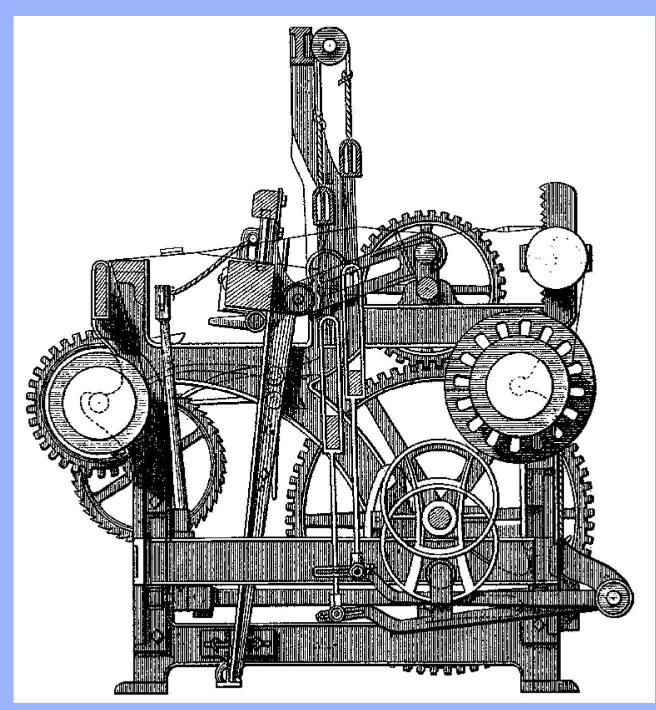
OMM

The Object Module Manager

Includes AmperWorks



Morgan Davis Group

Copyright

© 1992 Morgan Davis Group. All Rights Reserved.

http://www.morgandavis.net

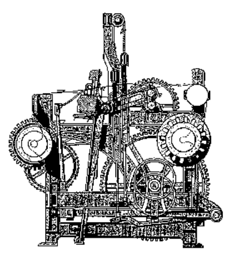
No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the author. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this product, the author assumes no responsibility for errors or omissions.

The product names mentioned in this manual are the trademarks or registered trademarks of their manufacturers.

ProDOS and ProDOS BASIC are copyrighted programs of Apple Computer, Inc. licensed to the Morgan Davis Group to distribute for use only in combination with this product. Apple software shall not be copied onto another diskette (except for archive purposes) or into memory unless as part of execution of this product. When this product has completed execution, Apple software shall not be used by any other program.

Apple Computer, inc. makes no warranties, either express or implied regarding the enclosed software package, its merchantability or its fitness for any particular purpose. The exclusion of implied warranties is not permitted in some states. The above exclusion may not apply to you. This warranty provides you with specific legal rights. There may be other rights that you may have that vary from state to state.

First printing — February 1992 — U.S.A.



The Power Loom

From the 1851 Edition of *The Iconographic Encyclopedia of Science, Literature and Art*

Printed on 20% post-consumer recycled paper.

Contents

Introduction **5**
Credits 6

Chapter One: Using the OMM **7**
What You Should Know 7
Memory Usage 7
Loading the OMM 8
Module Entries 8
OMM Commands 10
LOAD GET 11
LOAD PRINT 12
LOAD FRE 12
LOAD PEEK 13
LOAD TRACE/NOTRACE 13
LOAD CALL 14
Sample Programs 14

Chapter Two: Building A Module **15**
OMM Features 15
Module Format 15
Header 16
Code Section 17
Immediate Mode Table 18
Data Section 18
IMC 19
Messages 20
Utility Functions 23
Interfaces & Sources 28
Buffer Space 28
Absolute References 29
Opcode Usage 29

Chapter Three: AmperWorks **31**
Introduction 31
Command Summary 32
/ (Get Info) 33
\ (Set Info) 34
< (Parent Directory) 35

ADD 35
ASC 36
COPY 36
ERASE 36
FILES 37
GET 38
HLIN 38
LCASE 39
LIST 39
LEFT$ 40
MID$ 40
MLI 41
ONERR 42
POKE 42
POP 43
POS 43
PRINT 44
READ 44
REPT 45
RESTORE 46
RESTORE GOTO 46
RIGHT$ 47
SPC 48
SRT (Sort) 48
STORE 49
STORE CLEAR 50
SWAP 50
TFILES 50
TIME 51
UCASE 51
UNTIL 52
VAL 52
VLIN 52

Appendix A: ASCII Chart 53
Appendix B: ProDOS File Types 55
Appendix C: Error Codes 57
Appendix D: Licensing 59
Appendix E: BASIC Syntax 61

Index 63

Introduction

What is the Object Module Manager (OMM)?

For the avid BASIC programmer, the OMM allows you to make use of multiple machine language programs (e.g. utilities, editors, debuggers, graphics tools, modem and printer drivers, etc.) without the usual hassle of *where* to load them into memory. Without the impending disaster normally accompanied with putting two or more incompatible utilities into memory at once.

Throughout this manual the following symbols are used to denote keys on your keyboard:

[◁ reset]	Reset
[option]	Option or solid-apple
[command]	Command or open-apple
[control]	Control
[esc]	Escape
[return]	Return
[delete]	Delete
[↑]	Up arrow or [control]-K
[↑]	Down arrow or [control]-J
[←]	Left arrow or [control]-H
[→]	Right arrow or [control]-U
[tab]	Tab or [control]-I
[shift]	Shift

Hyphenated key references, such as [command]-[esc], tell you to press and hold the first key while typing the second.

Credits

The OMM would not be in your hands now if it were not for the encouragement and support I have received over the years. My foremost gratitude goes to the many customers and friends who have made it possible for me to create Apple II software. I cannot express enough appreciation for Tim Swihart, humble servant of Apple Computer, who is always there to assist. Finally, I am forever grateful for the love and tireless patience from my wife, Dawn, and children, Kristi and Ryan. To you all, I express my most sincere thanks.

—Morgan Davis

Using the OMM

This chapter shows you how to get started with the Object Module Manager (OMM). You'll learn how to load it into memory and work with it using Applesoft BASIC.

What You Should Know

The OMM is a utility for Applesoft BASIC. It efficiently manages machine language programs that enhance the BASIC environment. A BASIC program can load machine language programs (*object modules*) from disk into memory, and unload them when no longer needed. It allows modules to send messages to each other using *intermodule communication* (IMC). With the OMM, programmers can easily create highly integrated applications for use in the BASIC environment.

Since the OMM is a BASIC programmer's utility, a working knowledge of Applesoft and ProDOS BASIC (BASIC.System) is essential.

If you plan to create your own modules for use with the OMM, you'll need to know assembly language for the 6502-series microprocessors. You'll also need an assembler.

Memory Usage

The OMM resides in the main 48K memory bank. It does not use any auxiliary memory or space used by a RAM disk. To make the most efficient use of memory, the OMM moves itself as high as possible into free memory, giving optimum space to BASIC programs and modules.

As modules are loaded, the OMM places them just below itself and any previously loaded modules, building downward into memory. The OMM protects modules from being overwritten by file commands, such as BLOAD.

If a module is removed, any modules residing below it are moved up to recover unoccupied memory.

Loading the OMM

Since the OMM is used by many software packages, getting it into memory may depend on the program you are using. Simply booting the program's diskette may do the job.

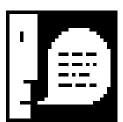
If you plan to use the OMM with your own projects, here is how to get it into memory. From the Applesoft prompt (*immediate mode*) type:

```
]-OMM.Loader   return
```

From a running program (*deferred mode*) use:

```
10 PRINT CHR$(4) "-OMM.Loader"
```

In each case, you may need to supply the full pathname to the OMM.Loader file if it is in a directory other than the current directory.

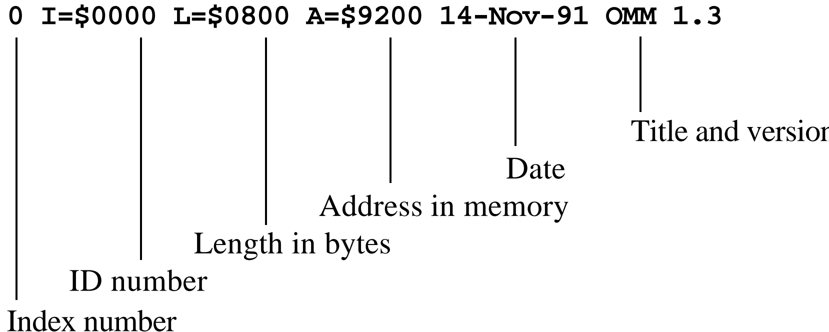


NOTE: If a copy of the OMM.Loader is already in memory, a DUPLICATE FILE ERROR occurs.

Module Entries

Once the OMM.Loader is activated, a line of information about the OMM is displayed:

```
0 I=$0000 L=$0800 A=$9200 14-Nov-91 OMM 1.3
```



Index Number. Each module, including the OMM itself, can be referenced by its index. Indexes start at 0 (the OMM) and increase by one for each subsequently loaded module.

ID Number (I). Each module (and the OMM) is identified by a unique ID number. Modules can be referenced by index or by ID when working with the OMM.

Length (L). Indicates the size of the module and the amount of memory a module uses.

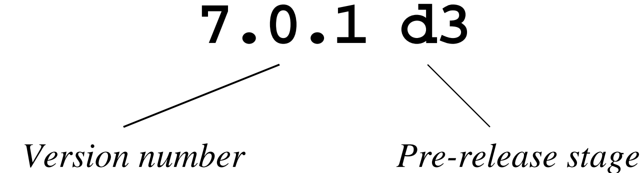
Address (A). Indicates the address in memory where the module resides.

Date. Version date of the module. The programmer who created it will update the date each time the module is changed.

Title and Version. The name of the module and its version number.

About Version Numbers

Version numbers are useful for indicating the various stages of enhancements and corrections to computer programs. Here is a guide for interpreting them:



The first digit in the version indicates the major release number (e.g., 7, as shown above). This would mean the 7th release of the program in which major changes were made.

The second digit is the minor release number (e.g., 0, in the example). In this case, there were no changes to warrant increasing the minor release digit since the 7.0 version. When enough minor changes occur, the major release digit increases.

Continued . . .

***About Version Numbers** (continued) . . .*

The third digit is the maintenance level (e.g., 1, in the example). This indicates very minor corrections made to the program. When enough maintenance fixes occur, the minor release number increases.

