APPLEWORKS 5 ULTRAMACROS 4.x REFERENCE MANUAL
Version 1.0 - Includes Standard 'DOT' Commands

Table of Contents **Page**

APPLEWORKS 5 ULTRAMACROS 4.x REFERENCE MANUAL
Version 1.0 - Includes Standard 'DOT' Commands

Table of Contents **Page**

APPLEWORKS 5 ULTRAMACROS 4.x REFERENCE MANUAL
Version 1.0 - Includes Standard 'DOT' Commands
Copyright (C) 1996 Joe Walters - Released as Freeware

Table of Contents **Page**

APPLEWORKS 5 ULTRAMACROS 4.x REFERENCE MANUAL
Version 1.0 - Includes Standard 'DOT' Commands
Copyright (C) 1996 Joe Walters - Released as Freeware

Table of Contents **Page**

Table of Contents **Page**

APPLEWORKS 5 ULTRAMACROS 4.x REFERENCE MANUAL
Version 1.0 - Includes Standard 'DOT' Commands
Copyright (C) 1996 Joe Walters - Released as Freeware

Table of Contents **Page**

Table of Contents **Page**

APPLEWORKS 5 ULTRAMACROS 4.x REFERENCE MANUAL
Version 1.0 - Includes Standard 'DOT' Commands

Table of Contents **Page**

Ultra 4 -- What's It To Ya?

Well, for one thing its free :-) And hopefully you have been nursing a passion to write UltraMacros and just didn't know where to begin. This is the place to begin; your very own Ultramacros Manual.

If you don't have a clue as to what UltraMacros is all about then this is the chapter for you! It starts with a high level overview which is followed by the UltraMacros Tutorial which is an overview at a somewhat lower level.

This is followed by Chapter 2 which delves into the miscellaneous details of UltraMacros and its environment.

Chapter 3 is the reference manual for UltraMacros built in commands (tokens). For the most part these are the (often enhanced) commands that came with UltraMacros 3.

Chapter 4 is the reference manual for "dot" commands. Dot commands are not built into UltraMacros so they are often referred to as external commands. These commands reside in files in your AW.INITS folder and their name always begins with "I." There can be one to many dot commands in any init file.

Happy macro writing!

What are these?
These are macro and data files needed to generate a manual for using and writing macros in AppleWorks for the Apple][(not the Mac or the AppleWorks GS program). It describes UltraMacros 4.x and as such can be used with AppleWorks 3.0 through AppleWorks 5.1.

This manual was written because there is no single source of documentation for UltraMacros 4.x. The documentation that came with UltraMacros 4.0 described the changes and additions to Ultramacros 3.0. It depended on the user also having Ultramacros 3.0 and its associated documentation.

Note: The next four sections are from Will Nelken's "Ultra to the max!" manual so the "I" mentioned is Will.

Now What Am I Going To Do With A MACRO?

"Macro" is short for "macro-instruction," meaning that a single keystroke can launch a series of actions, thereby compressing a great deal of user activity (dozens of keystrokes, decisions, data entry, and so on) into just one macro-instruction. Software publishers offer macro programs for use on various computer platforms (Apple II, Amiga, IBM, Macintosh). Some simply record keystrokes (they store your series of key presses in memory and "play them back" when you request it); some allow you to write macros, which is more powerful, because you may include commands normally unavailable at the keyboard. Some work in multiple programs (hiding in the background of memory until invoked); others, by design, work only within a specific program. The latter arrangement is limited to one program, but it usually takes advantage of the maximum possibilities within the environment.

With the introduction of Alan Bird's AutoWorks and *Randy Brandt*'s MacroWorks in 1986, macros for AppleWorks became a reality. Super MacroWorks soon followed and dominated the market because of its features, including the seeds of a genuine programming language for serious macro writers. Meanwhile, Alan Bird devoted himself to developing the TimeOut host for AppleWorks accessory programs, which permitted a wide variety of enhancements to run within AppleWorks, never requiring rebooting. Today, the list of such TimeOut modules available for AppleWorks users exceeds 80, including a thesaurus, file and disk management, graphing for spreadsheets, a drawing program, graphic font printing of text, telecommunications, report generation, and a host of smaller, but no less significant enhancements to the world of AppleWorks creativity.

With the release of TimeOut from Beagle Bros, *Randy Brandt* also released the first version of UltraMacros. Due to the creative genius and dedicated talent of Randy Brandt, this package of macro tools continues to evolve in response to users' requests. In August, 1992, JEM Software, Randy Brandt's own publishing venture, released a further upgrade to this already potent application. The extensions and additions amazed all of us. Instead of simply defining new limits, Ultra 4 opened the door to nearly unlimited addition of new macro commands and user development of macro programs.

Turbo-charge Your AppleWorks!
UltraMacros 4.x, once installed, immediately provides the first-time user with more speed and many new capabilities within the AppleWorks environment. However, unlike other TimeOut applications, UltraMacros is not limited to the original "program" design, but can be thoroughly customized by and/or for the end user into a multiplicity of forms and functions. I personally use UltraMacros to do the following among hundreds of other operations inside AppleWorks, each launched with a single keystroke:

: Check grammar usage in a word processor document

: Automate printing in text or SuperFonts

: Provide word-wrapping in the spreadsheet

: Automate billing in a business

: Save/print/remove all documents on the Desktop

: Locate a "lost" file on my hard disk

: Play Hangman

: Auto-hyphenate a word processor document

: Control various printer functions from the keyboard

: Add or subtract columns of numbers onscreen

: Make a file copy of any AppleWorks screen

: Write checks and manage a general ledger

: Automate TimeOut Calendar functions

: Create an outline document

: Convert foreign text for printing with SuperFonts

: Instantly create or destroy files on the Desktop

: Print word processor documents in two or more columns

: Print two-sided word processor documents

: Print a single label from a database file

: Set an alarm clock in AppleWorks

: Create pull-down or pop-up menus

: Change the case of a word/paragraph/document

: Instantly install a page-numbering footer

: Quickly add from/save to any drive or subdirectory

: Create a personalized dated letterhead or memo

: Enter MouseText into a word processor document

: Eliminate duplicate records in a database

Whatever your field, if you use AppleWorks regularly, you can benefit from adding UltraMacros. Since the release of UltraMacros 4.x, many inventive macro writers have arisen, supplying more utilities and specialized programs for AppleWorks users. And the horizon is growing still brighter.

Ultra 4 -- An Overview
For those familiar with UltraMacros -- its capabilities, commands and syntax -- Ultra 4 goes beyond your wildest macro dreams! Here's just a glimpse of its expanded features:

: Ten times as many variables as before!

: Enhanced < find > command now searches all menus, even in the background (behind frozen screen images)

: Improved taskfile launching permits more powerful file linking

: The four ampersand commands have been replaced with 45 new "dot commands" -- and many more to come

: Macro names and commands can be identified by user-defined labels

: Taskfile caching permits more rapid task switching

: Annotating macro code is easier than ever

: Single-stepping can now be activated/deactivated from within a macro or with a single keystroke

: Real for-next looping permits simpler looping and nested loops

: Menu creation has been simplified and enhanced

All of these will be covered in some detail in subsequent Chapters. For now, just sit back and let your imagination run.

What's Required? What's Compatible?
Ultra 4 is an upgrade to UltraMacros 3.x, not a new program. To benefit from it (except as a new coaster for your mug), you must be an owner of AppleWorks 3.0 through AppleWorks 5.1 (Ultra 4 does not support earlier versions), AND UltraMacros 3.x (Note:). If you have an Apple //e, it must be an enhanced model (65C02 CPU). You also need to have at least 192K of RAM (256K for AW 5), installed in your Apple (that means Apple //e owners must have at least 64K of auxiliary memory added). No, it's NOT possible to expand the AppleWorks program without limits and STILL have it fit on a 128K machine. Fortunately, Ultra 4 will take advantage of all the memory you have for your AppleWorks Desktop (up to 2 megabytes with "slinky" type RAM cards, like the AE RamFactor, or up to 8 megabytes of GS type memory or bank-switched cards).

Note: *Randy Brandt* has released UltraMacros 3 and 4 i.e., they are now freeware. To date they have not shown up on Delphi or any other public forum. I'm sure that will change in the near future.

Beyond that, all it needs to run is a careful examination of the manual and the sample files included on the disk, a spark of hope, a flash of inspiration, and patient persistence in learning. (Rather like life itself, isn't it?)

All task files must be recompiled from their source code (you DID save the source, didn't you?!) to work with Ultra 4. If you cannot find the original code, you may be able to decompile it. To do so, follow these steps:

1. Boot AppleWorks with UltraMacros 3.x.

2. Launch the task file. (If you can stop macro activity without restoring the default macros, then decompiling is possible. Otherwise, see below.)

3. Create a new AWP file (via the Main Menu).

4. From the AWP, call the TimeOut Menu by pressing <oa-Esc> and select Macro Compiler.

4. Select "2. Display current macro set". The macro set will decompile into the empty AWP.

5. Save the decompiled code.

6. Quit AppleWorks and reboot with Ultra 4.

7. Load the decompiled source code that you saved. Make the necessary changes to the code. Then recompile and create a new task file under Ultra 4.

Some commercial macro programs may not include source code or may have been "locked" by the author to prevent alterations and piracy. These must be revised by their authors before they can be used with Ultra 4.

NOTE: Any task files adapted with Randy Brandt's original "Macros to Menus" (which converted task files to appear as TimeOut applications in the TimeOut Menu) will NOT work with Ultra 4. A new version of "Macros to Menus" is included on the Ultra Extras disk; it must be used to recreate these macro applications.

You will need to revise some of your own macros and task files to work with Ultra 4, since Ultra 4 has changed a few commands and no longer supports a few others. At the same time, revising macros and task files to take advantage of the improved command structure and syntax of Ultra 4 will often enhance macro operations.

In order to make room for new features, Ultra 4 has reduced the macro table size limit from 4009 to 3984 bytes. Any macro sets that exceed this new limit must be revised. However, in actual practice, you will usually discover that updating the command structure of your macros and task files is all that is necessary to reduce the macro table size below the new limit, because Ultra 4 offers a more efficient syntax.

Ultra 4 is compatible with both TotalControl 2.x and DoubleData 2.x, eliminating some incompatibilities with these programs that existed in UltraMacros 3.x.

NOTE: Beginning with AppleWorks 4 the functions of TotalControl and DoubleData are built into AppleWorks itself so those two programs are no longer needed.

UltraMacros Tutorial

The attempt here is to give you an overview of UltraMacros that will make the UltraMacros Reference section more understandable. Most, but not all, of the following assumes that you are in a Word Processing file (UltraMacros can work everywhere in AppleWorks, but the Word Processor is a good, and understandable, place to test most things.)

What is This?

This is a manual for Ultramacros 4.x as it applies specifically to AppleWorks 5.1. Much of the 5.1 material also applies to AppleWorks 3 and 4. Some effort has been made to distinguish when

AppleWorks 5.1 differs from AppleWorks 4. Sorry if it isn't broad enough for you, but the scope of this work was too large as defined. Adding more work would have prevented my finishing this manual.

Main Sources For This Manual

1. TimeOut UltraMacros by Randy Brandt. This manual was for UltraMacros version 3.1. Published by Beagle Brothers Software.

2. Ultra 4.0 by Randy Brandt. An upgrade for TimeOut UltraMacros. Published by Jem Software.

3. Ultra to the Max! by Will Nelken. Published by Marin MacroWorks.

4. An extensive collection of posts to GEnie concerning AppleWorks &/or UltraMacros by uncounted people.

5. The /EXTRAS disk for AppleWorks 5.1. Permission to quote from it was given by Bill Carver (wgcarver@sqc.com).

6. Saved e-mail from people worldwide.

A great big thank you is in order to:

Randy Brandt: Who said I can quote any of his stuff if I don't sell it. With this in mind, please contain your passion to send me mega bucks for this manual.

I have quoted quite a bit of Randy's "stuff," beginning with the Ultramacros manuals, JEM 4.0 UM Manual, help files on AppleWorks 4 and AppleWorks 5 /EXTRAS disks, and numerous posts he has made on various GEnie AppleWorks &/or UltraMacros topics.

Will Nelken: Who said I could quote anything from his "Ultra to the Max!," manual if I kept my quoting mania within bounds. I do believe that I've done that. His examples of a macro command often inspired me to create a different, but (hopefully) equal example.

Gina Saikin of SQC fame: Gina was the one responsible for getting Bill Carver to agree to let me quote the SQC UltraMacros documentation. A big help! For this reason and a multitude of other ways she has supported the Apple][world over the years, a big thank you to Gina!

GEnie: Luckily, I managed to save much of the GEnie UM messages throughout the years. These folks are very much represented within this manual. Names upon request upon payment of a search fee ;-)

Delphi: I have had several meaningful inputs from Delphi folks. Unfortunately, the AppleWorks passion of GEnie has not been reflected on Delphi.

Bev Cadieux's Mail Group: I have tried as best as I can to NOT include anything that I learned while a member of Bev's Mail Group (MG), because she does not want me to include such information if it is to be uploaded to Delphi (This collection of files will definitely be uploaded to Delphi at some point).

The above said, I still recommend Bev's mail group to one and all that desire in-depth information about AppleWorks. There is nothing available that even closely resembles what Bev provides to the AppleWorks user. At this time you can contact her: [A2MG@AOL.COM].

Beta Testers

A heartfelt thanks to these folks. Without them this effort would have far more errors than it does. There were two groups of beta testers, experts and novices.

The experts in alphabetical order were: Roy Barrows, Kevin Noonan, and Bud Simrin (Special thanks to Bud. He really put my code through a wringer and it is much better for his efforts). Many thanks for your hard work and dozens and dozens of observations and suggestions on ways to improve this work.

The novice testers were needed to see if the manual made the subject matter clear to a new user. In alphabetical order they were: Mike Macchione and Gary Welsh. Again, thanks for pointing out a number of stumbling blocks that I'd placed in the new user's way.

Any and all errors remaining in the manual are mine and I apologize in advance for not catching them. Please do feed back so I can update the manual for other users.

Installing UltraMacros
No attempt has been made to give instructions on how to install or make backups of UltraMacros for two reasons:

1. For AppleWorks 3 you must have TimeOut installed. If you don't, then you need to purchase it somewhere since TimeOut is crucial to the operation of UltraMacros. TimeOut is not free or in the public domain. In fact, I know of no source for a new copy i.e., you can only buy it on the used market. (Sorry, I don't have a clue as to the latest version so I can't even help you in that respect.)

