

2.3.1 Invocation and Commands

LINK is used by executing the linker from the Monitor. Enter 'L' from the Pascal menu followed by [RETURN] and the linker will load. The linker will then prompt the user for the name of the main program and modules to be linked, separated by commas. The output is directed to the same diskette as the main program unless you specify an output file name followed by an equal sign before the main program name.

Example:

```
CALC,FPLIB/S,PASLIB/S

D2:CALC=CALC,FPLIB,PASLIB/S    (CALC.COM is written to D2:)
```

The above command will link one of the demo programs with the run-time package. The items to be linked may be preceded by a disk drive device prefix:

```
D2:CALC,D1:FPLIB,D1:PASLIB/S
```

2.3.2 Linker Option Switches

The linker lets you to place a number of "switches" following the file names in the list. Each switch is preceded by a slash (/) and is a single letter. There is a parameter on the /P and /D switches.

2.3.2.1 Run-time Library Search (/S)

The examples above show the use of the /S switch which, commands the linker to search the previously named relocatable file, PASLIB, as a library and extract only the necessary modules. The /S switch extracts modules only from libraries and does not extract procedures and functions from separately compiled modules. It is position dependent in that it must follow the name of the run-time library in the linker command line as in the examples above. PASLIB is a specially constructed, searchable library. Other .ERL files supplied with the system, unless explicitly specified, are not searchable. User-created modules are not searchable. The order of modules within a library is important.

Each searchable library must contain routines in the correct order and be followed by /S for searching to occur. If /S is not specified the entire contents of the library is loaded.

2.3.2.2 Memory Map (/M)

A /M following the last file named in the parameter list generates a map to the screen.

2.3.2.3 Load Map (/L) and Extended Load Map (/E)

A /L following the last module named causes the linker to display module code and data locations as they are being linked. A /E following the last module works as a modifier to /M and /L and causes the linker to display all routines including those beginning with $, ?, or @, which are reserved for run-time library routine names.

2.3.2.4 Program (/P) and Data (/D) Origin

To support relocation of object code and data areas, the linker supports the /P and /D switches. The /P switch controls the location of the object area (ROM) and the /D switch controls the location of the data area (RAM). The syntax is: /P:nnnn or /D:nnnn where "nnnn" is a hexadecimal number in the range 0...FFFF.

In addition, if you specify /D, the linker will not save any of the data area in the .COM file. This is a good way for reducing the data storage on diskette for programs, since only the code will be loaded from diskette and not uninitialized data areas. Note that local file operations are not guaranteed if this is used because the system depends on the linker zeroing the data area to make this facility work properly.

Also, if /D is used, more space is gained in the linking process because the data is not intermixed with the code as it is being linked. Using this switch is the first way to solve and "out of memory" messages displayed by the linker.

Using the /P switch and /D switch does not cause the linker to leave empty space at the beginning of the .COM file. The philosophy of the linker is that if the /P switch is used, you really want to move the program to another system for execution. This means that if you specify /P:8000, the first byte of the .COM file will be placed at location 8000H and not 32K of zeros before the first byte. In addition, if you specify /D the linker will not save any of the data area in the .COM file. This is a good way for reducing the data storage on diskette for programs since only the code will be loaded from a diskette and not uninitialized data areas.

The switches /P and /D are specified after the last routine to be loaded and may be in any order.

2.3.2.5 Continuation Lines (/C)

If a line needs to be continued enter /C after the last character on the line before pressing the [RETURN] key.

2.3.2.6 Linker Input Command File (/F)

The linker lets you enter data into a file and have the linker process the file names from the file. You specify a file with an extension of .CMD and follow this file name with a /F (e.g., CFILES/F). The linker will read input from this file and process the names just as if they were typed from the computer keyboard. If the file contains more than one line, you must use /C after each line. If you wish to return to

the computer console for more input you may place /C on the last line in the file. Data on the command line following the /F is ignored. A .CMD file may not contain a line containing /F.

2.3.2.7 Linker Switch Summary

/S	Search preceeding name as a library extracting only the required routines.
/L	List modules as they are being linked.
/M	List all entry points in tabular form.
/E	List entry points beginning with $, ? or @ in addition to other entry points.
/P:nnnn	Relocate object code to nnnnH.
/D:nnnn	Relocate data area to nnnnH.
/F	Take preceeding file name as a .CMD file containing file names (see above for syntax).
/C	Continuation Lines

2.3.2.8 Relocatable File Requirements

The distribution diskettes contain several .ERL files that must be linked into the program. The particular files depend on what group of routines the compiler must reference, based on the contents of your program. Below is a list of each file and the routines it contains. If you have any of these routines as an undefined reference, then link the appropriate relocatable file to resolve the undefined reference.

FPLIB	Floating point real numbers @ XOP, @RRL, @WRL (searchable)
PASLIB	Comparisons, I/O, arithmetic support, etc.
GRSND	Graphics, sound, and controllers support

2.3.2.9 Linker Error Messages

The linker allows up to forty names on the command line (or command file input) for files to be linked.

Errors encountered in the linking process are usually self-explanatory, such as "unable to open input file: xxxxxxxx" and "Duplicate symbol- xxxxxxx." Duplicate symbol means that a run-time routine or variable and user routine or variable have the same name. Undefined reference indicates the appropriate relocatable file has not been included. Refer to the preceeding paragraph on Relocatable File Requirements.

If you run out of memory while linking, you may remove the data from the code space with the /D switch. You may need to run a test link with the /D switch set very high to find out what the code size is, then relink with the /D switch set just above the last code address (with some room for code expansion).

2.3.2.10 Attributes of Linkable Modules

The linker will bind together ATARI Pascal main programs, Atari Pascal modules, and assembly language modules created by an appropriate assembler.

2.4 Object Program Execution

Once the source program has been successfully compiled and linked with the appropriate run-time libraries you may execute or "Run" the program.

When you select "R" for Run from the Pascal menu, you will then be asked for the object filename to run.

Example:

```
D2:CALC.COM
```

The object program will then be loaded into memory and executed.

2.5 ATARI Program-Text Editor (MEDIT)

The ATARI Program-Text Editor is a versatile tool that can be used to create and modify source programs written in ATARI Pascal. This product may be ordered through the ATARI Program Exchange (APX-20075) or may be purchased with the ATARI Macro Assembler (CX8121)

2.5.1 Running the ATARI Program-Text Editor

The Pascal menu provides an option of calling the ATARI Program-Text Editor. The default value of this option is disk drive 2. Prior to using this option you must first make the following modifications.

1) Copy MEDIT from the distribution diskette to a blank diskette on disk drive 2.

2) Load D2:MEDIT from the DOS menu using the "/N" option to prevent it from running (this will require the temporary presence of MEM.SAV which can be deleted afterwards).

3) Save it back from DOS as follows: D2:MEDIT/A, 2600, 2601.

This append operation tells the "Pascal" program pointer to begin execution at the MEDIT entry point.

Note: The append operation may also be used to run any assembly language file from Pascal. The file must be appended with the start address and start address plus one. If the file consists of many disconnected modules scattered throughout the program, make sure the appended start address used is the run-time entry point.

CHAPTER 3: ATARI PASCAL LANGUAGE SYSTEM EXTENSIONS

This chapter describes the function and use of ATARI Pascal extensions.

It covers the following areas:

3.1 Modular Compilation

3.2 Data Allocation and Parameter Passing

3.3 Program Segmentation - Chaining

3.4 Built-in Procedures

3.5 Non-Standard Data Access

3.6 Imbedded Assembly Code

3.7 Graphics and Sound Extensions

3.1 Modular Compilation

ATARI Pascal supports a flexible modular compilation system. Programs may be developed in a monolithic fashion until they become too large to manage (or compile) and then split into modules at that time. The ATARI Pascal modular compilation system allows full access to procedures and variables in any module from any other module. A compiler toggle is provided to allow you to "hide" (i.e. make private) any group of variables or procedures. See section 2.2.3.1 for a discussion of the $E toggle.

The structure of a module is similar to that of a program. It begins with the reserved word MODULE, followed by an identifier and semi-colon (e.g., MODULE TEST1;) and ends with the reserved word MODEND, followed by a period (e.g., MODEND.). In between these two lines you may declare label, constant, type, variable, procedure and function sections just as in a program. Unlike a program, however, there is no BEGIN..END section after the procedure and function declarations, just the word MODEND followed by a period (.).

Example:

```
MODULE MOD1;

<label, const, type, var declarations>

<procedure / function declarations and bodies>

MODEND.
```

To access variables, procedures and functions in other modules (or in the main program) a new reserved word, EXTERNAL, has been added and is used for two purposes.

First, the word EXTERNAL may be placed after the colon and before the type in a GLOBAL variable declaration denoting that this variable list is not actually to be allocated in this module but rather in another module. No storage is allocated for variables declared in this way.

Example:

```
I,J,K, : EXTERNAL INTEGER; (* in another module *)

R:       EXTERNAL RECORD   (* again in another module *)
                    ...      (* some fields *)
                 END;
```

You MUST BE responsible for matching declaration identically, because the compiler and linker do not have the ability to type check.

Second, the EXTERNAL word is used to declare procedures and functions which exist in other modules. These declarations must appear before the first normal procedure or function declaration in the

module/program. Externals may only be declared at the global (outermost) level of a program or module.

Just as in variable declarations, the ATARI Pascal language requires you to make sure the number and type of parameters match exactly and the returned type matches exactly for functions, because the compiler and linker do not have the ability to type check across modules. External routines may NOT have procedures and functions as parameters.

Note that in ATARI Pascal external names are significant only to seven characters and not eight. When interfacing to assembly language, limit the length of identifiers accessible by assembly language to six characters.

Listed below are a main program skeleton and a module skeleton. The main program references variables and subprograms in the module, and the module references variables and subprograms in the main program. The only differences between a main program and a module are that at the beginning of a main program there are 16 bytes of header code and a main program body following the procedures and functions.

Main Program Example:

```
PROGRAM EXTERNAL_DEMO;

<label, constant, type declarations>

VAR

   I,J :  INTEGER;              (* AVAILABLE IN OTHER MODULES *)

   K,L :  EXTERNAL INTEGER;   (* LOCATED ELSEWHERE *)

EXTERNAL PROCEDURE SORT (VAR Q:LIST; LEN: INTEGER);

EXTERNAL FUNCTION  IOTEST: INTEGER;

PROCEDURE PROC1;
BEGIN
  IF IOTEST = 1 THEN
     (* CALL AN EXTERNAL FUNC NORMALLY *)
    ...
END;

BEGIN
  SORT(....);
  (* CALL AN EXTERNAL PROC NORMALLY *)
END.
```

Module Example: (Note these are separate files)

```
MODULE MODULE_DEMO;
```

```
<label, const, type declarations>
VAR
  I,J : EXTERNAL INTEGER;        (* USE THOSE FROM MAIN PROGRAM *)
  K,L : INTEGER;                 (* DEFINE THESE HERE *)
EXTERNAL PROCEDURE PROC1;        (* USE THE ONE FROM THE MAIN PROG *)
PROCEDURE SORT(...);             (* DEFINE SORT HERE *)
    ...
FUNCTION IOTEST:INTEGER;         (* DEFINE IOTEST HERE *)
<maybe other procedures and functions here>
MODEND.
```

3.2 Data Allocation and Parameter Passing

3.2.1 Data Allocation

In addition to accessing variables by name, you must know how variables are allocated in memory. Section 5.1 discusses the storage allocation and format of each built-in scalar data type. Variables allocated in the GLOBAL data area are allocated essentially shown here. However, variables in an identifier list before a type (e.g., A, B, C : INTEGER) are allocated in reverse order (i.e., C first, following by B, followed by A).

Example:

```
A         : INTEGER;
B         : CHAR;
I,J,K     : BYTE;
L         : INTEGER;
```

STORAGE LAYOUT:

```
+0 A LSB
+1 A MSB
+2 B
+3 K
+4 J
+5 I
+6 L LSB
+7 L MSB
```

Structured data types: ARRAYs, RECORDs and SETs require additional explanation. ARRAYs are stored in ROW major order. For example
A: ARRAY [1..3,1..3] OF CHAR is stored as:

```
+0 A[1,1]
+1 A[1,2]
+2 A[1,3]

+3 A[2,1]
+4 A[2,2]
+5 A[2,3]

+6 A[3,1]
+7 A[3,2]
+8 A[3,3]
```

This is logically a one-dimensional array of vectors. In ATARI Pascal all arrays are logically one-dimensional arrays of some other type.

RECORDs are stored in the same manner as global variables.

SETs are always stored as 32-byte items. Each element of the set is stored as one bit. SETs are byte-oriented and the low order bit of each byte is the first bit in that byte of the set. Shown below is the set 'A'..'Z':

Byte number

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 ... 1F
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --    --
00 00 00 00 00 00 00 00 FE FF FF 07 00 00 00 00 00 ... 00
```

The first bit is bit 65 ($41) and is found in byte 8, bit 1. The last bit is bit 90 and is found in byte 11, bit 2. In this dicussion bit 0 is the least significant bit in the byte.

3.2.2 Parameter Passing

When calling an assembly language routine from ATARI Pascal or calling an ATARI Pascal routine from assembly language, parameters are passed on the stack. The parameter passing stack in ATARI Pascal is different than the 6502 hardware stack. This software stack is at locations $600 through $6FF in memory. The hardware X register must be saved and restored during execution of assembly language routines and is used as the pointer to the software stack. You may load the top of the stack using "LDA $600,X", etc. Upon entry to the routine, the top of the hardware stack contains the return address. On the software stack, in reverse order the declaration, (A,B:INTEGER;C:CHAR), would result in C on top of B on top of A. Each parameter requires at least one 16-bit WORD of stack space. A character or boolean is passed as a 16-bit word with a high order byte of 00. VAR parameters are passed by address. The address represents the byte of the variable with the lowest memory address.

Non-scalar parameters (excluding SETs) are always passed by address. If the parameter is a value parameter then code is generated by the compiler in a Pascal routine to move the data. SET parameters are passed by value on the stack and then the interpreter is used to store them.

The example below shows a typical parameter list at entry to a procedure:

```
PROCEDURE DEMO (I,J : INTEGER; VAR Q:STRING; C,D:CHAR);

AT ENTRY STACK ($600,X):

                +0    D
                +1    BYTE OF 00
                +2    C
                +3    BYTE OF 00
                +4    ADDRESS OF ACTUAL STRING
                +5    ADDRESS OF ACTUAL STRING
                +6    J  (LSB)
                +7    J  (MSB)
                +8    I  (LSB)
                +9    I  (MSB)
```

The assembly language program must remove all parameters from the evaluation stack before returning to the calling routine.

SETs are stored on the stack with the least significant byte on bottom (high address).

Function values are returned on the stack. They are placed "logically" underneath the return address before the return is executed. They therefore remain on the top of the stack after the calling program is re-entered following the return. Assembly language functions may only return the scalar types INTEGER, REAL, BOOLEAN and CHAR.

3.3 Program Segmentation-- Chaining

There are times when programs exceed the memory available and also many times when segmentation of programs for compilation and maintenance purposes is desired. ATARI Pascal provides a "chaining" mechanism in which one program may transfer control to another program.

You must declare an untyped file (FILE;) and use the ASSIGN and RESET procedures to initialize the file. You may then execute a call to the CHAIN procedure, passing the name of the file variable as a single parameter. The run-time library routine will then perform the appropriate functions to load in the file you opened using the RESET statement. Program size does not matter. A small program may chain to a large one and a large program may chain to a small one. If you desire to communicate between the chained program you may choose to communicate in two ways: shared global variables and ABSOLUTE variables.

If you use the shared global variable method, you must guarantee that at least the first section of global variables is the same in the two programs wishing to communicate. The remainder of the global variables need not be the same and the declaration of external variables in the global section will not affect this mapping. In addition to having matching declarations, you must use the /D option switch available in the linker (see section 2.3.2.4) to place the variables at the same location in all programs wishing to communicate.

To use the ABSOLUTE variable method you would typically define a record used as a communication area and then define this record at an absolute location in each module. This method does not require using the /D switch in the linker but does require knowledge of the memory used by the program and system.

Listed below are two example programs that communicate with each other using the ABSOLUTE variable method. The first program will CHAIN to the second program, which will print the results of the first program's execution:

Example:

```
PROGRAM PROG1;

TYPE
 COMMAREA = RECORD
              I,J,K :  INTEGER
            END;

VAR
  GLOBALS :  ABSOLUTE [$8000] COMMAREA;
  CHAINFIL:  FILE;

BEGIN (* MAIN PROGRAM #1 *)
  WITH GLOBALS DO
    BEGIN
      I := 3;
      J := 3;
      K := I * J
    END;

  ASSIGN(CHAINFIL,'D1:PROG2.COM');
  RESET(CHAINFIL);
  IF IORESULT <> 0 THEN
    BEGIN
      WRITELN('UNABLE TO OPEN D1:PROG2.COM');
      EXIT
    END;

   CHAIN(CHAINFIL)
 END.                  (* END PROG1 *)


(* PROGRAM #2 IN CHAIN DEMONSTRATION *)

PROGRAM PROG2;

TYPE
  COMMAREA = RECORD
               I,J,K :  INTEGER
             END;

VAR
  GLOBALS :  ABSOLUTE [$8000] COMMAREA;


BEGIN (* PROGRAM #2 *)
  WITH GLOBALS DO
   WRITELN('RESULT OF ',I,' TIMES ',J,' IS =', K)
END.                    (* RETURNS TO OPERATING SYSTEM WHEN COMPLETE *)
```

3.4 Built-in Procedures and Parameters

This section describes ATARI Pascal's built-in procedures and functions. Each routine is described syntactically, followed by a description of the parameters and an example program using the procedure of the function. Section 3.4.2.5 is a quick reference of all built-in procedures and functions.

3.4.1 MOVE, MOVERIGHT, MOVELEFT

```
PROCEDURE MOVE      (SOURCE, DESTINATION, NUM_BYTES)
PROCEDURE MOVELEFT  (SOURCE, DESTINATION, NUM_BYTES)
PROCEDURE MOVERIGHT(SOURCE, DESTINATION, NUM_BYTES)
```

These procedures move the number of bytes contained in NUM_BYTES from the location named in SOURCE to the location named in DESTINATION. MOVE is a synonym for MOVELEFT. MOVELEFT moves from the left end of the source to the left end of the destination. MOVERIGHT moves from the right end of the source to the right end of the destination (the parameters passed to MOVERIGHT specify the left hand end of the source and destination).

Use MOVELEFT and MOVERIGHT to transfer a byte from one data structure to another or to move data around within a single data structure. The move is done on a byte level so the data structure type is ignored. MOVERIGHT is useful for transferring bytes from the low end of an array to the high end. Without this procedure, a FOR loop would be required to pick up each character and put it down at a higher address. MOVERIGHT is also much, much faster. MOVERIGHT is ideal to use in an insert character routine whose purpose is to make room for characters in a buffer.

MOVELEFT is useful for transferring bytes from one array to another, deleting characters from a buffer, or moving the values in one data structure to another.

The source and destination may be any type of variable and both need not be of the same type. These may also be pointers to variables or integers used as pointers. They may not be named or literal constants. The number of bytes is an integer expression greater than zero.

Watch out for these problems:

1. Since no checking is performed as to whether the number of bytes is greater than the size of the destination, spilling over into the data storage adjacent to the destination will occur if the destination is not large enough to hold the number of bytes.

2. Moving zero bytes moves nothing.

3. No type checking is done.

Example:

```
PROCEDURE MOVE_DEMO;
CONST
  STRINGSZ = 80;
VAR
 BUFFER : STRING[STRINGSZ];
 LINE : STRING;

PROCEDURE INSRT(VAR DEST : STRING; INDEX : INTEGER; VAR SOURCE :
STRING);
BEGIN
  IF LENGTH(SOURCE) <= STRINGSZ - LENGTH(DEST) THEN
    BEGIN
     MOVERIGHT(DEST[ INDEX ], DEST[ INDEX+LENGTH(SOURCE) ],
               LENGTH(DEST)-INDEX+1);
     MOVELEFT(SOURCE[1], DEST[INDEX], LENGTH(SOURCE));
     DEST[0] :=CHR(ORD(DEST[0]) + LENGTH(SOURCE))
    END;
END;

BEGIN
  WRITELN('MOVE_DEMO.....');
  BUFFER := 'Judy J. Smith/ 335 Drive/ Lovely, Ca. 95666';
  WRITELN(BUFFER);
  LINE := 'Roland ';
  INSRT(BUFFER, POS('5',BUFFER)+2,LINE);
  WRITELN(BUFFER);
END;
```

THE OUTPUT FROM THIS PROCEDURE:

```
MOVE_DEMO.....
Judy J. Smith/ 355 Drive/ Lovely, Ca. 95666
Judy J. Smith/ 355 Roland Dive/ Lovely, Ca. 95666
```

3.4.2 EXIT

```
PROCEDURE EXIT;
```

EXIT is the equivalent of the RETURN statement in FORTRAN or BASIC. It will leave the current procedure/function or main program. EXIT will also load the registers and re-enable interrupts before exiting if EXIT is used in an INTERRUPT procedure. It is usually executed as a statement following a test.

Example:

```
PROCEDURE EXITTEST;
 (*EXIT THE CURRENT FUNCTION OR MAIN PROGRAM.*)

   PROCEDURE EXITPROC(BOOL : BOOLEAN);

   BEGIN
     IF BOOL THEN
       BEGIN
        WRITELN('EXITING EXITPROC');
        EXIT;
       END;
     WRITELN('STILL IN EXITPROC, ABOUT TO LEAVE NORMALLY');
   END;

BEGIN
  WRITELN('EXITTEST.......');
  EXITPROC(TRUE);
  WRITELN('IN EXITTEST AFTER 1ST CALL TO EXITPROC');
  EXITPROC(FALSE);
  WRITELN('IN EXITTEST AFTER 2ND CALL TO EXITPROC');
  EXIT;
  WRITELN('THIS LINE WILL NEVER BE PRINTED');
END;
```

Output:

```
EXITTEST.......
EXITING EXITPROC
IN EXITTEST AFTER 1ST CALL TO EXITPROC
STILL IN EXITPROC, ABOUT TO LEAVE NORMALLY
IN EXITTEST AFTER 2ND CALL TO EXITPROC
```

3.4.3 TSTBIT, SETBIT, CLRBIT

```
FUNCTION  TSTBIT(    BASIC_VAR, BIT_NUM) : BOOLEAN;
PROCEDURE SETBIT(VAR BASIC_VAR, BIT_NUM);
PROCEDURE CLRBIT(VAR BASIC_VAR, BIT_NUM);
```

TSTBIT returns TRUE if the designated bit in the basic_var is on, and returns FALSE if the bit is off. SETBIT sets the designated bit in the parameter. CLRBIT clears the designated bit in the parameter.

BASIC_VAR is any 8 or 16 bit variable such as integer, char, byte, word, or boolean. BIT_NUM is 0..15 with bit 0 on the right. Attempting to set bit 10 of an 8 bit variable does not cause an error but has no effect on the end result.

These procedures are useful for generating wait loops or altering incoming data by flipping a bit where needed. Another application is in manipulating a bit mapped screen.

Example:

```
PROCEDURE TST_SET_CLR_BITS;

VAR
  I : INTEGER;
BEGIN
  WRITELN('TST_SET_CLR_BITS.......');
  I := 0;
  SETBIT(I,5);
  IF I = 32 THEN
    IF TSTBIT(I,5) THEN
      WRITELN('I=',I);
  CLRBIT(I,5);
  IF I = 0 THEN
    IF NOT (TSTBIT(I,5)) THEN
      WRITELN('I=',I);
END;
```

Output:

```
TST_SET_CLR_BITS.......
I=32
I=0
```

3.4.4 SHR, SHL

```
FUNCTION SHR(BASIC_VAR, NUM) : INTEGER;
FUNCTION SHL(BASIC_VAR, NUM) : INTEGER;
```

SHR shifts the BASIC_VAR by NUM bits to the right, inserting 0 bits.
SHL shifts the BASIC_VAR by NUM bits to the left, inserting 0 bits.
BASIC_VAR is an 8 or 16 bit variable. NUM is an integer expression.

The uses of SHR and SHL are generally obvious. For example, suppose a 10 bit value is to be obtained from two separate input ports. You can use SHL to read them in:

```
VAR
  PORT1 : ABSOLUTE [$D000] BYTE;
  PORT2 : ABSOLUTE [$D232] BYTE;


X := SHL(PORT1 & $1F, 3) ! (PORT2 & $1F);
```

The above example reads from port1, masks out the three high bits returned from the INP array, and shifts the result left. Next, this result is logically OR'd with the input from port2, which has also been masked.

The following procedure demonstrates the expected result of executing these two functions.

Example:

```
PROCEDURE SHIFT_DEMO;
VAR I : INTEGER;
BEGIN
  WRITELN('SHIFT_DEMO ........');
  I := 4;
  WRITENLN('I=', I);
  WRITELN('SHR(I,2)=', SHR(I,2));
  WRITELN('SHL(I,4)=', SHL(I,4));
END;
```

Output:

```
SHIFT_DEMO........
I=4
SHR(I,2)=1
SHL(I,4)=64
```

3.4.5 HI, LO, SWAP

```
FUNCTION   HI(BASIC_VAR) : INTEGER;
FUNCTION   LO(BASIC_VAR) : INTEGER;
FUNCTION SWAP(BASIC_VAR) : INTEGER;
```

HI returns the upper 8 bits of BASIC_VAR (an 8 or 16 bit variable) in the lower 8 bits of the result. LO returns the lower 8 bits with the upper 8 bits forced to zero. SWAP returns the upper 8 bits of BASIC_VAR in the lower 8 bits of the result and the lower 8 bits of BASIC_VAR in the upper 8 bits of the result. Passing an 8 bit variable to HI causes the result to be 0 and passing 8 bits to LO does nothing.

These functions enhance ATARI Pascal's abilities to read and write to I/O ports. If a data item has 16 bits of information to send to a port that can handle 8 bits at a time, use LO and HI to send the low byte followed by the high byte. Similarly, reading 16 bits of data from a port that sends 8 bits at a time may be performed by SWAPping the first 8 bits into the high byte:

```
     VAR
       PORT6 : ABSOLUTE [$D234] BYTE;

   PORT6 := LO(B);
   PORT6 := HI(B);
   B := SWAP(PORT6) ! PORT6;
```

The following example shows what the expected results of these functions should be:

Example:

```
PROCEDURE HI_LO_SWAP;
VAR
  HL : INTEGER;
BEGIN
  WRITELN('HI_LO_SWAP.......');
  HL := $104;
  WRITELN('HL=',HL);
  IF HI(HL) = 1 THEN
    WRITELN('HI(HL)=',HI(HL));
  IF LO(HL) = 4 THEN
    WRITELN('LO(HL)=',LO(HL));
  IF SWAP(HL) = $0401 THEN
    WRITELN('SWAP(HL)=',SWAP(HL));
END;
```

Output:

```
HI_LO_SWAP.......
HL=260
HI(HL)=1
LO(HL)=4
SWAP(HL)=1025
```

3.4.6 ADDR

```
FUNCTION ADDR(VARIABLE REFERENCE) : INTEGER;
```

ADDR returns the address of the variable referenced. Variable reference includes procedure/function names, subscripted variables and record fields. It does not include named constants, user defined types, or any item that does not occupy code or data space.

This function is used to return the address of anything: compile time tables generated by INLINE, the address of a data structure to be used in a move statement, and so on.

Example:

```
PROCEDURE ADDR_DEMO(PARAM : INTEGER);
VAR
  REC : RECORD
          J : INTEGER;
          BOOL : BOOLEAN;
        END;
  ADDRESS : INTEGER;
  R : REAL;
  S1 : ARRAY[1..10] OF CHAR;

BEGIN
  WRITELN('ADDR_DEMO.....');
  WRITELN('ADDR(ADDR_DEMO)=',ADDR(ADDR_DEMO));
  WRITELN('ADDR(PARAM)=',ADDR(PARAM));
  WRITELN('ADDR(REC)=',ADDR(REC));
  WRITELN('ADDR(REC.J) ', ADDR(REC.J));
  WRITELN('ADDR(ADDRESS)=',ADDR(ADDRESS));
  WRITELN('ADDR(R)=',ADDR(R));
  WRITELN('ADDR(S1)=',ADR(S1));
END;
```

Output is system dependent.

3.4.7 SIZEOF

FUNCTION SIZEOF(VARIABLE OR TYPE NAME) : INTEGER;

SIZEOF returns the size of the parameter in bytes. It is used in move statements for the number of bytes to be moved. With SIZEOF you need not keep changing constants as the program evolves. Parameter may be any variable: character, array, record, etc, or any user-defined type.

Example:

```
PROCEDURE SIZE_DEMO;
VAR
  B : ARRAY[1..10] OF CHAR;
  A : ARRAY[1..15] OF CHAR;
BEGIN
  WRITELN('SIZE_DEMO.......');
  A := '***************';
  B := '0123456789';
  WRITELN('SIZEOF(A)=',SIZEOF(A),' SIZEOF(B)=',SIZEOF(B));
  MOVE(B,A,SIZEOF(B));
  WRITELN('A= ',A);
END;
```

Output:

```
SIZEOF(A)=15 SIZEOF(B)=10
A= 0123456789*****
```

3.4.8 FILLCHAR

PROCEDURE FILLCHAR(DESTINATION, LENGTH,CHARACTER)

This procedure fills the DESTINATION (a packed array of characters) with the number of CHARACTERs specified by LENGTH. DESTINATION is packed array of characters. It may be subscripted. LENGTH is an integer expression. If LENGTH is greater than the length of DESTINATION, adjacent code or data is overwritten. Also, if it is negative, adjacent memory can be overwritten. CHARACTER is a literal or variable of type char.

The purpose of FILLCHAR is to provide a fast method of filling in large data structures with the same data. For instance, blanking out buffers is done with FILLCHAR.

Example:

```
PROCEDURE FILL_DEMO;
VAR
    BUFFER : PACKED ARRAY[1..256] OF CHAR;
BEGIN
  FILLCHAR(BUFFER,256,' ');              {* BLANK THE BUFFERS *}
END;
```

3.4.9 LENGTH

```
FUNCTION LENGTH( STRING) : INTEGER;
```

This function returns the integer value of the length of the string.

Example:

```
PROCEDURE LENGTH_DEMO;
VAR
  S1 : STRING [40];
BEGIN
  S1 := 'This string is 33 characters long';
  WRITELN('LENGTH OF ',S1,'=',LENGTH(S1));
  WRITELN('LENGTH OF EMPTY STRING =',LENGTH(''));
END;
```

Output:

```
LENGTH OF This string is 33 characters long=33
LENGTH OF EMPTY STRING = 0
```

3.4.10 CONCAT

```
FUNCTION CONCAT ( SOURCE1, SOURCE2, .... , SOURCE) : STRING;
```

This function returns a string in which all sources in the parameter list are concatenated. The sources may be string variables, string literals, or characters. A SOURCE of zero length can be concatenated with no problem. If the total length of all SOURCES exceeds 56 bytes the string is truncated at 256 bytes. See the note under COPY in the next section concerning restrictions when using both CONCAT and COPY.

Example:

```
PROCEDURE CONCAT_DEMO;
VAR
  S1,S2 : STRING;
BEGIN
  S1 := 'left link, right link';
  S2 := 'root root root';
  WRITELN(S1,'/',S2);
  S1 := CONCAT(S1,' ',S2,'!!!!!!');
  WRITELN(S1);
END;
```

Output:

```
left link, right link/root root root
left link, right link root root root !!!!!!
```

3.4.11 COPY

```
FUNCTION COPY ( SOURCE, LOCATION, NUM_BYTE) : STRING;
```

Copy returns a string containing the number of characters specified in NUM_BYTES from SOURCE beginning at the index specified in LOCATION. SOURCE must be a string. LOCATION and NUM_BYTES are integer expressions. If LOCATION is out of bounds or is negative, no error occurs. If NUM_BYTES is negative or NUM_BYTES plus LOCATION exceeds the length of the SOURCE, truncation occurs.

Example:

```
PROCEDURE COPY_DEMO;
BEGIN
  LONG_STR := 'Hi from Cardiff-by-the sea';
  WRITELN (COPY(LONG_STR,9,LENGTH(LONG_STR)-9+1));
END;
```

Output;

```
Cardiff-by-the-sea
```

Note:

COPY and CONCAT are "pseudo" string returning functions and have only one statically allocated buffer for the return value. Therefore, if these functions are used more than once within the same expression, the value of each occurrence of these functions becomes the value of the last occurrence. For instance, "IF (CONCAT(A,STRING1) = (CONCAT(A,STRING2))" will always be true because the concatenation of A and STRING1 is replaced by that of A and STRING2. Also, "WRITELN (COPY(STRING1,1,4), COPY(STRING1,5,4))" writes the second set of four characters in STRING1 twice.

3.4.12 POS

```
FUNCTION POS( PATTERN, SOURCE ) : INTEGER;
```

This function returns the integer value of the position of the first occurrence of PATTERN in SOURCE. If the pattern is not found, a zero is returned. SOURCE is a string and PATTERN is a string, a character, or a literal.

Example:

```
PROCEDURE POS_DEMO;
VAR
  STR,PATTERN : STRING;
  CH : CHAR;
BEGIN
  STR := 'ABCDEFGHIJKLMNO';
  PATTERN := 'FGHIJ';
  CH := 'B';
  WRITELN('pos of ',PATTERN,' in ',STR,' is ', POS(PATTERN,STR));
  WRITELN('pos of ',CH,' in ',STR,'is ' ,POS(CH,STR));
  WRITELN('pos pf ''z'' in ',STR,' is ',POS('z',STR));
END;
```

Output:

```
pos of FGHIJ in ABCDEFGHIJKLMNO is 6
pos of B in ABCDEFGHIJKLMNO is 2
pos of 'z' in ABCDEFGHIJKLMNO is 0
```

3.4.13 DELETE

```
PROCEDURE DELETE (TARGET, INDEX, SIZE);
```

This procedure is used to remove SIZE characters from TARGET, beginning at the byte named in INDEX. TARGET is a string. INDEX and SIZE are integer expressions. If SIZE is zero, no action is taken. If it is negative, serious errors result. If the INDEX plus the SIZE is greater than the TARGET or if the TARGET is empty, the data and surrounding memory can be destroyed.

Example:

```
PROCEDURE DELETE_DEMO;
VAR
  LONG_STR : STRING;
BEGIN
  LONG_STR :='      get rid of the leading blanks';
  WRITELN(LONG_STR);
  DELETE(LONG_STR,1,POS('g',LONG_STR)-1);
  WRITELN(LONG_STR);
END;
```

Output:

```
      get rid of the leading blanks
get rid of the leading blanks
```

3.4.14 INSERT

```
PROCEDURE INSERT( SOURCE, DESTINATION, INDEX);
```

This procedure is used to insert the SOURCE into the DESTINATION at the location specified in INDEX. DESTINATION is a string. SOURCE is a character or string, literal or variable. INDEX is an integer expression. SOURCE can be empty. If INDEX is out of bounds or DESTINATION is empty, destruction of data occurs. If inserting SOURCE into DESTINATION causes DESTINATION to be longer than allowed DESTINATION is truncated.

Example:

```
PROCEDURE INSERT_DEMO;
VAR
  LONG_STR : STRING;
  S1 : STRING [10];
BEGIN
  LONG STR := 'Remember  May 9';
  S1 := 'Mother's Day, ';
  INSERT(S1,LONG_STR,10);
  WRITELN(LONG_STR);
  INSERT('to celebrate',LONG_STR,10);
  WRITELN(LONG_STR);
END;
```

Output:

```
Remember Mother's Day, May 9
Remember to celebrate Mother's Day, May 9
```

3.4.15 ASSIGN

```
PROCEDURE ASSIGN (FILE, NAME);
```

Use this procedure to assign an external filename to a file variable prior to a RESET or REWRITE. FILE is a filename, NAME is a literal or a variable string containing the name of the file to be created. FILE must be of type TEXT to use the special device names below.

Note that standard Pascal defines a "local" file. ATARI Pascal implements this facility using temporary filenames in the form PASTMPxx where "xx" is sequentially assigned, starting at zero at the beginning of each program. If an external file REWRITE is not preceeded by an ASSIGN, then a temporary filename will also be assigned to this file before creation.

NAME is normally a diskette filename in the standard format: dn:filename.ext but can also be a special device name.

Device Names

E:	Console screen editor device
S:	Console screen output device
K:	Console keyboard input device
P:	Printer output device

NOTE: Cassette (C:) files are not supported by ATARI Pascal.

Examples of ASSIGN usage:

```
ASSIGN(PRINTFILE, 'P:');
ASSIGN(F, 'D2:MT280.OVL');
ASSIGN(KEYBOARD, 'K:');
ASSIGN(CRT, 'S:');
```

Note: After ASSIGN(CRT, 'S:') you must use REWRITE, as the assign does not open the file.

3.4.16 WNB, GNB

```
FUNCTION GNB(FILEVAR: FILE OF PAOC):CHAR;
FUNCTION WNB(FILEVAR: FILE OF CHAR; CH:CHAR) : BOOLEAN;
```

These functions allow you to have BYTE-level access to a file in a high speed manner. PAOC is any type that is fundamentally a Packed Array Of Char. The size of the packed array is optimally in the range 128..4095.

GNB will let you read a file a byte at a time. It returns a value of type CHAR. The EOF function will be valid when the physical end-of-file is reached but not based upon any data in the file.

WNB will let you write a file a byte at a time. It requires a file and a character to write. It returns a boolean value that is true if there was an error while writing that byte to the file. No interpretation is done on the bytes that are written.

GNB and WNB are used (as opposed to F^, GET/PUT combinations) because they are significantly faster.

3.4.17 BLOCKREAD, BLOCKWRITE

```
BLOCKREAD (F:FILEVAR; BUF:ANY; VAR IOR:INTEGER; SZ,RB:INTEGER);
BLOCKWRITE(F:FILEVAR; BUF:ANY; VAR IOR:INTEGER; SZ,RB:INTEGER);
```

These procedures are used for direct diskette access. FILEVAR is an untyped file (FILE;). BUF is any variable large enough to hold the data. IOR is an integer that receives the returned value from the DOS. SZ is the number of bytes to transfer and RB should always be 0.

The data is transferred either to or from the user's BUF variable for the specified number of bytes.

3.4.18 OPEN

```
PROCEDURE OPEN (FILE, TITLE, RESULT);
```

The OPEN procedure increases the flexibility of ATARI Pascal. FILE is any file type variable. TITLE is a string containing the filename. RESULT is a VAR INTEGER parameter and upon return from OPEN has the same value as IORESULT. The maximum number of files that may be opened at any one time is three not including Console (E:, S:, or K:) files.

The OPEN procedure is the same as executing an ASSIGN(FILE, TITLE), RESET(FILE) and RESULT := IORESULT sequence.

Examples:

```
OPEN (INFILE, 'D:FNAME.DAT', RESULT);
```

3.4.19 CLOSE, CLOSEDEL

```
PROCEDURE CLOSE    ( FILE, RESULT );
PROCEDURE CLOSEDEL ( FILE, RESULT );
```

The CLOSE and CLOSEDEL procedures are used for closing and closing-with-delete respectively. The CLOSE procedure must be called to guarantee that data written to a file using any method is properly purged from the file buffer to the diskette. The CLOSEDEL is normally used on temporary files to delete them after use. FILE and RESULT are the same as used in OPEN (see section 3.4.18).

Files are implicitly closed when an open file is RESET.

The CLOSE procedure is used in the file section of the appendix.

3.4.20 PURGE

```
PROCEDURE PURGE (FILE );
```

The PURGE procedure is used to delete a file whose name is stored in a string. You must first ASSIGN the name to the file and then execute PURGE.

Example:

```
ASSIGN(F,'D2:BADFILE.BAD');
PURGE(F);                              (* DELETE D2:BADFILE.BAD *)
```

3.4.21 IORESULT

FUNCTION IORESULT : INTEGER

After each I/O operation the value returned by the IORESULT function is set by the run-time library routines. On the ATARI Home Computer, the general rule is that a non-zero value means an error and zero is a good result.

Example:

```
ASSIGN(F,'D2:HELLO');
RESET(F);

IF IORESULT <> 0 THEN
  WRITELN('C:HELLO IS NOT PRESENT');
```

3.4.22 MEMAVAIL, MAXAVAIL

```
FUNCTION   MEMAVAIL : INTEGER;
FUNCTION   MAXAVAIL : INTEGER;
```

The functions MEMAVAIL and MAXAVAIL are used in conjunction with NEW and DISPOSE to manage the HEAP memory area in ATARI Pascal. The MEMAVAIL function returns the largest total available memory at any given time irrespective of fragmentation. The MAXAVAIL function will first garbage collect and then report the largest block available. The MAXAVAIL function can be used to force a garbage collection before a time-sensitive section of programming.

The ATARI Pascal system fully supports the NEW and DISPOSE mechanism defined by the Pascal Standard. The HEAP area grows from the end of the data area and the stack frame (for recursion) grows from the top of memory downward.

3.4.23 Quick Reference Guide to Built-in Procedures and Parameters

(Alphabetical within each group:)

Character array manipulation routines

```
PROCEDURE FILLCHAR ( DESTINATION, LENGTH, CHARACTER);
PROCEDURE MOVELEFT ( SOURCE, DESTINATION, NUM_BYTES);
PROCEDURE MOVERIGHT( SOURCE, DESTINATION, NUM_BYTES);
```

Bit and byte manipulation routines

```
PROCEDURE CLRBIT( BASIC_VAR, BIT_NUM);
FUNCTION  HI    ( BASIC_VAR )           : INTEGER;
FUNCTION  LO    ( BASIC_VAR )           : INTEGER;
PROCEDURE SETBIT( BASIC_VAR, BIT_NUM);
FUNCTION  SHL   ( BASIC_VAR, NUM)       : INTEGER;
FUNCTION  SHR   ( BASIC_VAR, NUM)       : INTEGER;
FUNCTION  SWAP  ( BASIC_VAR )           : INTEGER;
FUNCTION  TSTBIT( BASIC_VAR, BIT_NUM) : BOOLEAN;
```

String handling routines

```
FUNCTION  CONCAT  ( SOURCE1, SOURCE2,...,SOURCEn ) : STRING;
FUNCTION  COPY    ( SOURCE, LOCATION, NUM_BYTES)   : STRING;
PROCEDURE DELETE  ( TARGET, INDEX, SIZE );
PROCEDURE INSERT  ( SOURCE, DESTINATION, INDEX);
FUNCTION  LENGTH  ( STRING )                       : INTEGER;
FUNCTION  POS     ( PATTERN, SOURCE)               : INTEGER;
```

File handling routines

```
PROCEDURE ASSIGN    ( FILE, NAME );
PROCEDURE BLOCKREAD ( FILE, BUF, IOR, NUMBYTES, RELBLK);
PROCEDURE BLOCKWRITE( FILE, BUF, IOR, NUMBYTES, RELBLN);
PROCEDURE CLOSE     ( FILE, RESULT );
PROCEDURE CLOSEDEL  ( FILE, RESULT );
FUNCTION  GNB       ( FILE )        : CHAR
PROCEDURE IORESULT                  : INTEGER;
PROCEDURE OPEN      ( FILE, TITLE, RESULT );
PROCEDURE PURGE     ( FILE );
FUNCTION  WNB       ( FILE, CHAR ) : BOOLEAN;
```

Miscellaneous routines

```
FUNCTION  ADDR ( VARIABLE REFERENCE )    : INTEGER;
PROCEDURE EXIT;
FUNCTION  MAXAVAIL : INTEGER;
FUNCTION  MEMAVAIL : INTEGER;
FUNCTION  SIZEOF( VARIABLE OR TYPE NAME) : INTEGER;
```

3.5 Non-Standard Data Access

3.5.1 Absolute Variables

<absolute var> ::= ABSOLUTE [<constant>] <var>

ABSOLUTE variables may be declared if you know the address at compile time. You declare variable(s) to be absolute using special syntax in a VAR declaration. ABSOLUTE variables are not allocated any space in your data segment by the compiler and you are responsible for making sure that no compiler-allocated variables conflict with the absolute variables. NOTE: STRING VARIABLES MAY NOT EXIST below [$100] in memory.

Examples:

```
I:      ABSOLUTE [$8000] INTEGER;
SCREEN: ABSOLUTE [$C000] ARRAY[0..15] OF ARRAY[0..63] OF CHAR;
```

3.6 INLINE

ATARI Pascal has a very useful built-in feature called INLINE. This feature lets you insert data in the middle of an ATARI Pascal procedure or function. In this way small machine code or P-code sequences and constant tables may be inserted into an ATARI Pascal program.

3.6.1 Syntax

The syntax for the INLINE feature is very similar to that of a procedure call in Pascal. The word INLINE is used followed by a left parenthesis "(" followed by any number of arguments separated by the slash "/" character and terminated by a right parenthesis ")". The arguments between the slashes must be constants or variable references that evaluate to constants. These constants can be of any of the following types : CHAR, STRING, BOOLEAN, INTEGER or REAL. Note that a STRING in quotes does not generate a length byte but simply the data for the string.

Literal constants of type integer will be allocated one byte if the value falls in the range 0 to 255. Named, declared, integer constants which will always be allocated two bytes.

3.6.2 Applications

The INLINE facility can be used to insert code or to build compile time tables. The following two sections give examples of each of these uses.

The program fragment below demonstrates how the INLINE facility can be used to construct a compile time table.

Example:

```
PROGRAM DEMO_INLINE;

TYPE
    IDFIELD = ARRAY [1..4] OF ARRAY [1..10] OF CHAR;

VAR
  TPTR : ^IDFIELD;

PROCEDURE TABLE;
BEGIN
  INLINE(     'ATARI      ' /
              'HOME       ' /
              'COMPUTER   ' /
              'SYSTEMS...' );
END;

BEGIN (* MAIN PROGRAM *)
   TPTR := ADDR(TABLE)+5;          (* +5 for P-code only *)

   WRITELN(TPTR^[3]);            (* SHOULD WRITE 'COMPUTER   ' :)

END.
```

3.7 Graphics and Sound Documentation

The graphics, sound, and controller package consists of an include file, GSPROCS, and a Pascal module, GRSND.ERL. The include file defines the entry points available in the Pascal module. The Pascal module must be linked with your program.

To use the package, type (*$ID:GSPROCS*) following the global variables of your program, and execute INITGRAPHICS as the first statement in your main program.

Example:

```
PROGRAM GRSND;

LABEL
    .....;

CONST
    .....;

TYPE
    .....;

VAR
    .....;

(* INCLUDE THE GRAPHICS AND SOUND DEFINITIONS *)
(*$ID:GSPROCS*)

(* LOCAL PROCEDURES *)

PROCEDURE XXXX;
  BEGIN
    .....;
  END;

PROCEDURE YYYY;
  BEGIN
    .....;
  END;

(* MAIN PROGRAM *)
BEGIN
  INITGRAPHICS(5);  (* INITIALIZE GRAPHICS PACKAGE WITH A MAXIMUM
                       GRAPHICS MODE OF 5 *)

  ......;
END.
```

The following sections describe each of the items available in the graphics and sound package.

3.7.1 Screen Types

TYPEs:

```
SCRN_TYPE = (SPLIT_SCREEN, FULL_SCREEN);
CLEAR_TYPE = (CLEAR_SCREEN, DO_NOT_CLEAR_SCREEN);
```

These screen types are used by the GRAPHICS procedure to define the type of screen and whether or not the screen will be cleared during the GRAPHICS procedure.

3.7.2 Variables

VARs:

```
SCRNFILE : EXTERNAL TEXT;
GRRESULT : EXTERNAL INTEGER;
```

SCRNFILE may be used to do standard Pascal I/O to the screen such as:

```
WRITE(SCRNFILE,'A');
```

This variable will send an "A" to the screen and depending on the current mode, the "A" will be displayed in some manner. Note this technique is normally used only in graphics modes 1 and 2. For the other graphics modes, use the procedures described below.

GRRESULT is used to determine if any errors occurred during one of the graphics procedures. The following are the procedures and functions that alter GRRESULT.

INITGRAPHICS	GRRESULT = 0 OK, 255 = ERROR
GRAPHICS	GRRESULT = 0 OK, 255 = ERROR
PLOT	GRRESULT = RESULT FROM XIO CALL
LOCATE	GRRESULT = RESULT FROM XIO CALL
FILL	GRRESULT = RESULT FROM XIO CALL
DRAWTO	GRRESULT = RESULT FROM XIO CALL

3.7.3 Graphic Procedures and Functions

3.7.3.1 Initialize Procedure

```
PROCEDURE INITGRAPHICS(MAX_MODE: INTEGER);
```

INITGRAPHICS must be the first statement of a program that uses the graphics and sound module. There is one parameter:

MAX_MODE	Maximum mode used by this program should be a value from 0 to 9.

If an error occurs, the GRRESULT = 255; otherwise; GRRESULT = 0.

3.7.3.2 Graphics Procedure

```
PROCEDURE GRAPHICS(MODE: INTEGER; SCREEN: SCRN_TYPE; CLEAR: CLEAR_TYPE);
```

GRAPHICS performs the same function as the GRAPHICS statement in ATARI BASIC, except it has three parameters instead of one.

MODE	The desired graphics mode 0 to MAX_MODE
SCREEN	FULL_SCREEN or SPLIT_SCREEN
CLEAR	CLEAR_SCREEN or DO_NOT_CLEAR_SCREEN

If an error occurs, then GRRESULT = 255; otherwise, GRRESULT = 0.

3.7.3.3 Textmode Procedure

```
PROCEDURE TEXTMODE;
```

TEXTMODE closes "S:" and reopens "E:". GRRESULT is unchanged.

3.7.3.4 Setcolor Procedure

```
PROCEDURE SETCOLOR(REGISTER, HUE, LUMINANCE: INTEGER);
```

SETCOLOR performs the same function as the SETCOLOR statement in ATARI BASIC. GRRESULT is unchanged.

REGISTER	A value from 0 to 4. Refer to section 9 of the ATARI 400/800 BASIC Reference Manual under SETCOLOR.
HUE	A value from 0 to 15. Refer to section 9 of the ATARI 400/800 BASIC Reference Manual under SETCOLOR.
LUMINANCE	A even value from 0 to 14. Refer to sectio 9 of the ATARI 400/800 BASIC Reference Manual under SETCOLOR.

3.7.3.5 Color Procedure

```
PROCEDURE COLOR(COLOR_VALUE: INTEGER);
```

COLOR performs the same function as the COLOR statement in BASIC.

COLOR_VALUE A value from 0 to 255. Refer to section 9 of the ATARI 400/800 BASIC Reference Manual under COLOR.

3.7.3.6 Plot Procedure

```
PROCEDURE PLOT(X,Y: INTEGER);
```

PLOT performs the same function as the PLOT statement in ATARI BASIC. It plots a point in the current color at the screen position X,Y.

X	the horizontal coordinate on the screen.
Y	the vertical coordinate on the screen.

GRRESULT = value of an XIO PUT character call.

3.7.3.7 Locate Procedure

```
FUNCTION  LOCATE(X,Y: INTEGER): INTEGER;
```

LOCATE performs the same function as the LOCATE statement in ATARI BASIC. It returns the pixel value at the screen position X,Y.

X	the horizontal coordinate on the screen.
Y	the vertical coordinate on the screen.

GRRESULT = value of an XIO GET character call.

3.7.3.8 Position Procedure

```
PROCEDURE POSITION(X,Y: INTEGER);
```

POSITION performs the same function as the POSITION statement in ATARI BASIC. It moves the invisible graphics cursor to position X,Y. Note the cursor is not moved until the next I/O function is performed.

X	the horizontal coordinate on the screen.
Y	the vertical coordinate on the screen.

3.7.3.9 Drawto Procedure

```
PROCEDURE DRAWTO(X,Y: INTEGER);
```

DRAWTO performs the same function as the DRAWTO statement in ATARI BASIC. It draws a line from the current graphics position to position X,Y in the current color.

X	the horizontal coordinate on the screen.
Y	the vertical coordinate on the screen.

GRRESULT = value of an XIO DRAWTO call.

3.7.3.10 Fill Procedure

```
PROCEDURE FILL(X,Y: INTEGER);
```

FILL performs the same function as the XIO 18 call in ATARI BASIC except it performs a plot at position X,Y to move the cursor to X,Y at the end of the FILL.

X	the horizontal coordinate on the screen.
Y	the vertical coordinate on the screen.

GGRESULT = value of an XIO FILL call.

3.7.4 Sound Procedures and Functions

3.7.4.1 Sound Procedure

```
PROCEDURE SOUND(VOICE,PITCH,DISTORTION,VOLUME: INTEGER);
```

SOUND performs the same function as the SOUND statement in ATARI BASIC. It turns on the sound channel indicated by VOICE at the indicated PITCH, DISTORTION, and VOLUME.

VOICE	One of the four sound channels at 0 to 3.
PITCH	A value between 0 and 255. Refer to section 10 of the ATARI BASIC manual under SOUND.
DISTORTION	A even value from 0 to 14. Refer to section 10 of the ATARI BASIC manual under SOUND.
VOLUME	A value from 0 to 15. 0 is off; 15 is maximum volume.

3.7.4.2 Soundoff Procedure

```
PROCEDURE SOUNDOFF;
```

SOUNDOFF turns off the sound to all the sound channels.

3.7.5 Controller Functions

3.7.5.1 Paddles

3.7.5.1.1 Paddle Function

```
FUNCTION PADDLE(PDLNUM: INTEGER): INTEGER;
```

PADDLE performs the same function as the PADDLE statement in ATARI BASIC. It returns the current value of one of the eight paddles.

PDLNUM — Is the paddle number to return; must be a value between 0 and 7.

3.7.5.1.2 Trigger Function

```
FUNCTION PTRIG(PDLNUM: INTEGER): INTEGER;
```

PTRIG performs the same function as the PTRIG statement in ATARI BASIC. It returns the current trigger value of one of the eight paddles.

PDLNUM — Is the paddle number to return; must be a value between 0 and 7.

3.7.5.2 Joysticks

3.7.5.2.1 Stick Function

```
FUNCTION STICK(STKNUM: INTEGER): INTEGER;
```

STICK performs the same function as the STICK statement in ATARI BASIC. It returns the current value of one of the four joysticks.

STKNUM — Is the joystick number to return; must be a value between 0 and 3.

CHAPTER 4: RUN-TIME ERROR HANDLING

The ATARI Pascal system supports two types of run-time checking: range and exception.

Range checking is performed on array subscripts and on subrange assignments. The default condition of the system is that these checks are disabled. You may enable them around any section of coding desired using the $R and $X toggles (see sections 2.2.3.4 and 2.2.3.5). These sections describe the implementation of this mechanism and how you may take advantage of this mechanism to handle run-time errors in a non-standard manner.

The general philosophy is that error checks and error routines will set Boolean flags. These Boolean flags along with an error code will be loaded onto the stack and the built-in routine @ERR is called with these two parameters. The @ERR routine will then test the Boolean parameter. If it is false then no error has occurred and the @ERR routine will exit back to the compiled code and execution continues. If it is true the @ERR routine will print an error message and lets you continue or abort.

Listed below are the error numbers passed to the @ERR routine:

Value	Meaning
1	Divide-by-0 check
2	Heap overflow check
3	String overflow check
4	Range check

4.1 Range Checking

When range checking is enabled the compiler generates calls to @CHK for each array subscript and subrange assignment. The @CHK routine leaves a Boolean value on the stack and the compiler generates calls to @ERR after the @CHK call. If range checking is disabled and a subscript falls outside the valid range, unpredictable results will occur. For subrange assignments, the value will be truncated at the byte level.

4.2 Exception Checking

When exception checking is enabled, the compiler will load the error flags (zero divide, string overflow, and heap overflow) as needed and call the @ERR routine after each operation that can set the flags. If exception checking is disabled the run-time routines attempts to provide a friendly action if possible: divide by zero results in a

maximum value being returned, heap overflow does nothing, and string overflow truncates.

4.3 User Supplied Handlers

You can write your own @ERR routine to be used instead of the system routine. You should declare the routine as:

```
PROCEDURE @ERR(ERROR:BOOLEAN; ERRNUM:INTEGER);
```

The routine will be called, as mentioned above, each time an error check is needed and this routine should check the ERROR variable and exit if it is FALSE. You may decide the appropriate action if the value is true. The values of ERRNUM are as shown in section 9.0.

4.4 Fatal Errors

"Fatal Errors" message can be deciphered for debugging purposes but may be confusing. The error can be translated to the Pascal error message and to the ATARI standard error message. The following example will illustrate the translation process:

```
  Fatal Error 64.88 --> Pascal Error . ATARI Error

Using base 16 (non-standard, 64    -- 100  and 88    --  136
                               16       10      16        10
```

A Pascal 100 error for our system refers to an operating system error. In this example we would then look at the ATARI Error 136 message to see that our error relates to an "EOF".

The following are predefined Pascal fatal errors.

64: Error while chaining.
65: Bad pseudo code.
66: Bad pseudo code.
67: Undefined pseudo opcode.
68: Stack overflow (program too complex).

CHAPTER 5: STRUCTURE/FORMAT OF A PASCAL PROGRAM

This chapter describes the data types and how they are stored. It also discusses the use of strings.

A description of the layout of a .COM file in memory under DOS 2.0S is presented.

5.1 Data Types

This section describes how the standard Pascal data types are implemented in ATARI Pascal. Table - summarizes the data types.

Data Type	Size	Range
CHAR	1 8-bit-byte	0..255
BOOLEAN	1 8-bit-byte	false..true
INTEGER	1 8-bit-byte	0..255
INTEGER	2 8-bit-bytes	-32768..32767
BYTE	1 8-bit-byte	0..255
WORD	2 8-bit-bytes	0..65535
FLOATING REAL	4 8-bit-bytes	10E-98..10E+98
STRING	1..256 bytes	----------
SET	32 8-bit-bytes	0..255

5.1.1 CHAR

The data type CHAR is implemented using one 8-bit byte for each character. The reserved word PACKED is assumed on arrays of CHAR. CHAR variables may have the range of CHR(0).. CHR(255). When pushed on the stack, a CHAR variable is 16 bits, with the high-order byte containing 00. This is to allow OD, ODD, CHR, and WRD to work together.

5.1.2 BOOLEAN

The data type BOOLEAN is implemented using one 8-bit byte for each BOOLEAN variable. When pushed on the stack, 8 bits of 0 are pushed to provide compatibility with built-in operators and routines. The reserved word PACKED is allowed but does not compress the data structure any more than one byte per element (this occurs with and without the packed instruction). ORD(TRUE) = 0001 and ORD(FALSE) = 0000. The BOOLEAN operators AND, OR and NOT operate only on ONE byte. Refer to the & and ! operators for 16-bit boolean operators.

```
--------------------
|X|X|X|X|X|X|X|X|0/1|    (X means don't care)
--------------------
```

5.1.3 INTEGER

The data type INTEGER is implemented using two 8-bit bytes for each INTEGER variable. MAXINT = 32767 and INTEGERS can be in the range -32768..32767. An integer subrange declared to be within the 0..255 range occupies only one byte of memory instead of two bytes. Integer constants may be hexadecimal numbers by preceeding the hex number with a dollar sign (e.g. $0F3B).

5.1.4 REAL

The implementation of the data type REAL in ATARI Pascal is the same as that used by ATARI BASIC. Six bytes of data are required to implement a floating point number. The first byte contains the mantissa sign, the exponent in excess-64. The base of the exponent is 100. The remaining five bytes contain the mantissa in binary coded decimal. The precision is approximately 8 digits.

ms = most significant bits
ls = least significant bits

5.1.5 Byte

The BYTE data type occupies a single byte. It is compatible with both INTEGER and CHAR types. This compatibility can be very useful when manipulating control characters, handling character arithmetic, etc. Characters and integers may be assigned to a BYTE.

5.1.6 Word

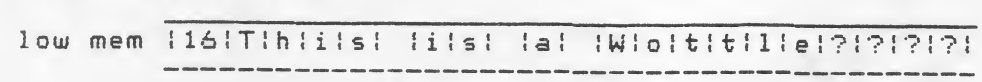
WORD is an unsigned, native machine word. All arithmetic and comparisons performed on expressions of type WORD are unsigned.

5.1.7 String

5.1.7.1 Definition

The pre-declared type STRING is like a packed array of characters in which the byte 0 contains the dynamic length of the string, and bytes 1 through n contain the characters. Strings may be up to 255 characters in length. The default length is 80 characters that may be altered when a variable of type STRING is declared (see example below).

The string "This is a Wottle" is 16 characters long. The following diagram shows how these characters are stored in a string declared to be 20 characters long.



If the number of characters in the string is less than the declared length, the bytes on the end are not defined. Note that the length is stored in the first byte and the total number of bytes required for the string is 17.

Example:

```
VAR
  LONG_STR:       STRING;        (This may contain up to 80 characters)
  SHORT_STR:      STRING[10];    (This may contain up to 10 characters)
  VERY_LONG_STR:  STRING[255];   (This may contain up to 255 characters,
                                  the maximum allowed.)
```

5.1.7.2 Assignment

Assignment to a string variable may be made via the assignment statement, reading into a string variable using READ or READLN, or the pre-defined string functions and procedures.

Example:

```
PROCEDURE ASSIGN;
VAR
  LONG_STR  : STRING;
  SHORT_STR : STRING [12];
BEGIN

  LONG_STR := 'This string may contain as many as eighty characters';
  WRITELN(LONG_STR);

  WRITE('type in a string 10 characters or less : ');
  READLN(SHORT_STR);
  WRITELN(SHORT_STR);

  SHORT_STR := COPY(LONG_STR,1,11);
  WRITELN('COPY(LONG_STR..)=',SHORT_STR);
END;
```

Output:

```
This string may contain as many as eighty characters
type in a string 10 characters or less : {123456} (USER INPUT) .
123456
COPY(LONG_STR..)=This string m
```

The individual characters in a string variable are accessed as if the string were an array of characters. Thus, normal array subcripting via constants, variables, and expressions allows assignment and access to individual bytes within the string. Access to the string over its entire declared length is legal and does not cause a run-time error even if an access is made to a portion of the string beyond the current dynamic length. If the string is actually 20 characters long and the declared length is 30 then STRING [25] is accessible.

Example:

```
PROCEDURE ACCESS;
VAR
  I : INTEGER;
BEGIN
  I := 15;
  LONG_STR := '123456789abcdef';
  WRITELN(LONG_STR);
  WRITELN(LONG_STR[6], LONG STR[ i-5 ]);
  LONG_STR[16] := '*';
  WRITELN(LONG_STR[16]);
  WRITELN(LONG_STR); (* will still only write 15-characters *)
END;
```

Output:

```
123456789abcdef
6a
*
123456789abcdef
```

5.1.7.3 Comparisons

Comparisons are valid between two variables of type STRING (regardless of their length) or between a variable and a literal string. Literal strings are sequences of characters between single quotation marks. Comparisons may also be made between a string and a character. The compiler "forces" the character to become a string by using the CONCAT buffer; therefore, comparison of the result of the CONCAT function and a character is not meaningful because this comparison would always be equal.

Example:

```
PROCEDURE COMPARE;

VAR
  S1,S2 : STRING[10];
  CH1   : CHAR;

BEGIN
  S1 := '012345678';
  S2 := '222345678';

  IF S1 < S2 THEN
    WRITELN(S1,' is less than ',S2);

  S1 := 'alpha beta';
  IF S1 = 'alpha beta ' THEN
    WRITELN('trailing blanks don''t matter')
  ELSE
    WRITELN('trailing blanks count');
  IF S1 = '    alpha beta' THEN
    WRITELN('blanks in front don''t matter')
  ELSE
    WRITELN('blanks in front do matter');
  IF S1 = 'alpha beta' THEN
    WRITELN(S1,' = ',S1);
  S1 := 'Z';
  CH1 := 'Z';
  IF S1 = CH1 THEN
   WRITELN('strings and chars may be compared');
END;
```

Output:

```
012345678 is less than 222345678
trailing blanks don't matter
blanks in front do matter
alpha beta = alpha beta
strings and chars may be compared
```

5.1.7.4 Reading and Writing Strings

Strings may be written to a text file using the WRITE or WRITELN procedure. WRITELN will cause a carriage return and line feed following the string. Reading a string is always done via the READLN statement because strings are terminated with a carriage return and line feed. Using READ will not work, because the end-of-line characters are incorrectly processed. Tabs are expanded when they are read into a variable of the STRING type.

5.1.8 Set

The SET data type is always stored as a 32 byte item. Each element of the set is stored as one bit. The low order bit of each byte is the first bit in that byte of the set. Shown below is the set "A".."Z" (bits 65..122)

Byte number	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	...	1F
	--	--	--	--	--	--	--	--	--	--	--	--	--		--
Contents	00	00	00	00	00	00	00	00	FE	FF	FF	07	00	...	00

CHAPTER 6: COMPATIBILITY

Pascal is considerably more standardized than BASIC. Nearly every version of Pascal is based on a definition of the language contained in "Pascal User Manual and Report", by Kathleen Jensen and Niklaus Wirth, Springer-Verlag, 1974. The Pascal Language System is a superset of the Pascal described in this book. In addition, ATARI Pascal meets a more recent standard, namely the ISO standard (International Standards Organization, similar to ANSI). It is expected that any Pascals developed from now on will certainly be compared to this standard, and will strive to meet it. ATARI has learned the importance of compatibility from its experience with ATARI BASIC. A Pascal that meets the newly developed ISO standard is a very positive step toward compatibility.

A possible compatibility problem is that the ATARI Pascal Language System is not entirely compatible with UCSD Pascal. UCSD Pascal has attained considerable popularity on small computers. While it is true that ATARI Pascal is not completely compatible with UCSD Pascal, it should be remembered that both versions are written around a common core-- Pascal as defined by Jensen and Wirth. The differences, though present, are not as significant as, for example, the differences in various BASICs. In addition, the superiority of the Pascal Language System justifies the incompatibilities involved.

A brief comparison of the features that differ between the two Pascals follows. Parts of this comparison is necessarily somewhat technical, as most of the differences are deep in the details of the language.

6.1 Incompatabilities With UCSD Pascal

1. The predefined type INTERACTIVE is available only in UCSD Pascal. On the ATARI Computer, any file associated with the computer console is automatically interactive, and therefore this type is not needed and would only clutter the language unnecessarily.

2. The predefined procedure SEEK is available only in UCSD Pascal.

3. UCSD Pascal uses UNITS to implement modular compilation. They are easy to understand, but are much more restrictive than ATARI Pascal's implementation of modular compilation.

4. UCSD Pascal provides SEGMENT procedures to allow overlays from diskette. ATARI Pascal will use the standard DOS methods for invoking overlays.

5. Sets can be considerably larger in UCSD Pascal. They are considerably faster in ATARI Pascal. The ATARI Pascal implementation is more in keeping with the spirit of the Jensen and Wirth standard.

6. UCSD Pascal includes bit-level packing on PACKED structures. Bit-level packing costs in both the size of the interpreter, and the speed of execution of the program (particularly on a machine based on the 6502 microprocessor which does not contain multiply and divide).

7. UCSD Pascal has a construct EXIT <procedure name> that is not included in ATARI Pascal, although ATARI Pascal permits EXIT without the procedure name. Many Pascal purists feel that the construct as implemented by UCSD is not a structured construct, and is therefore counter to the philosophy of the language.

8. UCSD Pascal includes the type LONG INTEGER that is not available in ATARI Pascal.

9. Several features in UCSD Pascal are operating system dependent, e.g., long file names, and unit I/O (similar to XIO). These have not been implemented in ATARI Pascal.

6.2 Additional Features Available with the ATARI Pascal Language System

1. The ATARI Pascal Language System is a complete ISO standard Pascal. Some of the ISO features not included in UCSD Pascal are conformant array handling, procedures and functions as parameters, local files, PACK and UNPACK procedures, READ and WRITE for non-text files, WRITE and WRITELN of Boolean expressions, and GOTO out of a procedure into a surrounding procedure.

2. The Pseudo code implemented in ATARI Pascal was optimized for the 6502 microprocessor.

3. ATARI Pascal uses the same operating system as all other ATARI programs. ATARI Pascal and ATARI BASIC files are the same format, and data files can be read by either language. You do not have the inconvenience of learning two different and incompatible operating systems, as you do with UCSD Pascal. In addition, ATARI Pascal allows access to I/O in a manner very similar to ATARI BASIC. XIO, graphics, sound, game controllers, and named devices are all implemented.

4. UCSD segment procedures are limited to six per program which limits the development of large applications. ATARI Pascal should allow the development of more complex applications.

5. ATARI Pascal has nine or ten digits of precision on real numbers. UCSD Pascal has only 6.5 digits of precision.

6. ATARI Pascal permits the programmer to trap errors, and prevent programs from aborting.

7. ATARI Pascal provides protection when reading in a string. If the string is too long for the receiving variable, ATARI Pascal will truncate the string. UCSD Pascal will overwrite the bytes following the string in memory, resulting in undefined program errors.

8. ATARI Pascal has extended the CASE statement by adding an ELSE clause. If the case selecting expression would not result in the execution of a statement with the CASE, the ELSE clause is executed. ELSE simplifies error checking. Execution of a similar unmatched CASE in UCSD Pascal causes an undefined result.

9. Modular compilation is much more flexible in ATARI Pascal. Local static variables, external procedures and functions located in the main program, and external global variable usage are all missing from UCSD Pascal.

10. ATARI Pascal has a built in BYTE data type. This data type eliminates the use of confusing CASE variant records when manipulating characters as integers.

11. ATARI Pascal has a built-in WORD data type. An unsigned 16-bit data type is very useful for address arithmetic and machine-level programming.

12. UCSD Pascal does not fully implement compatibility between strings and characters. Strings and characters are totally compatible in ATARI Pascal.

13. For system dependent applications, ATARI Pascal allows relaxation of type checking rules. This relaxation allows machine I/O and memory manipulation to be done without cluttering the program with confusing CASE variant records.

14. ATARI Pascal has the built-in bit-manipulation routines TSTBIT, SETBIT, CLRBIT, SHL, and SHR. Bit manipulation in UCSD Pascal must be done with CASE variant records, which are confusing and machine dependent.

15. In both ATARI Pascal and UCSD Pascal, the GET/PUT file I/O is quite slow. ATARI Pascal also contains GNB and WNB, which are high-speed I/O routines for byte I/O.

16. ATARI Pascal fully implements the NEW and DISPOSE procedures, including fragmentation management and re-use of disposed areas. UCSD Pascal implements a much more restricted version of these procedures. This feature is vital to any program doing dynamic data management.

17. ATARI Pascal allows full use of files. UCSD Pascal does not allow local files, files in records, or arrays of files.

18. ATARI Pascal includes the ADDR function. This returns the address of a variable, procedure, or function. This function is useful when doing machine dependent programming.

19. ATARI Pascal has a built-in INLINE feature that can be used to generate compile-time constant data. This feature eliminates run-time initialization of constant tables, increasing execution speed and decreasing code size.

20. ATARI Pascal allows output in any number base from two through sixteen.

21. ATARI Pascal allows input of either decimal or hex numbers.

22. ATARI Pascal has not extended the parameter list on any ISO standard routine (specifically RESET and REWRITE). For acessing external files, a new procedure (ASSIGN) has been added.

CHAPTER 7: LANGUAGE DEFINITION

7.1 Introduction

Chapter 7 defines the language features of ATARI Pascal that are common to each implementation of the compiler. It is assumed here that you are familiar with Jensen and Wirth's "Report" and/or the ISO draft standard (DPS/7185). The ATARI Pascal features that differ from those in the ISO standard and in Jensen and Wirth's "Report", are described by section. In each section, BNF (Backus Normal Form) syntax is provided for reference. The complete BNF description of the language is present in an appendix. Each section corresponds to Wirth's "Report".

7.2 Summary of the ATARI Pascal Language

Features of the ISO Pascal include the data types REAL, INTEGER, CHAR, BOOLEAN, multidimensional ARRAYS, user-defined RECORDS, POINTERS types, file variables, user-defined TYPES and CONSTANTS, and SETS (implemented in this version with a base type of 256 one byte elements). ENUMERATED types, and SUBRANGE types.

Also included in ISO Pascal are PROCEDURES, FUNCTIONS, and PROGRAMS. Passing procedures and functions as parameters to a Pascal routine are part of the ISO standard, as well as conformant arrays. Arrays of the same index type and element type but different sizes may be passed to the same procedure. External parameters with the PROGRAM statement are supported at the syntax level.

TYPED and TEXT files are supported as defined in the standard using the Pascal routines RESET, REWRITE, GET, PUT, READ, WRITE, READLN, and WRITELN. The default I/O files INPUT and OUTPUT are defined.

All ISO statements are supported including WITH, REPEAT...UNTIL, CASE, WHILE loops, FOR loops, IF..THEN..ELSE, and GOTO.

PACK and UNPACK are supported, but do not affect the outcome of the program (data structures are always packed at the byte level). NEW and DISPOSE are implemented; they allocate and deallocate HEAP space.

Modular compilation is an extension of the ATARI compiler. Variables and routines may be made public and/or private and may be called from any other module or from the main program.

The extended data types STRING, BYTE, and WORD are implemented in the ATARI Pascal compiler. The STRING type includes a length byte followed by the maximum number of bytes possible. Routines are supplied to INSERT a character or a string, DELETE from a string, find the POSition of a character in a string, COPY a portion of one string to another, and CONCATenate two or more strings and/or characters together. BYTE is a one-byte data item for representing numbers from 0 to 255. WORD is two bytes for the 8-bit CPU.

Additional procedures to manage files on diskette are implemented. A file on diskette is associated with an internal file and may be closed or deleted.

Manipulating BITS and BYTES is done using routines to TEST, SET, CLEAR, SHIFT RIGHT, and LEFT, return HI or LOW of a variable, and SWAP the high and low bytes of a variable.

Miscellaneous routines to access items in a program are to return the address of a variable or routine, return the size of a variable or type, move a given number of bytes from one memory location to another and fill a data item with a certain character. Also, the amount of HEAP space available at any given time is accessible. Garbage collection on the HEAP is supported.

Logical operators for non-Booleans are implemented.

HEX literal numbers may be used with a dollar sign ($).

Include files are supported.

An ELSE clause on the CASE statement is provided.

Program CHAINING is supported. Chaining is such that the code for one program is totally replaced by code for the next program, but heap space may be maintained across a CHAIN.

7.3 Notation, Terminology, and Vocabulary

```
<letter> ::= A | B | C | D | E | F | G | H | I | J |
             K | L | M   N | O | P | Q | R | S | T |
             U | V | W | X | Y | Z | a | b | c | d |
             e | f | g | h | i | j | k | l | m | n |
             o | p | q | r | s | t | u | v | w | x |
             y | z | @

<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
             A | B | C | D | E | F | (only allowed in HEX numbers)

<special symbol> ::= (reserved words are listed in the appendix)
             +  | -  | * | / | =  |<> | < | > |
             <= | >= | ( | ) | [  | ] | ^ |
             := | .  | , | ; | :  | ' |

            (the following are additional or substitutions:)
            ( . | .) | \ | ? | !  | |  | $ | &

            (. is a synonym for [
            .) is a synonym for ]
            ? and \, are synonyms
            !, and | are synonyms
            &
```

Extensions:

The symbol "@" is a legal letter in addition to those listed in the "Report". This symbol has been added because the run-time library routines are written using this special character as the first letter of their name. By adding "@" conflict with user names is avoided but users are allowed to call these routines. See section 7.4 for further information.

A comment beginning with "(*" must end with "*)".

```
<comment> ::= (* < any characters > *)
```

7.4 Identifiers, Numbers, and Strings.

```
<identifier>          ::= <letter> {<letter or digit or underscore>}
<letter or digit>     ::= <letter> | <digit> | _

<digit sequence>      ::= <digit> {<digit>}
<unsigned integer>    ::= $ <digit sequence> |
                          <digit sequence>

<unsigned real>       ::= <unsigned integer> . <digit sequence>  |
                          <unsigned integer> . <digit sequence>
                          E <scale factor>                       |
                          <unsigned integer> E <scale factor>
<unsigned number>     ::= <unsigned integer | <unsigned real>
<scale factor>        ::= <unsigned integer> | <sign><unsigned integer>
<sign>                ::= + | -

<string>              ::= '<character> {<character>}' | ''
```

All identifiers are significant to eight characters. External identifiers are significant to either six or seven characters depending upon usage. The underscore character (_) is legal between letters and digits in an identifier and is ignored by the compiler (i.e., A_B is equivalent to AB). Identifiers may begin with an "@". To allow declaration of external run-time routines within a Pascal program. Users are, in general, advised to avoid the "@" character to eliminate the chance of conflict with run-time routine names.

Numbers may be hex as well as decimal. Placing a "$" in front of an integer number causes it to be interpreted as a hex number by the compiler. The symbol <digit> now includes: "A", "B", "C", "D", "E" and "F". These may be upper or lower case.

7.5 Constant Definitions

```
<constant identifier> ::= <identifier>
<constant>            ::= <unsigned number>              |
                          <sign><unsigned number>        |
                          <constant identifier>          |
                          <sign><constant identifier>    |
                          <string>
<constant defination> ::= <identifier> = <constant>
```

In addition to all constant declarations available in standard Pascal, ATARI Pascal supports declaration of a null string constant:

Example:

```
nullstr = '';
```

7.6 Data Type Definitions

```
<type>            ::= <simple type>      |
                      <structured type>  |
                      <pointer type>
<type definition>::= <identifier> = <type>
```

7.6.1 Simple Types

```
<simple type>      ::= <scalar type>      |
                       <subrange type>    |
                       <type identifier>
<type identifier> ::= <identifier>
```

7.6.1.1 Scalar Types

```
<scalar type>  ::= ( <identifier  {, <identifier>} )
```

7.6.1.2 Standard Types

The following types are standard in ATARI Pascal.

INTEGER
REAL
BOOLEAN
CHAR

BYTE
WORD
STRING

Three additional standard types exist in ATARI Pascal. Refer to the Appendix for information on representation and usage of all standard and structured types.

STRING : Packed array [0..n] of char
byte 0 is dynamic length byte
bytes 1..n are characters

BYTE : Subrange 0..255 with special attribute that it is compatible also with CHAR type.

WORD : Unsigned native machine word

7.6.1.3 Subrange Types

```
<subrange type> ::= <constant> .. <constant>
```

7.6.2 Structured Types

```
<structured type>              ::= <unpacked structured type>          |
                                    PACKED <unpacked structured type>
<unpacked structured type>     ::= <array type>    |
                                   <record type>   |
                                   <set type>      |
                                   <file type>
```

The reserved word PACKED is detected and handled by the ATARI Pascal compiler as follows:

All structures are packed at the BYTE level even if the PACKED reserved word is not found.

7.6.2.1 Array Types

```
<array type>          ::= <normal array>  |
                          <string array>
<string array>        ::= STRING <max length>
<max length>          ::= [ <intconst> ]  |
                          <empty>
<inconst>             ::= <unsigned integer>  |
                          <int const id>
<int const id>        ::= <identifier>
<normal array>        ::= ARRAY [ <index type> {, <index type>}] OF
                          <component type>
<index type>          ::= <simple type>
<component type>      ::= <type>
```

Variables of type STRING have a default length of 81 bytes (80 data characters). A different length can be specified in square brackets following the word STRING. The length must be a constant (either literal or declared, e.g., STRING[5] or STRING[xyz] (where xyz is a constant (xyz=10)). It represents the length of the DATA portion (i.e, one more byte is actually allocated for the length).

7.6.2.2 Record Types

```
<record type>          ::= RECORD <field list> END
<field list>           ::= <fixed part>                        |
                           <fixed part> ; <variant part>       |
                           <variant part>
<fixed part>           ::= <record section> {;<record section>}
<record section>       ::= <field identifier> {,<field identifier>} :
                           <type> | <empty>
<variant part>         ::= CASE <tag field> <type identifier> OF
                           <variant> {;<variant>}
<variant>              ::= <case label list> : (<field list>)   |
                           <empty>
<case label list>      ::= <case label> {,<case label>}
<case label>           ::= <constant>
<tas field>            ::= <identifier> : |
                           <empty>
```

7.6.2.3 Set Types

```
<set type>    ::= SET OF <base type>
<base type>   ::= <simple type>
```

The maximum range of a base type is 0..255. For example, a set of [0..10000] is not legal. The set of CHAR or set of 0..255 is legal but set of 0..256 is not.

7.6.2.4 File Types

<file type> ::= file {of <type>}

Untyped files are allowed. They are used for CHAINING and are also used with BLOCKREAD and BLOCKWRITE procedures. Be extremely careful when using untyped files.

When you wish to read a file of ASCII characters and use implied conversions for integers and real numbers use the pre-defined type TEXT. TEXT is NOT the same as FILE OF CHAR. It has conversion implied in READ and WRITE procedure calls and also may be used with READLN and WRITELN. A file of type TEXT is declared in the following manner: "VAR F : TEXT". The INCORRECT syntax for declaring a TEXT file is "VAR F : FILE OF TEXT". See the appendix on Pascal file handling.

7.6.3 Pointer Types

<pointer type> ::= ^<type identifier>

Pointer types are identical to the standard except that weak type checking exists when the RELAXED type checking feature of the compiler is enabled (the default). In this case, pointers and WORDS used as pointers are compatible in all cases.

7.6.4 Types and Assignment Compatibility

The most common standard Pascal question concerns type conflict errors messages from the compiler. Types must be identical if the variable is passed to a VAR parameter. Types must be compatible for expressions and assignment statements. To understand the difference between compatible and identical types, think of types as pointers to compile-time records. If you declare a type (such as T=ARRAY [1..10] OF INTEGER), then anything declared as type T really points to the record describing type T. If, on the other hand, you declare two variables as follows:

```
VAR
  A1 : ARRAY [1..10] OF INTEGER;
  A2 : ARRAY [1..10] OF INTEGER;
```

they are not identical. The compiler created a new record for each type and therefore A1 and A2 do not point to the same record in memory at compile-time. The general rule is that if you cannot find your way back to a type definition, then the types are not identical.

CHR, ORD, and WRD are type converson operators that generate no code but tell the compiler that the following 8-bit data item is to be considered type CHAR, INTEGER, or WORD respectively. These operators may be used in expressions and with parameters except VAR parameters.

Below is a section from the ISO draft standard (DPS-7185) which is available from the American National Standards Institute. The ISO standard definition of compatible types is as follows:

> Types T1 and T2 shall be designated compatible if any of the four statements that follow is true.
> (a) T1 and T2 are the same type.
> (b) T1 is a subrange of T2 or T2 is a subrange of T1, or both T1 and T2 are subranges of the same host type.
> (c) T1 and T2 are designated packed or neither T1 nor T2 is designated packed.
> (d) T1 and T2 are string-types* with the same number of components.
>
> ...Assignment compatibility. A value of type T2 shall be designated assignment-compatible with a type T1 if any of the five statements that follow is true.
> (a) T1 and T2 are the same type, that is neither a file-type nor a structured-type with file component (this rule is to be interpreted recursively).
> (b) T1 is the real-type and T2 is the integer-type.
> (c) T1 and T2 are compatible ordinal-types** and the value of type T2 is in the closed interval specified by the type T1.
> (d) T1 and T2 are compatible set-types and all the members of the value of type T2 are in the closed interval specified by the base-type of T1.
> (e) T1 and T2 are compatible string-types*.

At any place where the rule of assignment-compatibility is used:
(a) It shall be an error if T1 and T2 are compatible ordinal-types** and the value of type T2 is not in the closed interval specified by the type T1.
(b) It shall be an error if T1 and T2 are compatible set-types and any member of the value of type T2 is not in the closed interval specified by the base-type of the type T1.

* String-types in ISO Pascals are arrays of characters.
** Ordinal types are named subranges of numbers or enumerations.

7.7 Declaration and Denotations of Variables

```
<variable>       ::= <var>                 |
                     <external var>        |
                     <absolute var>

<external var>   ::= <EXTERNAL <var>

<absolute var>   ::= ABSOLUTE [ <constant> ] <var>

<var>            ::= <entire variable>     |
                     <component variable>  |
                     <referenced variable>
```

ABSOLUTE variables may be declared if you know the address at compile time. You declare variable(s) to be absolute using special syntax in a VAR declaration. ABSOLUTE variables are not allocated any space in your data segment by the compiler and you are responsible for making sure that no compiler-allocated variables conflict with the absolute variables. NOTE: STRING VARIABLES MAY NOT EXIST AT LOCATIONS <= $100.. This is done so that the run-time routines can detect the difference between a string address and a character on the top of the stack. Characters have the high byte of 0 when present on the stack. After the colon (:) and before the type of variable(s), you place the keyword ABSOLUTE followed by the address of the variable in brackets ([..]):

Examples:

```
I:      ABSOLUTE [$800] INTEGER;
SCREEN: ABSOLUTE [$C000] ARRAY[0..15] OF ARRAY[0..63] OF CHAR;
```

7.7.1 Entire Variables

```
<entire variable>       ::= <variable identifier>
<variable identifier>   ::= <identifier>
```

7.7.2 Component Variables

```
<component variable> ::= <indexed variable> |
                         <field designator> |
                         <file buffer>
```

7.7.2.1 Indexed Variables

```
<indexed variable> ::= <array variable> [<expression> {,<expression>}]
<array variable>   ::= <variable>
```

STRING variables are to be treated as a PACKED array of CHAR for subscripting purposes. The valid range is 0..maxlength, where maxlength is 80 for a default length.

7.7.2.2. Field Designators

```
<field designator> ::= <record variable> . <field identifier>
<record variable>  ::= <variable>
<field identifier> ::= <identifier>
```

7.7.2.3 File Buffers

```
<file buffer>     ::= <file variable>^
<file variable>   ::= <variable>
```

7.7.3 Referenced Variables

```
<referenced variable> ::= <pointer variable>^
<pointer variable>    ::= <variable>
```

7.8 Expressions

```
<unsigned constant> ::= <unsigned number>                                    |
                        <string>                                             |
                        NIL                                                  |
                        <constant identifier>
<factor>            ::= <variable>                                           |
                        <unsigned constant>                                  |
                        <function designator>                                |
                        ( <expression> )                                     |
                        <logical NOT operator> <factor>
                        <set>
<set>               ::= [ <element list> ]
<element list>      ::= <element> {,<element>}                               |
                        <empty>
<element>           ::= <expression>                                         |
                        <expression> .. <expression>
<term>              ::= <factor> <multiplying operator> <factor>
<simple expression> ::= <term>                                               |
                        <simple expression> <adding operator> <term>         |
                        <adding operator> <term>
<expression>        ::= <simple expression>                                  |
                        <simple expression> <relational operator>
                            <simple expression>
```

An additional category of operators on 16-bit variables are &, ! (also |), (also \ and ?) denoting AND, OR and ONE's complement NOT, respectively. These have the same precedence as their equivalent boolean operators and accept any type of operand with a size <= 2 bytes.

7.8.1 Operators

7.8.1.1 The Operator NOT

<logical NOT operator> ::=NOT | \ | ?

\ and ? are NOT operators for non-Boolean operators.

7.8.1.2 Multiplying Operators

<multiplying operator> ::= * | / | DIV | MOD | AND | &

& is an AND operator on non-Boolean operators.

7.8.1.3 Adding Operators

<adding operator> ::= + | - | OR | | | !

! (synonym |) is an OR operator on non-Boolean operators.

7.8.1.4 Relational Operators

<relational operators> ::= = | <> | < | <= | >= | IN

7.8.2 Function Designators

<function designator> ::= <function identifier> |
<function identifier> (<parm> {,<parm>}
<function identifier> ::= <identifier>

7.9 Statements

```
<statement>                 ::= <label> : <unlabelled statement>  |
                                <unlabelled statement>
<unlabelled statement>      ::= <simple statement>                |
                                <structured statement>
<label>                     ::= <unsigned integer>
```

7.9.1 Simple Statements

```
<simple statement>  ::= <assigned statement>  |
                        <procedure statement> |
                        <goto statement>      |
                        <empty statement>
<empty statement>   ::= <empty>
```

7.9.1.1 Assignment Statements

```
<assignment statement>  ::= <variable> := <expression>              |
                            <function identifier> := <expression>
```

To the list of exceptions to assignment compatibility add:

1. Integer expressions may be assigned to variables of type pointer. For example:

```
TYPE X = RECORD
           (* field declarations *)
         END;
VAR P : ^X;
    I : INTEGER;
.................
P := I+1;
```

2. Expressions of type CHAR may be assigned to variables of type STRING.

3. Variables of type CHAR and literal characters may be assigned to variables of type BYTE.

4. Expressions evaluting to the type WORD may be assigned to pointer variables.

5. Expressions evaluating to the type INTEGER may be assigned to variables of type WORD.

7.9.1.2 Procedure Statements

```
<procedure statement> ::= <procedure identifier>{(<parm> {,<parm>}) |
                          <procedure identifier>
<procedure identifier>::= <identifier>
<parm>                ::= <procedure identifier>  |
                          <function identifier    |
                          <expression>            |
                          <variable>
```

The maximum number of parameters for a procedure or function is fifty (50).

7.9.1.3 GOTO Statements

```
<goto statement>  ::= goto <label>
```

7.9.2 Structured Statements

```
<structured statement> ::= <repetitive statement>   |
                           <conditional statement>  |
                           <compound statement>     |
                           <with statement>
```

7.9.2.1 Compound Statements

```
<compound statement> ::= BEGIN <statement> {;<statement>} END
```

7.9.2.2 Conditional Statements

```
<conditional statement> ::= <case statement>  |
                            <if statement>
```

7.9.2.2.1 If Statements

```
<if statement> ::= IF <expression> THEN <statement> ELSE <statement> |
                   IF <expression> THEN <statement>
```

7.9.2.2.2 Case Statements

```
<case statement>  ::= CASE <expression> OF
                      <case list> {;<case list>}
                      {ELSE <statement>}
                      END

<case list>       ::= <label list> : <statement>  |
                      <empty>
<label list>      ::= <case label> {,<case label>}
<case label>      ::= <non-real short scalar constant>
```

ATARI Pascal implements an ELSE clause on the CASE statement. In addition, if the selecting expression does not match any of the case

selectors, the program flow will "drop through" the CASE statement. The standard says this condition is an error.

Example:

```
CASE CH OF
  'A' : WRITELN('A');
  'Q' : WRITELN('Q');
  ELSE
    WRITELN('NOT A OR Q')
END
```

7.9.2.3 Repetitive Statements