If the program version is not for public release, it includes a pre-release stage version. This begins with a code letter:

d	*in-house development version (not yet released)*
a	*alpha release (for a small group of testers)*
b	*beta release (for a larger group of testers)*

Following the letter is a pre-release stage number. In the example on the previous page, 7.0.1d3 indicates that the program is the 3rd developmental version for the 7.0.1 release. It will go through alpha (e.g., 7.0.1a8) and beta (e.g., 7.0.1b12) stages. After all the testing is complete, it will drop the pre-release numbers to become the final 7.0.1 version, or the first maintenance release of the 7th major version of the program.

OMM Commands

The OMM provides the programmer with the following commands, accessed through the ampersand (&) feature of Applesoft BASIC:

`& LOAD GET`	Gets modules from disk into memory.
`& LOAD PRINT`	Prints a table of the loaded modules.
`& LOAD FRE`	Removes modules from memory.
`& LOAD NOTRACE`	Turns off reporting of loading and unloading activity.
`& LOAD TRACE`	Turns on reporting of module loading and unloading.
`& LOAD PEEK`	Peeks at various Loader information.
`& LOAD CALL`	Calls a module, passing various parameters.

These commands are described next. Refer to Appendix E, “BASIC Syntax” for a description of the various types of parameters that can be used by ampersand commands.

LOAD GET

`& LOAD GET strexp [,numvar] [AND ...]`

To bring modules from disk into memory, use the LOAD GET command. *strexp* is a string expression containing the pathname to a module. Modules have a file type of REL ($FE) on disk. Should an error occur (such as the module not being found), the appropriate error is generated through BASIC. You can trap for any disk errors using standard Applesoft ONERR handling.

To load modules without error reporting, use the optional numeric variable, *numvar*, method:

```
& LOAD GET "module",N
```

This tells the OMM to return disk errors in the result variable, N. If no error occurs, zero is returned. Any non-zero result is an error as described in Appendix D, "Error Codes." It is up to you to handle errors when using this method.

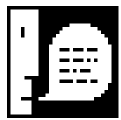
A program can make as many LOAD GET calls as needed to load the modules, but it may also specify a list of modules in one LOAD GET call by using the AND keyword. Example:

```
& LOAD GET "module1" AND "module2" AND "module3"
```

This is also valid:

```
& LOAD GET "module1",R1 AND "module2",R2
```

Up to 24 modules can be loaded into memory at any one time.



NOTE: If an attempt is made to load a module with an ID of a previously loaded module, a DUPLICATE FILE NAME error occurs (or a result variable contains 19).

This command must be used from deferred mode (while in a running program). Attempting to use it from immediate mode returns an ?ILLEGAL DIRECT ERROR.

LOAD PRINT

To display a table of loaded modules, use:

```
& LOAD PRINT
```

A typical table might look like this:

```
0 I=$0000 L=$0800 A=$9200 14-Nov-91 OMM 1.3
1 I=$7761 L=$0D00 A=$8500 17-Oct-91 AmperWorks 3.0
2 I=$776D L=$0A00 A=$7B00 13-Nov-91 ModemWorks 3.0
3 I=$7474 L=$0300 A=$7800 12-Nov-91 TimeGS 1.0
4 I=$7463 L=$0500 A=$7300 13-Nov-91 Console 1.0
5 I=$6D74 L=$0200 A=$7100 13-Nov-91 Terminal 1.0
6 I=$735D L=$0300 A=$6E00 12-Nov-91 Store256 1.0
7 I=$7470 L=$0500 A=$6900 12-Nov-91 SerialGS 1.0
8 I=$746D L=$0500 A=$6400 13-Nov-91 Intel 9600EX 1.0
```

You can select the numeric format used by including a 0 or 1 argument:

```
& LOAD PRINT 0 :REM 0 selects DECIMAL format
& LOAD PRINT 1 :REM 1 selects HEX format (default)
```

When an argument is given, nothing is printed—the formatting mode is set for subsequent &LOAD PRINT displays.

LOAD FRE

To remove a module from memory, use LOAD FRE:

& LOAD FRE ***numexp***

where *numexp* is the ID of the module. The ID can also be the index number. For example, the AmperWorks module shown in the table above could be freed using either of these commands:

```
& LOAD FRE 1
& LOAD FRE 30561
```

The index number is 1, and the ID number for AmperWorks is 30561 ($7761). The ID of a module will always be greater than 255, so anything less than 256 is considered the index of a loaded module.

To remove everything—the OMM and any modules—use:

```
& LOAD FRE
```

When the ID argument is omitted, an ID of zero is assumed. This shuts down the OMM and all modules loaded. Index 0 references the Loader itself, and when the Loader goes, everything goes.

LOAD TRACE/ NOTRACE

Each time a module is loaded with LOAD GET, or removed with LOAD FRE, the OMM displays the module's information entry.

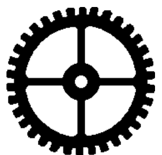
To show a module's entry when loaded or removed, use:

```
& LOAD TRACE
```

To keep the OMM "silent" when modules are loaded or removed, use:

```
& LOAD NOTRACE
```

LOAD PEEK



LOAD PEEK obtains OMM information about loaded modules. It requires a number to specify the type of information desired. Additional arguments may be needed depending on the type of information requested. Examples:

```
& LOAD PEEK 0, N
```

LOAD PEEK 0 returns a count of loaded modules into the numeric variable which follows.

```
& LOAD PEEK 1, ID, A$
```

LOAD PEEK 1 returns a module's information entry. The module is identified by ID (or index), and the entry is returned in a string variable. If an invalid ID or index is specified, an empty string is returned. The numbers are in hexadecimal, unless the &LOAD PRINT 0 command is used first.

LOAD CALL

This is an advanced OMM command, typically used when testing a module in development. Use LOAD CALL to pass a message and a function number to a module from within BASIC. (Messages and functions are discussed in Chapter Two, "Building A Module"). Example:

```
& LOAD CALL ID, message, function [, ... ]
```

This command requires at least three arguments: the ID (or index) of the module, a message code number, and a function number. If the intended function requires any extra parameters they can be included as needed.

Sample Programs

The sample program below illustrates the process of working with a module once the OMM is active. It uses the HexDec module to perform hex-to-decimal and decimal-to-hex number conversions. When done, it removes the HexDec module.

```
100  & LOAD GET "/OMM/Modules/HexDec"
110  PRINT : PRINT "HexDec module loaded"
120  PRINT : INPUT "Enter a decimal number: ";D
130  & HEX(D),H$
140  PRINT "The hex equivalent of ";D;" is $";H$
150  PRINT : INPUT "Enter a hex number: $";H$
160  & DEC(H$),D
170  PRINT "The decimal equivalent of $";H$;" is ";D
180  PRINT
190  & LOAD PEEK 0, LAST
200  & LOAD FRE LAST
210  PRINT : PRINT "HexDec module released"
```

Additional sample programs can be found on the OMM diskette.

Building A Module

This chapter is for assembly language programmers interested in creating their own modules for use with the OMM. The OMM is perfect for the Applesoft programmer with machine language routines in excess of 200 bytes. Entire integrated systems can be built based on the OMM.

OMM Features

Cooperating with ProDOS 8 and ProDOS BASIC, the OMM offers these services:

- ◊ Manages machine language routines (code modules)
- ◊ Allows code modules to peacefully coexist in memory
- ◊ Makes efficient use of memory
- ◊ Protects program modules from being overwritten
- ◊ Quickly relocates modules from disk into memory
- ◊ Allows for loading and unloading of modules as needed
- ◊ Provides intermodule communication (IMC) technology
- ◊ Includes utility routines useful to most modules
- ◊ Supports built-in ampersand command parsing
- ◊ Dynamically relocates modules to reclaim unused memory
- ◊ Compatible with ProDOS, BASIC.System, and Applesoft

The OMM does some neat things that, until now, could not be integrated into the operating system under ProDOS BASIC. The ability to parse ampersand commands makes it easy to add machine language routines to Applesoft. And, since modules can communicate with one another, the possibilities are endless for a highly integrated environment.

Module Format

A module follows a specific internal format, composed of four major parts:

- ◊ Header
- ◊ Code section
- ◊ Immediate mode reference table
- ◊ Data section

A condensed layout of a typical module would follow this source code outline:

```
HEADER  equ     *       ;16-byte (version 0) header
        :
START   equ     *       ;start of module code
        :
        db      $00     ;code ends with a $00 byte
IMMED   equ     *       ;immediate mode table starts
        :
        dw      $0000   ;ends with two $00 bytes
DATA    equ     *       ;data section starts
        :
END     equ     *       ;end of module
```

(For example purposes, *db* is an assembler pseudo-opcode that defines one byte. The *dw* pseudo-opcode defines one word — two bytes or 16-bits.)

Header

The header consists of eight fields. Each field is 16-bits in size:

```
HEADER  equ     *           ;start of header
hVERS   dw      $0000       ;Loader header version
hID     dw      $7475       ;16-bit unique ID number
hSIZE   dw      END-START   ;size of the module
hORG    dw      START       ;origin at start of module
hAMPC   dw      AMPERCT     ;ampersand command table
hKIND   dw      $0000       ;module kind
hRSRV1  dw      $0000       ;reserved for future use
hRSRV2  dw      $0000       ;reserved for future use
```

hVERS. Normally $0000. If your module uses inline, two-byte BRK instructions, use a version of $0001.

hID. A unique 16-bit ID number that identifies the module.

hSIZE. The size of the module (not including the header). This is easily computed at assembly time by using labels at the beginning (START) and end (END) of your program to calculate the difference.

hORG. The value of the program counter at the START of the module. This value must always be page aligned. This means that the code at START must begin on a page boundary ($1000, $1100, $4600, etc., where the low-byte of the address is zero). To do this with most assemblers, set the ORG to a value like $0FF0 before the header. The module's header, being 16 bytes, places the code at the START of the program at $1000.

hAMPC. The address of an optional table of data and tokens for processing ampersand commands. If no ampersand parsing is to be done for the module, set this field to $0000. If a table address is given, the table itself is located in the data section of the program and must follow this format:

```
AMPERCT asc     'TYPE'
        db      $00
        asc     'SPOOL'
        db      $00
        db      $FF
```

(*asc* is a pseudo-opcode that inserts a stream of characters).

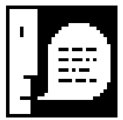
Each command name (i.e., "TYPE" or "SPOOL") is stored as ASCII text (MSB off), and is terminated by a $00 byte. Applesoft tokens can be used in addition to ASCII text strings. The table itself is terminated by one $FF byte.

hKIND. Use $0000 — the OMM currently supports a single format.

hRSRV1 and **hRSRV2**. Reserved for future use. They both should be $0000.