2. TimeOut is built into AppleWorks 4 and 5 so installation is not an issue.

Nor is activation of UltraMacros in AppleWorks 4 or 5. For AppleWorks 4 it is covered in the manual. For AppleWorks 5 it is covered in the Delta manual that accompanied your two disks from Scranton Quality Computers.

So What's a Macro?
The simplistic answer is, "A macro is a single keystroke that does the work of many keystrokes." While the above is true, UltraMacros can do many things that are not possible from the keyboard. We will start out with examples of a single keystroke replacing many and ease into the esoteric a little later.

The keyboard of a //e and //c has two meta keys, the first is the outline of an apple and the second is an apple filled in i.e., black aka solid. From this the UltraMacros convention of an Open Apple Key (oa-), Solid Apple Key (sa-), and Both Apple Keys (ba-), was derived. Though we use the convention oa-, sa-, and ba- you do NOT press the - key on your keyboard.

An AppleWorks macro is a Solid-Apple (sa-) key command; you simply hold down the sa- key while pressing another key and a predefined sequence of keystrokes is performed. For example, you can set up a macro such as sa-N that types your name and address, or use a macro such as sa-I to indent a paragraph three spaces (one keystroke instead of the usual seven). Macros save you a lot of typing and a lot of time. Also, fewer keystrokes means fewer chances for making errors.

The sa- key on the Apple //e and //c has been renamed the OPTION key on the IIgs. If you have a IIgs, think OPTION whenever this manual mentions sa-. (Or, ignore what is printed on the key cap and think sa-, its your call.)

We have said that macros are invoked by pressing sa- and some other key. While that is true, there is more to this story. In addition to the sa- commands there are Both-Apple (ba-) and Solid-Apple-Control (sa-ctrl-) macros.

The ba- macros are invoked by holding down three keys: the Open-Apple (named the Command on the IIgs), Solid-Apple and the wanted macro key.

The sa-ctrl- macros are invoked by holding down three keys: the sa-, control, and the wanted macro key.

At one time there was yet another combination: Both-Apple-Control (ba-ctrl-). Because of a great number of programing difficulties, these macros are now reserved by UltraMacros and should not be called or defined by users. If you do, dire things may happen to AppleWorks and your files!

IIgs One-Key Macros in AppleWorks 3
In UltraMacros 3.x, IIgs users had the ability to run sa- keypad and function key macros without pressing the Option (sa-) key by poking a 0 into a keypad table. (Function key macros are only available if you have an extended keyboard.)

In Ultra 4.x, keypad and function keys automatically run ba- macros. No special tables, no tricks. Just compile ba- macros or press a keypad or function key for recording and you use the one-key macros without having to press another key. (See Recording Macros for clarification)

Each keypad macro is predefined in UM4.0.System to print the key characters. For example, if you press the keypad "=" key, it will run the macro <ba-=>:all>=! which prints a "=".

Having the keypad always call macros can be a hassle if you want to use the keypad to enter numbers from various macro sets without defining ba- macros for each keypad entry.

To *disable/enable one key macros:*

```
AW 3.x      Disable                  Enable
           --------                --------
        <poke $B504,$39>      <poke $B504,$27>
```

IIgs One-Key Macros in AppleWorks 4 and 5.
In AppleWorks 4 and 5 the keypad keys do not automatically run ba- macros so the default macros do not have macros predefined for each keypad key i.e., keypad macros default to off. (I *think* this is a function of the version of UltraMacros 4, not the AppleWorks version. However, since the versions of UltraMacros 4 work only with their intended versions of AppleWorks the question is moot.)

The following is an example of how to enable/disable keyboard macros for AW 4 or AW 5. For these AW versions the memory location contains a number of other flags that must not be disturbed when changing the keyboard macro enable/disable status.

```
// Multiple variables were used so you can use the debugger
// (oa-ctrl-X), to see the values at each stage of the game. See
// sa-C for a less variable intensive version of sa-A.

// sa-A enables keyboard macros. We have to assure that bit 4
// is zero first so the D = C + 16 does not generate a carry from bit
// 4 to 5, etc., etc.

//Bits in this diagram number from right to left. Bit 4 is zero

               //                E    F
<sa-A>:<all :  //             7  4 3  0
A = $EF :  //$EF = binary 1110 1111
B = peek $0FFD :
C = .andbits A,B : //Assure bit 4 is zero
D = C + 16 :   //Now set bit 4
poke $0FFD,D : //Write flags back
>!
```

```
//This disables keyboard macros
<sa-B>:<all :  //             E    F
A = $EF :  //$EF = binary 1110 1111
B = peek $0FFD :
C = .andbits A,B : //Assure bit 4 = 0
poke $0FFD,C : //Write flags back
>!

//This is identical to sa-A in its effect. Uses less variables.
<sa-C>:<all :
A = peek $0FFD :   //Flags with bit 4 either 1 or 0
A = .andbits A,$EF :   //Assure bit 4 is 0
A = A + 16 :   //Now set bit 4
poke $0FFD,A : //Write the flags back
>!
```

Recording Your Own Macros
This section tells you how to record your keystrokes so that they can be played back later with one keypress. It assumes that you have already booted AppleWorks. Note that you do not have to be in a WP, DB or SS application to start recording a macro. Anyplace in AW, even a TimeOut menu will work.

You can record a macro for anywhere from 1 to 3984 or so keystrokes, depending on how many macro keystrokes are already in memory from previously recorded macros and the macro set in memory at boot time.

The only exception is macro 0 (zero). You can always enter up to 80 keystrokes, but it will automatically stop recording at 80 keystrokes.

Any macros recorded using oa-X are lost when you exit AW or launch another task file.

Macro 0 (zero) is even more volatile. Its definition is lost each time an assignment is made to string 0 ($0). Assignment to string 0 is often out of your control so never record a macro 0 for anything except a minimal task that doesn't extend over more than a few minutes time.

1. Press oa-X. The *Escape Map* (at the top of the screen), will say: Review/Add/Change (if you are in a WP document). The left side of the bottom line will say: Press macro key:. If you press a key that is unassigned in the current macro set i.e., W, you will see: Recording W on the right side of the bottom line.

   If you do NOT see "Recording W" then that macro key is defined as part of the current macro set and UltraMacros will ignore the request. My memory on the subject is that Mark Munz told Randy to not redefine existing macros and Randy said, "Pshaw," and forged ahead. After a number of problems surfaced Randy said, "I give," and modified UltraMacros such that it would refuse to redefine macros defined in the current set (without any sort of error indication other than the "Recording W" not showing up).

   We need to make a distinction between macros defined in the current macro set and those defined using oa-X. If W is already assigned in the current macro set then UltraMacros will ignore the oa-X request e.g., "Recording W" will NOT appear, however, if you previously defined W using oa-X then you will be allowed to redefine it. See step 5 for details.

2. You are now in the record mode and whatever you type (mouse moves are included - but forget them, they are unpredictable), will be memorized. Type your name and press Return.

3. Now press oa-X to end the macro definition, (Recording W will disappear from the bottom line).

4. You have just recorded your first macro!. To use it, hold down the sa- key and press W (sa-W) and your name will be typed out at supersonic speed! This macro can now be used in any AppleWorks application i.e., WP, DB, or SS.

5. Now press oa-X and press W again. The message "Replace global macro W? "Yes No" appears on the bottom line. Ultramacros lets you decide if you want to destroy this previously recorded macro. Use arrow keys to highlight your choice and hit return or simply type the first letter Y/N. (Press N this time.)

6. Press oa-X to record another macro.

7. In response to the "Press macro key:" prompt, press RETURN. You will hear a beep. This is because RETURN (Control-M), is a reserved macro name.

8. Press oa-X to record another macro.

9. Press oa-W. The Recording W message will appear with no indication that you are defining a ba-W macro.

10. Enter: a friends's name and press oa-X to end the macro.

11. Press sa-W and your name will be typed out.

12. Press ba-W and your friend's name will appear.

All (non-reserved) keys can have both a sa- and ba- definitions. In addition many of the keys can also have sa-ctrl- definitions. To record a sa-ctrl- macro press: oa-X, hold the control key while pressing the wanted macro key i.e., W. You will get the Recording^W (note the carat preceding the ^W).

NOTE: See the file "Mac.AllPossible" for the final word on what is and what isn't reserved.

The sa-ctrl keys are A through Z, @, [, \,], ^, and _ with the following exceptions:

1. sa-ctrl-M (RETURN), sa-ctrl-#, and sa-ctrl-[(ESC) are reserved.

2. sa-ctrl-Y attempt results in sa-Y being recorded so in effect, sa-ctrl-Y is reserved.

3. sa-ctrl-^ attempt results in a beep as if reserved, however, the Recording^^ message appears and stays on even though if one repeatedly hits oa-X, esc, etc., and you cannot record any other macros. Basically, save your files and exit AppleWorks is the order of the day. Stay far away from this one!

Macros that are reserved sa- commands, such as sa-RETURN, generally can be defined as ba-macros. (ba-$ is reserved. See the file *Mac.AllPossible*.)

Any macros recorded using oa-X are lost when you exit AppleWorks.

1. Use the Ultra Options to "Save the current macros as: 1) The default set 2) A Task file. This saves all active macros "as is". Not normally a smart thing to do. See #2 for a better approach.

2. Create a new WP file. Name it anything since it isn't going to be around all that long. Type oa-ESC and select "Ultra Compiler" and then select #2, "Display current macro set." This will dump all the macros for the set into your new file.

   Find the new macros and copy them, with added comments, to the source file for the decompiled macro set. Compile that set and save the result as the default, task file, or invisible task file. Whichever is appropriate. (See, "Making Macros Permanent," for more on the three types of macro files.)

Creating Custom Macros
This section tells you how to create custom macros by editing a macro file, compiling the changed macros, and then saving them on disk.

Default Macros
The default macros (called "built-in" macros in previous manuals), are those macros which are part of SEG.UM and are available whenever you start AppleWorks. These macros can be changed at any time to anything you wish.

Newcomers to macros should study and use these default macros before attempting to create compiled (non-recorded) macros.

Naturally, you should never change the original source files that shipped with AppleWorks. Always work with a (renamed) copy.

1. Boot AppleWorks and insert the /EXTRAS disk. Add the files /EXTRAS/FILES/SEG.UM.source and /EXTRAS/FILES/SEG.AX.source to the desktop. These are the files you need to study before continuing on to change/add your own macros. (Load another file, "/EXTRAS/FILES/SEG.NA.source," to the desktop. Try and figure out when sa-Ctrl-Z is entered and how. What is its stated purpose? What is its real purpose? And they are different!)

2. Notice that in seg.ax on or about line 3 there is a line with the word "labels," which is followed on the next line by ".seg.ax". This is the name that the compiled file will be saved as in your AppleWorks main directory. Next, notice in file seg.um, on or about line 20 there is a single line "labels." Note that the next line is blank i.e., no name to save this file as. This is because this is the default set and as such is automatically saved as seg.um when you tell Ultra Options to save the current macros as the default set.

3. The reason for having two files for the default set is because UltraMacros has a limit on how large a compiled macro set can be. To get around this limitation UltraMacros allows the set in memory to "call" macros in one or more other macro sets. These called macro sets are known as *Task Files.* Return is made to the calling macro when the called macro terminates. See the < call > command in Chapter 3 for more details.

4. Print out both SEG.UM.source and SEG.AX.source (along with SEG.NA.source if you desire). Examine the printout while reading this manual's description on how macros are constructed. You can modify these (renamed) files to create your own custom macros. Modifying existing macros is a good way to learn about writing your own macros.

Creating a Macro File
A macro file is any AppleWorks Word Processor file which contains macro definitions. You can create a custom macro file by one of three ways:

1) Adding an existing macro file to the desktop and changing the definition of an existing macro definition or by adding a completely new macro to the file. This is the best way to go since macro comments are preserved in the source.

2) By using the Macro Compiler's "Display current macro set," option to list the current macros into a WP file and then adding to or modifying an existing macro definition. This is *not* the best way to go since the comments for all macros, except the added macros, is lost.

3) Create a completely new Word Processor file and start your own custom macro file. A good place to start is to copy the file *Try* (see Note: Try) which is included as part of this package and use oa-N to change the name to your new custom macro package. Next, you need to change the name the compiled file will be saved as. At the top of the file you will see a line just after the

< Labels > command you will see the line ".TryStuff" Change this to the name you choose for your package. Note the initial period.

There's nothing magical about the macro definitions in the Word Processor. They must be "compiled" into true macro codes to be used by UltraMacros.

Creating your very own UltraMacro
Here's a step-by-step look at creating your first custom macro definition and making it a permanent part of AppleWorks.

1. Start up AppleWorks with UltraMacros and TimeOut installed and activated.

2. Load the "Macros Ultra" from the Extras disk (AW 4 & 5), or UltraMacros disk (AW 3), to the desktop. Also load in the file "Try" from this manual's disk.

3. Go to the AppleWorks Main menu and make a new Word Processor file called TEST or FRED or any name that strikes your fancy. I'm going to assume that you chose TEST. (I usually choose x cause I know the darned thing isn't going to live to see the light of day anyway :-)

4. Press oa-Q and return to the "Macros Ultra" file.

5. Use the oa-F command to find "B: < awp". You will see the following macro definition:

```
B:<awp sa-% : rtn rtn>
Date:<tab date : rtn rtn rtn>
From:<tab print $96  :rtn:
ifnot $97 = "" tab tab print $97:  rtn: endif :
tab tab print $98  :
rtn tab tab print $99  :
rtn rtn rtn rtn rtn>To:<tab tab>!
```

6. Copy this macro to the clipboard and then oa-Q to the "Try" file, oa-9 to the bottom, oa-C from the clipboard the macro copied above.

7. oa-ESC and select Ultra Compiler. Respond to the prompts in the following way: Compile a new set of macros, Pause each line? No, Compile from the beginning. When it completes, press space to continue.

   If unsuccessful, fix whatever problem the compiler pointed out to you and this time press ba-C instead of all of the nonsense you had to go through in step 7. In fact, even if successful, try ba-C in the Try file a couple of times. You will like it! See the file *Try.Docs* for a tutorial about what is in Try and why you probably want to include most of it as the top part of every macro package you write.