```
<repetitive statement> ::= <repeat statement>  |
                           <while statement>   |
                           <for statement>
```

7.9.2.3.1 While Statements

```
<while statement>  ::= WHILE <expression> DO <statement>
```

7.9.2.3.2 Repeat Statements

```
<repeat statement>  ::= REPEAT <statement> {;<statement>} UNTIL
                           <expression>
```

7.9.2.3.3 For Statements

```
<for statement> ::= FOR <ctrlvar> := <for list> DO <statement>
<for list>      ::= <expression> DOWNTO <expression> |
                    <expression>   TO   <expression>
<ctrlvar>       ::= <variable>
```

7.9.2.4 With Statements

```
<with statement>        ::= WITH <record variable list> DO <statement>
<record variable list> ::= <record variable> {,<record variable>}
```

Note that the ISO standard differs from Pascal defined by Jensen and Wirth in that only LOCAL variables are allowed as FOR loop control variables. This prevents such programming errors as the inadvertent use of a GLOBAL variable as a FOR control variable when nested five levels deep.

You're limited to 16 FOR and/or WITH statements in a single procedure/function. This limitation is so that the compiler can allocate a fixed number of temporary locations (16 words) in the data segment for the procedure/function.

7.10 Procedure Declarations

```
<procedure declaration>     ::= EXTERNAL <procedure heading>     |
                                <procedure heading> <block>
<block>                     ::= <label declaration part>
                                <constant definition part>
                                <type definition part>
                                <variable declaration part>
                                <procfunc declaration part>
                                <statement part>

<procedure heading>         ::= PROCEDURE <identifier> <parmlist>  |
                                PROCEDURE <identifier>;


<parmlist>                  ::= ( <fparm> {,<fparm>} )

<fparm>                     ::= <procedure heading>     |
                                <function heading>      |
                                VAR <parm group>        |
                                <parm group>

<parm group>                ::= <identifier> {,<identifier>} :
                                   <type identifier>                |
                                 <identifier> {,<identifier>} :
                                   <conformant array>

<conformant array>          ::= ARRAY [ <indxtyp> {;<indxtyp} ] OF
                                <conarray2>

<conarray2>                 ::= <type identifier>  |
                                <conformant array>

<indxtyp>                   ::= <identifier>..<identifier> : <ordtypid>

<ordtypid>                  ::= <scalar type identifier>     |
                                <subrange type identifier>

<scalar type identifier>    ::= <identifier>

<subrange type identifier>  ::= <identifier>

<label declaration part>    ::= <empty>     |
                                LABEL <label> {,<label>} ;

<constant definition part>  ::= <empty>     |
                                CONST
                                  <constant definition>
                                  {;<constant definition>} ;

<type definition part>      ::= <empty>     |
                                TYPE
                                  <type definition>
                                  {;<type definition>} ;
```

```
<variable declaration part>::= <empty>     |
                               VAR
                                 <variable declaration>
                                 {;<variable declaration>} ;

<procfunc part>            ::= {<proc or func> ; }

<proc or func>             ::= <procedure declaration>   |
                               <function declaration>

<statement part>           ::= <compound statement>
```

7.10.1 Standard Procedures

The following is a list of ATARI Pascal built-in procedures. See Chapter 3 for parameters and usage. These procedures are pre-declared in a scope surrounding the program. Therefore, any user routines of the same name will take precedence.

NEW	DISPOSE	EXIT	INSERT
DELETE	COPY	CONCAT	FILLCHAR
MOVELEFT	MOVERIGHT	CLRBIT	HI
LO	SETBIT	SHL	SHR
SWAP	TSTBIT	LENGTH	POS
ADDR	MOVE	MAXAVAIL	MEMAVAIL
SIZEOF			

7.10.1.1 File Handling Procedures

All standard file handling procedures are included. In addition the procedure ASSIGN(F,string) is added where "F" is a file and "string" is a literal or variable string. ASSIGN assigns the external file name contained in the string to file F. It is used preceding a RESET or REWRITE. See Section 3.4.15 for more details.

Listed below are the names of the file handling procedures:

GET	PUT	RESET	REWRITE
ASSIGN	CLOSE	CLOSEDEL	PURGE
OPEN	BLOCKREAD	BLOCKWRITE	READ
CHAIN	PAGE	IORESULT	
GNB	WNB	WRITELN	
WRITE	READLN		

7.10.1.2 Dynamic Allocation Procedures

NEW DISPOSE

In addition to NEW and DISPOSE, MEMAVAIL and MAXAVAIL are also included.

7.10.1.3 Data Transfer Procedures

PACK (A,I,Z) UNPACK (Z,A,I)

7.10.2 FORWARD

Forward procedure declarations are implemented in ATARI Pascal. It is recommended that this feature not be used unless strict Pascal conformance is required. The three pass compiler, makes forward declarations unnecessary.

7.10.3 CONFORMANT ARRAYS

Note that the ISO standard has added the CONFORMANT ARRAY SCHEMA for passing arrays of similar structure (i.e., same number of dimensions, compatible index type, and same element type), but different upper and lower bounds. You may now pass, for example, an array dimensioned as 1..10 and an array 2..50 to a procedure expecting an array. You define the array as a VAR parameter and in the process of declaring the array, you also define variables to hold the upper and lower bound of the array. These upper and lower bound items are filled in at RUN-TIME by the generated code. To pass arrays in this manner, the index type must be compatible with the type of the conformant array bounds.

Below is an example of passing two arrays to a procedure that displays the contents of the arrays on the file OUTPUT:

```
PROGRAM DEMOCON;

TYPE
  NATURAL = 0..MAXINT;   (*FOR USE IN CONFORMANT ARRAY DECLARATION *)

VAR
  A1 : ARRAY [1..10] OF INTEGER;
  A2 : ARRAY [2..20] OF INTEGER;

PROCEDURE DISPLAYIT (

    VAR AR1 : ARRAY [LOWBOUND]..HIBOUND:NATURAL] OF INTEGER

                      );

(* THIS DECLARATION DEFINES THREE VARIABLES:

      AR1       : THE PASSED ARRAY
      LOWBOUND  : THE LOWER BOUND OF AR1 (PASSED AT RUN-TIME)
      HIBOUND   : THE UPPER BOUND OF AR1 (PASSED AT RUN-TIME)

*)

VAR
  I : NATURAL;
(* COMPATIBLE WITH THE INDEX TYPE OF THE CONFORMANT ARRAY *)

BEGIN
  FOR I := LOWBOUND TO HIBOUND DO
    WRITELN('INPUT ARRAY[',I,']=',AR1[I])
END;

BEGIN (* MAIN PROGRAM *)

  DISPLAYIT(A1);    (* CALL DISPLAYIT AND PASS A1 EXPLICITLY AND
                       1 AND 10 IMPLICITLY *)
```

```
    DISPLAYIT(A2)     (* CALL DISPLAYIT AND PASS A2 EXPLICITLY AND
                         2 AND 20 IMPLICITLY *)

END.
```

7.11 Function Declarations

```
<function decl>    ::= EXTERNAL <function heading> ;
                       <function heading> <block>

<function heading>::= FUNCTION <identifier><parmlist>:<result type>; ;
                      FUNCTION <identifier> : <result type> ;

<result type>      ::= <type identifier>
```

7.11.1 Standard Functions

Listed below are the names of the standard functions supported:

ABS	SQR	SIN	COS
EXP	LN	SQRT	ARCTAN
ODD	TRUNC	ROUND	ORD
WRD	CHR	SUCC	PRED
EOLN	EOF	IORESULT	MEMAVAIL
MAXAVAIL	ADDR	SIZEOF	POS
LENGTH			

7.11.1.1 Arithmetic Functions

7.11.1.2 Predicates

7.11.1.3 Transfer Functions

WRD(x) : The value x (a variable or expression) is treated as the WORD (unsigned integer) value of x. Integer literal constants are not of type WORD. Therefore, if W is of type word, W:=3 is illegal, and you must use W := WRD(3).

7.11.1.4 Further Standard Functions

File handling: (F is a file variable. See files in appendix.)

PUT(F) GET(F) RESET(F) REWRITE(F) PAGE(F) EOF(F) EOLN(F)

Dynamic Allocation: (Tn is a variant record selector, P is a pointer)

NEW(P) NEW(P,T1,T2,...,Tn) DISPOSE(P) DISPOSE(P,T1,T2,...,Tn)

Data Transfer Procedures: (See page 106 of Jensen and Wirth for a more complete description.)

PACK(A,I,Z) UNPACK(Z,A,I)

Arithmetic functions:

ABS(X) OR ABS(I) - special returns the type of its argument
SQR(X) OR SQR(I) - if passed integer returns integer, etc.

Transfer functions: (SC is a non-real short scalar)

Implemented at compile-time and generate no code:

ODD(SC) : BOOLEAN ORD(SC) : INTEGER CHR(SC) : CHAR WRD(SC) : WORD

Implemented at run-time and do generate code:

SUCC(<any scalar type except real>) PRED(<any scalar type except real>)

7.12 INPUT AND OUTPUT

ATARI Pascal supports all standard Pascal I/O facilities.

7.12.1 THE PROCEDURE READ

Reading into subranges is implemented but no range checking is performed, even with range checking turned on.

7.12.2 THE PROCEDURE READLN

```
<readcall>        ::= Read or readln> {( {<filevar> ,} {<varlist>} )}

<read or readln>::= READ | READLN

<filevar>         ::= <variable>

<varlist>         ::= <variable> {,<variable>}
```

7.12.3 THE PROCEDURE WRITE

7.12.4 THE PROCEDURE WRITELN

```
<writecall>          ::=<write or writeln> {( {<filevar> ,} {exprlist})}

<write or writeln> ::= WRITE | WRITELN

<exprlist>           ::= <wexpr> {,<wexpr>}

<wexpr>              ::= <expression> {:<width exp> {:<dec expr>}}

<width expr>         ::= <expression>

<dec expr>           ::= <expression>
```

To write integers with a base other than ten use a negative decimal place field specifier.

For example:

```
WRITE(I:15:-16)
(* this writes I in HEX*)
```

You may not use functions that perform input or output as a parameter to a WRITE or WRITELN statement. These include access routines such as GNB. The file pointers become modified by the reading routines, causing the output to be done to the input file.

7.12.5 ADDITIONAL PROCEDURES

See Section 7.10.1.1

WORD input and output is not performed with the standard READ and WRITE procedures. Two new procedures are READHEX and WRITEHEX. These

new procedures allow Hex I/O on variables of any one-, two-, or four-byte type such as integer, char, byte subrange, enumerated, word, and long integer. See the section in Chapter 3.4 on ATARI Pascal extensions.

7.13 PROGRAMS

```
<program>           ::= <program heading> <block> .   |
                        <module heading>
                            <label declaration part>
                            <constant definition part>
                            <type definition part>
                            <variable declaration part>
                            <procfunc declaration part>
                            MODEND .

<program heading>   ::= PROGRAM <identifier> ( <prog parms> ) ;

<module heading>    ::= MODULE <identifier> ;

<prog parms>        ::= <identifier> {,<identifier>}
```

The above is identical to the standard with the addition of modules. Refer to Chapter 3.

APPENDIX A: LANGUAGE SYNTAX DESCRIPTION

```
<letter> ::= A | B | C | D | E | F | G | H | I | J |
             K | L | M | N | O | P | Q | R | S | T |
             U | V | W | X | Y | Z | a | b | c | d |
             e | f | g | h | i | j | k | l | m | n |
             o | p | q | r | s | t | u | v | w | x |
             y | z | @ |


<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
             A | B | C | D | E | F   (only allowed in HEX numbers)


<special symbol> ::= {reserved words are listed in appendix B}
             +  | -  | * | / | = | <> | < | > |
             <= | >= | ( | ) | [ |  ] | ^ |
             := | .  | , | ; | : | '  |

            {the following are additional or substitutions:}
            ( . | .) | \ | ? | ! | |  | $ | & |

            (. is a synonym for [
            .) is a synonym for ]
            \, and ? are synonyms
            !, and | are synonyms
            &

<identifier>        ::= <letter> {<letter or digit or unscore>}

<letter or digit>   ::= <letter> | <digit> | _

<digit sequence>    ::= <digit> {<digit>}

<unsigned integer>  ::= $ <digit sequence> |
                        <digit sequence>

<unsigned real>     ::= <unsigned integer> . <digit sequence>    |
                        <unsigned integer> . <digit sequence>
                        E <scale factor>                         |
                        <unsigned integer> E <scale factor>

<unsigned number>   ::= <unsigned integer> | <unsigned real>

<scale factor>      ::= <unsigned integer> | <sign><unsigned integer>

<sign>              ::= + | -

<string>            ::= ' <character> {<character>}' | ''

<constant identifier> ::= <identifier>

<constant>              ::= <unsigned number>           |
                            <sign><unsigned number>     |
```

```
                          <constant identifier>       |
                          <sign><constant identifier> |
                          <string>

<constant definition> ::= <identifier> = <constant>

<type>            ::= <simple type>     |
                      <structured type> |
                      <pointer type>

<type definition> ::= <identifier> = <type>

<simple type>     ::= <scalar type>     |
                      <subrange type>   |
                      <type identifier>

<type identifier> ::= <identifier>

<scalar type> ::= ( <identifier> {, <identifier>} )

<subrange type> ::= <constant> .. <constant>

<structured type>           ::= <unpacked structured type>        |
                                PACKED <unpacked structured type>

<unpacked structured type> ::= <array type>  |
                               <record type> |
                               <set type>    |
                               <file type>

<array type>     ::= <normal array> |
                     <string array>

<string array>   ::= STRING <max length>

<max length>     ::= [ <inconst> ] |
                     <empty>

<inconst>        ::= <unsigned integer> |
                     <int const id>

<int const id>   ::= <identifier>

<normal array>   ::= ARRAY [ <index type> {,<index type>}] OF
                     <component type>

<index type>     ::= <simple type>

<component type> ::= <type>

<record type>    ::= RECORD <field list> END

<field list>     ::= <fixed part>                    |
                     <fixed part> ; <variant part> |
```

```
                        <variant part>

<fixed part>      ::= <record section> {;<record section>}

<record section>  ::= <field identifier> {,<field identifier>}: <type>|
                      <empty>

<variant part>    ::= CASE <tag field> <type identifier> OF
                      <variant> {;<variant>}

<variant>         ::= <case label list> : (<field list>) |
                      <empty>

<case label list>::= <case label> {,<case label>}

<case label>      ::= <constant>

<tag field>       ::= <identifier> : |
                      <empty>

<set type>        ::= SET OF <base type>

<base type>       ::= <simple type>

<file type>       ::= file {of <type>}

<variable>        ::= <var>             |
                      <external var>    |
                      <absolute var>

<external var>    ::= EXTERNAL <var>

<absolute var>    ::= ABSOLUTE [ <constant> ] <var>

<var>             ::= <entire variable>     |
                      <component variable>  |
                      <referenced variable>
```

Declaration of variable of type STRING:

```
<identifier> {,<identifier>} : STRING {[<constant>]}

<entire variable>      ::= <variable identifier>

<variable identifier> ::= <identifier>

<component variable>  ::= <indexed variable>  |
                          <field designator>  |
                          <file buffer>

<indexed variable> ::= <array variable> [<expression> {,<expression>}]

<array variable>   ::= <variable>
```

```
<field designator> ::= <record variable> . <field identifier>

<record variable>  ::= <variable>

<field identifier> ::= <identifier>

<file buffer>      ::= <file variable>^

<file variable>    ::= <variable>

<referenced variable>::= <pointer variable>^

<pointer variable>   ::= <variable>

<unsigned constant>  ::= <unsigned number>          |
                         <string>                   |
                         NIL                        |
                         <constant identifier>

<factor>             ::= <variable>                 |
                         <unsigned constant>        |
                         <function designator>      |
                         ( <expression> )           |
                         <logical not operator> <factor>
                         <set>

<set>                ::= [ <element list> ]

<element list>       ::= <element> {,<element>}       |
                         <empty>

<element>            ::= <expression>                 |
                         <expression> .. <expression>


<term>               ::= <factor> <multiplying operator> <factor>

<simple expression>  ::= <term>                                          |
                         <simple expression> <adding operator> <term>|
                         <adding operator> <term>

<expression>         ::= <simple expression>                             |
                         <simple expression> <relational operator>
                             <simple expression>

<logical not operator> ::= NOT | \ | ?

        \ and ? is are NOT operators for non-Booleans.

<multiplying operator> ::= * | / | DIV | MOD | AND | &

       & is an AND operator on non-Booleans.

<adding operator> ::= + | - | OR | | | !
```

```
        ! (synonym !) is an OR operator on non-Booleans.

<relational operators> ::= = ! <> ! < ! <= ! > ! >= ! IN

<function designator> ::= <function identifier>                    !
                          <function identifier> ( <parm> {,<parm> )

<function identifier> ::= <identifier>

<statement>           ::= <label> : <unlabelled statement>    !
                          <unlabelled statement>

<unlabelled statement>::= <simple statement>                       !
                          <structured statement>

<label>               ::= <unsigned integer>

<simple statement>    ::= <assignment statement>  !
                          <procedure statement>   !
                          <goto statement>        !
                          <empty statement>

<empty statement>     ::= <empty>

<assignment statement>::= <variable> := <expression>               !
                          <function identifier> := <expression>

<procedure statement> ::= <procedure identifier> ( <parm> {,<parm>} ) !

                          <procedure identifier>

<procedure identifier>::= <identifier>

<parm>                ::= <procedure identifier>  !
                          <function identifier>   !
                          <expression>            !
                          <variable>

<goto statement>      ::= goto <label>

<structured statement>::= <repetitive statement>    !
                          <conditional statement>   !
                          <compound statement>      !
                          <with statement>

<compound statement>  ::= BEGIN <statement> {,<statement>} END

<conditional statement> ::= <case statement>  !
                            <if statement>

<if statement> ::= IF <expression> THEN <statement> ELSE <statement> !
                   IF <expression> THEN <statement>
```

```
<case statement> ::= CASE <expression> OF
                     <case list> {;<case list>}
                     {ELSE <statement>}
                     END

<case list>      ::= <label list> : <statement> |
                     <empty>

<label list>     ::= <case label> {,<case label>}

<repetitive statement>  ::= <repeat statement>  |
                            <while statement>   |
                            <for statement>


<while statement> ::= WHILE <expression> DO <statement>

<repeat statement> ::= REPEAT <statement> {;<statement>} UNTIL
                       <expression>

<for statement> ::= FOR <ctrlvar> := <for list> DO <statement>

<for list>       ::= <expression> DOWNTO <expression>  |
                     <expression>   TO   <expression>

<ctrlvar>        ::= <variable>

<with statement> ::= WITH <record variable list> DO <statement>

<record variable list> ::= <record variable> {,<record variable>}

procedure declaration> ::= EXTERNAL <procedure heading>  |
                           <procedure heading> <block>

<block>                ::= <label declaration part>
                           <constant definition part>
                           <type definition part>
                           <variable declaration part>
                           <procfunc declaration part>
                           <statement part>

<procedure heading>  ::= PROCEDURE <identifier> <parmlist>   |
                         PROCEDURE <identifier> ;            |
                         PROCEDURE INTERRUPT [ <constant> ] ;

<parmlist>           ::= ( <fparm> {;<fparm>} )

<fparm>              ::= <procedure heading> |
                         <function heading>  |
                         VAR <parm group>    |
                         <parm group>

<parm group>         ::= <identifier> {,<identifier>} :
                           <type identifier>            |
```

```
                          <identifier> {,<identifier>} :
                            <conformant array>

<conformant array>    ::= ARRAY [ <indxtyp> {;<indxtyp} ] OF
                          <conarray2>

<conarray2>           ::= <type identifier>      |
                          <conformant array>

<indxtyp>             ::= <identifier> .. <identifier> : <ordtypid>

<ordtypid>            ::= <scalar type identifier>     |
                          <subrange type identifier>

<label declaration part>    ::= <empty>  |
                                LABEL <label> {,<label>} ;

<constant definition part> ::= <empty>  |
                                CONST
                                  <constant definition>
                                  {;<constant definition} ;

<type definition part>      ::= <empty>  |
                                TYPE
                                  <type definition.
                                  {;<type definition>} ;

<variable declaration part>::= <empty>  |
                                VAR
                                  <variable declaration>

<indxtyp>                   ::= <identifier>..<identifier> : <ordtypid>

<ordtypid>                  ::= <scalar type identifier>  |
                                <subrange type identifier>

<label declaration part>    ::= <empty>  |
                                LABEL <label> {,<label>} ;

<constant definition part> ::= <empty>  |
                                CONST
                                  <constant definition>
                                  {;<constant definition>};

<type definition part>      ::= <empty>  |
                                TYPE
                                  <type definition>
                                  {;<type definition>} ;

<variable declaration part>::= <empty>  |
                                VAR
                                  <variable declaration>
                                  {;<variable declaration>} ;
```

```
<procfunc part>              ::= {<proc or func> ; }

<proc or func>               ::= <procedure declaration>  |
                                 <function declaration>

<statement part>             ::= <compound statement>

<function decl>              ::= EXTERNAL <function heading> |
                                 <function heading> <block>

<function heading>::= FUNCTION <identifier><parmlist>:<result type>; |
                      FUNCTION <identifier> : <result type> ;

<result type>      ::= <type identifier>

<readcall>         ::= <read or readln> {( {<filevar> ,} {<varlist>} )}

<read or readln>   ::= READ | READLN

<filevar>          ::= <variable>

<varlist>          ::= <variable> {,<variable>}

<writecall>        ::=<write or writeln> {( {<filevar> ,} {exprlist} )}

<write or writeln> ::= WRITE | WRITELN

<exprlist>          ::= <wexpr> {,<wexpr}

<wexpr>             ::= <expression> {:<width expr> {:<dec expr>}}

<width expr>        ::= <expression>

<dec expr>          ::= <expression>

<program>           ::= <program heading> <block> .   |
                        <module heading>
                           <label declaration part>
                           <constant definition part>
                           <type definition part>
                           <variable declaration part>
                           <procfunc declaration part>
                           MODEND .