Code Section

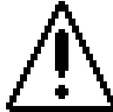
The code in the START segment of the module cannot contain any data storage. Only valid opcodes and their operands can be present. In addition, you cannot make immediate references to labels within the program, as they cannot be resolved correctly by the OMM's code relocator. The end of this section is terminated by a $00 byte (or BRK). This informs the relocator that the immediate mode table begins.



NOTE: If your module has a version 1 header (hVERS is $0001), the code section ends with three (3) zero bytes.

Immediate Mode Table

The IMMED table consists of two-byte addresses that reference locations within the module (between START and END). The table ends with two $00 bytes. This allows code to access labels within the module using absolute references.



CAUTION: An immediate reference to a label within a module is not allowed:

```
        lda     #<LABEL         ;this is NOT allowed!
        ldy     #>LABEL         ;ERROR!  ERROR!
        :
LABEL   equ     *
```

The OMM code relocator cannot resolve internal immediate references. To the relocator, it appears that the code is simply loading the A and Y registers with two constant values—whatever the assembler assigned to them. Since LABEL can be located anywhere in memory at runtime, the value stored in Y will be invalid. To overcome this, the address of LABEL can be stored in the IMMED table:

```
IMMED   equ     *
a_LABEL dw      LABEL
```

The code can now load A and Y with the runtime address of LABEL using absolute references to the two bytes at a_LABEL:

```
        lda     a_LABEL
        ldy     a_LABEL+1
```

Data Section

The DATA section can contain anything. The relocator does not touch it. Code is stored in the DATA section may not run properly if it makes references within the module. Code can be stored here as long as it makes no references to itself (other than conditional branches), or makes references to locations outside the module (e.g., ProDOS, Applesoft, monitor ROM, etc.).

IMC

The OMM features a technology called intermodule communication (IMC). IMC allows modules to send messages to each other to exchange information or to initiate special tasks. The OMM itself makes use of this in order to instruct the module to perform certain OMM-related tasks, such as initializing itself after it has been loaded into memory.

This assumes, of course, that the module begins with code to process commands and messages from the OMM (or other modules). At the START of a typical module, the following code is included to operate correctly in the OMM environment:

```
START   cmp     #MSG_AMPR       ;ampersand command?
        beq     doampr          ;yes

        cmp     #MSG_INFO       ;get INFO string?
        beq     doinfo          ;yes

        cmp     #MSG_INIT       ;INIT request?
        beq     doinit          ;yes

        cmp     #MSG_QUIT       ;QUIT request?
        beq     doquit  ;yes

        rts
```

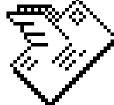
(The values for the MSG constants are kept in an interface file that the source brings into the assembly before this point. They're discussed in the next section).

The OMM calls a module with a command code in the accumulator (A-register), and it is up to the module to service the command based on the message being sent. While there are many different kinds of messages that a module may elect to service, all modules must be able to service the INFO message.

Messages

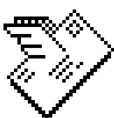
The messages that the OMM may send are:

MSG_INIT	Initialize module
MSG_QUIT	Quit (shutdown) module
MSG_AMPR	Execute ampersand service routine
MSG_USER	Intermodule service request
MSG_REL1	Alert module that it is about to be relocated
MSG_REL2	Alert module after it has been relocated
MSG_KILL	Notification that a module is to be killed
MSG_DIED	Notification that a module just died
MSG_BORN	Notification that a module has just been born
MSG_IDLE	Idle event for module (not implemented yet)
MSG_INFO	Asks module to return an information string



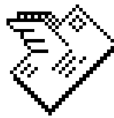
MSG_INIT ($00)

A module receives the MSG_INIT message from the OMM as soon as it is established into memory. The module can use this opportunity to set itself up before handling other messages. This is the second message that the module will get, the first being the MSG_INFO message.



MSG_QUIT ($01)

This is the last message a module gets before it is removed from the system. The module should close any open files, and disable interrupt sources it turned on. In essence, it must clean house and put all the toys away. In doing this, it may instruct other modules to prepare for its demise by passing information to them.

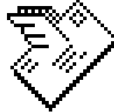


MSG_AMPR ($02)

If an ampersand command is encountered in Applesoft, control passes to the OMM where it attempts to locate the command in tables in each module. When found, the OMM sends the MSG_AMPR message to the corresponding module.

The Y-register contains a number that denotes the index of the ampersand command in the table. If Y is zero, the first command in the table was issued. If Y were five, then the sixth

command in the table had been encountered. In this manner, Y can be used as an index into a table of addresses for ampersand service routines.

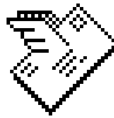


MSG_USER ($03)

When another module wants to communicate via IMC, it performs the following steps:

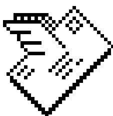
```
ldx     moduleIndex     ;X = index to the module
ldy     #A_FUNCTION     ;Y = a user-defined function
jsr     OMMVEC          ;call the OMM
```

The X-register holds the index of the module to receive the message. (Obtaining an index to a module is discussed later). The Y-register holds a value that is passed to the module. This value is known by both the calling module and the receiving module. The receiving module's START code would detect the MSG_USER code in the A-register, pass control to its handler, which services the function based on the number in the Y-register.



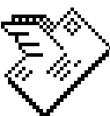
MSG_REL1 ($04)

When a module is purged, any modules that are to be relocated to fill in the gap are sent the REL1 message. This message is sent *before* the relocation begins. The module can take the opportunity to do whatever is necessary to allow the OMM to relocate the module properly. It may also use this time to inform other modules that it is about to move, in case the other modules may expect the to-be-relocated module to stay in place due to absolute references. The module may also wish to remove any interrupt vectors allocated to it at this time.



MSG_REL2 ($05)

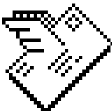
After a module is relocated it receives the REL2 message. The module can basically reverse the steps taken for the MSG_REL1 event in order to return to its normal state.



MSG_KILL ($06)

This message is sent to all loaded modules just prior to a module being freed, including the to-be-killed module. This allows all the modules to cooperate in whatever measures are necessary to prepare for the loss of a module which is still alive at this point.

Upon entry, the Y register will contain the index to the module being freed.

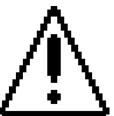


MSG_DIED ($07)

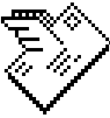
After a module is removed, this message is sent to all loaded modules. If a module used IMC with the dead module, it will know that is no longer in the system, and no further IMC calls can be made to it.

Upon entry, the Y register will contain the index to the module that was freed.

Each module using IMC should call the OMM to get new indexes for the remaining modules for which it requires indexes.



WARNING: Indexes shift when a module is freed. If a module assumes that an IMC index is valid, the system will probably crash. It is important to call the OMM to get new indexes after any module is freed. This can only be done safely after receiving a MSG_DIED message.

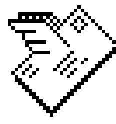


MSG_BORN ($08)

After a new module has been loaded, the BORN message is sent to all loaded drivers (including the one just loaded). The unique 16-bit ID of the new module is placed at $3C.

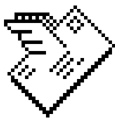
Upon entry, the Y register will contain the index to new module.

At this time, each module using IMC should call the OMM to get indexes for the modules for which it requires indexes. In this manner, the loading sequence of modules can be completely arbitrary, yet all modules will know when new ones come to life.



MSG_IDLE ($09)

This message is reserved for future use.



MSG_INFO ($0A)

The OMM sends the INFO message to a module immediately after it is loaded. All modules must service this request for the OMM to operate correctly. The module should perform only one task at this time, and that is to place the address of an information string into locations $3C and $3D and return.

The information string contains the revision date, title, and version of the module, in high-order ASCII (bit 7 set) using the following format:

```
16-May-90 AmperWorks 3.0
```

The version number should follow format of version numbers as described in Chapter One.

The string and the MSG_INFO handler must remain intact and unmodified for the duration of the module's life in memory, as it can be requested at any time.

Utility Functions

To modules, the OMM itself appears to be a module. Using IMC, modules can call upon utility routines built into the OMM to simplify common tasks:

OMM_GETID	Gets the index of a module based on ID
OMM_XOAMP	Executes original ampersand vector
OMM_FREE	Frees a module
OMM_PUTWORD	Stores a word value to a BASIC variable
OMM_PUTSTR	Stores a string to a BASIC variable
OMM_GETSTR	Gets the descriptor of a string expression
OMM_PADDEC	Prints a word in decimal, right justified
OMM_C2PSTR	Copies string to a Pascal formatted string
OMM_COUNT	Returns the count of loaded modules
OMM_GETINFO	Gets the info string of a module

These functions are performed by calling the OMM just as you would call another module. The OMM's index is represented by the assembler equate called OMM_ID ($00). Calls to the OMM are made by calling OMMVEC ($3F8). Example:

```
ldy     #function       ;Y = OMM function
ldx     #OMM_ID         ;X = OMM index
jsr     OMMVEC          ;call the OMM vector
```

Some functions require arguments or return arguments in memory locations, flags, or registers.

OMM_GETID ($00)

For a module to communicate with another, it must be able to tell the OMM which module it wishes to reference. This is done by using the loaded module's index. A program can obtain the index for a module by asking the OMM to look it up based on its unique ID. To obtain the index, place the ID of the module into locations $3C and $3D, and then call the OMM with the OMM_GETID function.

Getting a Number from BASIC

A function for obtaining the numeric value from an Applesoft expression or variable is not included in the OMM's repertoire since it is easy to do this using two methods listed here:

```
jsr  GETBYTE    ;($E6F8) Put 8-bit value into X
```

or

```
jsr  FRMNUM     ;($DD67) Evaluates a 16-bit number
jsr  GETADR     ;($E752) Stores word at $50 and $51
                ;  also: Y = low byte, A = high byte
```

For example, to obtain the index for the AmperWorks module, the following code can be used:

```
lda     #'a'            ;AmperWorks ID is 'aw'
sta     $3C             ; (or $7761)
lda     #'w'
sta     $3D
ldy     #OMM_GETID      ;Y = function number
ldx     #OMM_ID         ;X = OMM index
jsr     OMMVEC          ;call the OMM
stx     awIndex         ;save the index
```

If the ID was found, the module's index is returned in the X-register and the carry flag is clear. The OMM returns with $00 in the X-register and the carry flag set if the ID search fails.

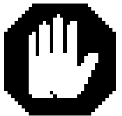
OMM_XOAMP ($01)

This function causes the OMM to call the address of the original ampersand handler installed before the OMM was launched. It is included in case your module does further ampersand parsing, then discovers that the command the user has given is not one of its own.