8. Press sa-B to see a sample "begin a memo" macro. If, at this point there is a whole lot of screen flashing going on and TEST ends up containing pretty much garbage this means that you sloughed off typing in your name and address information the first time you started AppleWorks. In this case: 1) Use the file utilities to delete (if it exists), the file NAME.ADDRESS from the AppleWorks folder. 2) Compile SEG.UM.source and save it as the default set. 3) Compile SEG.AX.source and save it as an invisible task file. 4) Compile SEG.NA.source and save it as an invisible task file. 5) Exit AppleWorks and boot it again. This time, answer the questions. 6) Come back to this section when you have completed the above. (I'm not giving more detailed instructions on how to do the above because you have proven that you don't follow instructions when they are given ;-)

9. Well, we have not done all that much about creating your own UM macro so far. Lets give it a try now.

10. Enter the Mark Munz UM *debugger* (oa-Ctrl-X or oa-Clear).

11. Type: oa-M to see the names of the presently defined UM macros. You are shown a list of these macros in the order that they are defined in the source file. Selecting "Alphabetical," shows a list of defined macros in alphabetical order. Note that a tilde i.e., sa-~B means sa-Ctrl-B macro.

12. Selecting "Alphabetical" we see that ba-C has not been defined so we can use it as our new macro. (If ba-C is assigned for you, then you are not using a "out-of-the-box-" AppleWorks. Pick an unused letter and follow along.)

13. Many macro examples you will see start: B: < all :... This works, however, it is not my preference. My way is: < sa-B > : < all :... My way doesn't cost more or less, just more typing on your part. You will have to decide on the format for your macros. For ba- macros, you have to do it my way ;-) Here is my macro to compile the current file. I put this macro in all of my UM files. You want to put it as the third macro in the file so it gets compiled and can be used even if there are errors further down the file. For most macros you should write them in the fashion shown so crucial steps can be commented or risk having a really cool macro become "write only" some time in the future when you try and figure out just what it was you were trying to do. NOTE: "Write Only," is a computer nerd joke, so laugh uproariously so folks think you got it.

NOTE: Try The "Try" file is provided so you can copy any of the examples from this manual to the bottom and compile and see just how the macro works. The "Try" file has the bare bones needed for any custom macro package. In fact, it has the following < ba-C > macro already so there is no need for you to copy this one over there.

Compile current WP file macro

```
Compile current wp file macro
<ba-C>:<awp :
$0 = "Ultra Comp" :    //Default to AW 4/5
A = peek $1003 :   //AW version byte
if A < 40 then :
   $0 = "U4 Comp" :    //Nope, AW 3 so change name
endif :

oa-esc :
print "1" :    //Put cursor at top
Z = 0 :
find :
if Z = 0 then :    //Compiler not found
   msg $0  + " not found" :
   bell :
   stop :
endif :
rtn :      //Select it
oa-rtn :   //No questions, just do it!
>!
```

1. In the above you see where Z is set to 0: You (may) have seen a number of quotes from Randy where you needn't worry about the value of Z if you do not use it in your macros as a general purpose variable since all < find > 's returned a value in the "at start" range. Bzzzt! Thank you for playing - you lose. That was the "old" rule before dot commands. Now there are a NUMBER of macro commands that return values in Z that mess up following < find > commands that depend on Z being in the "at start" range. So, the rule is: If you have not set/used Z in a find within your macro, set the sucker to 200 for an exact match or 0 for a match

from the beginning. To use 200 in this example we would have to spell out "Ultra Compiler" exactly as it appears in the menu.

2. In the above it said mentioned that the ba-C macro should be the third macro. You probably wondered what are the first two macros? Here they are from the AW 5.1 /EXTRAS/MACROS/SEG.UM.source file. While you are told to study these macros to gain understanding, there are no comments for many of the crucial commands. This is the version that ships with AW 5.1. No code changed, comments added. (Well, I did rearrange a few things, upper cased the variables for readability, etc.)

```
<ba-]>:<all:
   poke $11AC,0 :  //Ignore user presses of Esc
   $1 = screen 36,1,9 :    //Line 1, col 36 for 9 characters
   ifnot $1 = "MAIN MENU" then : //No idea why this needed
       input :
       rtn :
       Rpt :   //Loop until at main menu
   endif :
   X = PeekWord $B560 :    //Zero = no Inits. Mine = $2045
   if X = 0 then bell :    //$EF00 is start of macro table.
       poke $EF70,$1E :    //No idea what $EF70 does. Mine = $DD
       poke $11A2,0 :  //Mark UM as not active. Normally = 1
       oa-Q oa-S >5< rtn : //Init Mgr options See Note 1:
       msg ' You must activate Inits and reboot to use UltraMacros ' :
       stop :  //Again, cannot use #Key2Stop yet.
   endif :
   // Get to Main Menu if auto-startup active,
   // even if 5.25" drive and no clock
   goto ba-[ :
>!

<ba-[>:<all:
poke $11AC,0 : //Don't allow esc from this macro
F = 0 :        //Init F
call ba-K in "seg.ax": //See if seg.ax online
if F = 0 sa-_ :    //No it isn't. Prompt user for AW disk
   Rpt :       //Keep trying until it is
endif :

P(1) = peekword $A90 : //Pointer to Name.Address data
if P(1) > 0 exit endif:    //Exit to <endif> behind <Rpt> below
msg ' Please Wait ' :
Begin :
   $1 = .awpath :  //Path AW booted from
   .online $1 :    //Should not fail. Just booting.
   if Z = 0 sa-_ then :    //Whoops, it failed.
       Rpt :   //Rpt will go to the Begin statement
   endif:

//The following <if> is because the one above only took us to the
//<endif> after the <Rpt> so we have to ask again to get the
//<uncache> behind the next <Rpt>. Confused yet?
//See Chapter 3 exit/rpt trick

if P(1) > 0 exit endif:
   $1 = $1 + "/NAME.ADDRESS" :
   .loadvar $1, 190 :  //Load strings $90 through $99
   if Z = 0 then : //Pointer there but file gone??
```

```
       goto ba-# : //Remake the NAME.ADDRESS file
endif :

//Move Name.Address info from $96-$99 to desktop and set $A90 to
//point to it.
sa-! :
exit :     //Exit/Rpt trick.
Rpt :      //Never goes to the top of this macro

//Remove the file named in $95 from cache. SEG.NA uses this facility.
.uncache $95 :

sa-% :     //Load NAME.ADDRESS to desktop
msg ' Default Macros Successfully Installed - Press any key ':
wait 9000 :
msg "" :
poke $11AC,$1B :   //Put esc back
>!
```

Note 1:
Daddy, what can I do from the oa-Q (Desktop Index) menu?
Glad you asked little Billy. In addition to file selection &/or Desktop Index selection via < tab > , here is what I've discovered. Uncle Randy has turned the Desktop Index < oa-Q > into a mini-meta key. Not all of these can be found in the AW 4 manual. (By me at least.)

1. oa-A goes to: Main Menu, Add Files menu

2. oa-B goes to: Main Menu, Other Activities, Standard Settings, Print Buffer options.

3. oa-C goes to: Main Menu, Other Activities, Clipboard Options

4. oa-D goes to: Main Menu, Other Activities, Disk Activities. (Tab to File Activities.)

5. oa-F goes to: Main Menu, Other Activities, File Activities. (Tab to Disk Activities.)

6. oa-P goes to: Main Menu, Other Activities, Change Current Disk.

7. oa-S goes to: Main Menu, Other Activities, Standard Settings.

8. oa-V goes to: A view of the contents of all three desktops.

Making (Recorded) Macros Permanent
This could be subtitled, "What *Types of Macro files* are there Daddy?" The answer is three little Billy.

1. The *default set* i.e., the macros that are present each time you boot AppleWorks.

2. A *task file* that is launched either by manual means (you, your fingers, and the keyboard or from within another macro package.

3. An *invisible task file*. These can never be launched as a package since the name of the package never appears in a list of available task files. Only individual macros within the invisible task file can be called from within another macro package. (An individual macro within the invisible task file can be selected by < *link* > which leaves the invisible task file in control after the linked to macro completes which would allow selection of others, etc., etc. So by getting tricky, you can < *launch* > an invisible task file using < link > .)

There are two ways to make recorded macros permanent:

1. Type oa-ESC to bring up the TO menu, select "Ultra Options" and then select #4 (The default set). This saves all active macros "as is". This is not a smart thing to do since your recorded macro(s) will be saved without there being any source telling your what is really in your default set. This means you have to remember what you recorded each and every time you record and save as the default set - and you know how bad your memory is.

2. Create an empty WP file named "Junk," or some such. It isn't going to be around all that long.

3. While within "Junk" type: oa-ESC, select "Ultra Compiler," select "Display current macro set." This will cause the current macro set to be placed in WP file Junk. There will be no comments, etc.

4. oa-9 to the bottom of the file and the recorded macro(s) should be last in the file. Copy them to the clipboard and oa-Q to the macro source file.

5. Go to the point in the source where you want the recorded macro(s). (Some folks go alphabetically, others go function, etc., etc.) When at that point oa-C From the clipboard the macro(s).

6. Add comments and whatever type of formatting you normally use.

7. Add a terse line for each new macro at the appropriate point in the beginning of the program that describes the macros function. The lines have the form: \sa-W (tab) Change to 10 chars/inch

8. Use ba-C to compile the package, sa-Ctrl-T to save it appropriately. (If you have not copied ba-C and/or sa-Ctrl-T from the Try file, now would be a good time to do that. Read the notes at the sa-Ctrl-T macro for the three types of sa-Ctrl-T macros (default set, task file, or invisible task file).

TimeOut Applications AW 3.x, AW 4.x, AW 5.x

AppleWorks 3.x
There are two TimeOut applications on the Ultra 4.0 disk. UM 4.0 Options replaces Macro Options and UM 4.0 Compiler replaces Macro Compiler. Using new names allows you to have booth versions on your TimeOut menu, so that you can launch ULTRA.SYSTEM and run UltraMacros 3.x, or launch UM4.O.SYSTEM to run Ultra 4.0 through 4.2 on the same AppleWorks disk.

AppleWorks 4.x and AW 5.x
For AppleWorks 4 and 5 UltraMacros version 4.3 (AW 4.x) and UltraMacros 4.4 (AW 5.x), reside in a file named seg.um which is loaded at boot time if UltraMacros is activated. It is no longer possible to run either UM 3.x or Ultra 4.x for AppleWorks 4 and beyond.

UM 4.x Options
The Macro Options application (TO.UM.OPTIONS on disk for pre AW 5 and TO.ULT.OPTIONS for AW 5) contains Ultra 4.0's options. From inside of AppleWorks, press oa-Escape and select UM 4.0 options or Ultra Options to get the following menu for AW 5.

```
File: CH2A           ULTRA OPTIONS        Escape: Review/Add/Change
====================================================================

        Current macro set: seg.um

        1.  Launch a Task file
        2.  Launch default macros

        Save current macros as:

        3.  A Task file
        4.  The default set


                  UltraMacros 4.4 Copyright 1994 Randy Brandt
------------------------------------------------------------------
Type number, or use arrows, then press Return         4986K Avail.
```

The first two options on this menu deal with Task files.

TASK FILES
Task Files are compiled sets of macros that have been saved as system files. They are called "Task Files" because they allow you to quickly and easily execute a specific task.

Task files can be launched several different ways:

1. From within AppleWorks by using the TimeOut Macro Options application. (See "Launch a Task File" section that follows.)

2. From within AppleWorks by using the <call>, <link>, or <launch> commands.

   A. <call> loads the macro set and then executes the specified macro which returns to the calling macro when macro activity stops.

   B. <link> loads the macro set and then executes the specified macro which is identical to <call>. However, return is not made to the calling macro i.e., the linked to macro set remains the active set when macro activity stops.

C. < launch > loads the macro set and then executes the second macro. As with < link >, return is not made to the calling macro i.e., the launched macro set remains the active set when macro activity stops.

3. From outside AppleWorks by using a program selector such as Bird's Better Bye, the GS/OS finder, ProSel or EasyDrive.

4. From outside of AppleWorks by typing -TASK.NAME from Basic with the AppleWorks STARTUP disk in the current drive.

Launch a Task File

This option reads the Appleworks disk and looks for macro sets (Task files). The names are shown on the screen. Press Escape to return to the menu, or, using the arrow keys, select the macro package to launch. Press Return to run the macro set, or oa-Return to load the macros without running them. If macro sets are present in memory Options will list them, along with an additional option to read the disk for more task names.

If you press oa-Return to select this option, you will be prompted with a pathname ending in UM4.0.SYSTEM (seg.um for AW 4 and 5). If you press Return, your default boootup macros will be reinstalled.

You may also type in a different pathname to launch any other file. The advantage of this option is that your Task file doesn't have to be on the AppleWorks startup disk, and Options doesn't have to read the disk to build a list of available Task files.

NOTE: Ultra 4.x Task files are not compatible with earlier versions. Earlier task files won't appear on the UM 4.x Options list. To convert macros, you will need to load the source and recompile.

When a task is launched from outside of AppleWorks, it first loads UM4.0.SYSTEM, which in turn loads AppleWorks. The first macro in the set of macros i$ then executed.

The second macro should actually begin the task. This is the macro that is executed when the task is launched from within AppleWorks.

Create a Task File

This option takes whatever macros are currently active in Ultra and saves them as a Task file on your AppleWorks startup disk.

First you are prompted to enter a name for the new Task file. The macro set's name is provided as a default. Enter a legal Prodos name (you know, 1 to 15 characters beginning with a letter and containing only letters, periods or numbers).

You are then asked if you want to create a hidden task file. If you choose No a system file is created which can be launched from program selectors. if Yes is chosen, a binary file is created and will not show up on program selectors such as Bird's Better Bye or the AppleWorks selector. These BIN files will show up in Finder but double clicking them will not "launch" unless you have some sort of BIN launcher in place.

If the AppleWorks startup disk is not found, you will be prompted to insert it. Put the startup disk in the drive and press Return, or press escape to cancel. When the macros have been updated, put the previous disk back in the drive.

You also have the option of entering a full ProDOS pathname in order to save the Task file on a different disk or subdirectory.

Save macros as default Set
This optIon takes whatever macros are currently active in UltraMacros and saves them into the file SEG.UM ("UM4.0.SYSTEM" AW 3), on your AppleWorks startup disk. These macros will be then be available whenever you start AppleWorks. No compiling will be necessary to reuse them.

Before the macros are saved you are asked if you want to activate the auto startup macro. if you choose Yes, the first macro in the macro set will be automatically run when AppleWorks is started. Current conventions says that the first macro is always ba-] and the second is ba-[. Normally, ba-] just calls ba-[. However, this convention does allow you to do something different on loading the default set.