<program heading>   ::= PROGRAM <identifier> ( <prog parm> ) ;

<module heading>    ::= MODULE <identifier> ;

<prog parms>        ::= <identifier> {,<identifier>}
```

APPENDIX B: RESERVED WORDS

The following are the reserved words in ATARI Pascal

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

ATARI Pascal also has extended reserved words:

ABSOLUTE EXTERNAL PREDEFINED

APPENDIX C: ERROR MESSAGES

Recursion stack overflow: evalution stack collision with symbol table; correct by reducing symbol table size, simplifying expressions.

1: Error is simple type
Self-explanatory.

2: Identifier expected
Self-explanatory.

3: 'PROGRAM' expected
Self-explanatory.

4: ')' expected
Self-explanatory.

5: ':' expected
Possibly an = used in a VAR declaration.

6: Illegal symbol (possibly missing ';' on line above)
Symbol encountered is not allowed in the syntax at this point.

7: Error in parameter list
Syntactic error in parameter list declaration.

8: 'OF' expected
Self-explanatory.

9: '(' expected
Self-explanatory.

10: Error in type
Syntactic error in TYPE declaration.

11: '[' expected
Self-explanatory.

12: ']' expected
Self-explanatory.

13: 'END' expected
All procedures, functions, and blocks of statements must have an 'END'. Check for mismatched BEGIN/ENDs.

14: ';' expected (possibly on line above)
Statement separator required here.

15: Integer expected
Self-explanatory.

16: '=' expected
Possibly a : used in a TYPE or CONST declaration.

17: 'BEGIN' expected
Self-explanatory.

18: Error in declaration part
Typically an illegal backward reference to a type in a pointer declaration.

19: Error in <field-list>
Syntactic error in a record declaration

20: '.' expected
Self-explanatory.

21: '*' expected
Self-explanatory.

50: Error in constant
Syntactic error in a literal constant

51: ':=' expected
Self-explanatory.

52: 'THEN' expected
Self-explanatory.

53: 'UNTIL' expected
Can result from mismatched begin/end sequences.

54: 'DO' expected
Syntactic error.

55: 'TO' or 'DOWNTO' expected in FOR statement
Self-explanatory.

56: 'IF' expected
Self-explanatory.

57: 'FILE' expected
Probably an error in a TYPE declaration.

58: Error in <factor> (bad expression)
Syntactic error in expression at factor level.

59: Error in variable
Syntactic error in expression at variable level.

99: MODEND expected
Each MODULE must end with MODEND.

101: Identifier declared twice
Name already in visible symbol table.

102: Low bound exceeds high bound
For subrange the lower bound must be <= high bound.

103: Identifier is not of the appropriate class
A variable name used as a type, or a type used as a variable, etc. can cause this error.

104: Undeclared identifier
The specified identifier is not in the visible symbol table.

105: Sign not allowed
Signs are not allowed on non-integer/non-real constants.

106: Number expected
This error can often come from making the compiler totally confused in an expression as it checks for numbers after all other possibilities have been exhausted.

107: Incompatible subrange types
(e.g. 'A'..'Z' is not compatible with 0..9).

108: File not allowed here
File comparison and assignment is not allowed.

109: Type must not be real
Self-explanatory.

110: <tagfield> type must be scalar or subrange
Self-explanatory.

111: Incompatible with <tagfield> part
Selector in a CASE-variant record is not compatible with the <tagfield> type.

112: Index type must not be real
An array may not be declared with real dimensions

113: Index type must be a scalar or a subrange
Self-explanatory.

114: Base type must not be real
Base type of a set may be scalar or subrange.

115: Base type must be scalar or a subrange
Self-explanatory.

116: Error in type of standard procedure parameter
Self-explanatory.

117: Unsatisified forward reference
A forwardly declared pointer was never defined.

118: Forward reference type identifier in variable declaration
You attempted to declare a variable as a pointer to a type not yet declared.

119: Re-specified parameters not OK for a forward declared procedure
Self-explanatory.

120: Function result type must be scalar, subrange or pointer
A function has been declared with a string or other non-scalar type as its value. This is not allowed.

121: File value parameter not allowed
Files must be passed as VAR parameters.

122: A forward declared function's result type can't be re-specified
Self-explanatory.

123: Missing result type in function declaration
Self-explanatory.

125: Error in type of standard procedure parameter
This error is often caused by not having the parameters in the proper order for built-in procedures or by attempting to read/write pointers, enumerated types, etc.

126: Number of parameters does not agree with declaration
Self-explanatory.

127: Illegal parameter substitution
Type of parameter does not exactly match the corresponding formal parameter.

128: Result type does not agree with declaration
When assigning types to a function result, the types must be compatible.

129: Type conflict of operands
Self-explanatory.

130: Expression is not of set type
Self-explanatory.

131: Tests on equality allowed only
Occurs when comparing set for other than equality.

133: File comparison not allowed
File control blocks may not be compared because they contain multiple fields unavailable to the user.

134: Illegal type or operand(s)
The operands do not match those required for this operator.

135: Type of operand must be Boolean
The operands to AND, OR and NOT must be BOOLEAN.

136: Set element type must be scalar or subrange
Self-explanatory.

137: Set element types must be compatiable
Self-explanatory.

138: Type of variable is not array
A subscript has been specified on a non-array variable.

139: Index type is not compatible with the declaration
Occurs when indexing into an array with the wrong type of indexing expression.

140: Type of variable is not record
Attempting to access a non-record data structure with the 'dot' form or the 'with' statement.

141: Type of variable must be file or pointer
Occurs when an up arrow follows a variable which is not of type pointer or file.

142: Illegal parameter solution
Self-explanatory.

143: Illegal type of loop control variable
Loop control variables may be only local non-real scalars.

144: Illegal type of expression
The expression used as a selecting expression in a CASE statement must be a non-real scalar.

145: Type conflict
Case selector is not the same type as the selecting expression.

146: Assignment of files not allowed
Self-explanatory.

147: Label type incompatible with selecting expression
Case selector is not the same type as the selecting expression.

148: Subrange bounds must be scalar
Self-explanatory.

149: Index type must be integer
Self-explanatory.

150: Assignment to standard function is not allowed
Self-explanatory.

151: Assignment to formal function is not allowed
Self-explanatory.

152: No such field in this record
Self-explanatory.

153: Type error in read
Self-explanatory.

154: Actual parameter must be a variable
Occurs when attempting to pass an expression as a VAR parameter.

155: Control variable cannot be formal or non-local
The control variable in a FOR loop must be LOCAL.

156: Multidefined case label
Self-explanatory.

157: Too many cases in case statement
Occurs when jump table generated for case overflows its bounds.

158: No such variant in this record
Self-explanatory.

159: Real or string tagfields not allowed
Self-explanatory.

160: Previous declaration was not forward

161: Again forward declared

162: Parameter size must be constant

163: Missing variant in declaration
Occurs when using NEW/DISPOSE and a variant does not exist.

164: Substitution of standard procedure/function not allowed

165: Multidefined label
Label more than one statement with same label.

166: Multideclared label
Declare same label more than once.

167: Undeclared label
Label on statement has not been declared.

168: Undefined Label
A declared label was not used to label a statement.

169: Error in base set

170: Value parameter expected

171: Standard file was re-declared

172. Undeclared external file

174: Pascal function or procedure expected
Self-explanatory.

183: External declaration not allowed at this nesting level
Self-explanatory.

187: Attempt to open library unsuccessful
Self-explanatory.

191: No private files
Files may not be declared other than in the GLOBAL variable section of a program or module because they must be statically allocated.

193: Not enough room for this operation
Self-explanatory.

194: Comment must appear at top of program

201: Error in real number - digit expected
Self-explanatory.

202: String constant must not exceed source line

203: Integer constant exceeds range
Range on integer constants are -32768...32767

250: Too many scopes of nested identifiers
There is a limit of 15 nesting levels at compile-time.
This includes WITH and procedure nesting.

251: Too many nested procedures or functions
There is a limit of 15 nesting levels at execution time.

253: Procedure too long
A procedure has generated code that has overflowed the internal procedure buffer. Reduce the size of the procedure and try again.

259: Expression too complicated
Your expression is too complicated (i.e. too many recursive calls needed to compile it). Reduce the complication using temporary variable.

397: Too many FOR or WITH statements in a procedure
Only 16 FOR and/or WITH statements are allowed in a single procedure (in recursive mode only)

400: Illegal character in text
A non-Pascal special character was found outside a quoted string.

401: Unexpected end of input
"End." encountered before returning to outer level.

402: Error in writing code file, not enough room
Self-explanatory.

403: Error in reading include file
Self-explanatory.

404: Error in writing list file, not enough room
Self-explanatory.

405: Call not allowed in separate procedure
Self-explanatory.

406: Include file not legal
Self-explanatory.

407: Symbol Table Overflow

497: Error in closing code file.
An error occured when the .ERL file was closed.
Make more room on the destination diskette and try again.

500: A non-standard feature has been used when the T+ or W+ toggles are enabled. This is a non-fatal information-only error.

APPENDIX D: ATARI PASCAL FILE I/O

The sections in this appendix describe ATARI Pascal files and how to use them. Since working from an example will be the most effective way of describing these concepts, program examples have been included for each area of file handling.

- The first section defines the terms used, such as "file," "window variable," and "TEXT."
- The second section shows how to use all the file operation procedures with examples. These include ASSIGN, RESET, REWRITE, sequential file access procedures, CLOSE, etc.
- The third section defines Pascal TEXT files. Sample programs demonstrate the use of built-in Boolean functions EOLN and EOLF, READLN, WRITELN, formatted I/O, and writing to the printer.
- The fourth section presents some less frequently used file operations.

1. DEFINITIONS

The terms and definitions included here are arranged to logically discuss the concepts of files as you read through.

FILE

A file is data arranged in logical, equal-sized elements very much like an open-ended array accessed via a pointer. The size and arrangement of the data is controlled by your program. A file is generally stored on a secondary storage medium. For the purpose of this documentation, secondary storage is assumed to be a diskette. You may write data to a file or read data from a file using the file operation procedures provided with ATARI Pascal. This data in the file may be accessed sequentially (record 1 accessed before record 2, record 2 is accessed before record 3, etc), or directly.

FILENAME

The filename is the name of the file on diskette. It is the name displayed in the directory listing of the storage medium. In ATARI Pascal the filename is represented in a program by a STRING (a dynamic sequence of ASCII characters). For example, "D2:TEST.PAS" is the filename in literal string format for the file located on drive "D2" with the name of "TEST" and the extension of ".PAS".

TYPE

The type of file defines the size and format of the individual file elements, the smallest accessible units of a file. For example, a file of type INTEGER (2 8-bytes) may be visualized as:

```
+--------+--------+--------+--------+--------+--------+
|00001000|00000000|00100001|00000001|00000001|00000000|
+--------+--------+--------+--------+--------+--------+
|  record 0       |   record 1      |    record 2     |
```

This file contains the integers 8, 33, and 1 (stored inverted in this sample). The smallest retrievable element is two bytes. See the explanations of untyped files or byte files it you want to treat this file differently than a file of integers. Files may be of the standard Pascal scalar types: BOOLEAN, INTEGER, CHAR, or REAL. Files may also be of the structured types STRING, arrays, and records. The predefined type TEXT is used for ASCII files. Text files are similiar to FILE of CHAR except that they are subdivided into lines, and numbers written to them are converted to ASCII (and may be formatted), and numbers read from them are converted to binary. A line is a sequence of characters terminated by an end-of-line character, which is usually a carriage return/line feed. Also, unlike FILE of CHAR, TEXT files will accept PACKED ARRAY[1..N] OF CHAR or ARRAY[1..N] OF CHAR (writing an UNPACKED ARRAY is not ISO standard), and STRINGS as data. A Boolean value is converted to the ASCII sequence "TRUE" or "FALSE" on write but the reverse is not true. For further explanations on typed and text files, see the operations section.

A non-ISO standard concept regarding files is the UNTYPED file. This concept is used for fast block input and output (entire sectors are read or written) regardless of the kind of data contained in the file.

FILE INFORMATION BLOCK (FIB)

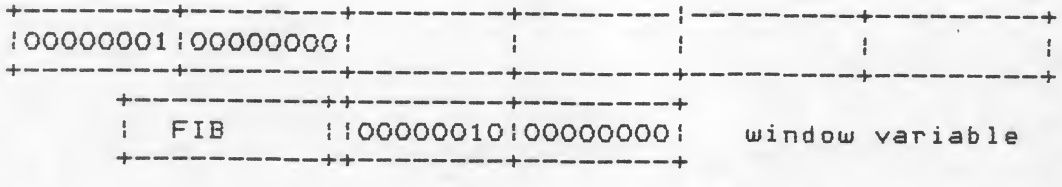
The FIB contains information necessary for the run-time routines to perform file operations on a disk file. The filename, the type of the file, the access type (read or write), end-of-file and end-of-line flags, and a diskette buffer (the size of one diskette sector) are among the kinds of information kept in the FIB.

WINDOW VARIABLE OR WINDOW POINTER

The window variable is a buffer the size of a file element and is located just past the FIB in ATARI Pascal. A way to think of it is that it moves along the file and acts as a 'window' to the element of the file to be read or written. For this reason, it is considered a pointer to the file element being accessed. It is denoted as "F^" where "F" is the name of a file variable. To read from a file, the element which is accessible is moved to the window variable. To write to a file, the data must be transferred from the window variable to the file.

FILE VARIABLE

The file variable consists of a FIB and a window variable. It is the actual data item allocated by the compiler and referenced in a Pascal program. An example will clarify what a file variable is, as well as what the FIB and window variable are. The statement, "VAR F : FILE OF INTEGER;" causes the compiler to create a file variable F with its own FIB in the data area and its own window variable (2 bytes) to hold a 16-bit integer. The window variable is denoted by F^. Suppose "I" is an integer in the same program and has the value 2. Suppose also that the file already contains the value 1 in the first element as below:



To write the contents of I to the file, the window variable must contain 2 (F^ := I puts the contents of I into the window variable) and be "positioned" over the second element of the file. Given the command PUT(F) described in the operations section, the number 2 is written to the file.

2. FUNDAMENTAL FILE OPERATIONS

Sample programs and explanations demonstrate the use of file operation procedures in ATARI Pascal. You will see how to open, create, read, write, delete, and close files. Demonstrated also are the use of typed and text files; the file status functions IORESULT, EOF, and EOLN; and how to assign to a window variable.

Figure D-1 lists a program named WRITE_READ_FILE_DEMO that creates a typed file on diskette, writes data to the file, closes the file, then re-opens the file to read the data back. The procedures used to perform these are ASSIGN, REWRITE, RESET, IORESULT, PUT, GET, and CLOSE. WRITE is used to display the results on the terminal. The output work is done in WRITEFILE and the input work is done in READFILE. Creating, opening, and closing the file is done in the main body of the program.

The WRITELN statements on lines 37, 43, 46, and 49 write the string passed to them to the default OUTPUT file (the console). This procedure and READLN are discussed later in this section under TEXT files.

First note the form of the declaration of OUTFILE. It is declared to be of type CHFILE, which is defined as a FILE OF CHAR in the TYPE declaration section (lines 3 and 4). This is done because the file is passed as a parameter to the WRITEFILE and READFILE routines and a parameter list cannot declare a new type. For example, the following parameter declaraction is illegal in Pascal because only type identifiers are allowed in a parameter list:

```
PROCEDURE WRITEFILE( VAR F : FILE OF CHAR);
```

```
1       0       PROGRAM WRITE_READ_FILE_DEMO;
2       0
3       0       TYPE
4       1         CHFILE = FILE OF CHAR;
5       1       VAR
6       1         OUTFILE : CHFILE;
7       1         RESULT  : INTEGER;
8       1         FILENAME: STRING[16];
9       1
10      1         PROCEDURE WRITEFILE( VAR F : CHFILE);
11      1         VAR CH: CHAR;
12      2         BEGIN
13      2           FOR CH:= '0' TO '9' DO
14      2             BEGIN
15      3               F^ := CH;(*CHR(I + ORD('0'));*)
16      3               PUT(F)
17      3             END;
18      2         END;
19      1
20      1         PROCEDURE READFILE( VAR F : CHFILE);
21      1         VAR I : INTEGER;
22      2             CH : CHAR;
```

```
23      2         BEGIN
24      2           FOR I : 0 TO 9 DO
25      2             BEGIN
26      3               CH := F^;
27      3               GET(F);
28      3               WRITELN(CH);
29      3             END;
30      2         END;
31      1
32      1       BEGIN
33      1         FILENAME := 'TEST.DAT';
34      1         ASSIGN(OUTFILE,FILENAME);
35      1         REWRITE(OUTFILE);
36      1         IF IORESULT = <> 0 THEN
37      1           WRITELN('Error creating ',FILENAME)
38      1         ELSE
39      1           BEGIN
40      2             WRITEFILE(OUTFILE);
41      2             CLOSE(OUTFILE,RESULT);
42      2             IF RESULT = <> 0 THEN
43      2               WRITELN('Error closing ',FILENAME)
44      2             ELSE
45      2               BEGIN
46      3                 WRITELN('Successful close of ',FILENAME);
47      3                 RESET(OUTFILE);
48      3                 IF IORESULT = <> 0 THEN
49      3                   WRITELN('Cannot open ',FILENAME)
50      3                 ELSE
51      3                   READFILE(OUTFILE)
52      3               END;
53      2             END;
54      1         END.
```

Figure D-1: File Input and Output.

PROCEDURE ASSIGN(VAR F: FILE VARIABLE;STR : STRING);

Purpose: Associate the file variable F with an external file on diskette named in STR.

ASSIGN is the first file operation to be executed in line 34. This procedure associates a file variable (OUTFILE) with an external file on a diskette given in FILENAME (in this case it is "TEST.DAT"). The string passed to ASSIGN is placed into the FIB and the name is interpreted. After executing the ASSIGN procedure, the file variable passed to the ASSIGN procedure is always associated with the diskette file named in the name parameter until, or unless, another ASSIGN is done to the file variable.

PROCEDURE REWRITE(VAR F : FILE VARIABLE);

Purpose: Create a file on diskette using the name in the FIB (either filled in by the ASSIGN statement previously or null (if null, a temporary file is created.).

The REWRITE procedure is called in line 35 of Figure D-1. Executing this procedure causes the creation of a file with the name contained in the FIB of F. Any existing files by that name are deleted so NEVER use REWRITE on a file which contains usable data. In this example, the file on diskette will be named "TEST.DAT" and is located on the default diskette (because no other diskette was specified in the file name string passed to ASSIGN).

If no previous ASSIGN had been performed, the name field of the FIB is empty and a temporary file is created with the name "PASTMP00.$$$." Temporary files are generally used for scratch pad memory and data which is not needed after execution of the program. The digits at the last two positions in the name are used to give each temporary file a unique name.

The EOF function and the EOLN function return true because OUTFILE is an output file. OUTFILE is open only for writing sequentially and is ready to receive data into its first element. If the operation is not successful, the IORESULT function returns a non zero in this case (see line 36).

FUNCTION IORESULT : INTEGER;

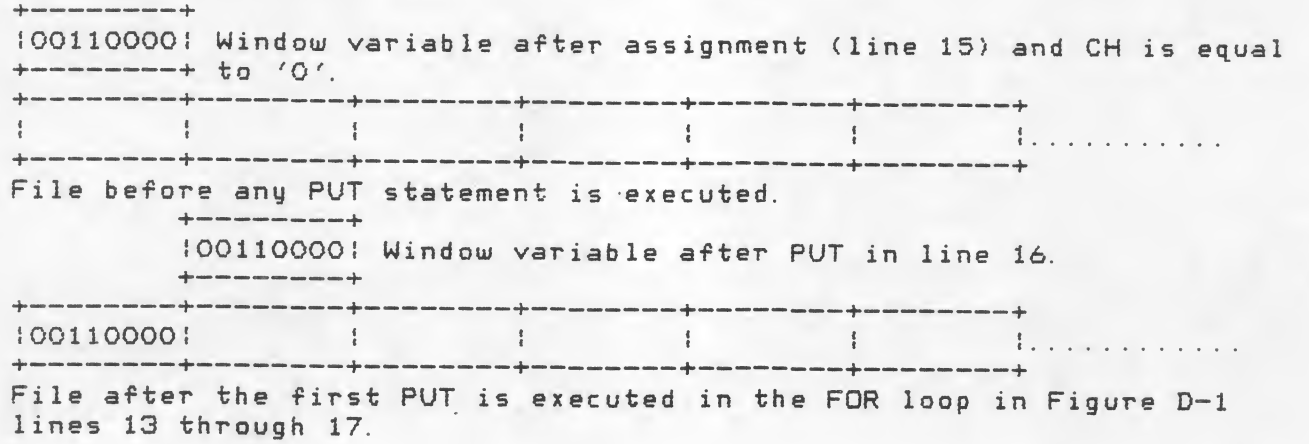
Purpose : Return the integer value indicating status of file operation.

The value of this function is set after any input or output operation and may be checked at any time. Note in Figure D-1 it is called after each file operation in lines 36, 42, and 46. It is used here to stop the program if a file operation did not work as planned. Note that you cannot "WRITE(IORESULT)" because IORESULT is reset to 0 after each I/O operation. The meaning of the values returned by IORESULT is presented in Chapter 3.

```
PROCEDURE PUT(VAR F : FILE VARIABLE);
```

Purpose : Transfer the contents of the window variable associated with F to the next available record in the file.

Procedure WRITEFILE, beginning on line 9 of Figure D-1, writes the characters "0" to "9" to the TEST.DAT file. The PUT procedure causes the data to be written to the file. Always before executing a PUT, an assignment is made to the window variable as in line 15. Following is a diagram of what is occurring:




```
PROCEDURE WRITE;
PRODEDURE WRITE(expression,...,expression);
PROCEDURE WRITE(VAR F:FILE VARIABLE,expression,...,expression);
```

Purpose : Shorthand for 'F^ := data; PUT(F);' also performs conversions to ASCII on numbers when F is a TEXT file.

Expression includes contents of variables, strings, array elements, constants, and expressions. When a file variable is not specified, the default OUTPUT file is assumed. The WRITE procedure does the same operations on the file as lines 15 and 16. It executes an assignment followed by a PUT and is merely a shorthand version. GET and PUT are provided because the ISO standard requires them and in some versions of Pascal, such as UCSD Pascal, WRITE can only be used on TEXT files.