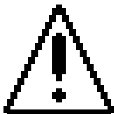
OMM_FREE ($02)

A module can instruct the OMM to free another module by placing the index of the module to purge at location $3C. Example:

```
lda     awIndex         ;get rid of AmperWorks
sta     $3C             ;put the index here
ldy     #OMM_FREE       ;free function number
ldx     #OMM_ID         ;the OMM's index
jsr     OMMVEC          ;call the OMM
```



WARNING: OMM_FREE should never *be used by a module that resides lower in memory than the module to purge. When the OMM is finished relocating the remaining modules, control returns to the calling module. If the module has moved, the return address is invalid, and the system crashes.*



CAUTION: This function should be avoided, unless you really know what you're doing, as the order in which modules are loaded cannot be safely assumed. It is possible to push a valid return address onto the stack (e.g., the RESET handler at $FA62) and then JMP to OMMVEC. This ensures a valid return address and no crashing after removing a module.

OMM_PUTWORD ($03)

To store a numeric value to an Applesoft variable (pointed to by the Applesoft text pointer), the OMM_PUTWORD function can be used. Example:

```
lda   #5       ;store a 5 (low byte)
sta   $3C
lda   #0       ;zero (high byte)
sta   $3D
ldy   #OMM_PUTWORD
ldx   #OMM_ID
jsr   OMMVEC
```

Store the value into locations $3C and $3D. If a byte value is to be stored, put the byte into $3C and write $00 to $3D.

OMM_PUTSTR ($04)

To store string data to an Applesoft variable pointed to by the text pointer, use the OMM_PUTSTR function. It requires a string descriptor at location LOWTR ($9B). A string descriptor consists of three bytes of information: a length byte, and two bytes that point to the first character of the string. Example:

```
lda    #15     ;store 15 characters
sta    $9B
lda    #$200   ;location is at input buffer
sta    $9C
lda    #>$200
sta    $9D
ldy    #OMM_PUTSTR
ldx    #OMM_ID
jsr    OMMVEC
```

OMM_GETSTR ($05)

This function evaluates the string expression at the Applesoft text pointer and returns its descriptor in location LOWTR ($9B). LOWTR will hold the length of the string, and at LOWTR+1 is a two-byte pointer to the first character in the string.

OMM_PADDEC ($06)

Printing a value in decimal, right-justified (space-padded), is a chore for any machine language program. This function makes it easy. Put the width of the field, in which the number should be padded, into location LOWTR ($9B). Put the value into locations $9C and $9D (LOWTR+1). Then make this function call. After printing the number through COUT, the cursor follows the last digit printed.

OMM_C2PSTR ($07)

Copies a string of characters to a buffer that will begin with a count byte, followed by the string itself. In other words, it makes a Pascal-formatted string. These are used frequently when dealing with ProDOS pathnames.

The descriptor of the source string is stored at LOWTR ($9B), which is a count byte followed by a two-byte pointer to the string. Put the target buffer address into locations $3C and $3D.

OMM_COUNT ($08)

This function returns the count of the loaded modules in the A-register.

Tip: Integration With Many Modules

If your module must keep track of indexes for one or more modules, have it service the MSG_BORN and MSG_DIED messages by a single routine. That routine would simply call the OMM to obtain indexes for the modules it desires with the OMM_GETID function.

f

OMM_GETINFO ($09)

Use this function to obtain the info string from a module. Before making the call, put the module's index (not ID) number in the byte at $3C:

```
lda     moduleIndex
sta     $3C
ldy     #OMM_GETINFO
ldx     #OMM_ID
jsr     OMMVEC
```

On return, $3C and $3D contains a pointer to the requested module's info string.

Interfaces & Sources

The OMM disk comes with interface files containing equates for the OMM. If you use ORCA/M or APW, the ORCA directory contains a file called OMM.EQU with all the equates you need. If you use Merlin, the MERLIN directory contains the OMM.S file with equates for Merlin.

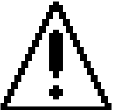
Source code templates for your own modules can be found in the ORCA (TEMPL.ASM) and MERLIN (TEMPL.S) directories. These are skeleton programs with a "fill in the blanks" format to make it easier to create new modules. Also included in these directories is a sample program module (hex/decimal number conversion) with sources for both ORCA/M and Merlin.

Buffer Space

Modules should include internal buffers for space needed during the course of their lifetime. This is done by defining storage at assembly time. However, it may be safe to allocate dynamic buffers at runtime by calling ProDOS BASIC's "GetBufr" routine, but only if the buffer is used temporarily and then discarded with "FreeBufr" between OMM service calls. Long-term preservation of a buffer allocated through ProDOS BASIC is not guaranteed due to the nature of the OMM and the way it dynamically manages modules.

Absolute References

Care should be taken when an absolute reference is made to a location within a module. Since modules are "slippery", pointers to data items in other modules may become invalid.



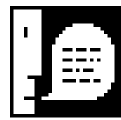
CAUTION: Pointers to interrupt service routines (in modules that have been moved) can have disastrous effects.

Whenever possible, all communication between modules that share common data should use the IMC to transfer pointers to data items. When modules shift, utilize IMC to update the pointers for those data items. The OMM provides mechanisms for updating absolute references in movable code with the REL1 and REL2 messages.

Opcode Usage

Your programs can use 6502, 65C02, and even 65816 instructions. But, 65816 programmers note that the OMM cannot properly relocate 65816 code that uses the 16-bit immediate mode references for the accumulator and index registers. It is possible to place such code into the data section where it will execute correctly, so long it does not use absolute references to locations within the module.

If you need to use a BRK instruction in your code section, be sure to use a version 1 header (hVERS = $0001). Format considerations of a version 1 module are the same as for version 0, except that the end of the code section ends with three zero bytes instead of one. Also, inline BRKs occupy two bytes, not just one. Users of the 6502 and 65C02 should, therefore, use two BRK instructions in a row. 65816 programmers should use the standard two-byte BRK instruction.



NOTE: Programs that include 65816 opcodes will relocate properly even when run on a CPU that does not support 65816 instructions. The OMM allows you to put CPU dependent code into your programs. You must, however, make sure that the program can run on the machine in use.

AmperWorks

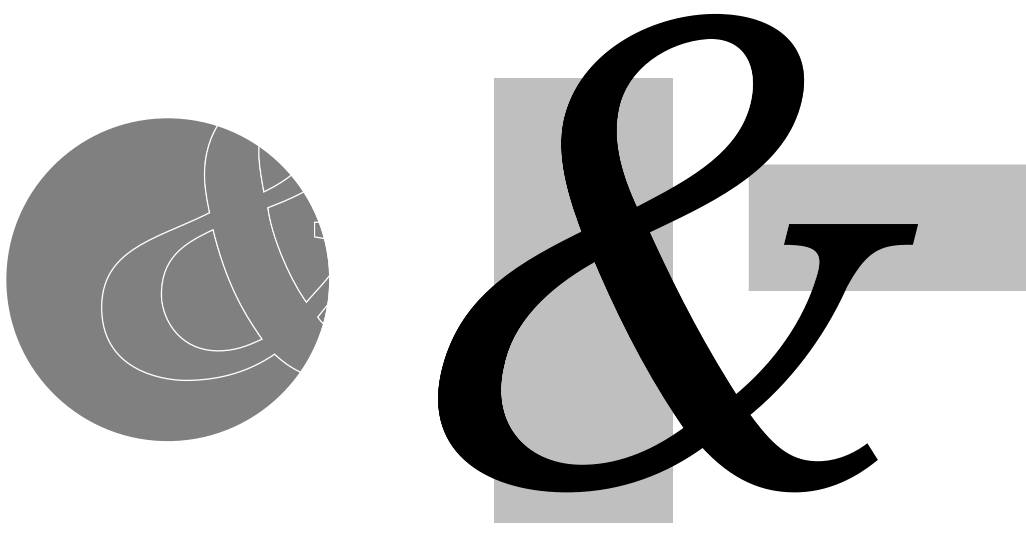
This chapter describes AmperWorks, an OMM module providing extended commands for Applesoft. Using AmperWorks commands in your BASIC programs can speed them up dramatically, as well as provide functionality that is impossible or difficult to do with BASIC alone.

Introduction

Most AmperWorks commands are stored in the AmperWorks module. Use the OMM's &LOAD GET command to load it into memory.

A few AmperWorks commands are kept in other modules. For example, the string storage commands (e.g., &STORE and &RESTORE) are found in Store256, Store512, and StoreGS. The &TIME command is kept in the Time and TimeGS modules. If your program will use these commands, decide which module is best for your computer and use &LOAD GET to load it into memory.

Abbreviations are used for different types of arguments that AmperWorks commands require. Refer to Appendix E, "BASIC Syntax", for an explanation of the abbreviations.



Command Summary

AmperWorks consists of the following commands:

Command	Description
/	Get file information
\	Set file information
<	Return the parent path of a file
ADD	Add a file to the end of another file
ASC	Convert a string to ASCII text
COPY	Copy a file to another file
ERASE	Erase an array from memory
FILES	Put a directory's filenames into an array
GET	Get characters into a string
HLIN	Draw a horizontal line with a character
LCASE	Convert a string to lowercase
LEFT$	Left-justify a string within a field
LIST	Display the contents of a file
MID$	Change the mid-portion of a string
MLI	Perform a ProDOS MLI function
ONERR	Fix ONERR bug and get error information
POKE	Poke a list of values into memory
POP	Reset Applesoft's stack
POS	Find a pattern within a string
PRINT	Print to the screen during file output
READ	Read characters into a string
REPT	Start a REPT-UNTIL loop
RESTORE	Restore a storage cell to a string
RESTORE GOTO	Set the next DATA statement line number
RIGHT$	Right-justify a string within a field
SPC	Strip spaces from the ends of a string
SRT	Sort an array
STORE	Store a string into a memory storage cell
STORE CLEAR	Erase all strings in storage
SWAP	Swap the values of two variables
TFILES	Put a directory's filenames into an array
TIME	Return the date and time
UCASE	Convert a string to uppercase
UNTIL	Mark the end of a REPT-UNTIL loop
VAL	Evaluate an expression and return the result
VLIN	Draw a vertical line with a character

/
(Get Info)

`& / strexp, strvar`

Gets information on the file described by *strexp*. Eighteen characters of information are returned in *strvar*. If *strvar* comes back empty (equal to “”), then the pathname was invalid or non-existent. This is useful as an alternative to ProDOS BASIC’s VERIFY command.