< ba-] > : < all: goto ba-[> !   get to Main Menu if auto-startup active,

Here is the second macro. It can do as much or as little as needed. For example, the "call sa-K in seg.ax" isn't needed if you have your AppleWorks disk (HD) on-line at all times. However, if you are booting from a 3.5" disk on a single 3.5" drive system, there is a possibility that seg.ax won't be available on disk when it is needed so it is preloaded here.

```
<ba-[>:<all :
// This message is displayed if auto-startup is used, or if
// these macros are used as a Task file.
poke $11AC,0 : //Do not allow ESC key to abort
F = 0 :
call sa-K in "seg.ax" :    //Empty macro to get seg.ax into cache
if F = 0  sa-_ :   //sa-K &/or seg.ax not found
   Rpt :       //Go to beginning of macro
endif :

msg 'Default macros installed.' :
poke $11AC,27 :    //Put ESC back
wait 1500 : msg ''>!

<sa-_>:<asr:
.say "Place your AW STARTUP disk in a drive and press Return" :
K = peek $C000 : if K = 27 stop >!
```

if the SEG.UM (UM4.0.SYSTEM AW 3), file is not found, you will be asked to insert your AppleWorks startup disk. You can press Escape to cancel, or you can insert the startup disk and press Return. When the macros have been updated, put the previous disk back in the drive.

Key click on: Yes
NOTE: Options 4 through 12 are specific for AW 3. These option do exist for AW 4/5, however, they are handled from the "Other Activities" menu and are covered in the AW 4 manual.

Because the AppleWorks 3.0 cursor blinks while macros are active, the speaker clicks on each keystroke so you know that a macro is in control. To cancel the clicking, use this option.

Cursor blink A
Cursor blink B
These two options control the speed at which the AppleWorks cursor flashes (blinks). Because there are so many hardware/software combinations, it's impossible to come up with defaults that work for everyone. Your cursor should blink at a pace that "feels right" to you Turning the mouse on or off greatly affects the blink rate.

Blink A is the time off for the overstrike cursor, and the time on for the insert cursor. blink 8 is the time on for the overstrike, and the time off for the insert cursor. Sound confusing? lt is. Usually the first number is higher than the second, and smaller numbers will make the blink faster. Experiment

Mouse on: Yes
Use this option to ignore your mouse. lt is useful for //c users who don't have a mouse. //c's have mouse cards built-in, and the cards sometimes give false readings as though a mouse was being moved. if you occasionally experience random cursor moves on any kind of Apple][, try turning the mouse off.

Mouse button delay: 2
Choose this option to adjust how long the mouse button delays after it's used to select menu options. if you find yourself jumping several menu steps at a time when you press the mouse button, you should increase the delay.

Mouse horizontal: 16
Mouse vertical: 32
These options control how far the mouse has to travel horizontally or vertically before the AppleWorks cursor moves. Low numbers are more responsive and high numbers mean you have to move farther

Apple IIgs users can also use the Control Panel Options to change the high speed mouse option to "yes" or "no"

Screen blanker on: Yes
The screen blanker automatically blanks your screen if there has been no key press or mouse move for a specified amount of time. This avoids monitor "bum-in" (i.e. scorching of the phosphor monitor screen). On a IIgs, even the border color is changed to black.

A potential problem was noted by one of the beta testers: "With *Desktop Utils* (which has a screen blanker … and possibly *Twilight II* though I have not used that for years, and possibly others) if the screen is already black then those colours are stored by the second screen blanker :) … so when you restore by pressing a key you restore to black."

Thre are three remedys:

1. Two quick presses on the esc key < esc : esc > , will set the world right again.

2. Set the AppleWorks Screen Blanker: Off

3. Disable the non-AppleWorks screen blanker

Screen blanker delay: 20
Choose this option to adjust how long the screen preserver waits before it blanks the screen. The delay is related to the number of cursor blinks, so your cursor blink rate directly affects the delay time. A higher number means a longer delay before the screen blanks.

UM 4.0 Compiler
The UM 4.0 Compiler (TO.UM.COMPILER on disk) allows you to create new macro programs by scanning a Word Processor document containing macro definitions and converting them into form usable by ultra 4.0. It can also display the current macro set listing the active macro definitions into a Word Processor file.

The basic operation of the compiler hasn't changed from the description in the UltraMacros manual. However, some new features have been added.

C++ Style Comments
C++ comments. C++ (a "real" programming language according to Mark Munz) allows you to enter // and type comments after that. The compiler ignores everything on the line following the // so your ending " > !" must be on another line.

```
<sa-A>:<all :
oa-Q : esc :   //get to main menu from anywhere
rtn : rtn  //add files
>!
```

Compiler Labels
Compiler labels Ultra 4.0 offers a powerful new compiler feature called < labels > , related to the *address labels* found in UltraMacros 3.x. The name is the same because both are names starting with a "#" sign. Labels allow you define a series of commands by name and then use that name throughout the macro file.

Labels are defined at the beginning of your macro file before the < start > token. As with < start > and < end > , the word < labels > must start in column 1, on a line by itself, with no trailing blanks.

A macro set (task file), may be named by a period followed by a valid ProDOS file name.

labels
.TryStuff   //Name of this macro set if saved to disk

Lines beginning with "\" are *Macro Titles.* (A "kind" of label.) They list macro names (ba-C) and the function they perform (Compile UltraMacros). Pressing *< sa-Esc >* from the keyboard or calling *< .MacroNames >* from a macro causes a window to display a scrollable list of the available Macro Titles. Hitting Return while one is highlighted will cause it to run. (Esc will abort the process.)

Each Macro Title can have up to 27 characters of description + 2 for overhead. There are a maximum of 512 characters for AW 4 (UM 4.0 through 4.3). In UM 4.4 (AW 5), the total has been increased to 768 bytes.

Using the full 29 characters per macro title would restrict you to 17 Macro Titles for AW 4 and 26 for AW 5. No indication is given if you exceed the number of characters. Your only recourse is to compile the macro set, press sa-Esc to get the available macros menu, oa-9 takes you to the bottom where you can see if the last macro title made it. If not, figure out how to shorten the description of one or more macro titles and try again. (I currently have 31 defined for AW 5.1 so it pays to keep those titles as short as possible.)

```
\ba-C               Compile Ultramacros
\sa-F               Find Text. Clr prev search
\ba-L               Launch seg.um
\sa-Ctrl-T          Make this file a task file
```

Note that there are three correct ways to specify the macro name: \sa-A (tab/spaces) Comment, \ba-A (tab/spaces) Comment, and \sa-Ctrl-A (tab/spaces) Comment. In the scrollable list they will be displayed as, "Comment" followed by one or two mousetext Apple symbols (the solid apple for sa-macros and both the solid and open apple for ba- macros), a dash, a caret for the control key, and finally the macro letter.

Labels must be on one line only.
Next we have Compiler Labels, which is what this section is titled. Labels start with "#" in column 1, one or more tab/spaces, and equal sign (=), one or more tab/spaces, followed by the text to be assigned to the label. As shown below, the text can be the name of a macro to call or one or more macro tokens.

```
#Key2Stop          = ba-!                 //Call macro ba-!
#addfiles          = oa-q : esc : rtn : rtn :  //Go to add files menu
#FindMain          = sa-B                 //Call macro sa-B
```

Next we have the *<start>* token which is followed by macro definitions.

```
start
//The word start must start in column 1, be on a line by itself,
//and have no trailing spaces. Everything from this point to the
//token <end> is either comments or macro definitions/tokens.

//The first two macros, by convention, are ba-], followed by ba-[.
//See "Compile current WP file macro" section in Chapter 1 for
//a detailed look at the two macros as shipped with AppleWorks

<sa-A>:<all :
#addfiles :
#findMain :                    //Loads file "Main"
>!

<#FindMain>:<asr :             //Really sa-B
$0 = "Main" :
Z = 0 :                        //Match at the beginning
find :
if Z = 0 then :                //If Z = 0 didn't find file Main
  bell :                       //Wake bozo up
  stop :                       //Stop all macro activity
endif :
rtn :                          //It was found & highlighted so load it
>!
```

end

End of macro definitions. Any type of text (macro definitions, comments, your laundry list, etc.), can come after the *<end>* token and it will be ignored by Ultramacros. Like start, it must begin in column 1, be on a line by itself and have no trailing spaces.

One way I use it is to copy a macro I'm going to be drastically changing behind the <end> token so I have a fall back position if all else fails.

Note: A personal preference is to assign the top row shift keys (!, @, #, …, to these label macros because these keys are harder to press than sa-B, etc.

```
#Lab1   = sa-!
#Lab2   = ba-!
#Lab3   = sa-@
etc.
```

Warning: There are a couple of reserved macros along the way. (Oh, but you have looked at *Mac.AllPossible* haven't you ;-)

Appleworks address labels.

The Ultra 4.0 Compiler recognizes some AppleWorks address and special values by name. These labels are mostly useful for use with peek, peekword, poke, and pokeword. For example, the following displays a message with the desktop file count:

```
<sa-A>:<all :
msg "There are " + str$ peek #filecount + " files an the desktop." :
>!
```

Always follow the label with a space or " > " or the compiler won't recognize it as valid. < poke #keypad + 7 > is ok; < poke #keypad + 7 > won't work. The compiler doesn't recognize "#keypad + " as a label

The compiler won't list the label when decompiling. The address will be given in decimal as always.

TIP: Just use decompiling for getting to your recorded macros, and then add them to your commented source file where everything, hopefully, is clear

The compiler allows the use of literal key names to represent values in some cases. Double quotes tell the compiler to use the OA versions of keys (hi bit on), single quotes tell the compiler to use the normal versions of keys (hi bit off), and carets tell it to use the control equivalents.

```
<sa-A>:<all :                //See Note 1
W = #"a" :                   //High bit on
$1 = "W = " + str$ W :
X = #'a' :                   //High bit off
$1 = $1 + "   X = " + str$ X :
Y = #'A' :                   //High bit off
$1 = $1 + "   Y = " + str$ Y :
Z = #^a^ :                   //Control-A
$1 = $1 + "   Z = " + str$ Z :
A = #^+^ :                   //See Note 2
$1 = $1 + "   A = " + str$ A :
msg $1 + "  " + %J% + " Key " + %K% :
#Key2Stop :
>!
```

Note 1: After testing this macro I somehow turned the colon behind the < all > token into an upper case ell (L). Rerunning the macros later "just to make sure" and W kept coming up with a value of zero. Turns out UltraMacros thought I was using a variable named LW (second and subsequent letters ignored), so it gave the value to L. Sheesh!

Note 2: There is no such thing as a Control- + but we got an answer of 11. See the AWP file "KeyChart" and you will see that we got the answer for Control-K. See where " + " is on the chart and you will see what happened.

This is very handy for checking for specific keys:

```
<sa-A>:<all : X = key : if X = #"P" then msg "oa-P was pressed">!
```

Instead of using 1 and 0, you can use more descriptive labels, #true and #false, or #on and #off.

Name	Value	Example
#true	1	`<sa-A>:<all : find : if Z = #true then rtn>!`
#false	0	`<sa-A>:<all : if X = #true then Y = #false>!`
#on	1	`<sa-A>:<all : display #on>!`
#off	0	`<sa-A>:<all : display #off>!`

Address Labels
Here's a list of supported address labels that will generally yield internal AppleWorks locations: (No, True and False are not internal locations ;-)

Since these are labels, they must be prefixed with a "#" for use in a macro.

Generally you would do: "A = peek #Label" However, there are some of these labels that point to values that can be greater than 255. For those, you need to use: "A = PeekWord #Label"

Those that I see that require a PeekWord are DBRecs, #DBRules, DBSelrecs, and FreeMem. Might be more, but I doubt it. (A good way to learn more about these labels is to join *Bev Cadieux's Mail Group:*)

The way most folks use these labels is:

```
<sa-A>:<all :
A = peek #CursChar :         //Get character under cursor
A = A - 128 :                //Remove the high bit
$1 = chr$ A :                //Put the string value into $1
>!
```

Copy to Try, compile and run. Look in the *debugger* for the address.

```
<sa-A>:<all :
clear 255 :        //Clear all integers to 0 & strings to null
A = #day :         //These tokens are used for bypassing
B = #month :       //the clock and forcing
C = #year :        //UltraMacros to use the
D = #dclock :      //date or time of your choice
E = #tclock :      //when displaying the date
F = #hour :        //or time. See the file
G = #minute :      //Macros Ultra for examples.
H = #ccwidth :     //width of current spreadsheet cell
I = #colwidths :   //width table for all columns
J = #curhor :      //horizontal cursor position
K = #curschar :    //character under the cursor
L = #cursver :     //vertical cursor position
M = #dbfields :    //total categories
N = #dbrecs :      //total records
O = #dbrpts :      //total reports
P = #dbrules :     //are rules active?
Q = #dbselrecs :   //records selected by find and /or rules
R = #dbzoom :      //zoom status
S = #exitflag :    //oa-Q or oa-S that caused bail from current file?
T = #false :       //Constant 0
U = #filecount :   //files on desktop
V = #filestatus :  //status of current file
W = #findtype :    //default matching for live finds. See <find>
X = #freemem :     //K free on desktop
Y = #kbtype :      //type of data on clipboard
Z = #key :         //last key pressed
A(1) = #msgh :     //msg horizontal value
B(1) = #msgv :     //msg vertical line
C(1) = #off :      //Constant 0
D(1) = #on :       //Constant 1
```

```
E(1) = #openfile:  //current file's number in desktop index
F(1) = #socursor:  //1 = overstrike cursor, 0 = insert cursor
G(1) = #sszoom :   //zoom status
G(1) = #true :     //Constant 1
H(1) = #waitkey    //default for <wait>
I(1) = #workstr :  //current spreadsheet cell (if string value)
J(1) = #worktype:  //current spreadsheet cell id  <128 means label
K(1) = #workval :  //spreadsheet cell value (in SANE format)
L(1) = #wpwa :     //length/cr byte for current wp line
M(1) = #wpzoom :   //zoom status
N(1) = #totalfiles //Totalfiles on all three desktops
>!
```

Debug
This Mark Munz masterpiece is not a TimeOut application, but it seemed like a logical place to put it. Debug is accessed by oa-Control-X (oa-clear on the IIgs) or the < debug > token from within a macro.

Numeric variables are displayed at the top left of the screen, thirteen at a time. Use up, down, oa-up or oa-down to scan through the variables. You may also press oa-0 though oa-9 to jump to the start of the various arrays. Press Return on any variable you wish to modify. Values may be entered in decimal, or in hex if you precede the number with a "$" sign. Use Tab to switch to the string variables (and another Tab to switch back to the numeric variables).

String variables are displayed at the top right of the screen preceded by the string number and length. Only the first 34 characters are displayed in this window. Using the same commands as with the numeric commands to highlight a string (oa-0 through oa-9 to display a group of ten strings and up and down arrows to select an individual string), you will see the first 79 characters of the string displayed on the next to last line of the screen. i.e., there is no room to display the 80th character. You can see the 80th character by using the "msg StringName," in any type of AW file or by print StringName in a WP file.