```
PROCEDURE CLOSE(VAR F : FILE VARIABLE; RESULT : INTEGER);
```

Purpose : Flush the buffer in the FIB associated with F so all data is written to the diskette.

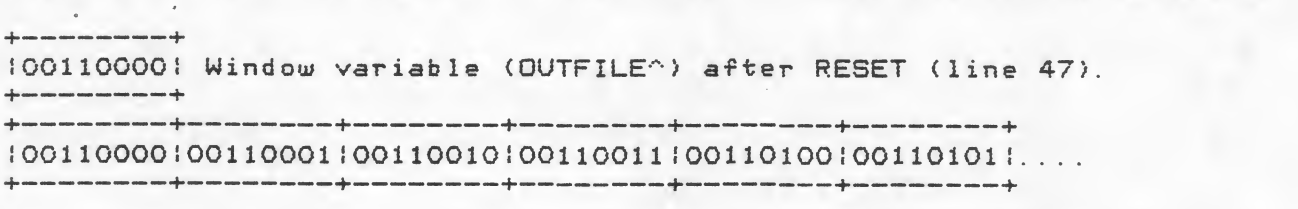
The next statement to be executed after returning from WRITEFILE is line 41, where the file is closed. CLOSE must be executed to assure that the data written to "TEST.DAT" is actually saved on the diskette. Up until this point the data is written to the buffer in memory and now must be saved by flushing the buffer. RESULT is the value returned

by the Operating System indicating whether the close is successful. It is included as a parameter to maintain compatibility with previous versions of the compiler. In this program a value of non zero means an error closing the file, and any other value indicates success.

PROCEDURE RESET(VAR F : FILE VARIABLE);

Purpose : Open an existing file for reading. The window variable is moved to the beginning of the file.

After checking RESULT, the procedure RESET is called in line 47. RESET opens an existing file for reading and resets the window variable to the beginning of the file. F^ is assigned the first element of F. If F is already open, RESET calls CLOSE. EOF and EOLN return FALSE. If a RESET is done on a file that does not exist, IORESULT contains a non zero. All other values of IORESULT indicate success. In the sample program, OUTFILE is opened by the RESET procedure so that it may be read. Below is a diagram of the file and window variable after the RESET is executed in line 47. Note that with non-computer console typed files, such as OUTFILE, the procedure RESET does an initial GET, which moves the first element of the file (in this case the ASCII value for the number 0) into the window variable.



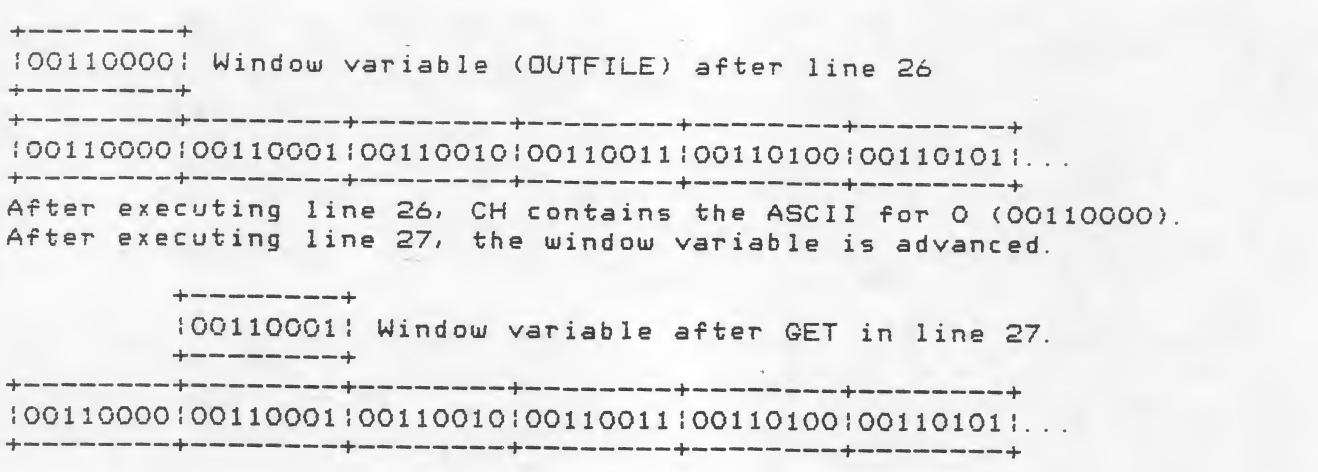
The initial GET is not performed on console files or untyped files. You would always have to type a character before your program could execute, because the GET procedure is waiting for a character.

PROCEDURE GET(VAR F : FILE VARIABLE);

Purpose : Transfer the currently accessible record to the window variable and advance the window variable.

After checking that the RESET procedure is successful, procedure READFILE is called in line 51. This procedure reads each element of the file passed to it (in this case the element is a character) and writes that element to the screen. READFILE begins on line 20. The work is done in the FOR loop of lines 24 through 29.

The GET procedure advances the window variable by one element and moves the contents of the file pointed to into the window variable. If no next element exists, EOF becomes TRUE. See Section 3 on TEXT files for more details on GET and TEXT files. The diagram below describes what is happening within the FOR loop on lines 26 and 27 the first time through the loop.



Line 28 writes the contents of CH to the default output file which is the computer console. Procedure READFILE displays the characters "0" through "9" in a column on the computer console. Calling CLOSE after a RESET is not necessary in the sequential case, because the file already exists on the diskette and has not been altered in any way. If OUTFILE is accessed randomly, a CLOSE might be necessary.

```
PROCEDURE READ(data, data,...,data);
PROCEDURE READ(VAR F : FILE VARIABLE , data, data, ..., data);
```

Purpose: When used with non-computer console files execute "data := F^; GET(F);" for each data item read. When used with computer console files, execute "GET(F); data :=F^;". If F is not specified the default INPUT file is used. See the section on TEXT files for information on conversions.

The READ procedure is the same as an assignment and a call to GET. If READ is used rather than GET in the current example, the FOR loop body would look like this:

```
FOR I := 0 TO 9 DO
  BEGIN
    READ(CH);
    WRITELN(CH)
  END;
```

Reading past end-of-file on computer console input results in a system crash.

3. TEXT FILES

DEFINITION

A TEXT file is a file of ASCII characters subdivided into lines. A line is a sequence of characters terminated by a nonprintable end-of-line indicator, usually a carriage return and a line feed character. It is similar to a file of CHAR except that automatic conversion of numbers is performed when they are read from and written to the file. Also, variables of type STRING may be read from a text file and BOOLEANs, STRINGs, and PACKED ARRAYs may be written to text files. Access to a TEXT file is via GET and PUT for character I/O (which do not do conversions), READ and WRITE, which have been defined earlier in this section, and READLN and WRITELN, which are used in Figure D-2 and defined in this section.

The format of a TEXT file in memory is a FIB and a 1-byte window variable. On diskette, the file looks like the sample below in which a carriage return is represented by ">", linefeed by "/" and end of file by "#."

```
+-------------------------------------------------------------+
This is a line>/This is the next line>/This is the last line>/#
+-------------------------------------------------------------+
```

```
FUNCTION EOLN : BOOLEAN;
FUNCTION EOLN(VAR F : TEXT) BOOLEAN;
```

Purpose: Indicate the state of the file be returning true only when the window variable is over the end-of-line character. When no file is specified the default INPUT file is assumed.

This function returns true on diskette text files when the last valid character on a line is read using a READ statement. Because the sequence of statements for a READ (on non-computer console files) is "CH := F^; GET(F);", the window variable is positioned over the end-of-line character immediately after the last character is read. Thus, EOLN returns TRUE on NON-COMPUTER CONSOLE TEXT files when the last character is read . Also, a BLANK character is returned instead of the end-of-line character. The above sequence is reversed on computer CONSOLE files (READ is an initial call to GET followed by an assignment from the window variable). When you use computer CONSOLE files, EOLN will return true after the carriage return / line feed is read instead of after the last character as in disk files. A blank is still returned in the character.

```
FUNCTION EOF;
FUNCTION EOF(VAR F : FILE) : BOOLEAN;
```

Purpose: Indicate the state of a file by returning true only when the window variable is over an end-of-file character. When no file is specified, the default INPUT file is assumed.

EOF is a function that returns true when the end-of-file character is read. It is similar to EOLN in that the last character read will set EOF to true on NON-COMPUTER CONSOLE files. On computer CONSOLE files EOF is true only when the end-of-file indicator is entered. Reading past end-of-file on computer console files is not supported (the system can crash). Reading past the end of the file on diskette files is not supported. A blank is returned by the window variable when EOF is true. Also, note that on non-text files, EOF may not become true at the end of the valid data because the data may not fill up the entire last sector of the file.

Figure D-2 is a program that writes data to a text file and reads it back to be displayed on the output device. The procedure WRITEDATA actually writes to the TEXT file and the procedure READDATA retrieves the information stored in the file. The program is divided into a main body and two procedures to demonstrate the usefulness of breaking up code into blocks that perform certain functions. This method makes code much easier to read and debug.

The file is declared in line 3. Note that the declaration is NOT "VAR F : FILE of TEXT". TEXT is treated as a special version of FILE of CHAR, so FILE of TEXT translates to FILE of FILE of CHAR (nonsensical).

The program begins execution on line 25 with a call to the ASSIGN procedure. Lines 25 through 29 create a TEXT file named TEXT.TST on the logged-in drive. If the file creation is successful, then the sample data is initialized in lines 31 and 32, followed by a call to the WRITEDATA routine in line 33. WRITEDATA uses the WRITELN procedure, which is only used with TEXT files.

```
PROCEDURE WRITE;
PROCEDURE WRITELN;
PROCEDURE WRITELN(expr,expr,...expr);
PROCEDURE WRITELN(F);
PROCEDURE WRITELN(F,expr,expr,...expr);
```

Purpose: Put the data into the file associated with F, ending the output with an end-of-line character. If no file is specified the expressions are written to the OUTPUT file. A WRITELN with no expressions merely outputs a carriage return / line feed. The WRITE procedure is redefined as a conversion rather than a replacement for PUT.

This procedure writes the data passed to it to the file named, placing an end-of-line character after the last item of data written. If no file is named, the file is written to the default OUTPUT file. Data may be literal and named constants, integers, reals, subranges, enumerated, Booleans, strings, and packed arrays of characters, but may not be structured types such as records. Numeric data is converted to ASCII and strings are treated as arrays of characters (the length byte is not written to the file).

Formatted Output

In Figure D-2 three lines that make up the body of WRITEDATA (9, 10, and 11) do the actual file output. Line 9 sends the contents of the variable string S followed by a carriage return / line feed to the TEXT file F. Line 10 formats the contents of I in a field of four spaces and sends this formatted output to the file F. The real number literal in line 11 is formatted into a field of nine spaces, four of which must be to the right of the decimal place. This formatted number is then written to the file F. The field format may be specified for any data type. For non-real numbers only the field width is specified, not the number of places after the decimal point. The data is right justified in the field. If a number is larger than the 6.5 significant digits can represent, the output is always expressed in exponental notation. Also, if the field width is too small to express the number it is written in exponential notation. For further information on formatting consult a Pascal textbook and experiment.

The body of the WRITEDATA procedure could have been written as follows with the same results.

```
WRITELN(F,S);
WRITELN(F,I:4, 45.6789 : 9 : 4);
```

Control returns to the main body of the program and line 34 is executed. If the CLOSE is successful, the RESET in line 39 opens the file F (which is still associated with "TEXT.TST" on the diskette), moving the window variable to the beginning in preparation for reading data from the file F. Following a successful RESET, the procedure READDATA is called to read back the information placed in "TEXT.TST" and display it at the computer console.

```
Statement     Nest     Source Statement

 1              0       PROGRAM TEXT10_DEMO;
 2              0
 3              0       VAR F : TEXT;
 4              1           I : INTEGER;
 5              1           S : STRING;
 6              1
 7              1         PROCEDURE WRITEDATA;
 8              1         BEGIN
 9              2           WRITELN(F,S);
10              2           WRITE(F,I:4);
11              2           WRITELN(F,45.6789:9:4);
12              2         END;
13              1
14              1         PROCEDURE READDATA;
15              1         VAR R : REAL;
16              2         BEGIN
17              2           READLN(F,S);
18              2           READ(F,I);
19              2           READ(F,R);
20              2           WRITELN(S);
21              2           WRITELN(I:4,' ',R:9:4);
22              2         END;
23              1
24              1       BEGIN
25              1         ASSIGN(F,'TEXT.TST');
26              1         REWRITE(F);
27              1         IF IORESULT <> 0 THEN
28              1           WRITELN('Error creating')
29              1         ELSE
30              1           BEGIN
31              2             I := 35;
32              2             S := 'THIS IS A STRING';
33              2             WRITEDATA;
34              2             CLOSE(F,I);
35              2             IF IORESULT <> 0 THEN
36              2               WRITELN('Error closing')
37              2             ELSE
38              2               BEGIN
39              3                 RESET(F);
40              3                 IF IORESULT <> 0 THEN
41              3                   WRITELN('Error opening')
42              3                 ELSE
43              3                   READDATA;
44              3               END;
45              2           END;
46              1       END.
46              0       ------------------------------------
46              0       Normal End of Input Reached
```

Figure D-2 Text Files

```
PROCEDURE READ;
PROCEDURE READLN;
PROCEDURE READLN(F);
PROCEDURE READLN( F, variable, variable,...,variable);
```

Purpose: Read from the file associated with F into the variables listed. In all cases, read until an end-of-line character is found, skipping any unread data, and advance to the beginning of the next line. READ is redefined to perform conversion of reals, Booleans, and integers.

READLN, like WRITELN, has as parameters an optional file variable and any number of variables to receive the data from the file. If the file variable is not specified, input is taken from the default INPUT file, the keyboard. The variables in the parameter list are the same type as the data being read from the file. However, no type checking is done, so it is up to you to construct a parameterlist compatible with the format of your file. Any numbers are converted on input but the formatting is lost. Numbers must be separated from each other and other data types by a blank or a carriage return line feed.

READLN recognizes but does not transmit the end-of-line character. The action is to read data until it encounters an end-of-line and character. The action is to read data until it encounters an end-of-line and advance the window variable to the beginning of the next line. The data in "TEXT.TST" looks like the following:

```
THIS IS A STRING>/
  35  45.6789>/#
```

After reading the string in the first line to read the integer 35, you must use READ and not READLN. If a READLN were used here, the 35 would be read properly because the first blank terminates the number. However, the window variable would be advanced past the real number to the end of the file. Then, if you try to read the real, all one gets is EOF, and then you wonder what happened to the real number known to be out there.

STRINGS must always be read with a READLN because they are terminated with end-of-line characters. If the data to the file had been 'THIS IS A STRING 35>/', the value returned for S would be the entire line, including the ASCII 35.

Lines 20 and 21 write the data to the computer console in the same format as it is contained in the file.

After executing READDATA, the program is finished. A CLOSE is not necessary because the data in "TEXT.TST" is not altered in any way since the last CLOSE on that file.

Writing to the Printer

Writing to the printer is very simple, as demonstrated in Figure D-3. A file variable is declared to be of type TEXT as in line 5 of Figure D-3. This file variable is ASSIGNed to the printer in line 11. The filename 'P: passed to ASSIGN means that F is to be associated with the list device so that all data written to F is routed to the printer. REWRITE is called to open the list device for writing. Note that a CLOSE is not necessary since the data has already been written and the buffer does not need to be flushed. Lines 23 and 25 use standard Pascal formatting directives. In line 23, R is to be written in a field seven characters long, with three digits to the right of the decimal place.

```
Statement    Nest      Source Statement

 1            0        PROGRAM PRINTER;
 2            0        (*WRITE DATA AND TEXT TO THE PRINTER *)
 3            0
 4            0        VAR
 5            1          F : TEXT;
 6            1          I : INTEGER;
 7            1          S : STRING;
 8            1          R : REAL;
 9            1
10            1        BEGIN
11            1          ASSIGN(F, 'P: ');
12            1          REWRITE(F);
13            1          IF IORESULT <> 0 THEN
14            1            WRITELN('Error rewriting file')
15            1          ELSE
16            1            BEGIN
17            2              S := 'THIS LINE IS A STRING';
18            2              I := 55;
19            2              R := 3.141563;
20            2              WRITE(F,S);
21            2              WRITE(F,I);
22            2              WRITELN(F);
23            2              WRITELN(F,R:7:3);
24            2              WRITE(F,I,R);
25            2              WRITE(F,I:4,R:7:3);
26            2              WRITELN(F);
27            2              WRITELN(F,'THIS IS THE END.')
28            2            END
29            0        END.
29            0        ----------------------------
29            0        Normal End of Input Reached
```

Figure D-3 Writing to a Printer and Number Formatting

4. MISCELLANEOUS FILE ROUTINES

A sample program is not provided for the following routines.

```
PROCEDURE OPEN (F: FILE VARIABLE, TITLE : STRING; VAR RESULT :
INTEGER);
```

Purpose : Identical to the sequence 'ASSIGN(F,TITLE) ; RESET(F); '.

```
PROCEDURE CLOSEDEL (F : FILE VARIABLE; VAR RESULT : INTEGER);
```

Purpose : Close file F and delete it. Used with temporary files. Exactly the same as CLOSE followed by PURGE.

```
PROCEDURE PURGE (F : FILE VARIABLE);
```

Purpose : Delete the file associated with F from the Diskette. An ASSIGN must be executed sometime before the call to PURGE so that the file control block for F contains the name of the file to be deleted. On some operating systems, the file may be required to be closed before this procedure can function properly. In this case CLOSEDEL is a useful procedure.

APPENDIX E: BIBLIOGRAPHY

Grogono, Peter, Programming in Pascal , Addison-Wesley, Reading, Massachusetts, 1978.
A good introduction for self-teaching.

Wilson, I.R, and Addyman, A.M, A Practical Introduction to Pascal , Springer-Verlag, New York, 1979.
An advanced textbook

Jensen, Katheleen, and Wirth, Niklaus, Pascal User Manual and Report , Springer-Verla, New York, 1974.
First definition of Pascal. Best used as a reference document.

"Draft Proposal ISO/DP 7185; Programming Languages-Pascal"
Not designed for the novice. A precise language definition.
May be obtained from American National Standards Institute,
International Sales Department,
1430 Broadway
New York, New York 10018

Findley, William, and Watt, David A., PASCAL: An Introduction to Methodical Programming , Computer Science Press, Potomac, Maryland, 1978.

Conway, Richard, Gries, David, Zimmerman, E. Carl, A Primer on Pascal Winthrop Publishers, Cambridge, Massachusetts, 1976.

Miller, Alan R., Pascal Programs for Scientists and Engineers Sybex, Inc., Berkeley, CA., 1981.

De Re Atari, "A Guide to Effective Programming", APX-90008

ATARI 400/800 Disk Operating System II Reference Manual, C016347

ATARI 400/800 BASIC Reference Manual, C015307

APPENDIX F: Player/Missile Demo Program

The Player/Missile Demo program may be entered using the ATARI Program-Text Editor and used as an example for modular compilation and use of the built-in graphics and sound procedures. Compile each of the modules separately (PMDEMO, PMMIS, PEEKPOKE, PMSND). Then link these modules together along with the Graphics and Sound Library (GRSND). When the linker responds with the asterisk repond with the following:

```
D2:PMDEMO,D2:PMMIS,D2:PMSND,D2:PEEKPOKE,GRSND,PASLIB/S
```

Once linked together you may execute the program using the "Run" command. A joystick is required to move the player and fire the missile.

```
PROGRAM PLAYER/MISSILE (INPUT,OUTPUT);

(*
This program, written in Pascal, demonstrates the player/missile
capabilities of ATARI Pascal. It is based on the player/missile
demonstration program written in BASIC. Error checking has been
implemented so that the player does not cause system crashes when it
goes off the screen. The player is held just off the visible screen
until the input from the joystick changes its position to a point on
the visible screen. In addition a visible missile will be fired when
the button on the joystick is pressed. Also implemented are sounds
associated with the movement of both the player and the missile.

Four modules must be compiled separately and then linked together to
form the executable object file. These modules include PMSOUND
(D2:PMSND.PAS), PEEKPOKE(D2:PEEKPOKE.PAS), PMMISSILE(D2:PMMIS.PAS) and
program player/missile (D2:PMDEMO.PAS).

The executable file is D2:PMDEMO.COM and can be run by typing "R" in
the Pascal monitor. A joystick is required for program execution.
The player will respond to the joystick by moving vertically,
horizontally, and diagonally. The missile is fired by pressing the
button on the joystick. Both the player and the missile may be moving
simultaneously.
*)

TYPE
  SCRN_TYPE=(FULL_SCREEN,SPLIT_SCREEN);
  CLEAR_TYPE=(CLEAR_SCREEN,DO_NOT_CLEAR_SCREEN);

VAR
  PMBASE, (*PLAYER-MISSILE BASE ADDRESS*)
  X, (*PLAYER AND MISSILE HORIZONTAL POSITION*)
  Y, (*PLAYER VERTICAL POSITION*)
  MISY, (*MISSILE VERTICAL POSITION*)
  A:INTEGER;
  FIRED:BOOLEAN; (*FLAG SET TO TRUE WHEN MISSILE FIRED, RESET WHEN
                   MISSILE HAS MOVED OFF THE TOP OF THE SCREEN*)

EXTERNAL PROCEDURE INITGRAPHICS(MAX_MODE:INTEGER);

EXTERNAL PROCEDURE GRAPHICS(MODE:INTEGER;SCREEN:SCRN_TYPE;CLEAR:
CLEAR_TYPE);

EXTERNAL PROCEDURE SETCOLOR(REGISTER,HUE,LUMINANCE:INTEGER);

EXTERNAL PROCEDURE SOUND(VOICE,PITCH,DISTORTION,VOLUME:INTEGER);

EXTERNAL FUNCTION STICK(STKNUM:INTEGER):INTEGER;

EXTERNAL FUNCTION STRIG(STKNUM:INTEGER):INTEGER;

EXTERNAL PROCEDURE MAKENOISE;(*IN MODULE PMSOUND*)
```

```
EXTERNAL PROCEDURE BIGBANG; (*IN MODULE PMMISSILE*)

EXTERNAL PROCEDURE MOVEMISSILE; (*IN MODULE PMMISSILE*)

EXTERNAL PROCEDURE POKEBYTE(ADDR,VAL:INTEGER); (*IN MODULE PEEKPOKE*)

EXTERNAL FUNCTION PEEKBYTE(ADDR:INTEGER):INTEGER; (*IN MODULE
PEEKPOKE*)

PROCEDURE SETPLAYER;
(*SETPLAYER initializes the player by first clearing out the player's
section of memory and then initializing that memory with the proper
values so that the player takes on the shape printed below.*)
VAR I:INTEGER;
BEGIN
    (*CLEAR PLAYER AREA IN MEMORY*)
  FOR I:=PMBASE+512 TO PMBASE+640 DO POKEBYTE(I,0);
  POKEBYTE(704,108);  (*SET PLAYER COLOR TO PURPLE*)
  (*INITIALIZE PLAYER AREA WITH MISSILE SIZE, SHAPE*)
  I:=PMBASE+512+Y;
  POKEBYTE(I,153); (*PLAYER WILL LOOK LIKE THIS: *)
  I:=I+1;
  POKEBYTE(I,189);   (*          *)
  I:=I+1;            (*          *)
  POKEBYTE(I,255);   (*          *)
  I:=I+1;            (*          *)
  POKEBYTE(I,189);   (*          *)
  I:=I+1;
  POKEBYTE(I,153)
END;

PROCEDURE MOVERIGHT;
(*MOVERIGHT moves the player to the right on the screen by
incrementing the player's horizontal position register.*)

BEGIN
  IF X<214 THEN BEGIN (*MOVE RIGHT ONE COLOR CLOCK*)
    X:=X+1; (*INCREMENT*)
    (*POKE NEW VALUE INTO HORIZONTAL POSITION REGISTER*)
    POKEBYTE (53248,X)
  END  (*ELSE HOLD STILL, JUST OFFSCREEN AT RIGHT:)
END;