The information returned follows the structure of ProDOS’s GET_FILE_INFO parameter table. Since the information string contains control characters, your programs can convert their ASCII values to meaningful numbers.

`MID$ (INFO$, 1,1) =`	Parameter Count
`MID$ (INFO$, 2,2) =`	Address of Pathname
`MID$ (INFO$, 4,1) =`	Access Bits
`MID$ (INFO$, 5,1) =`	Filetype
`MID$ (INFO$, 6,2) =`	Auxiliary Filetype
`MID$ (INFO$, 8,1) =`	Storage Type
`MID$ (INFO$, 9,2) =`	Blocks Used
`MID$ (INFO$,11,2) =`	Modification Date
`MID$ (INFO$,13,2) =`	Modification Time
`MID$ (INFO$,15,2) =`	Creation Date
`MID$ (INFO$,17,2) =`	Creation Time

Sample Program

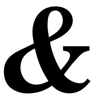
```
10 & / "/GRACELAND/ELVIS", I$
20 IF I$ = "" THEN PRINT "Elvis taken by aliens!"
30 IF I$ > "" THEN PRINT "Elvis lives!"
```

NOTE: To use a partial pathname, the ProDOS prefix must be set.

Also See

\

\ (Set Info)



& \ *strexp, strvar*

Sets information on the file described by *strexp*. Eighteen characters in *strvar* must follow the structure of ProDOS's SET_FILE_INFO parameter table (as illustrated on the previous page).

Sample Program

```
10 & / "DOWNLOAD", I$ : REM Get info on DOWNLOAD
20 & MID$ (I$, 5) = CHR$(6): REM Change filetype
30 & \ "DOWNLOAD", I$ : REM Set info on DOWNLOAD
```

Sample Run

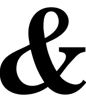
Line 10 gets file information on DOWNLOAD and places it into I$. Line 20 changes the fifth character in I$, the filetype field for the ProDOS SET_FILE_INFO parameter list. Line 30 uses &\ to set new information on DOWNLOAD. The ASCII value of 6 used in Line 20 sets DOWNLOAD to a BIN file.

NOTES: AmperWorks ignores the first three characters of the information string since they may have different meanings in future versions of ProDOS BASIC. To use a partial pathname with this command, the prefix must be set.

Also See

/

< (Parent Directory)



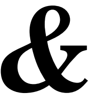
& < *strexp*, *strvar*

Separates the prefix from the file name in the complete pathname specified by *strexp*. The sample program demonstrates its features.

Sample Program

```
10 PN$ = "/a/dev/mw/install"
20 PRINT "Pathname:", PN$
30 & < PN$, P$
40 PRINT "Path:", P$
50 N$ = MID$ (PN$, LEN (P$) + 2)
60 PRINT "Name:", N$

]RUN

Pathname:      /a/dev/mw/install
Path:          /a/dev/mw
Name:          install
```

ADD



& ADD (*strexp1* TO *strexp2*)

Appends the filename described by *strexp1* to the filename described by *strexp2*. If the target file does not exist, it is created and the contents of the source file are copied. Any filetype of any size may be added—the target file retains its original file information. The ProDOS prefix must be set in order to use a partial pathname.

Samples

```
& ADD ("/ram/temp" TO "/disk/logfile")
& ADD (FILE$(1) TO FILE$(2))
```

If the disk becomes full, the target file is *not* deleted, nor is it left with part of the source file appended to it. AmperWorks leaves the target file unchanged in the event of a disk error.

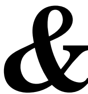
Also See
COPY

ASC

& ASC *strvar*

Converts the characters in a string variable to standard ASCII values (the high-bits are cleared). This is useful for programs working with strings containing non-ASCII characters.

COPY



& COPY (*strexp1* TO *strexp2*)

Copies the filename described by *strexp1* to the filename described by *strexp2*. If the target file exists, it is overwritten. Any file of any type and size may be copied. The ProDOS prefix must be set in order to use a partial pathname.

Samples

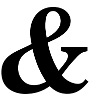
```
& COPY ("/dev/test" TO "/dev/test.bak")
& COPY (TAKEMEOUT$ TO THEBALLGAME$)
```

If the disk becomes full during a COPY, the operation is cancelled and the target file will not exist.

Also See

ADD

ERASE



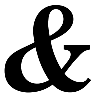
& ERASE (*arrayname*)

Removes an array (of any kind or dimension) from memory, allowing you to create and erase arrays as needed, giving your programs additional free memory. ERASE requires the name of an array only—no subscript is required.

Sample Program

```
10 DIM F$(300)
20 & FILES ("/RAM5", F$), N
30 GOSUB 1000: REM Work with the F$ array
40 & ERASE (F$): REM Now erase it from existence
```

FILES



```
& FILES (strexp, strvar [,numexp1, numexp2]),
        numvar1 [, numvar2]
```

Reads filenames from the directory described by *strexp* and stores them in an array described by *strvar*.

The number of names placed in the array is returned in *numvar1*, and the actual number of files residing in the directory (matching any selection criteria) is returned in the optional *numvar2*. *strvar* must be DIMensioned before using FILES to avoid an OUT OF DATA ERROR.

The optional *numexp1* is a filetype filter, used as search criteria. If *numexp* is 255 for example, only SYS-type names are placed in the array (the numeric value for a SYStem file is 255). If *numexp* is a negative number, the logic is reversed, placing all names in the array except for those having types equal to the absolute value of *numexp*. For example, to gather all the names that are not subdirectories (type 15), *numexp* would be -15.

The optional *numexp2* is an invisibility filter code:

0 Include visible files only (default)
1 Include all files
2 Include invisible files only

Example

```
& FILES ("/A", F$, , 2), N
```

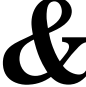
This reads all invisible names found on /A, placing them into the F$ array. The count is returned in N. Since the type filter option is omitted, all types are possible. If you include a type of 4 (TXT), it finds all invisible text files.

FILES cannot be used in immediate mode since the contents of the input buffer are destroyed. The ProDOS prefix must be set in order to use a partial pathname.

Also See

TFILES

GET



```
& GET [(numexp [,strexp])] [,"..."] [,strvar]
```

Gets data from an opened file. AmperWorks' GET is similar to Applesoft's, except it allows input of multiple characters, including commas, colons, and quotes. And unlike Applesoft's INPUT, it does not affect the display. Input is terminated after a carriage return is entered, or when the character input count reaches 255 (or the optional *numexp* limit). It follows the same format for its arguments as discussed in the &READ command.

Samples

```
& GET A$  :REM Gets up to 255 characters into A$
& GET (5)  :REM Gets 5 characters (discarded)
& GET (1),"Any Key"  :REM Gets 1 character w/prompt
& GET (0),"Press Return"  :REM Waits for a RETURN
& GET (15),"Code:",A$  :REM Gets up to 15 into A$
```

Also See

READ

HLIN

```
& HLIN numexp1,numexp2
```

Draws a horizontal line, the length determined by *numexp1,* with the character whose ASCII code is determined by *numexp2*. Both arguments must be numeric values from 0 to 255.

Samples

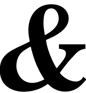
```
& HLIN 10,ASC("*")              **********
& HLIN 15,65                    AAAAAAAAAAAAAAA
& HLIN 20,65 + 1                BBBBBBBBBBBBBBBBBBBB
```

Creative use of &HLIN and &VLIN allows you to quickly draw boxes and borders.

Also See

VLIN

LCASE



& LCASE (*strvar*)

Converts a string variable's uppercase letters to lowercase.

Sample Program

```
10 A$ = MID$("AmperWorks", 1)
20 PRINT A$
30 & LCASE (A$)
40 PRINT A$

]RUN

AmperWorks
amperworks
```

Also See

UCASE

LIST

& LIST *strexp*

LIST displays the contents of the file described by *strexp*, useful for with files containing readable text. The listing may be paused with [control]-S and restarted with any key. Pressing [esc] stops the listing.

Samples

```
& LIST "/mail/msg.1234"

& LIST FILE$
```

This command cannot be used in immediate mode since the contents of the input buffer are destroyed. The ProDOS prefix must be set in order to use a partial pathname.

LEFT$

`& LEFT$ (`*`strexp, numexp1`*` [,`*`numexp2`*`]), `*`strvar`*

Left-justifies a string within a specified width—padding a string so that its length becomes a fixed value. This is useful when displaying tabular information, or when writing data to a random access text file.

strexp is the source string to be left-justified into *strvar*. The width of *strvar* is determined by *numexp1*, and is a value from 1 to 255. Spaces are used for padding, unless the optional *numexp2* is used; its ASCII value becomes the padding character. For example, to left-justify a string into a 20-character field with periods, use:

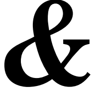
```
& LEFT$ (A$, 20, 46), B$
```

(46 is the ASCII value for the period character).

If the length of *strexp* is greater than the field width given in *numexp1*, the contents of *strexp* are truncated before being placed into *strvar*.

Also See
RIGHT$, SPC

MID$



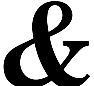
`& MID$ (`*`strvar, numexp1`*` [,`*`numexp2`*`]) = `*`strexp`*

Replaces the middle portion of a string with the *strexp* that follows the equal sign. It overlays *strvar* at the position specified by *numexp1*. The optional *numexp2* is the number of characters to overlay. If *numexp2* is omitted, the length of *strexp* is assumed. For example, if A$ = “AAAAA”, then the following changes it to “AZZZA”:

```
& MID$ (A$, 2, 3) = "ZZZZZZZZZ"
```

Also See
POS

MLI



`& MLI (numexp1, numexp2), numvar`

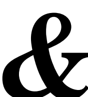
Performs a ProDOS Machine Language Interface (MLI) command specified by *numexp1*. The address of the MLI parameter table is given in *numexp2*. After the MLI command is performed, the result code is stored in *numvar*. It is useful for performing commands that ProDOS BASIC does not already provide, such as retrieving a file's length in bytes, reading a block of data from disk, and moving the "mark" (position) in an open file to a new offset, among many others. Further explanation of the MLI goes beyond the scope of this manual, though there are many good books on the subject.