Press Return on any string to edit it. The cursor moves to the bottom of the screen where you may enter new text. The ruler on the line above the string shows you the length of the string. A maximum of 79 characters can be entered. One short of a string's limit. Oh well.

Display Items
Several other items are displayed at the bottom right part of the screen.

Onerr Status indicates the current < *Onerr* > flag setting.

Sleep Macro displays the name of any defined sleeping macro, along with the time it is set to activate. Macros are put to sleep by < *Wait* >

Name displays the name of the current macro set.

Pr# displays the < print > output slot, normally 0, which is the screen.

Defined indicates how many macros are in the current set.

Length indicates the length of the current macro set.

```
None    UltraMacros Debug v2.6  Copyright 1995 Mark Munz & Randy Brandt
-----------------------------------------------------------------------
  A (0) : $0020, 32        |    00 : 05 :
  B (0) : $0001, 1         |    01 : 09 :Not found
  C (0) : $00EF, 239       |    02 : 76 :reports. But I was hoping to get m
  D (0) : $0000, 0         |    03 : 10 :/DATA/TXT/
  E (0) : $01DE, 478       |    04 : 00 :
  F (0) : $0001, 1         |    05 : 03 :ONE
  G (0) : $0046, 70        |    06 : 14 :/DATA/DOWNLOAD
  H (0) : $0000, 0         |    07 : 00 :
  I (0) : $0001, 1         |    08 : 00 :
  J (0) : $0002, 2         |    09 : 00 :
  K (0) : $0000, 0         |---------------------------------------------
  L (0) : $0002, 2         |Onerr Status: Off
  M (0) : $0000, 0         | Sleep Macro: None
---------------------------|---------------------------------------------
 └-D Dot cmds   └-X Xtnd nums |Trace Options          |     Name: seg.um
 └-M Macros     └-V View scrn |Numeric: No            |
 └-P Peek vals  └-B Break opt |Strings: No            |      Pr# : 0
 └-T Trace opt  └-S Save Info | Macros: No            |  Defined: 50
 └-N End macro  └-W Walk thru |  Break: Off           |   Length: $0976, 2422
----------------------------------------------------------------------------
```

oa-D show Dot commands
Use the arrow keys to see all of the installed dot commands. In the following a "#" denotes an integar or integer variable is required and "$" a quoted string or string variable is required.

Use Tab or the Right arrow key to move through the command lists. You can use oa-Tab and the Left arrow to move back. Press Escape to return to the main screen.

```
None    UltraMacros Debug v2.6  Copyright 1995 Mark Munz & Randy Brandt
---------------------------------------------------------------------------
                                      |
     DB.AND.SS                        |      DEFAULTS
                                      |
       .GetNames   #,#,#              |        .Beep       #,#
       .SetNames   #,#,#              |      $=.Caps       $
       .GetRec     #,#,#,#            |      #=.Eof
       .SetRec     #,#,#,#            |        .FindPO
     $=.GetCat     #,#                |      $=.GetFPath
       .SetCat     #,#,$              |      #=.ID
     $=.CatName    #                  |      $=.Lower      $
     #=.CatNum     $                  |        .Online     $
     #=.Column     $                  |      $=.PeekStr    #
     #=.ColWidth   #                  |        .PokeStr    $,#
     $=.GetCell    #,#,#              |        .PokeZp     #,#
       .SetCell    #,#,$              |        .SetDisk    $
     $=.CellID                        |        .SetFPath   $
     $=.LastCol                       |      $=.Upper      $
     $=.LastRow                       |        .ZoomIn
                                      |
                                      |                              -->
---------------------------------------------------------------------------
```

oa-M show Macro names

Displays the macro names currently defined in the order they are defined in the AWP macro source file.

all:ba-] = macro is valid in all locations in AppleWorks, including TimeOut.
awp:sa-F = macro is valid only in a Word Processor file.
all:sa-~D = The tilde denotes the control key = sa-Ctrl-D

In addition, you have the option of viewing the names in alphabetical order, making it easy to see if you've defined duplicate macros, or to find an unused name. Original puts the list back in the order they were defined in the file.

```
None  UltraMacros Debug v2.6  Copyright 1995 Mark Munz & Randy Brandt
------------------------------------------------------------------
all:ba-]     awp:ba-;     awp:sa-T     awp:sa-1
all:ba-[     asr:sa-&     awp:sa-U     awp:sa-2
awp:ba-C     awp:sa-/     all:ba-Q     awp:sa-3
adb:sa-F     awp:sa-J     awp:sa-G     awp:sa-4
all:sa-F     all:ba-J     awp:sa-I     asr:sa-%
all:ba-L     awp:sa-K     awp:sa-~I    asr:sa-^
awp:sa-~T    awp:ba-K     awp:ba-+     asr:ba-(
asr:ba-B     asr:sa-#     awp:ba-I
all:ba-!     awp:sa-S     all:sa-*
all:ba-@     all:sa-M     awp:sa-P
awp:ba-A     all:ba-M     awp:sa-O
all:sa-!     awp:sa-N     awp:sa-H
all:ba-Y     asr:ba-%     awp:sa-R
all:ba-Z     asr:ba-#     awp:ba-R
asr:sa-)     awp:sa--     all:ba-1
awp:sa-B     awp:ba--     all:ba-2
awp:sa-D     awp:sa-8     all:ba-3
all:sa-~D    awp:sa-9     all:ba-4
asr:sa-(     awp:sa-E     all:ba-5
awp:sa-;     asr:ba-E     asr:ba-6
-------------------------------------------------------------------
Viewing order?  Alphabetical  Original
```

NOTE: As part of this manual I generated all possible macro names using, what else, UltraMacros. As you might guess, the number of macros more than filled the above screen. What to do? Simply hit TAB to see the next screen. Is that *Mark Munz* a cool programmer or what!?! (The file to generate all possible UM names is part of this package.)

oa-P display Peek values
Displays 20 addresses along with the data found at that address. You may define each location as a Word (two bytes), Byte (one byte), String (length byte followed by characters) or Raw data (16 bytes). Press Return on a location to set the address and display type. See oa-S option for the way to save these setting between sessions.

If you wish to change the display from say, byte to word, hit Return and type one existing character followed by Return. Simply hitting Return, Return, will not give you the option to change the display mode.

```
None    UltraMacros Debug v2.6  Copyright 1995 Mark Munz & Randy Brandt
-------------------------------------------------------------------
 1.  $0000  W:$0000 , 0     ".."
 2.  $0000  W:$0000 , 0     ".."
 3.  $0000  W:$0000 , 0     ".."
 4.  $0000  W:$0000 , 0     ".."
 5.  $0000  W:$0000 , 0     ".."
 6.  $0000  W:$0000 , 0     ".."
 7.  $0000  W:$0000 , 0     ".."
 8.  $0000  W:$0000 , 0     ".."
 9.  $0000  W:$0000 , 0     ".."
10.  $0000  W:$0000 , 0     ".."
11.  $0000  W:$0000 , 0     ".."
12.  $0000  W:$0000 , 0     ".."
13.  $0000  W:$0000 , 0     ".."
14.  $0000  W:$0000 , 0     ".."
15.  $0000  W:$0000 , 0     ".."
16.  $0000  W:$0000 , 0     ".."
17.  $0000  W:$0000 , 0     ".."
18.  $0000  W:$0000 , 0     ".."
19.  $0000  W:$0000 , 0     ".."
20.  $0000  W:$0000 , 0     ".."
---------------------------------------------------------------------------
```

oa-T set Trace options
Allows you to set trace options for numeric and string variables, or for macro names. If macro names are being traced, Ultra 4.0 will pause each time a new macro is called, displaying the new macro name on the bottom of the screen.

When you choose to trace variables, you have the option of displaying the variable names and values each time a variable is set (modified) or whenever it is accessed (get or set).

This option changes only the bottom line of the screen as follows. (The highlighted options are from Roy Barrows' HiLight Tool Macro. Unfortunately, the hilights won't appear in the printed manual so we have placed a comma between options.)

```
Change Trace Option for?  Ä⌡ σ≥Θπ,   ô⌠≥Θετ,   ìßπ≥∩≤,   â∞σß≥áü∞∞,   é≥σßδáÉ⌠≤
```

If you choose "Break Pts" you must also select oa-B to set break points to break on.

oa-N End Macro
If a <debug> token is encountered in a macro or you reach a designated break point, you are put into the debugger. At that point you can simply hit ESC to return to the macro where execution will continue or you can type oa-N followed by ESC. In this later case the macro is aborted. There is no special screen for this option.

oa-X Extended Numeric Variables
oa-X displays the contents of the 26 extended math variables. Typing A through Z allows you to set the value of that extended variable. See the documentation for the dot commands (Chapter 4), that begin with an x i.e., <.xMath> for details.

```
None    UltraMacros Debug v2.6  Copyright 1995 Mark Munz & Randy Brandt
---------------------------------------------------------------------

                             Extended Variables

              `A :        0.00             `N :        0.00
              `B :        0.00             `O :        0.00
              `C :        0.00             `P :        0.00
              `D :        0.00             `Q :        0.00
              `E :        0.00             `R :        0.00
              `F :        0.00             `S :        0.00
```

```
               `G :         0.00            `T :         0.00
               `H :         0.00            `U :         0.00
               `I :         0.00            `V :         0.00
               `J :         0.00            `W :         0.00
               `K :         0.00            `X :         0.00
               `L :         0.00            `Y :         0.00
               `M :         0.00            `Z :         0.00

 xFixed    (fix decimal places at 0, 1, 2, or use 128 for appropriate):  2
 xIntegers (treat Ultra variables as hundredths if 0 or integers if 1):  1

--------------------------------------------------------------------------
Type variable name to modify:
```

oa-V View Screen
Typing oa-V from the debugger allows you to see the text screen as it was when the debugger was entered. Any key restores the debugger screen.

If < display 0 > is active you will see only the visible screen image, not the hidden one.

Randy said, "I can't show you the text screen without admitting that Bill Gates and I can see everything on any computer."

oa-B Break Options
Break Points tell the UltraMacros to break out of the macro to allow user interaction. A break point is when a designated variable or string contains a predefined value/string.

```
None    UltraMacros Debug v2.6  Copyright 1995 Mark Munz & Randy Brandt
-------------------------------------------------------------------------
Numeric Conditionals:

           1.  ==============           6.  ==============
           2.  ==============           7.  ==============
           3.  ==============           8.  ==============
           4.  ==============           9.  ==============
           5.  ==============          10.  ==============
-------------------------------------------------------------------------
String Conditionals:

           1.  ==============
           2.  ==============
           3.  ==============
           4.  ==============
           5.  ==============
           6.  ==============
           7.  ==============
           8.  ==============
           9.  ==============
          10.  ==============
--------------------------------------------------------------------------
Type number, or use arrows, then press Return            5254K Avail.
```

There are 10 user defined variables and 10 user defined strings available for break point use. The user defines what the break point value will be. When the variable or string reaches this predefined value you will be prompted at the bottom of the screen to either continue or enter the debugger.

The same variable and/or string can be defined for 10 different break points i.e., if variable A ever contains any of these 10 values, break.

The screen is divided into two halves, variable and string, and as usual, you toggle between them by the TAB key and navigate within the half with the arrow keys and Return to select.

Choosing a variable is a two step process:

1. Hitting return you are prompted to name the variable A-Z i.e., E.

2. Next you are asked to name which of the 10 Es you want by filling in () with 0 through 9 i.e., E(4).

Now you are asked to select with the arrow keys and Return one of the six relationship symbols:

<>	Not equal to
>=	Greater than or equal to
>	Greater than
=	Equal to
=<	Equal to or less than
<	Less than

Next you will be asked to provide the integer value.

For strings the process is somewhat easier. Here you select the position from the 10 available, provide the string number (0 through 99), select one of the three relationships (Starts With, Contains, or Equals). Next type in the string followed by Return.

If you wish to remove a particular break point, simply highlight it and hit oa-DEL. You will be prompted to see if you really want to delete it. Answer appropriately.

Once all the break points are defined you have to activate the feature. This is done from the main debugger window where you select oa-T. One of the options is: Break Points. Select it and answer "Yes," to the question. Naturally, answering "No," deactivates break points.

NOTE: A word of caution. One beta tester reported: "Some users have found that 'Break points' are not reliable and can cause crashes, especially in complex macros. Most Ultra users I know avoid them. 'Trace' is also a little harder to use than it should be because it doesn't actually stop after each macro command. It often skips several commands, leaving you to guess exactly where it is. I think Randy once explained to me that it skips over any commands that don't cause some change on the screen, or don't cause the screen to refresh, or something like that. It is still a very nice addition to the macro package. Most other macro language have nothing like it."

So there you have it. A two sided coin. When break points work, they work very well. When the don't, crashes ensue. So use break points in moderation and don't be surprised if a crash ensues.

oa-S Options

This option from the main Ultra Debug screen will save (remember), your current break point conditionals and peek value definitions for the next time you run AppleWorks.

The information is saved in …/AW.INITS/I.DEBUG so if you want to set the break point conditionals &/or peek values back to the default state (none), you can either manually remove them or copy a pristine copy of I.DEBUG from one of your copies of the original disk to …/AW.INITS.

oa-W Walk Through

The last option from the main Ultra Debug screen will turn on single-stepping. If you have visited Ultra Debug in the midst of an active macro (with a < debug > token or a break point), single stepping begins from the moment you leave the debugger.

It turns out that oa-# (Chapter 3), from the keyboard or a macro is the same thing as oa-W from the debug menu.

There are some caveats concerning oa-W/oa-#. See oa-# command in Chapter 3. It turns out that oa-W/oa-# isn't always what you want to use.

Variables
UltraMacros supports three types of variables:

Numeric Variables
Numeric variables contain integer values that can range from 0 to 65535.

String Variables
String variables can contain up to 80 ASCII characters. Each character can have an ASCII value of 0 through 255. (Not all ASCII values can be printed in all situations. Exceptions have been noted throughout this manual.)

Extended Numeric Variables
Extended Numeric Variables were slightly covered earlier under the description of the < debug > token. They are covered to much greater detail in Chapter 4 in the MathTools section.

Defining Numeric variables
UltraMacros has 260 numeric variables which are represented by the letters of the alphabet in ten arrays from A(0) to Z(9). Variable names with no array are assumed to be in array 0, so X is the same as X(0). Each variable can contain a value from 0 to 65535.