PROCEDURE MOVELEFT;
(*MOVELEFT moves the player to the left on the screen by decrementing
  the player's horizontal position register.*)

BEGIN
  IF X>40 THEN BEGIN (*MOVE LEFT ONE COLOR CLOCK*)
    X:=X-1; (*DECREMENT*)
    (*POKE NEW VALUE INTO HORIZONTAL POSITION REGISTER*)
    POKEBYTE(53248,X)
  END  (*ELSE HOLD STILL, JUST OFFSCREEN AT LEFT*)
END;
```

```
PROCEDURE MOVEUP;
(*MOVEUP moves the player up on the screen by moving the player up in
 the player's memory area.*)
VAR I:INTEGER;
BEGIN
  IF Y>1 THEN BEGIN (*MOVE PLAYER UP ONE UNIT IN MEMORY AND ON
    SCREEN*)
    FOR I:=0 TO 6 DO POKEBYTE(PMBASE+511+Y+I,
    PEEKBYTE(PMBASE+512+Y+I));
    Y:=Y-1 (*PLAYER HAS MOVED UP ONE UNIT*)
  END (*ELSE HOLD STILL, JUST OFFSCREEN AT TOP OF SCREEN*)
END;

PROCEDURE MOVEDOWN;
(*MOVEDOWN moves the player down on the screen by moving the player
  down in the player's memory area.*)
VAR I:INTEGER;
BEGIN
  IF Y<120 THEN BEGIN
    (*MOVE PLAYER DOWN ONE UNIT ON SCREEN AND IN MEMORY*)
    FOR I:=6 DOWNTO 0 DO POKEBYTE(PMBASE+512+Y+I,PEEKBYTE
          (PMBASE+511+Y+I));
    Y:=Y+1 (*PLAYER HAS MOVED DOWN ONE UNIT*)
  END  (*ELSE HOLD STILL, JUST OFFSCREEN AT BOTTOM OF SCREEN*)
END;

BEGIN (*MAIN PROGRAM*)
  INITGRAPHICS(0);
  GRAPHICS(0,FULL_SCREEN_,CLEAR_SCREEN); (*CLEAR SCREEN*)
  POKEBYTE(755,1);   (*POKE OUT CURSOR*)
  SETCOLOR(2,0,0);   (*SET BACKGROUND COLOR TO BLACK*)
  X:=120; (*SET HORIZONTAL COORDINATE OF PLAYER*)
  Y:=48; (*SET VERTICAL COORDINATE OF PLAYER*)
  A:=PEEKBYTE(106)-8;
  POKEBYTE(54279,A);(*SET PLAYER-MISSILE ADDRESS BASE REGISTER*)
  PMBASE:=256*A; (*SET PLAYER-MISSILE ADDRESS*)
  POKEBYTE(559,46); (*SET DMACTL IN OS SHADOW*)
  POKEBYTE(53277,3); (*SET GRACTL--ENABLE PLAYER AND MISSILE DMA TO
                       PLAYER AND MISSILE GRAPHICS REGISTERS*)
  POKEBYTE(53248,X);   (*SET PLAYER HORIZONTAL POSITION*)
  SETPLAYER; (*CLEAR AND SET PLAYER-MISSILE MEMORY AREA*)

  (* NOW FOR THE MOVEMENT AND MISSILE FIRING *)
  FIRED:=FALSE; (*INITIALIZE "FIRED" FLAG*)
  WHILE 4>2 DO BEGIN
    A:=STICK(0);
    IF A<>15 THEN MAKENOISE; (*GENERATE MOVEMENT SOUND*)
    (*MOVEMENT*)
    IF A=5 THEN BEGIN
       MOVERIGHT;
       MOVEDOWN
    END ELSE IF A=6 THEN BEGIN
       MOVERIGHT;
       MOVEUP
```

```
    END ELSE IF A=7 THEN MOVERIGHT
    ELSE IF A=9 THEN BEGIN
       MOVELEFT;
       MOVEDOWN
    END ELSE IF A=10 THEN BEGIN
       MOVELEFT;
       MOVEUP
    END ELSE IF A=11 THEN MOVELEFT
    ELSE IF A=13 THEN MOVEDOWN
    ELSE IF A=14 THEN MOVEUP
    ELSE IF A=15 THEN SOUND(0,182,2,0);
      (*PLAYER IS STANDING STILL, SO MAKES NO SOUNDS*)
    IF FIRED THEN MOVEMISSILE (*CONTINUE MISSILE ON ITS TRAJECTORY*)
    ELSE IF STRIG(0)=0 THEN BIGBANG; (*FIRE MISSILE*)
  END; (*WHILE*)
END.
```

```
MODULE PMMISSILE;
(*The routines in this module handle the firing and flight of the
  missile for the player/missile graphics demonstration program.*)

VAR PMBASE,X,Y,MISY:EXTERNAL INTEGER;
    FIRED:EXTERNAL BOOLEAN;

EXTERNAL FUNCTION PEEKBYTE(ADDR:INTEGER):INTEGER;

EXTERNAL PROCEDURE POKEBYTE(ADDR,VAL:INTEGER);

EXTERNAL PROCEDURE SOUND(VOICE,PITCH,DISTORTION,VOLUME:INTEGER);


PROCEDURE MOVEMISSILE;
(*Movemissile is called by procedure bigbang when the missile is
first fired, and later by the main program as the missile continues
its trajectory. The main program calls movemissile until the missile
has moved off the top edge of the screen and the "fired" flag has been
reset.*)
  VAR I:INTEGER;

BEGIN
  IF MISY>5 THEN BEGIN
    FOR I:=0 TO 1 DO POKEBYTE(PMBASE+383+MISY+I,PEEKBYTE(PMBASE+384+
          MISY+I));
    (*MOVE MISSILE UP IN MISSILE MEMORY*)
    MISY:=MISY-1 (*MISSILE HAS MOVED UP ONE*)
  END;
  IF MISY<=5 THEN FIRED:=FALSE (*MISSILE HAS MOVED OFF THE TOP EDGE
                                 OF THE SCREEN, SO RESET THE "FIRED"
                                 FLAG*)
END;

PROCEDURE BIGBANG;
(*Bigbang is called whenever the user presses the fire button on the
joystick. Bigbang launches the missile and starts it on its
trajectory.*)
  VAR I:INTEGER;

BEGIN
  FOR I:=PMBASE+384 TO PMBASE+512 DO POKEBYTE(I,0);
    (*CLEAR MISSILE AREA IN MEMORY*)
  SOUND(3,46,12,14); (*FIRE!! (BEGIN FIRING NOISE)*)
  POKEBYTE(53260,0); (*SET NORMAL MISSILE SIZE*)
  POKEBYTE(53252,X+3);
   (*SET MISSILE HORIZONTAL POSITION EQUAL TO PLAYER HORIZONTAL
     POSITION*)
  MISY:=Y-1; (*SET MISSILE VERTICAL POSITION EQUAL TO THE POINT JUST
               ABOVE PLAYER VERTICAL POSITION*)
  I:=PMBASE+384+MISY;
  POKEBYTE(I,3);  (*SET MISSILE SHAPE IN MEMORY*)
  FIRED:=TRUE; (*SET MISSILE FIRED FLAG TO SHOW THAT A MISSILE HAS
                 BEEN FIRED*)
```

```
  MOVEMISSILE; (*START MISSILE ON ITS TRAJECTORY*)
  SOUND(3,46,12,0) (*STOP THE FIRING SOUND*)
END;

MODEND.
```

```
MODULE PMSOUND;

(*This module contains procedure makenoise, which controls the sound
generation for the player's movement. This procedure was put into its
own module.*)

EXTERNAL PROCEDURE SOUND(VOICE,PITCH,DISTORTION,VOLUME:INTEGER);

 PROCEDURE MAKENOISE;
 (*GENERATE ENGINE SOUND WHEN PLAYER MOVES.*)
 BEGIN
  SOUND(0,182,2,6)
 END;

MODEND.
```

```
MODULE PEEKPOKE;
(*This module contains procedures for performing BASIC style PEEKs
  and POKEs.*)

PROCEDURE POKEBYTE(ADDR,VAL:INTEGER);
(*
  POKEBYTE: BASIC STYLE OF MEMORY LOCATIONS
    POKEBYTE PROVIDES A METHOD,SIMILAR TO THE BASIC POKE, FOR THE
    PASCAL USER TO SET MEMORY LOCATIONS.
  ENTRY: POKEBYTE(ADDR,VAL); (SAMPLE CALL)
         ADDR = ADDRESS TO BE POKED
         VAL = VALUE TO BE POKED INTO ADDRESS
  EXIT: CONTENTS OF ADDR IS NOW VAL
  CHANGES: ADDR (ADDRESS)
  CALLS: -NONE-
*)

  VAR
   PTR:^CHAR; (*POINTER TO ADDRESS TO BE CHANGED*)

  BEGIN
   PTR:=ADDR; (*SET PTR TO POINT AT DESIRED ADDRESS*)
   PTR^:=CHR(VAL) (*POKE NEW VALUE INTO ADDRESS POINTED TO BY PTR*)
  END;

FUNCTION PEEKBYTE(ADDR:INTEGER):INTEGER;
(*
  PEEKBYTE:  SIMPLE BASIC STYLE PEEK AT MEMORY LOCATIONS
    PEEKBYTE PROVIDES THE PASCAL USER WITH A METHOD, SIMILAR TO THE
    BASIC PEEK, TO FIND OUT THE CONTENTS OF MEMORY LOCATIONS.
  ENTRY: INTEGERVARIABLE := PEEKBYTE(ADDR); (SAMPLE CALL)
         ADDR = ADDRESS TO BE LOOKED AT
  EXIT:  PEEKBYTE = CONTENTS OF THE ADDRESS GIVEN BY ADDR
  CHANGES:  INTEGERVARIABLE IN THE CALLING ROUTINE
  CALLS:  -NONE-
*)

  VAR
   PTR:^CHAR; (*POINTER TO ADDRESS TO BE LOOKED AT*)

  BEGIN
   PTR:=ADDR; (*SET PTR TO POINT TO DESIRED ADDRESS*)
   PEEKBYTE:=ORD(PTR^) (*PEEKBYTE "PEEKS AT" AND
   RETURNS CONTENTS OF ADDRESS POINTED TO BY PTR*)
  END;

MODEND.
```

APPENDIX G: HELPFUL HINTS

The following are assorted statements that may prove to be useful when using the ATARI Pascal Language System.

1. Compilation of Pascal programs using Floating Point numbers (REALS) requires that the Include file FLTPROCS or STDPROCS be identified within the declaration body of the source. In addition the FPLIB must be linked with your compiled source and PASLIB. Failure to do so will cause your compilation and/or linking to error. Refer to the demo program CALC for an example.

2. Identifiers are significant to only eight characters.

3. CLOSEDEL can be used with any file so be careful. You may accidentally delete something that you didn't expect to.

4. While standard procedures are built into the compiler, others require the appropriate Include files for declaration purposes. Check these files to determine if you need them. These Include files may be listed on the printer by use of the copy option under DOS.

5. The reserved word "PREDEFINED" allows certain procedures and functions to become part of the scope surrounding the program. In addition any file parameter is passed as two parameters as required by the run-time routines.

INDEX

ABSOLUTE variables, 32, 59
ADDR 41
AND
  and 16 bit variables, 94
ARCTAN 104
ARRAY
  as procdural parameters 102
  storage 29
ASSIGN 50, 131
Assignment compatibility 90
Available memory message 8, 13

BCD REAL 71
Bit and byte manipulation 38, 93
BLOCKREAD 52
BLOCKWRITE 52
BOOLEAN 70
Built-in procedures
  ADDR 41
  ASSIGN 50
  BLOCKREAD 52
  BLOCKWRITE 52
  CLOSE 54
  CLOSEDEL 54
  CLRBIT 38
  CONCAT 45
  COPY 46
  DELETE 48
  EXIT 37
  FILLCHAR 43
  GNB 51
  HI 40
  INSERT 49
  IORESULT 56
  LENGTH 44
  LO 40
  MAXAVAIL 57
  MEMAVAIL 57
  MOVE 35
  MOVELEFT 35
  MOVERIGHT 35
  OPEN 53
  POS 47
  SETBIT 38
  SHL 39
  SHR 39
  SIZEOF 42
  summary of 58
  SWAP 40

TSTBIT 38
WNB 51
PURGE 55
BYTE 71, 86
Byte manipulation
(see Bit and byte manipulation)

CALC.PAS 7
Chaining 103
Chaining
absolute variable communication 32
example 33
global variable communation 32
how-to 32
maintain heap 32
CHAR 70
CHR 70, 90, 105
CLOSE 54, 132
CLOSEDEL 54, 141
CLRBIT 38
Comments
syntax 83
Compatibility with UCSD 77
Compiler control toggles
entry point control $E 14
listing controls $P/$L 15
run-time range checking control $R 15
run-time exception checking control $X 15
source code include mechanism $I 14
strict/relaxed type checking control $T/$W, 14
summary 16
syntax 14
Compiler
output 8, 13
. output 8, 13
available memory 8
compile time informational output 7, 13
execution 7, 12
operational description 12
PHASE 1 13, 18
PHASE 2 13
remaining memory 8
sample output 7
separate compilation 26
step-by-step instructions 7
system requirements 3
user table space 8
CONCAT 45
Confromant arrays 102
Constant data at compile-time 61
COPY 46

Data storage 70
Data types
BOOLEAN 70
BYTE 71
CHAR 70
INTEGER 71
range 70
REAL 71
SET 75
size 70
STRING 71
WORD 71
DELETE 48
Distribution disk
contents 4
minimum configuration 3

End of file 128, 134, 135
EOF 104, 133
EOLN 104, 133
Error handling
run-time 68
Error message
type conflict 90
Error messages 18, 118
Exception checking
(see Compiler control toggles)
EXIT 37
Extensions to ISO standard
(see ISO standard extensions)
Extensions
summary 81
EXTERNAL
and entry point symbols 14
and modular compilation 26
and procedures/functions 26
and variables 27
routines as parameters 26

FIB
(see File Information Block)
File Information Block 128
File variable 128
File variable untyped files are allowed 90
Filename
definition 127
Filenames
associating external and internal 50
compiler input 7, 12
linker input 9, 19
Files
ASCII text 89

ASSIGN procedure 50
associating files with external names 100
built-in procedures 100
chaining 151
closing 54, 132
creating 131
definition 127
deleting 54, 55
devices E:, S:, K:, P:, 50
error handling 56
example 130
fast byte routines 51
formatted output 137
hex output 106
implied conversions 90
local 50
local files and linker /D switch 20
opening (see also RESET) 53
pre-defined type TEXT 89
primitive file access 52
printer output 50, 14
temporary, (see local)
text 135, 138
untyped 89
window variable 128, 132, 133, 134
writing to printer 140
FILLCHAR 43
Floating Point REAL 71
Formatted output 106, 137
FORWARD 101
FPLIB.ERL 4, 9, 19, 71, 152

GET 133
GNB 51
GOTO 96
GSSND.ERL 4, 143

Heap management
ISO standard 142
MEMAVAIL and MAXAVAIL 57
parameters 104
Hexadecimal numbers 106
HI 40

I/O
(see Files)
Identifiers
and @ 83, 84
external signifigance 26
legal Pascal 83

Include files 4, 5, 8, 14, 152
INLINE
code examples 61
syntax 60
INSERT 49
INTEGER 71
IORESULT 56, 131, 138, 140
ISO standard extensions
absolute variables 91
additions to assignment compatibility rules 95
BNF syntax description of ATARI Pascal 109
built-in procedures and functions 34
chaining 32
concise list of ATARI Pascal facilities 1
ELSE clause on CASE statement 96
external procedures 98
INLINE 60
modular compilation 26
null strings 84
operators 94
WRD type transfer function 104
ISO standard
assignment compatibility 90
changes from Jensen and Wirth for FOR loops 97
draft used by ATARI 1
extensions for conformant arrays 102
summary of features 81
type compatibility 90

LENGTH 44
Line 135
Line numbers 18
Linker
/D and chaining 32
attributes of compatible modules 22
command file facility switch /F 20
data origin switch /d 19
effects of /P and /D on .COM file contents 20
effects of using /D on local files 20
extending map switch /E 19
gaining memory space 19
input filenames 19
invocation 19
library search switch /S 19
load map switch /L 19
program origin switch /P 20
sample 9
sample output 9
saving space by using /D 20
switch summary 21

switches 19
LINK 4
Listing 7
LO 40
Local files
(see Files)

MAXAVAIL 57
MEMAVAIL 57
Modular compilation
and $E toggle 26
and EXTERNAL 26
example 26
overview 26
syntax 26
MOVE 35
MOVELEFT 35
MOVERIGHT 35

NOT
and 16 bit variables 94

ODD 70, 104
OPEN 53, 133, 141
Operators
AND 70, 94
and 16 bit variables 93
NOT 70, 94
OR 70, 94
Option Switches
compiler 16
linker 21
OR
and 16 bit variables 93
ORD 70, 71, 90, 104
Output
formatted 137

PACKED 70, 86
PASLIB 9, 152
PASLIB.ERL 4
Pointers 89
Portability 14
POS 47
Printer
assignment 50
writing example 140
writing to 140
Program sample
CHAIN Demo 33

DEMOCON (conformant arrays) 102
DEMO_INLINE 61
External_Demo (Modular compilation) 27
PRINTER 140
Procedure ACCESS (strings) 73
Procedure ADDR DEMO 41
Procedure ASSIGN (strings) 72
Procedure COMPARE (strings) 74
Procedure CONCAT_DEMO 45
Procedure COPY_DEMO 46
Procedure DELETE_DEMO 48
Procedure EXITTEST 37
Procedure FILL_DEMO 43
Procedure HI_LO_SWAP 40
Procedure INSERT_DEMO 49
Procedure MOVE_DEMO 36
Procedure POS_DEMO 47
Procedure SHIFT_DEMO 39
Procedure SIZE_DEMO 42
Procedure TST_SET_CLR_BITS 38
Procedure TEXTIO_DEMO 138
Procedure WRITE_READ_FILE_DEMO 129
PURGE 55, 141
PUT 129

Range checking
(see Run-time)
READ 134
READLN 139
REAL
BCD 71
floating point 71
RECORD
storage 29
Remaining memory message 8
Requirements
run-time 4
system 3
Reserved words 117
RESET 133
REWRITE 131
Run-time Library
source 4
Run-time
error handling 68
exception checking 68
fatal errors 69
range checking 68

Scalars
storage 29
SET 30

SETBIT 38
SHL 39
SHR 39
SIZEOF 42
Space reduction
  and linker /D switch 20
STRING 135
STRING implementation details 71
STRING
  access 75
  and READLN 139
  assignment 71
  comparison 74
  CONCAT 45
  COPY 46
  default length 87
  definition 71, 86
  explicit length declaration 87
  null string 85
  run-time error 68
  use as arrays of characters 92
Strings
  DELETE 48
  INSERT 49
  LENGTH 44
  POS 47
SWAP 40
Symbols 83
Symbols
  identifier significance 84
  use of @ in identifiers 83
  use of hexadecimal numeric literals 84
  use of underscore in identifiers 84

TEXT files
  definition 135
TSTBIT 38
Type checking toggle 14
Type conflict
  error 89
Types
  ABSOLUTE attribute for variables 59
  data implementation 70
  extended 86
  file types 89
  implementation of PACKED 86
  pointers 89
  pre-defined 86
  range of SET type 88
  restrictions on use of ABSOLUTE
   with strings 59

User table space 8

Window variable
(see Files)
WNB 51
WORD 71, 86
WRITE 132
WRITELN
and text files 136

LIMITED WARRANTY ON MEDIA AND HARDWARE ACCESSORIES.

We, Atari, Inc., guarantee to you, the original retail purchaser, that the medium on which the APX program is recorded and any hardware accessories sold by APX are free from defects for thirty days from the date of purchase. Any applicable implied warranties, including warranties of merchantability and fitness for a particular purpose, are also limited to thirty days from the date of purchase. Some states don't allow limitations on a warranty's period, so this limitation might not apply to you. If you discover such a defect within the thirty-day period, call APX for a Return Authorization Number, and then return the product along with proof of purchase date to APX. We will repair or replace the product at our option.

You void this warranty if the APX product: (1) has been misused or shows signs of excessive wear; (2) has been damaged by use with non-ATARI products; or (3) has been serviced or modified by anyone other than an Authorized ATARI Service Center. Incidental and consequential damages are not covered by this warranty or by any implied warranty. Some states don't allow exclusion of incidental or consequential damages, so this exclusion might not apply to you.

DISCLAIMER OF WARRANTY AND LIABILITY ON COMPUTER PROGRAMS.

Most APX programs have been written by people not employed by Atari, Inc. The programs we select for APX offer something of value that we want to make available to ATARI Home Computer owners. To offer these programs to the widest number of people economically, we don't put APX products through rigorous testing. Therefore, APX produts are sold "as is", and we do not guarantee them in any way. In particular, we make no warranty, express or implied, including warranties of merchantability and fitness for a particular purpose. We are not liable for any losses or damages of any kind that result from use of an APX product.

ATARI PROGRAM EXCHANGE

REVIEW FORM

We're interested in your experiences with APX programs and documentation, both favorable and unfavorable. Many software authors are willing and eager to improve their programs if they know what users want. And, of course, we want to know about any bugs that slipped by us, so that the software author can fix them. We also want to know whether our documentation is meeting your needs. You are our best source for suggesting improvements! Please help us by taking a moment to fill in this review sheet. Fold the sheet in thirds and seal it so that the address on the bottom of the back becomes the envelope front. Thank you for helping us!

1. Name and APX number of program _____

2. If you have problems using the program, please describe them here.

_____

_____

_____

3. What do you especially like about this program?

_____

_____

_____

4. What do you think the program's weaknesses are?

_____

_____

_____

5. How can the catalog description be more accurate and/or comprehensive?

_____

_____

6. On a scale of 1 to 10, 1 being "poor" and 10 being "excellent", please rate the following aspects of this program?

_____ Easy to use
_____ User-oriented (e.g., menus, prompts, clear language)
_____ Enjoyable
_____ Self-instructive
_____ Useful (non-game software)
_____ Imaginative graphics and sound

7. Describe any technical errors you found in the user instructions (please give page numbers).

8. What did you especially like about the user instructions?

9. What revisions or additions would improve these instructions?

10. On a scale of 1 to 10, 1 representing "poor" and 10 representing "excellent", how would you rate the user instructions and why?

11. Other comments about the software or user instructions:

STAMP

ATARI Program Exchange
Attn: Publications Dept.
P.O. Box 50047
60 E. Plumeria Drive
San Jose, CA 95150

[seal here]