Sample Program

```
10 TBL = 768 : L = TBL + 2
20  DEF FN PL(X) = PEEK(X) + PEEK(X+1) * 256
         + PEEK(X+2) * 65536
30   PRINT CHR$(4) "OPEN TEST.FILE,TTXT"
40 REF = PEEK (48848)
50  & POKE TBL, 2, REF
60  & MLI (209, TBL), ERR : REM Get_EOF
70  PRINT CHR$(4) "CLOSE"
80  IF ERR THEN PRINT "Error: " ERR: END
90  PRINT "File's length is " FN PL(L)

]RUN

File's length is 32768
```

MLI Command	Hex	Dec	MLI Command	Hex	Dec
ALLOC_INT	$40	64	GET_PREFIX	$C7	199
DEALLOC_INT	$41	65	OPEN	$C8	200
QUIT	$65	101	NEWLINE	$C9	201
READ_BLOCK	$80	128	READ	$CA	202
WRITE_BLOCK	$81	129	WRITE	$CB	203
GET_TIME	$82	130	CLOSE	$CC	204
CREATE	$C0	192	FLUSH	$CD	205
DESTROY	$C1	193	SET_MARK	$CE	206
RENAME	$C2	194	GET_MARK	$CF	207
SET_INFO	$C3	195	SET_EOF	$D0	208
GET_INFO	$C4	196	GET_EOF	$D1	209
ONLINE	$C5	197	SET_BUF	$D2	210
SET_PREFIX	$C6	198	GET_BUF	$D3	211

ONERR



& ONERR [*numvar1, numvar2*]

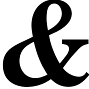
Performs two special tasks when placed at the beginning of an ONERR GOTO error handling routine. First, it fixes the stack that Applesoft's error handling leaves corrupted, avoiding subsequent RETURN WITHOUT GOSUB errors in subroutines. Second, if included, the error code is placed in *numvar1* and the program line where the error occurred is stored in *numvar2*.

Sample Program

```
10 ONERR GOTO 40
20 PRIMT "Misspelled PRINT, dummy!"
30 END
40 & ONERR CODE, LINE
50 PRINT "Error #"; CODE;
60 PRINT " in line "; LINE

]RUN

Error #16 in line 20
```

Since &ONERR affects Applesoft's stack, never issue &ONERR unless an error has occurred.

POKE



& POKE *numexp1, numexp2* [*,numexp3...*]

Pokes multiple numeric values into consecutive memory locations starting at the address specified by *numexp1*. Example:

```
& POKE 768,173,31,192,141,67,3,141, ...etc.
```

The above example stores 173 at location 768, 31 at location 769, 192 at location 770, and so on. This allows small assembly language programs to be stuffed quickly into memory.

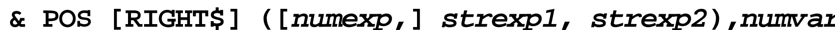
As long as their values are within 0 to 255, you can also POKE numeric variables and expressions into memory with &POKE.

POP

& POP

Removes all GOSUB-RETURNs, FOR-NEXT loops, and REPT-UNTIL loops from the stack. By contrast, Applesoft's POP statement removes only the most recent GOSUB's "return" line from the stack. Use POP at any point in your program where the normal flow has been incorrectly or artificially diverted.

POS



& POS [RIGHT$] ([*numexp*,] *strexp1*, *strexp2*),*numvar*

Searches for a pattern within a string. It returns the position of *strexp2* within *strexp1* starting at the optional *numexp* argument (the offset from the start of *strexp1*). If a match is found, the position is placed in *numvar*. If no match is found, *numvar* contains zero. Searching starts at the beginning of the string and continues to the end. If the optional RIGHT$ keyword is given, the search begins at the end of the string and works toward the start of the string.

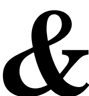
Sample Program

```
10  REM This program searches a string for all the
20  REM 'e' letters and points to each with a marker
30 OLDP = 0: REM Initialize offset
40 A$ = "The Apple IIGS Personal Computer": PRINT A$
50 & POS (OLDP + 1, A$, "e"),P
60 IF NOT P THEN END: REM Stop! No more e's found
70 PRINT SPC( P - OLDP - 1) "^";
80 OLDP = P: GOTO 50

]RUN

The Apple IIGS Personal Computer
  ^     ^       ^             ^
```

Also See

LCASE, UCASE

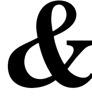
PRINT



& PRINT [...]

This is identical to Applesoft's PRINT, except that it displays its arguments to the screen while a file is open for output. With it, you can write to a file and print to the screen without having to switch off file output.

READ



& READ [(*numexp* [,*strvar*])] [,"..."] [,*strvar*]

Reads input until *return* is entered. Commas, colons, and quotes, normally refused by Applesoft, are allowed. Unlike AmperWorks' GET statement, READ ignores all control characters except for the following:

Code	Key	Action
8	←	Rubout
9	*tab*	Tab to next position (modulo 8)
13	*return*	End of input
23	*control*-W	Delete word
24	*control*-X	Clear input
127	*delete*	Rubout

The optional *numexp* defines the maximum number of characters that may be read. If that number is reached, further input is ignored.

If the optional *strvar* is included after *numexp*, word wrap is enabled. When the total number of characters allowed have been entered, word wrap is performed, stuffing the wrapped characters into the optional *strvar*.

Sample Program

```
10 REM Word Wrap Test Program
20 I$ = "": REM Stuff string starts out empty
30 & REPT
40   & READ (38,I$),":",A$
50 & UNTIL (A$ = "")
```

Sample Run

Text is entered at the ":" prompt. When input reaches column 39, words wrap around the screen to the next line. When this happens, the wrapped word is stored in I$ while the rest of the line is saved in A$. When program flow resumes at Line 40, the contents of I$ are stuffed into the input line, simulating real word wrap. The program stops when return is pressed on a new line.

If *numexp* is negative and a stuff string is present, the stuff string is placed into the input line, but word wrapping is disabled.

If *numexp* is zero, return is the only key accepted.

If (*numexp*) is omitted, up to 255 characters may be entered.

NOTE: The optional prompt string may be used to prompt the user for input. The final string variable is optional—input is discarded.

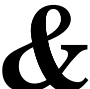
Samples

```
& READ A$ :REM Reads a line of text into A$
& READ (5) :REM Reads 5 characters (discarded)
& READ (1),"Any Key" :REM Reads 1 character
& READ (0),"Press Return" :REM Waits for a return
& READ (15),"Phone: ",A$ :REM Reads 15 into A$
& READ (N),C$ :REM Reads N characters into C$
& READ (78,W$),L$(I) :REM Reads with wordwrap
```

Also See

GET

REPT



```
& REPT
```

Begins a REPT-UNTIL loop. A REPT-UNTIL block is a group of statements or program lines, surrounded by &REPT and &UNTIL. The block executes repeatedly until the expression in the UNTIL statement is true. These blocks may be nested many levels deep.

Sample Program

```
10 N = INT (20 * RND(1)) + 1
20 & REPT
30   & REPT
40      INPUT "Guess my number (1-20) ";X
50   & UNTIL (X >= 1 AND X <= 20)
60   IF X <> N THEN PRINT "Wrong." : FOUND = 0
70   IF X = N THEN PRINT "You got it!": FOUND = 1
80 & UNTIL (FOUND)
```

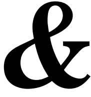
If you must leave a REPT-UNTIL loop prematurely, it is best to *prime* the UNTIL condition, and GOTO the line number where the UNTIL statement can be found. As with FOR-NEXT and GOSUB-RETURN, branching out of a REPT-UNTIL loop leaves garbage on the stack, causing your program to misbehave.

Also See

POP, UNTIL

RESTORE

& RESTORE *numexp* TO *strvar*



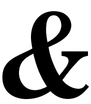
Retrieves a string from the storage cell identified by *numexp*, storing it in *strvar*. This command is available only if a Store module is loaded (e.g., Store256, Store512, or StoreGS).

Also See

STORE, STORE CLEAR

RESTORE GOTO

& RESTORE GOTO ...



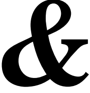
Selects the next program line where DATA is to be read with Applesoft's READ statement. Applesoft's RESTORE always resets the next DATA line to the beginning of your program. AmperWorks gives you more flexibility.

Sample Program

```
10 DATA A,B,C
20 & RESTORE GOTO 50
30 READ A$
40 PRINT A$
50 DATA E,F,G

]RUN

E
```

Applesoft initially sets the next DATA line to the first line in this program. Line 20 uses RESTORE GOTO to change to the next DATA statement in Line 50. Line 30 uses the READ statement, which actually reads DATA from Line 50, even though it has not read the DATA in Line 10 yet. Instead of “A”, an “E” is printed, which proves this sample works.

RIGHT$



& RIGHT$ (*strexp, numexp1* [*,numexp2*]), *strvar*

Right-justifies a string within a specified width, padding it with spaces so that the length becomes a fixed value.

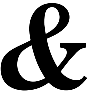
strexp is the source string, right-justified into *strvar*. The width of *strvar* is determined by *numexp1*, and has a value from 1 to 255. Spaces are used for padding, unless the optional *numexp2* is included; its ASCII value is the padding character.

If the length of the string is greater than the field width, the rightmost contents of *strexp* are placed into *strvar*.

Also See

LEFT$, SPC

SPC



& SPC (*strexp* [, *numexp*]), *strvar*

Strips leading and trailing spaces from *strexp* and stores the new string into *strvar*. Spaces are stripped, unless the optional *numexp* is given; characters with its ASCII value are stripped instead.

Sample Program

```
10 A$ = "    This is a test    "
20 PRINT "["; A$; "]"
30 & SPC (A$), A$
40 PRINT "["; A$; "]"

]RUN

[    This is a test    ]
[This is a test]
```

Also See

LEFT$, RIGHT$

SRT (Sort)

& SRT (*arrayname*, *numexp*)

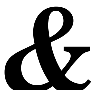
Sorts any kind of single-dimension array described by *arrayname* (the variable name without a subscript). The number of elements to sort is specified by *numexp*. The array must be DIMensioned, even if it has less than 10 elements.

Sample Program

```
10 E = 10
20  DIM N(E)
30  FOR I = 1 TO E
40    N(I) = INT (200 * RND(1))
60  NEXT
60  & SRT (N,E)
70  FOR J = 1 TO E
80    PRINT N(J); SPC(5)
90  NEXT
```

```
]RUN

3     8     34     37     65     118     190     195
```

STORE



& STORE *strexp* TO *numexp*

Stores a string described by *strexp* into a storage cell specified by *numexp*. This command is available only if a Store module is loaded, such as Store256, Store512, or StoreGS.

Stored data is not erased after running a program, or after typing NEW or CLEAR. The string is placed into the storage buffer at the first unused location. Subsequent strings are placed after any previous strings stored in the buffer. You can store up to 255 strings, as long as the strings do not exceed buffer capacity.