Here's a macro that uses variable C to print the numbers from 0 to 9

```
<sa-A>:<all : C = 0 (print C : spc  : C = C + 1) 10>!
```

Variable names can also be multiple characters; there are still just 26 of them, but you can use any alphanumeric characters after the variable name, along with _, [and] (underscore and the two brackets). Because the compiler simply ignores the characters after the name, no space is wasted, and listing the macros will list the actual variable name only. For example. our previous macro could be written like this:

```
<sa-A>:<all : Count = 0 (print count : spc : C = C + 1) 10>!
```

There is a danger involved in longer names since "Count" and "Constant" are the same variable, C, which isn't immediately obvious when as you try to debug your macros. Your current editor declines using longer names in any non-trivial e.g., a macro that calls another or doesn't fit on one screen.

Variable names may be used for the subscript of a variable, but array 0 is always assumed if no array number is given:

```
<sa-A>:<all :
oa-9 :                      //Bottom of file
S = 8 :                     //Setting up index for J(8)
J(8) = 89 :                 //Just a number
```

```
print JEM(Software) :        //AKA print J(8)
>!
```

We suggest leaving variables U, V, W, X, Y and Z as "throw-away" variables. Assume that they can be redefined indiscriminately by an and all macros. (Especially Z. See the <find>, <.AskYn>, etc., commands for the reason.)

In addition to the above, you should consider I, J, and K as loop variables in deference to old Fortran programmers ;-)

Start up AppleWorks and insert the /EXTRAS disk. Add all of the files named: "SEG.??.source" to the desktop. Examine the variable usage by the macros in them.

Here's a chart showing the various ways to define numeric variable and use them in conditional macros.

Not sure if I like this chart all that much, but Randy had it in two Ultra manuals so I'll have a go at explaining it.

1. A variable can only be changed by using the "=" operator.

2. Conditional tests, <if> and <ifnot>, test the relationship between a variable and an operand.

3. The operand can be one of three things: A) An absolute decimal or hexadecimal number B) another variable C) the output of *some* Ultra commands. Item C is somewhat slippery and I recommend testing thoroughly to make sure you are getting the results you think you are.

   Here is an example of C that works : if A > peek $10F5 then ... :

   Here is one that does not even though it (and several others), were initially in the chart below. : if A > len $1 then ... : Ultra views this statement as syntactically wrong and will not compile. Testing this particular relationship between A and the length of the string in $1 is a two step process : B = len $1 : if A > B then... :

4. There are three operators >, =, and < (greater than , equal to, and less than), the relationship between a variable and the operand is very straightforward when the <if> condition is tested i.e., : if A > B then... : when true means one thing, A is greater than B.

5. The <ifnot> condition, while straightforward, is somewhat broader than the <if> case i.e., : ifnot A > B then... : when true means that A is either equal to or less than B.

6. From the two preceding points we can see that with <if> and <ifnot> and the three operators you can test the relationship between the variable and the operand six different ways:

```
if A > B then... :            //A is greater than B
else ... :                    //A is either equal or less than B

if A = B then... :            //A is equal to B
else ... :                    //A is either greater or less than B

if A < B then... :            //A is less than B
else ... :                    //A is either equal or greater than B

ifnot A > B then...           //A is either equal or less than B
else ... :                    //A is greater than B

ifnot A = B then...           //A is either greater or less than B
else ... "                    //A is equal to B
```

```
ifnot A < B then...          //A is either equal or greater than B
else ... :                   //A is less than B
```

Examining the above you see that the < else > following an < if > has the same truth values as an < ifnot > testing the < else's > associated < if > condition. So an < else > after an < if > is really an < ifnot > and an < else > after an < ifnot > is really an < if > . (Head hurt yet?)

7. Think of the relationship between two variables as a pie cut into three equal slices labeled in turn > , = , and < . The < if > command selects the slice named after its operator and a companion < else > selects both of the remaining slices. The < ifnot > rejects the one its operator names and takes the other two while a companion < else > selects the named operator.

8. Hopefully, the above will help you with the following chart.

```
condition   var   operator operand
----------------------------------
                           X         variable
if          A(0)     >     7         decimal number
(define)    thru     =     $10       hexadecimal number
Ifnot       Z(9)     <     key       single key keyboard input
                           peek      value at an address
                           peekword  value at a 2 byte address
              .eof and probably scads of other dot commands
```

Remember those crazy mix-and-match animal cards when you were a kid? This is the same idea, except that a variable can only be defined using the "equals" operator. Otherwise you can pick any item out of each category and use them together in a macro (**Note:**).

Any number of operands can be chained together using the four basic math operators (+ - / *). No parentheses are allowed. The equations are strictly evaluated left to right with no other precedence. See < and > and < or > later in Chapter 3 for using multiple var-operator-operand sequences with a single condition.

Note:
Method A: < if A > peek $10F5 then ... >
Method B: < B = peek $10F5 : if A > B then ... >

Both of the above give identical results. The Method B: costs six more bytes of UltraMacros macro memory. Your editor believes that Method B: is somewhat clearer to future readers of your macros. You will have to decide which method to use.

Defining String Variables

UltraMacros has 100 string variables which are represented by the numbers 0-99. Each variable can contain up to 80 characters, usually made up of text (as opposed to control-characters).

$0 (pronounced "String Zero") is a special case. lt is the same thing as macro 0. lt can be printed at any time by pressing sa-0 (zero). The other strings have to be printed or executed from within a macro.

String variables may be defined in many different ways. Literal strings may be surrounded by single or double quotation marks:

```
<sa-A>:<all :
$88 = "This is a literal text string" :
>!
```

Strings may be defined as the current date or time in these formats

```
<sa-A>:<all :
$0 = "date    " + date :
$1 = "date2   " + date2 :
$2 = "time    " + time :
$3 = "time24 " + time24 :
SaveScr :
.Cls 1 :
.WriteStr 10,10,$0 :
.WriteStr 10,11,$1 :
.WriteStr 10,12,$2 :
.WriteStr 10,13,$3 :
msg 'Any key to continue' :
A = key :
RestScr :
esc : esc :
>!
```

Strings may be defined as the current Spreadsheet cell, Data Base category or Word Processor line:

```
<sa-A>:<all :
$8 = cell :
>!
```

A portion of the screen may be used to define a string (see the description of < screen > in Chapter 3.)

```
<sa-A>:<all :
$6 = screen 7,1,15 :
>!
```

A string may be defined by user input from the keyboard :

```
<sa-A>:<all :
$3 = getstr 15 :
>!
```

See the description of < getstr > for more information.

A string may be defined to be the same as another string. ln this example, $7 is made identical to the current value of $2:

```
<sa-A>:<all :
$7 = $2 :
>!
```

String variables can be referenced indirectly by using a numeric variable to specify the string to use. This is neat, so listen up!

```
<sa-A>:<all :                 //prints  Z  ONE
$0 = "Z" :
$1 = "ONE" :
oa-9 :                        //Go to the bottom
A = 0 :
print $(A) :
A = A + 1 :
print "  " + $(A) :
>!
```

A more common use of this technique is to process a series of strings in a loop of some sort. A for loop is perfect because it gives you the name of the string if you play your cards right.

```
<sa-A>:<all :
// Macro code to extract and concatenate 1st 4 characters of strings
//$10 through $24 into string $0

$0 = left $10,4 :            //Get first entry out of the way
for I = 11 to 24 :
  $0 = $0 + "," + left $(I),4 :       //process strings 11 through 24
next I :
>!
```

String tokens requiring one parameter can not use equations. The following macro would be illegal because "getstr 3 + 2" should be "getstr 5" or some other single parameter.

```
<sa-A>:<all :
msg "< " + getstr 3 + 2 + screen 1,1,9 + " >" :
>!
```

Changing per the above compiles fine, but the results are less than what is wanted here. Copy the following to Try, compile it, oa-9 to the bottom of the file and call sa-A. Type the word Hello when prompted by the > at the bottom of the screen.

```
<sa-A>:<all :
msg "< " + getstr 5 + screen 1,1,9 + " >" :
>!
```

The above gives pretty freaky results, doesn't it? The msg line got a message of < Hello1 and the screen at the cursor becomes: +" >":

Clearly, the "1" behind "Hello" is either the first or second "1" from the screen command while the +" >": must come from the compiled macro because there is no space between the " > " and the colon as there is in the source..

Here is one way to fix the problem. Another way would to leave the screen command where it is and move the getstr 5 prior to the msg command i.e., $1 = getstr 5 : and put + $1 + prior to the screen command. (I tried both ways and both worked.)

```
<sa-A>:<all :
$1 = screen 1,1,9 + " >" :
msg "< " + getstr 5  + $1:
>!
```

Parameters for strictly numeric tokens can be equations:

```
<sa-A>:<all :
A = 0 :
L = 2 :

$0 = "Keep your eye on The tab line at the top of the screen" :
hilight 9,L,len $(A) + 15,L :
msg %J% + $0 + "  Key" + %K% :
A = key :
esc : esc :
>!
```

Here's a chart showing some of the ways to define string variables and use them in conditional macros. What this chart is trying to say is:

If you have a string variable set to some value then you can compare it to literal text, another string, or the output of a number of commands that output a string? The answer is "yes," for the following list of commands. Nothing is implied for the myriad of dot commands now available and perhaps available in the future i.e., be suspicious. Check things out.

The following is my preferred way to test string variables. It costs several more bytes, but is much more understandable by your readers. Also, it is NOT guaranteed that all present or future dot commands that ouput a string will work properly within the syntax of:
if A > dotCommand then …

```
<sa-A>:<all :
msg '' :
$1 = getstr 5 :
$2 = "   B" :
$2 = right $2,1 :
if $1 > $2 :
  msg '$1 is bigger' :
endif :
>!
```

```
test      str  operator  operand
==================================================
                      "text"  a literal string
                        $2    another string variable
                       date   October 1 1 , 1989
                       date2  10/11/89
If         $0     >    time   2:33
(define)  thru    =    time24 14:33
Ifnot     $99     <    getstr keyboard input
                       cell   db category, ss cell, wp line
                       screen 80-column text screen
                       chr$   ASCii value of a variable
                       str$   string equivalent of a variable
                       mid    middle portion of a string
                       left   left portion of a string
                       right  right portion of a string
                       dots   a myriad of dot commands. See Chapter 4
```

Any number of operands can be chained together using concatenation (+). No parentheses are allowed. The equations are evaluated left to right with no other precedence. Any characters beyond 80 in a single string are ignored. See <and> and <or> descriptions later in Chapter 3 for using multiple str-operator-operand sequences with a single condition.

Testing String Relationships

Testing the relationships between strings can throw you for a loop unless you understand that when comparing two strings UltraMacros begins with the first character in each string, compares it, asks what relationship is being tested i.e., " > " and if string's character is bigger than the operand's character at this point in time then the condition is true and true is returned as the result for the test.

```
<sa-A>:<all :
$1 = "2" :
$2 = date2 :                //date returned in 10/15/99 format
if $1 > $2 then :
  msg '$1 is bigger' :
```

```
endif :
if $1 < $2 then :
  msg '$1 is smaller' :
endif :
>!
```

You will never see the message that $1 is smaller, no matter what month of the year, since the comparison of the first character shows $1 is larger.

The message here is to be careful when comparing strings. Don't think in terms of numeric superiority/inferiority. If the strings contain ASCII numbers then use <val> to extract the numbers into numeric variables and do the comparison there.

```
<sa-A>:<all :
$1 = "2" :
$2 = "1000000" :
if $1 > $2 then :
  msg "Comparing strings " + $1 + " is bigger than " + $2 + "  Key" :
  A = key :
endif :
A = val $1 :
B = val $2 :
if A < B then :
  $0 = "Comparing int var " + str$ A + " is smaller than " + str$ B :
  msg $0 + "  Key" :
  A = key :
  msg '' :
endif :
>!
```

One final example:

```
<sa-A>:<all :
$1 = "Apr 18, 1999" :
$2 = "Apr 18,  2000" :
if $1 > $2 then :
  msg "$1 is bigger because 1 is bigger than space" :
endif :
>!
```

First A Lot Of Background

This chapter explains in excruciating detail the capabilities of UltraMacros. In fact, it's everything you ever wanted to know about macros but were afraid to ask. (The above statement was true for UltraMacros 3. Since then, Randy has released UltraMacros 4 with its "dot" commands so now there is a lot more to learn as Chapter 4 will attest.)

The Anatomy of a Macro File

The macro file is a AppleWorks Word Processing file with the following sections:

Optional Comments
You can put comments in the beginning of the file. They have no formal structure i.e., you don't have to prefix them with "//" or surround them with curly brackets i.e., { comment }. The only restriction is that you cannot have < *start* > starting in column 1 and alone on the line i.e., "start your engines," is fine.

The mechanics of comments is covered more fully a bit later in this file.

< Labels >
Domain: All Modules
This defines the beginning of the < Labels > section of the file. See Chapter 2 section titled, "Compiler Labels" for detailed descriptions concerning the following.

Task File Name
Macro Titles
Label Definitions
< start >
Domain: All Modules
Macro definitions start here and continue until the end of the file or the optional token < end > .
< end >
Everything from here to the end of the file is ignored if the < end > token is used

The Anatomy of a Macro

Before you can start creating your own macros, you need to understand how a macro is built. The syntax of a command is the set of rules governing the organization and usage of that command. In an English sentence, "He here is" would be improper syntax because the "is" should precede the "here". In a like manner, macro commands must be organized in such a way that UltraMacros can understand what you want to have accomplished.

Tokens
Take a look at the macros in the Macros Ultra file. The macros come after the token < START > and before the optional token < END > . Each macro is made up of a series of normal characters and special tokens.

A token is a code word enclosed in < brackets > that represents a special keystroke or macro command. For example, the token < rtn > represents the RETURN key, and the token < left > represents the LEFT-ARROW key. The macro compiler converts these readable tokens into the equivalent invisible command codes within the macro.

Here's a macro a few lines into the Macro Ultra file.

Note: Many existing examples will show the following:
```
f:<awp : oa-f>T<oa-y>! //Find text; clear default first
```

```
a:<awp : a = $10c1>!
```

This manual will always show:

```
<sa-F>:<awp : oa-F>T<oa-Y>!    //Find text; clear default first
<sa-A>:<awp : A = $10C1>!

In fact, the preferred style is:
<sa-F>:<awp :  //Find text; clear default first
oa-F>T<oa-Y :
>!

<sa-A>:<awp :
A = $10C1 :    //No idea what this does ;-)
>!
```

Since this style allows one to attach meaningful comments to lines.

Expanding on the preceding. We will always show:

1. ALL numeric variables in upper case: A, D(2), etc.

2. ALL macro names in upper case: sa-B, ba-R, sa-ctrl-D, etc.

3. ALL hexadecimal numbers in upper case: $10C2, $FFFF, etc.

4. ALL tokens in lower case: < rtn > , < left > , etc. Well, almost always. We have a personal bias to show Begin/Rpt with an initial caps to aid the reader in picking out the extent of a loop. As an added anomaly to our actions, we also like an upper-case C when the control key is involved i.e., sa-Ctrl-D.

5. Colons between all tokens unless the token is immediately followed by a ″ > ″

6. More on the lower case tokens. They are shown almost universally, where they are defined, with an initial cap as an aid to the reader to understand where they belong alphabetically i.e., lc looks suspiciously like 1c, but is really Lc. Also., from the get go most folks began using caps for embedded words in dot commands i.e., .GetNames, .FindPO, etc., etc. Some of this usage has crept over to the non-dot commands. In any case, token names are caseless so it doesn′t matter if you use: .GeTnAmEs, .getnames, .GETNAMES, etc., etc, except as a matter of style.

Each macro begins with ″ < sa-″ (Solid Apple key, named Option on the IIgs), and a character that represents the key used, ″ < sa-F″, to activate the macro. In this example, the character ″F″ indicates that this macro is executed by pressing Solid-Apple-F.

Next comes ″ < sa-F > : < ″, followed by a token that designates where the macro will work ″ < sa-F > : < awp :″; this macro works only in the Word Processor.

Next come the keystrokes and tokens that actually make up the macro. In this example there are three keystrokes: Open-Apple-F, > T < and Open-Apple-Y. The ″ > ″ tells UltraMacros that what follows with not be a token, rather it should be considered as if the user had typed what follows (T in this case), and that it should be sent to AppleWorks. The ″ < ″ tells UltraMacros that it should consider what follows as UltraMacros tokens, not text to be sent to AppleWorks.

An exclamation mark (!) signals the end of the macro definition. Any text after the ″!″ is ignored. In this example the words, //Find text; clear default first″ describe what the macro does. They are not considered part of the macro.

Token equivalents for keyboard keys

<del>	**Delete key**
<esc>	**Escape key**
<rtn>	**Return key**

<go> Return key
This executes a < rtn > and deletes the --- > marker at a numbered menu. Also, if the display is turned off < display 0 > , it turns the display back on again. Try changing the < go > in sa-A to < rtn > and see how messy things get. Also, try to move the cursor off of the current screen page. Will not be able to do it. Call sa-B to turn display back on. A couple of < esc > keys should set any screen garbage right.

```
<sa-A>:<all :
oa-Q : //Desktop index menu
display 0 :    //Turn display off
go :   //Right back to the Try file
>!

<sa-B>:<all :
display 1 :    //Help the poor guy out and turn display back on
>!
```

<tab>	**Tab key**
<left>	**Left-Arrow key**
<right>	**Right-Arrow key**
<up>	**Up-Arrow- key**
<down>	**Down-Arrow key**
<spc>	**Space-Bar key**

The tokens for Open-Apple, Solid-Apple, Both-Apple AND Control commands use the abbreviations oa, sa, ba, and ctrl followed by a hyphen and the appropriate key. Here are some examples (The hyphen is never typed):

<oa-1>	**Open-Apple-1**
<sa-B>	**Solid-Apple-B**
<ba-right>	**Both-Apple-Right**
<sa-Ctrl-C>	**Solid-Apple-Control-C**
<ba-Ctrl-C>	**Both-Apple-Ctrll-C NOT supported**

Note: You might see examples (outside of this manual), where < ba-Ctrl-Char > macros are used. The message from *Randy Brandt* is, "Do NOT define ba-ctrl-Char macros. They will get you into deep, deep trouble when you least expect it."

UltraMacros adds a number of unchangeable Open-Apple and Solid-Apple commands to AppleWorks.

Tokens may be entered in upper or lower case, but no spaces are allowed between the letters making up the token. For example, < rtn > , < RTN > , and < Rtn > are all valid tokens for the RETURN key, but < r tn > is not valid.

Multiple consecutive tokens can be used without brackets around each individual token. Just separate the tokens with spaces and/or colons. For example, two Up-Arrow commands followed by a Left-Arrow can be represented as < up > < up > < left > , < up up left > , < up : up : left > , > or < up > , < up left > .

To address this point a bit more. The following macro compiles correctly:

```
<sa-A>:<all Begin C = key msg  C if C = 27 then endmacro endif Rpt>!

If you wish to clean up the screen as is done below you will have to
put in one colon i.e., msg '' : endmacro...
```

I maintain that the above is NOT an easy macro for a user to parse. Consider the same macro written this way:

```
<sa-A>:<all :
Begin :
  C = key :
  msg  C :
  if C = 27 then :
     msg '' :               //Clean up screen.
     endmacro :
  endif :
Rpt :
>!
```

Both consume the same number of bytes. While both are uncommented, we maintain that the second is much easier to understand.

It is perfectly legal (and we encourage it as an aid to clarity), to write: "if C = 27 then :"

THE OFFICIAL WORD:
The above concerning space versus colon is true in most cases. However, consider the following: Due to the complexity of some of the new compiler features, the compiler is a bit more finicky about syntax than older versions. To play it safe, you should end lines with a colon, not just a return.

THE UNOFFICIAL WORD:
Use colons after each token. They do NOT cost anything (compiled byte wise), and lead to better understanding of those reading your macros.

```
<sa-A>:<all : $2 = "Fourscore and seven" :
  $1 = left $2,4            // C++ comments are like a colon
  msg $1>!
```

Note that C++ style *optional comments* allow you to leave off the end of line colon. Putting in the colon in such a case costs nothing in macro space and shows consistency in coding style. (However, your faithful manual writer must confess that he has left off the colon, from time to time, if it kept the comment tabbed correctly - neatness counts too :-)

```
  $1 = left $2,4 :          // C++ comments are like a colon
```

Do pick a convention and stick to it for all your macros so that you and other readers can understand your macros at a later date. (And if you are updating another's macros try and keep with that person's style unless you deem it too ugly to live with; then you should modify their style to match yours.)

Personal preference is to include the colon. It costs nothing in macro space and makes clear where one token begins and ends. Easy enough to see for tokens without parameters, not always so easy with complex tokens.

The compiler also allows you to include comments between the < brackets > . Comments are surrounded by curly {brackets}. The previous example could include a comment like this:

```
<up : up : {this text gets ignored by the compiler} left>
```

Another (newer) comment convention is to use two slash characters (//) to end entry of tokens on that line i.e., anything after the two slash characters to the end of the line is ignored by the compiler. The previous example becomes:

```
<up : up :  //this text gets ignored by the compiler
left>
```

The macro compiler will ignore the curly brackets and everything between them (or the // and all characters after for that line). No macro table space is wasted by using comments. The previous sample will compile into three bytes - two Up-Arrow codes and one Left-Arrow code.

Note: If the curly brackets are not between token brackets, they will be treated as normal text. DO NOT!! use token brackets < > inside of the curly brackets {}.

```
<sa-A>:<all : {comment } stop>!       //This one is OK
<sa-A>:<all : {--> } stop>! //This one isn't
```

You can use curly brackets in comments after two slashes:

< sa-A > : < all : $1 = chr$ 123 > ! //$1 contains a { character

Local and Global Macro Tokens
Each macro must be classified as either local or global. A global macro is one that works anywhere. A local macro is one that works only within a specific application (Word Processor, Word Processor Outliner, Data Base, Spreadsheet, or Macro Subroutine macro.)

<all>	**All applications (global)**
<awp>	**AppleWorks Word Processor only**
<aol>	**AppleWorks WP Outliner (AW 5)**
<adb>	**AppleWorks DB**
<asp>	**AppleWorks SS**
<asr>	**Accessible only from other macros**

You cannot have more than one global macro with the same name (the second one will never be used), but you can give the same name to several local macros as long as they are for different applications.

```
<sa-F>:<adb : oa-F : >A<   oa-Y>!      //find text, clear previous word
<sa-F>:<all : oa-F : rtn : oa-Y>!      //find text, clear previous word
```

The order in which macro definitions appear in a file is important. When you select a macro, UltraMacros starts at the beginning of the macro table and searches for the first macro with the specified name. When a match is found, the application definition is checked.

1. If the macro is type < all > , it is executed regardless of where you are within AppleWorks or TimeOut.

2. If the macro is type <asr>, it is executed only if called from another macro. It can't be accessed from the keyboard. <asr> macros are for UltraMacros 3.0 and beyond.

3. If the macro is an AppleWorks application type, UltraMacros checks to see if you are in the specified application. If so, the macro is executed; if not, it keeps searching.

From the above you can see that if multiple macros are created with the same name, the subroutine <asr> must be first, followed by local, followed by the global macro. See the discussion of the oa-M option of Mark Munz's debugger for a way to quickly see the order that macros are defined.

NOTE: Both-Apple macros are not considered the same as Solid-Apple macros even if they use the same key. A key such as "A" could be conceivably have fifteen completely different definitions; a Both-Apple, Solid-Apple, and Solid-Apple-Ctrl command for: <awp>, <aol>, <adb>, <asp>, and <asr>. Keep in mind that there are only 32 different control characters (@, A-Z, [, \,], ^, and _), so all other characters can generate only ten definitions per character.)

Recorded macros (those defined using oa-X) are global by default.

You CANNOT redefine a compiled macro name so leave several key combinations "open," for later definition i.e., sa-W.

See "Recording Your Own Macros" in Chapter 1 for the full story about recording macros.

Assigning Macro Names

When considering what character to assign to a macro keep in mind:

1. Try for a character that is a mnemonic aid to the user i.e., sa-F for Find, sa-P for Printing, etc.

2. Try to assign all user callable macros to a sa-Char macro. The implications of this is to use non shift characters for the user and the shift characters (!,@, #, etc.) for sub macros. The sa-Ctrl-Char macros also fall into this category.

3. If it makes sense, use ba-Char macros to extend sa-Char macros i.e., the sa-N signs your name and address at the bottom of a letter. The ba-N macro calls the sa-N macro and then adds your telephone number and e-mail address.

4. Sometimes you need to set up constants, etc., and then perform some function. At the conclusion of the function the user wants to perform the same function elsewhere in the file. For these types of situations consider using a ba-Char or sa-ctrl-Char macro to query the user to set up string and numeric constants and then call the sa-Char macro. From that point on the user can simply call the sa-Char macro.

5. The sa-Char and ba-Char macros can compliment one another. Consider that sa-1 through sa-8 set markers 1 through 8 and ba-1 through ba-8 go to markers 1 through 8.

6. There are times when sa-Ctrl-Char is needed to further extend the scope of the sa- or ba- macros.

7. See the file "AllPossible," for a listing of all possible/legal macro names. As you might guess, there are some reserved names.

Calling Other Macros

One macro can call another macro in two different ways:

```
<sa-A>:<all :                 //Move cursor line to clipboard
sa-, :                        //Far left on line
oa-M>T<down :                 //Move To clipboard down selects all +
left :                        //first char next line. Left deselects
rtn :                         //first next line. Rtn executes
>!

<sa-9>:<awp :                 //Move the last line in a file
oa-9 : up :                   //to the clipboard
goto sa-A :
>!
```

In the first example, sa-A calls reserved macro sa-, to move the cursor to the left column; UltraMacros then returns to sa-A and the current line is moved to the clipboard.

In the second example, sa-9 goes to the last line in the file and then uses the < goto > command to send control to macro sa-A. UltraMacros never returns to sa-9 because < goto > is a "one-way" command.

Those with BASIC programming experience can think of the first example as a GOSUB and the second as a GOTO. Just remember that using a macro NAME will only continue the current macro when the called macro is finished, and that using GOTO NAME means the macro will never will come back.

"Macro nesting" occurs when a macro calls a macro which calls a macro… UltraMaCros has to remember where to back up to when the current level is finished. The limit is 17 levels. A macro which calls itself will execute 17 times and then stop.

```
<sa-A>:<all : oa-9>*<sa-A>! //Print 17 asterisks
```

To execute a procedure more often, use < begin > and < rpt > along with variables (they are explained later). (It is personal preference to indent those lines between a begin/rpt couplet in order to make the scope of the loop more easily apparent.)

```
<sa-A>:<all :
oa-9 :                        //Get below everything in the Try file
A = 120 :
Begin :
  print "*" : A = A - 1 :
  if A = 0 then stop : endif :
Rpt :
>!                            //Print 120 asterisks
```

CAUTION: When you're about to delete a macro from a file, make sure the macro isn't needed by another macro in the same file. Use the oa-F command to search for references to that macro. For example, if you plan to delete macro sa-B, search for "sa-B".

Actual Reference

This reference documents all of the commands built-in to UltraMacros. It assumes that you have some familiarity with UltraMacros 3.x commands. If not, then now is the time to read the section in this chapter titled, "The Anatomy of a Macro."

Replaced UltraMacros 3.x Commands

UltraMacros discarded or replaced several UltraMacros commands in order to make room for new features. Here are the commands replaced by a similarly named dot command:

```
Replaced: Old      New
         =======  ======
          cls      .cls
          id#      .id
          findpo   .findpo
          menu     .makemenu
```

Dropped UM 3.x Commands
< elseoff, < & > , < rem > , < inc > , < dec > , and < ifkey > .

< elseoff > and < endif > performed the identical function so the dropping of < elseoff > lost nothing.

< & > was dropped all together. If you used it and know what it did you can probably modify your code and use < jsr > .

< rem > allowed the inclusion of comments in compiled code. These comments performed no useful function. In fact, they hurt you by chewing up valuable bytes that could be used for macro code that performed a function.

< inc > and *< dec >* incremented the character under the cursor. You can, with a peek of $10F5, determine what the character is under the cursor and follow this with macro code to increment or decrement the character. See the following macro for one approach.

```
//Increment the first char on the line by two i.e., A becomes C, etc.
<sa-A>:<all :
oa-, :                      //Assure over first char
C = peek $10F1 :            //Cursor. 0 = insert, 1 = overstrike
poke $10F1,1 :              //Assure overstrike
A = peek $10F5 :            //Char under cursor
A = A - 126 :               //Increment by 2 and remove hi-bit
print chr$ A :
oa-, :                      //Put to first on line for user sanity
down :
poke $10F1,C :              //Put cursor back to insert/overstrike
>!
```

< ifkey >
Below is an implementation of < ifkey > from the Ultra.4.0 disk from file "Key Test Macro" modified a tad so it doesn't wipe out the top line in the file with the counter that is showing you that the macro can get your input and at the same time do other things i.e., < ifkey > .

```
<sa-A>:<all :
poke $11AC,0 :              //Keep Esc from aborting the macro
insert : oa-e :             //Overstrike cursor for our display
zoom :                      //Zoom out
oa-1 :                      //Top of file
$2 = cell :                 //Get the top line of Try
oa-Y :                      //Wipe it out
$1 = 'Press any key (Escape to stop) Last key: ' :
A = 0 :                     //Initialize a counter

Begin
  X = peek $C000 :          //peek the key location
  msg ' ' + $1 + str$ x + ' ' : //Show last key pressed
  ifnot X = 27  then A = A + 1 :
     oa-1 :                 //Back to the top again.
     print A :
     Rpt :                //Keep going
  else :  //Esc isn't aborting the macro, just passing control
     msg "" :  bell :       //Clear bottom and beep
     poke $11AC,27 :        //Restore Esc for stopping runaway macros
     oa-, :                 //Left of the screen
     oa-Y :                 //Wipe out the counter
     print $2 :             //Put back original line
  endif :                   //Not needed since at end of macro
>!
```

Changed Commands
Two commands were changed, <call> and <clear>. See their descriptions for details on how they work now.