Each string that AmperWorks puts into the buffer includes two bytes of overhead. The first byte is the string's ID number. The second byte is the length of the string. With Store256, providing a 256-byte buffer, you could have at least 84 one-character strings in storage. Or, you could have only one 254 character string filling the entire buffer. Storing a null string removes a previously stored string with the same ID. Null strings and their overhead are not stored, however. They occupy no space.

Strings may be stored with any arbitrary ID numbers from 0 to 255, and need not be stored in order. Examples:

```
& STORE "Foo" TO 100
& STORE "Bar" TO 20
```

Restoring a string by its ID returns the string as long as the string exists in storage, otherwise a null string is returned. Examples:

```
& RESTORE 100 TO A$    :REM restores "Foo"
& RESTORE 20 TO A$     :REM restores "Bar"
& RESTORE 42 TO A$     :REM restores "" (nothing)
```

Also See
RESTORE, STORE CLEAR

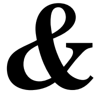
STORE CLEAR

& STORE CLEAR

Clears all strings from storage. This command requires a Store module to be loaded before it can be used.

Also See
RESTORE, STORE

SWAP



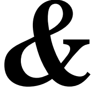
& SWAP (*var1, var2*)

Exchanges the values of two variables or array elements, indispensable for sorting and data processing. After SWAP is executed, the original value of *var1* is stored in *var2* and the original value of *var2* is stored in *var1*. Both *var1* and *var2* must be of the same variable type—string, integer, or floating point.

Sample Program

```
10  INPUT "Enter a value for X: "; X
20  INPUT "Enter a different value for Y: "; Y
30 & SWAP (X, Y)
40 PRINT "Now X = "; X; " and Y = "; Y

]RUN

Enter a value for X: 65
Enter a different value for Y: 816
Now X = 816 and Y = 65
```

TFILES

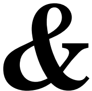


& TFILES (*strexp*, *strvar* [,*numexp*]), *numvar1*
[, *numvar2*]

TFILES is identical to FILES, except filenames placed into the string array may end with special characters. DIR files end with a slash (/). BAS, BIN, SYS, and CMD files end with an asterisk (*). TFILES is mostly useful for display purposes.

Also See
FILES

TIME



& TIME (*strvar*)

Returns the current day, date and time from the Apple IIGS built-in clock or ProDOS-compatible clock card. With string functions, you can strip out certain parts of the time string as needed for your application.

Sample Program

```
10 & TIME (T$)
20 PRINT "Today is: ";T$

]RUN

Today is: Mon,  8 Sep 86 03:04:05
```

Some clock systems do not support the day of week nor seconds through ProDOS. On these systems, TIME returns a string with the first four characters as spaces, and seconds are always “00”.

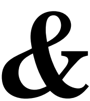
```
Today is:       8 Sep 86 03:04:05
```

If a clock is not installed, using TIME returns a string with a blank day of week and month, and everything else is zero:

```
Today is:       0      00 00:00:00
```

This command is available only if a Time module is loaded, such as Time or TimeGS.

UCASE

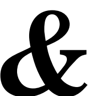


& UCASE (strvar)

Converts a string variable’s lowercase letters to uppercase.

Also See
LCASE

UNTIL

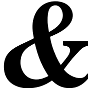


& UNTIL (*boolexp*)

UNTIL marks the end of a REPT-UNTIL block. The statements between the &REPT and &UNTIL markers repeat until the Boolean expression within parentheses is true. When the condition is met, program flow continues after the UNTIL statement.

Also See
REPT

VAL



& VAL *strexp* TO *numvar*

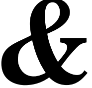
Evaluates the numeric expression contained in *strexp*, returning the result in *numvar*.

Sample Program

```
10 INPUT "Enter an expression: ";A$
20 & VAL A$ TO N
30 PRINT "The result is: ";N

]RUN

Enter an expression: sin(log(3)*10)   return
The result is: -.999955336
```

Also See
REPT

VLIN



& VLIN *numexp1, numexp2*

Draws a vertical line, the height determined by *numexp1,* with the character whose ASCII code is determined by *numexp2*. Both arguments must be numeric values from 0 to 255. This is identical to &HLIN, except that the line is drawn vertically.

Also See
HLIN

ASCII Chart

Low			High	
0	$00	^@	128	$80
1	$01	^A	129	$81
2	$02	^B	130	$82
3	$03	^C	131	$83
4	$04	^D	132	$84
5	$05	^E	133	$85
6	$06	^F	134	$86
7	$07	^G	135	$87
8	$08	^H	136	$88
9	$09	^I	137	$89
10	$0A	^J	138	$8A
11	$0B	^K	139	$8B
12	$0C	^L	140	$8C
13	$0D	^M	141	$8D
14	$0E	^N	142	$8E
15	$0F	^O	143	$8F
16	$10	^P	144	$90
17	$11	^Q	145	$91
18	$12	^R	146	$92
19	$13	^S	147	$93
20	$14	^T	148	$94
21	$15	^U	149	$95
22	$16	^V	150	$96
23	$17	^W	151	$97
24	$18	^X	152	$98
25	$19	^Y	153	$99
26	$1A	^Z	154	$9A
27	$1B	^[	155	$9B
28	$1C	^\	156	$9C
29	$1D	^]	157	$9D
30	$1E	^^	158	$9E
31	$1F	^_	159	$9F

Low			High	
32	$20	SPC	160	$A0
33	$21	!	161	$A1
34	$22	"	162	$A2
35	$23	#	163	$A3
36	$24	$	164	$A4
37	$25	%	165	$A5
38	$26	&	166	$A6
39	$27	'	167	$A7
40	$28	(	168	$A8
41	$29	)	169	$A9
42	$2A	*	170	$AA
43	$2B	+	171	$AB
44	$2C	,	172	$AC
45	$2D	-	173	$AD
46	$2E	.	174	$AE
47	$2F	/	175	$AF
48	$30	0	176	$B0
49	$31	1	177	$B1
50	$32	2	178	$B2
51	$33	3	179	$B3
52	$34	4	180	$B4
53	$35	5	181	$B5
54	$36	6	182	$B6
55	$37	7	183	$B7
56	$38	8	184	$B8
57	$39	9	185	$B9
58	$3A	:	186	$BA
59	$3B	;	187	$BB
60	$3C	<	188	$BC
61	$3D	=	189	$BD
62	$3E	>	190	$BE
63	$3F	?	191	$BF

Low				High	
64	$40	@		192	$C0
65	$41	A		193	$C1
66	$42	B	⬉	194	$C2
67	$43	C	⌛	195	$C3
68	$44	D	✓	196	$C4
69	$45	E	☑	197	$C5
70	$46	F	⏎	198	$C6
71	$47	G	≣	199	$C7
72	$48	H	←	200	$C8
73	$49	I	…	201	$C9
74	$4A	J	↓	202	$CA
75	$4B	K	↑	203	$CB
76	$4C	L	‾	204	$CC
77	$4D	M	↵	205	$CD
78	$4E	N	█	206	$CE
79	$4F	O	⇤	207	$CF
80	$50	P	⇥	208	$D0
81	$51	Q	⤓	209	$D1
82	$52	R	⤒	210	$D2
83	$53	S	—	211	$D3
84	$54	T	└	212	$D4
85	$55	U	→	213	$D5
86	$56	V	▒	214	$D6
87	$57	W	▓	215	$D7
88	$58	X	⊏	216	$D8
89	$59	Y	⊐	217	$D9
90	$5A	Z	▕	218	$DA
91	$5B	[	◆	219	$DB
92	$5C	\	‗	220	$DC
93	$5D	]	╪	221	$DD
94	$5E	^	⊡	222	$DE
95	$5F	_	▏	223	$DF

Low			High	
96	$60	`	224	$E0
97	$61	a	225	$E1
98	$62	b	226	$E2
99	$63	c	227	$E3
100	$64	d	228	$E4
101	$65	e	229	$E5
102	$66	f	230	$E6
103	$67	g	231	$E7
104	$68	h	232	$E8
105	$69	i	233	$E9
106	$6A	j	234	$EA
107	$6B	k	235	$EB
108	$6C	l	236	$EC
109	$6D	m	237	$ED
110	$6E	n	238	$EE
111	$6F	o	239	$EF
112	$70	p	240	$F0
113	$71	q	241	$F1
114	$72	r	242	$F2
115	$73	s	243	$F3
116	$74	t	244	$F4
117	$75	u	245	$F5
118	$76	v	246	$F6
119	$77	w	247	$F7
120	$78	x	248	$F8
121	$79	y	249	$F9
122	$7A	z	250	$FA
123	$7B	{	251	$FB
124	$7C	|	252	$FC
125	$7D	}	253	$FD
126	$7E	~	254	$FE
127	$7F	DEL	255	$FF

Low High Low High Low High Low High

ProDOS File Types

Type	Hex	Dec	Description
UNK	$00	0	Unknown
BAD	$01	1	Bad Blocks
PCD	$02	2	Apple /// Pascal Code
PTX	$03	3	Apple /// Pascal Text
TXT	$04	4	ASCII Text
PDA	$05	5	Apple /// Pascal Data
BIN	$06	6	General Binary
FNT	$07	7	Apple /// Font
FOT	$08	8	Graphics
BA3	$09	9	Apple /// BASIC Program
DA3	$0A	10	Apple /// BASIC Data
WPF	$0B	11	Word Processor
SOS	$0C	12	Apple /// SOS System
DIR	$0F	15	Folder
RPD	$10	16	Apple /// RPS Data
RPI	$11	17	Apple /// RPS Index
AFD	$12	18	Apple /// AppleFile Discard
AFM	$13	19	Apple /// AppleFile Model
AFR	$14	20	Apple /// AppleFile Report Format
SCL	$15	21	Apple /// Screen Library
PFS	$16	22	PFS Document
ADB	$19	25	AppleWorks Data Base
AWP	$1A	26	AppleWorks Word Processor
ASP	$1B	27	AppleWorks Spread Sheet
TDM	$20	32	Desktop Manager Document
8SC	$29	42	Apple II Source Code
8OB	$2A	43	Apple II Object Code
8IC	$2B	44	Apple II Interpreted Code
8LD	$2C	45	Apple II Language Data
P8C	$2D	46	ProDOS 8 Code Module
FTD	$42	66	File Type Names
GWP	$50	80	Apple IIGS Word Processor
GSS	$51	81	Apple IIGS Spread Sheet
GDB	$52	82	Apple IIGS Data Base
DRW	$53	83	Drawing
GDP	$54	84	Desktop Publishing
HMD	$55	85	Hypermedia
EDU	$56	86	Educational Data
STN	$57	87	Stationery
HLP	$58	88	Help
COM	$59	89	Communications
CFG	$5A	90	Configuration
ANM	$5B	91	Animation
MUM	$5C	92	Multimedia
ENT	$5D	93	Entertainment
DVU	$5E	94	Development Utility

Continued . . .

ProDOS File Types (Continued)

Type	Hex	Dec	Description
BIO	$6B	107	PC Transporter BIOS
TDR	$6D	109	PC Transporter Driver
PRE	$6E	110	PC Transporter Pre-Boot
HDV	$6F	111	PC Transporter Volume
WP	$A0	160	WordPerfect Document
GSB	$AB	171	Apple IIGS BASIC Program
TDF	$AC	172	Apple IIGS BASIC TDF
BDF	$AD	173	Apple IIGS BASIC Data
SRC	$B0	176	Apple IIGS Source
OBJ	$B1	177	Apple IIGS Object
LIB	$B2	178	Apple IIGS Library
S16	$B3	179	GS/OS Application
RTL	$B4	180	GS/OS Run-time Library
EXE	$B5	181	GS/OS Shell Application
PIF	$B6	182	Permanent Initialization
TIF	$B7	183	Temporary Initialization
NDA	$B8	184	New Desk Accessory
CDA	$B9	185	Classic Desk Accessory
TOL	$BA	186	Tool
DRV	$BB	187	Device Driver
LDF	$BC	188	Load File
FST	$BD	189	GS/OS File System Translater
DOC	$BF	191	GS/OS Document
PNT	$C0	192	Packed Super Hi-Res Picture
PIC	$C1	193	Super Hi-Res Picture
ANI	$C2	194	Animation
PAL	$C3	195	Palette
OOG	$C5	197	Object Oriented Graphics
SCR	$C6	198	Script
CDV	$C7	199	Control Panel
FON	$C8	200	Font
FND	$C9	201	Finder Data
ICN	$CA	202	Icons
MUS	$D5	213	Music Sequence
INS	$D6	214	Instrument
MDI	$D7	215	MIDI
SND	$D8	216	Sampled Sound
DBM	$DB	219	Relational Data Base File
LBR	$E0	224	Archival Library
ATK	$E2	226	AppleTalk Data
R16	$EE	238	EDASM 816 Relocatable File
PAS	$EF	239	Pascal Area
CMD	$F0	240	BASIC Command
LNK	$F8	248	EDASM Linker
OS	$F9	249	GS/OS System File
INT	$FA	250	Integer BASIC Program
IVR	$FB	251	Integer BASIC Variables
BAS	$FC	252	Applesoft BASIC Program
VAR	$FD	253	Applesoft BASIC Variables
REL	$FE	254	Relocatable Code
SYS	$FF	255	ProDOS 8 System Application

Error Codes

0 **NEXT Without FOR**: a NEXT was encountered which had no matching FOR.
2 **Range Error**: an invalid argument value was specified.
3 **No Device Connected**: the given slot has no disk drive installed.
4 **Write Protected Disk**: unable save data unless write-enabled.
5 **End of Data**: an attempt was made to read data past the end of a file.
6 **Path Not Found**: the path to a filename was not found.
7 **File Not Found**: the specified file was not found.
8 **I/O Error**: the drive went offline or the disk has a media defect.
9 **Disk Full**: no room exists on the disk storing more data.
10 **File Locked**: the file is protected against modification or removal.
11 **Invalid Option**: an option not allowed for a certain command was used.
12 **No Buffers Available**: not enough memory for further disk functions.
13 **File Type Mismatch**: an invalid attempt was made to access a special file.
14 **Program Too Large**: you've written a FAT and SLOPPY program.
15 **Not Direct Command**: command was issued from immediate mode.
16 **Syntax Error**: a filename is illegal or a program statement misspelled.
17 **Directory Full**: the root volume contains too many filenames.
18 **File Not Open**: an attempt was made to read or write from an closed file.
19 **Duplicate File Name**: a RENAME or CREATE used on an existing filename.
20 **File Busy**: an attempt to re-OPEN or modify an OPEN file's name was made.
21 **File Still Open**: upon entering immediate mode, a file was found OPEN.
22 **RETURN Without GOSUB**: a RETURN with no matching GOSUB.
42 **Out of Data**: an attempt was made to READ past the last DATA item.
53 **Illegal Quantity**: an out-of-range value was used with a certain command.
69 **Overflow**: you used an awfully BIG or amazingly SMALL number.
77 **Out of Memory**: program code and variables have used up all free memory.
90 **Undef'd Statement**: a line number which does not exist was referenced.
107 **Bad Subscript**: an array subscript is larger than the given DIMension.
120 **Redim'd Array**: an attempt was made to reDIMension an existing array.
133 **Division by Zero**: division by zero is undefined (remember your algebra?)
163 **Type Mismatch**: a numeric or string value was used incorrectly.
176 **String Too Long**: the given string was larger than was allowed.
191 **Formula Too Complex**: go easy on the machine, Einstein.
224 **Undef'd Function**: reference to an undefined FuNction was made.
254 **Reenter**: user input was not of the type or format required.
255 **Control-C Interrupt**: control-C was pressed.

Licensing

As stated on the inside cover of this manual, this is a copyrighted software product. It may not be distributed in any way without permission of the Morgan Davis Group. To obtain authorization to include Morgan Davis Group software with your commercial products, write or call and request a Universal Software Licensing Agreement. Be sure to include the title of the Morgan Davis Group software you wish to license:

http://www.morgandavis.net

BASIC Syntax

Throughout this manual some abbreviations are used to clarify special syntaxes or conditions for command usage. This appendix quickly explains what they mean and how they're used.

strexp

A string is defined as a group of letters, numbers, symbols, or control codes. A string expression, or *strexp* as used in this manual, is any combination of strings and their various forms in BASIC. Examples of string expressions:

```
X$
"Hello, World."
"this" + "that"
CHR$(4) + "OPEN" + FILE$
CHR$(ASC(MID$(Q$, I, 1)) - 2) + "yuck!"
```

strvar

With some ampersand commands that return string information, a string variable, *strvar*, is required. When a *strvar* is called for, a string expression is *not* allowed. Examples of string variables:

```
X$
NAME$(7)
```

boolexp

A Boolean expression, *boolexp*, is any logical operation that results in a TRUE or FALSE numeric value. This includes numeric or string operations used *conditionally*. In BASIC, a TRUE value is anything other than zero (usually one), while FALSE is *always* zero. Some examples:

```
"A" = B$
"A" < "B" OR "B" < "C"
((I - J) OR Q) AND C < (D + 33 * (NOT X))
```

numexp

A numeric expression, *numexp*, is any combination of numbers, numeric variables, or arithmetic functions that result in a numeric value. Examples:

```
X
2 + 2
ASC(MID$(B$, 5, 1)) + 64 * (C / 2)
PI - INT(LOG(X) / SIN(Y) * Y * 20))
```

numvar

A numeric variable, *numvar*, is used when a command returns a numeric value. Examples:

```
X
Q2%
J(3 + I)
```

Optional Arguments

Some commands accept optional parameters, shown within [ ] brackets in this manual. Do not include the brackets when you enter the commands into BASIC.

Index

A

absolute reference 29
ampersand
  command table 16, 17
AmperWorks 31
  command summary 32
  loading 31
APW 28
ASCII chart 53
assembly language 7, 15, 29

B

boolexp 61

E

error codes 57

F

file type 11, 55
function
  OMM_C2PSTR 23, 27
  OMM_COUNT *23*, 27
  OMM_FREE 23, 25
  OMM_GETID 23, 24, 27
  OMM_GETINFO 23, 28
  OMM_GETSTR 23, 27
  OMM_PADDEC 23, 27
  OMM_PUTSTR 23, 26
  OMM_PUTWORD 23, 26
  OMM_XOAMP 23, 25

H

header
  fields 16
  version 16, 18, 29

I

ID number 9, 12
IMC 19
  updating pointers *29*
immediate mode table 17, 18
immediate reference 17, 18
index number 9, 12
intermodule communication. ***See IMC***
interface files 28

L

licensing 59

M

memory 7
- auxiliary memory 7
- internal buffers 28
- removing modules 12

Merlin 28

message
- MSG_AMPR 20
- MSG_BORN 20, 22, 27
- MSG_DIED 20, 22, 27
- MSG_IDLE 20, 23
- MSG_INFO 20, 23
- MSG_INIT 20
- MSG_KILL 20, 22
- MSG_QUIT 20
- MSG_REL1 20, 21, 29
- MSG_REL2 20, 21, 29
- MSG_USER 20, 21

messages 19, 20

module
- address 9
- building a 15
- data section 18
- format of a 15
- header 16
- IMC handler 19
- information 8, 12, 13
- length 9
- loading 8
- memory usage 7
- removing 12

N

numexp 62

numvar 62

O

OMM
- code relocator 17, 18, 29
- Commands
  - LOAD CALL 14
  - LOAD FRE 10, 12
  - LOAD GET 10, 11
  - LOAD NOTRACE 10, 13
  - LOAD PEEK 10, 13
  - LOAD PRINT 10, 12
  - LOAD TRACE 10, 13
- features 15
- OMM.Loader 8

opcode usage 29
- BRK 29
- immediate mode reference 29

ORCA/M 28

S

strexp 61

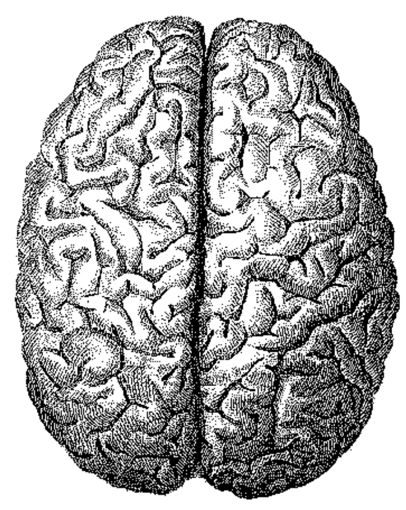
strvar 61

V

version numbers 9

BASIC Memory Management Solution

How did we take the headache out of enhancing BASIC?



We used our brains!

If you've ever tried to improve Applesoft BASIC, the OMM is for you! The OMM eliminates memory conflicts and gives you added power! Use your favorite assembler to easily create relocatable modules. Let the OMM do the brain work!

- Accellerates Applesoft
- Intermodule Communication
- Efficiently manages memory
- Includes AmperWorks™
- Sample programs
- Supports all assemblers

Morgan Davis Group
www.morgandavis.net