Reserved Macros

UltraMacros reserves all Both-Apple-Control macros. You will see several references throughout this manual that in many cases ba-Ctrl-x, where x is any one of "spc" through "?", will indeed work. However, Randy has said that you should NOT use them because there are situations where they will get you into deep doo-doo. If you have run out of macro names (not likely), and feel compelled to use a ba-Ctrl-x macro, make sure it is a "simple" macro and is not called from within a loop from another macro.

The special macros listed below cannot be recorded, changed or deleted; You must use them "as is"

You can use these macros at any time (unless otherwise noted): directly from the keyboard (press the appropriate key along with sa- (Option on the IIgs), while recording a macro (press the key along with sa), or in a macro definition (use the appropriate token).

<Ahead> sa-.
Finds the first blank space to the right of the cursor position. This macro works wherever AppleWorks allows you to edit characters, Including Word Processor files, Data base categories, at Find prompts, and when AppleWorks prompts you to enter names.

I have used this command once in a written macro. I use it all the time in keyboard macros to modify text where there is a pattern. Copy The text below and the sa-A macro to Try. Compile Try, put the cursor on the first of the four lines with an unwanted word at the beginning nd end the line, and call sa-A.

```
UnwantedWord Text that is wanted UnwantedWord
UnwantedWord Text that is wanted UnwantedWord
UnwantedWord Text that is wanted UnwantedWord
UnwantedWord Text that is wanted UnwantedWord

//Writing a macro to do this function isn't really worth it
//since this pattern probably won't show up for a long time,
//if ever.
```

```
<sa-A>:<all :
oa-. :                        //however, a keyboard macro is easy
sa-, :                        //Same as <back> command
oa-Y :                        //Delete trailing word
oa-, :                        //Left side of line
sa-. :                        //Same as <ahead>
oa-Del :                      //Delete the space between words
rtn :                         //Wanted text to next line
up :                          //up to line with Unwanted word
oa-D :
oa-. :                        //Right side of word
rtn :                         //Delete it
down :                        //Next unchanged line
>!
```

< Back > sa-,
Finds the first blank space to the left of the cursor position. As with < ahead > , I've found just one use for this in a permanent macro. However, I find it invaluable in keyboard macros.

See < ahead > command for an example of its use.

< Date > sa-'
Displays the date in this approximate format: August 26, 1992 (handy for dating letters or Data base and Spreadsheet reports). The exact format is controlled by the AppleWorks Standard Settings.

< Date2 > sa-"
Displays the date in this approximate format: 08/10/87 (handy for dating transactions in the Spreadsheet), based on the AppleWorks Standard Settings option.

< Time > sa- =
Displays the time in this format: 1:42 pm. if you don't have a clock, the time will always be 12:00 am.

< Time24 > sa- +
Displays the time in this format: 13:42. ln the Data base, if a category includes the word TiME, AppleWorks converts 24-hour times to 12 hour times. For example, 21 :406 is converted to 9 :46 PM.

< Find > sa-Return
ln the Word Processor, moves the cursor to the next carriage return marker no matter what the initial value of Z is and Z remains unchanged. Also, UltraMacros no longer zooms the screen for this command.

< find > is a powerful command which searches menus and lists for the text in string 0 (zero). lt can search the oa-Q file menu and the AppleWorks Add/List/Delete files listings with the screen display on or off. < find > also searches TimeOut menus with the display on or off, and will scan all possible menus automatically.

< find > searches any inverse bar menu with the screen display on.

< find > can search the Spreadsheet column the cursor is in, although this should only be done "live," since it doesn't know when to quit. If a match is not found you have to hit < esc > to stop it.

Reports say that it sometimes stops erratically on mousetext. (I had no idea how to put MouseText into the SpreadSheet and said so. Kevin Noonan, a tester of this manual pointed out tha on can simply copy text from a WP into a SS. It works. Now of course, you are subject to erratic stops :-)

For two consecutive cells with the identical contents, < find > will stop at the second one. (In my trials, it did stop at the first occurrence.) To keep searching, press the down arrow followed by sa-Rtn. **Ed:** All in all, I'd lose < find > in a spreadsheet unless you "know" the data is in the column.)

When < find > fails, Z is set to 0; if a match is found, Z equals the position where the match was found. For example. if < find > is set to match anywhere (see table below), and you're searching for "men", Z would be set to 6 when it reached the file "Docs.MenuTools". (Note: This is not the sixth file, it is the sixth character in the filename.)

UltraMacros's Version 4 < find > allows much more flexibility than the old version. The setting of variable Z determines how < find > searches for a match to the contents of string $0:

Past manuals showed a single value in Z to effect the wanted < find > behavior. In fact, there are a range of values. None in the range are any better than the nominal; so use the nominal.

Type of Match	Nominal Value of Z	Range of Values of Z	
at start	0	0-127	Partial match at beginning
at end	150	128-159	Partial match at end of string
anywhere	160	160-191	Partial match anywhere in string
exact	200	>192	Lengths & case must be the some

If you are doing a series of "match at start" finds (most common), you would set Z = 0 prior to the first find. Since Z always returns a result that falls into the "at start" range (0 or 1), you don't need to pre-define Z before doing subsequent < find > s that are supposed to match at the start. Earlier versions of Ultra required that you set Z to 0 prior to each find.

To spell this out a bit more for success returns Z will equal:

1. At Start Z will equal 1
2. At end Z will equal 1 through 15
3. Anywhere Z will equal 1 through 15
4. Exact Z will equal 1

Many dot commands (< .AskYN > , < .GetInput > , < .GetValue > , etc.), return values in excess of 127 (155, 191, 209,...), under some circumstances so be sure you understand what UM commands are being called between finds. Initialize Z prior to the < find > if in doubt.

A "live" < find > using sa-Return always assumes a "match at start" search unLess you change the FindType flag. For example, if you want exact matches only when using sa-Return, use this macro to set find:

```
<sa-A>:<all : poke #findtype, 200>!
```

and from then on during this AppleWorks session, live finds will have to match string 0 exactly. Naturally, the next time AppleWorks is booted #findtype will revert to the default "match at start."

TIP: If You're at a file list and want to find a file in a hurry, press oa-0 (zero) to define string 0 with the file name and then press sa-Return to < find > the file. From within a macro, you can use this command to automatically load files by name.

TIP: Use the ability to search a menu to < find > printers by name when you aren't sure what order they'll be in.

< sa-esc >
< ba-esc >
Each of these brings up the *Available Macros* window. See Labels in this chapter for some discussion of *Macro Titles* Chapter 4 under < .*MacroNames* > for an associated dot command and Chapter 2 under Macro Titles for a more than thorough discussion on this subject. (Sorry it got spread around so much.)

Open-Apple Commands

The following commands can be used directly from the keyboard as well as from within macros. if you're recording a macro, press the appropriate key along with oa-whatever to use the command in a macro definition.

< oa-# >
Activates single-stepping, so that UltraMacros pauses and waits for a key between each command it processes. Press Return for normal speed or any other key (except ESC, which aborts the macro) to keep single-stepping. (The spacebar is a convenient key and is in keeping with the interface of a > number of other applications. One user made a good case for the "clear" key. However, since that key doesn't appear on a //e or //c I'm staying with the spacebar.)

To activate single-stepping from within a macro, use oa-#, rather than "step" because the word "step" is used in < for-next > loops. Of course, you can always use compiler labels to define: #step = oa-# and then use #step.

If you wish to step through an existing macro simply type oa-# from the keyboard followed by the macro call keystrokes i.e., sa-A. Do **not** hit Return after oa-# since that will turn oa-# off. (Unless you want to turn off oa-# prior to calling the macro.)

A long time ago a user asked *Randy Brandt* why oa-# wasn't showing him what he wanted. Here is Randy's answer:

> Chris, I believe stepping pauses each time something is passed back to AppleWorks, since there's no way for you to see anything happening when internal macro calculations are being made. Therefore a macro that does nothing with AppleWorks won't pause at all. I see your point that the macro could put up a message of its own independent of AW, and therefore stepping could be desirable, but unfortunately that never occurred to me at development time. Again, I've never been much a single-step user, so I haven't supported it as well as you might like. I'm a Debug advocate.

So, the bottom line is: Randy likes debug over oa-#. Your Mileage May Vary (YMMV).

To get a feel for oa-#, copy these two macros to the bottom of the Try file. Compile the file then call sa-A.

The messages that prompt you are predictions about what will happen next, not about what has happened. Please study the code and understand this point.

Initially, I had some other code in sa-B without a rtn command. When I compiled, ran sa-A, and subsequently called ba-C to compile the Try file again, I was still in single step mode. Very confusing until you figure it out.

```
<sa-A>:<all :

//Uncomment the debug call, comment out the oa-# call and compile.
```

```
//when you enter debug type oa-W followed by oa-Q to exit debug.
//This will show you that oa-# from a macro and oa-W from the
//debug screen are identical.

//debug :
oa-# :                      //Will not stop here
A = 1 :                     //Nor here
msg 'Press space to allow oa-Q to complete' : //Nor here
oa-Q :                      //Stops here

A = 2 :                     //Will not stop here
msg 'Press space to allow esc to complete' :  //Nor here
esc :                       //Stops here

A = 3 :                     //Will not stop here
sa-B :                      //Nor here
>!

<sa-B>:<asr :
msg 'Press space to allow 2nd oa-Q to complete' : //Not here
oa-Q :                      //Stops here
msg 'Press space to allow rtn to complete and shut off oa-#' :
rtn :                       //Stop oa-#
msg '' :
>!
```

< oa-X >
Begin/stop recording a macro. This command must be used from the keyboard only; it can't be used within a macro. See the section in Chapter 1 titled: **Recording Your Own Macros** for details on oa-X.

< oa-0 >
Note that oa-0 is a zero. Presents a " > " prompt on the bottom screen line, allowing up to 60 characters to be entered for defining macro 0 (zero). This command is used from the keyboard only. Do not use it while recording a macro. See the description of the < getstr > token.

A trivial use of this might be a case where you wish to enter the same text a number of places in your WP, DB, or SS.

1. Type oa-0, type the text, end with Return.

2. Move to where you want the text and type sa-0

3. Repeat step #2 as often as desired.

Another use is to copy data from one WP line, SS cell, or DB category to another.

1. Put the cursor on the line, cell, or category and type oa-- This sets macro 0 (string 0) to the contents of that line, cell, or category. (See oa-- later in this chapter.)

2. Move the cursor to the line, cell, or category where you want to copy the data and type sa-0. You can repeat this step as many times as you want in different lines, cells, or categories.

< oa-Ctrl-@ >

Sends a CONTROL-@ to AppleWorks. Use this while recording or defining a macro. if you just use CONTROL-@ the macro will stop at that point. Control-@ is used only for printer and interface definitions.

< UC >
< oa-: >
Changes the character at the cursor to upper case. No longer forces the overstrike cursor on.

< LC >
< oa-; >
Changes the character at the cursor to lower case. No longer forces the overstrike cursor on.

< First >
< oa-, >
< oa- < >
Moves the cursor to the first DB category, Spreadsheet column or Word Processor column. < oa- < > is identical and gives an intuitive "feel" to the direction the cursor is about to take. Originally < oa- < > was the keyboard equivalent for the now removed command < store > .

< Last >
< oa-. >
Moves the cursor to the last Data Base category, Spreadsheet column containing data, or Word Processor column. *< oa- >* > is identical and gives an intuitive "feel" to the direction the cursor is about to take. Originally, < oa- > > was the keyboard equivalent for the now removed command < recall > .

< Insert >
< oa-! >
Turns on the insert cursor (the blinking underscore).

Note: oa-E is a built-in AppleWorks command, not UM, that toggles the cursor between insert and overstrike. < insert > is a UM command that assures the insert cursor. To assure overstrike: < insert : oa-E >

< Read >
< oa-^ >
From the keyboard, oa-^ will read the character at the current cursor position into string 0 (zero) and advance the cursor. You can also use the arrow keys to move the cursor to a new position before reading another character. < read > no longer forces the overstrike cursor on.

When $0 reaches 79 characters oa-^ stops advancing the cursor and rings a bell for each subsequent press of oa-^. In the following the cursor stops after 79 characters, with no bell being rung.

```
<sa-A>:<all : (read) 85>!
```

As stated, the cursor stops when 79 characters have been read via oa-^ with a bell for all subsequent oa-^'s. The only way I have found to reset oa-^ from the keyboard is to enter the debugger < oa-X > , and select $0 and null the string by oa-Y, Return. Next, I have to call macro 0 via sa-0. Then and only then will oa-^ accept further keyboard input. Any clarification from readers will be appreciated.

While recording a macro, oa-^ will read the character at the current cursor position into the macro being recorded (the character will become text in the macro definition).

In a macro definition, < read > will read the character at the current cursor position and add it to string 0 (zero).

If you want each < read > to reset string 0 after you have used it, put another < read > in another macro and call that macro.

In the ancient days (UM 3), < read > echoed back the screen character in order to advance the cursor. The carriage return "blot" is the delete character in ASCII codes, hence with it visible, you'll delete when it gets played back. The message here? Don't use read until you update to UM 4. (Do you think this is a huge amount of text about a command nobody you know has ever used?)

< Zoom >
< oa-@ >
Forces zoom OUT (hides printer options in the Word Processor, shows values rather than labels in the spreadsheet, and shows multiple-record layout in the Data Base). To zoom in, use < .zoomin > (See Chapter 4.)

< Disk >
< oa-& >
Reads the current AppleWorks disk name (pathname if a subdirectory is included) into string 0 (zero). This command can be used at any time. A brief flash at the top left of the screen indicates that the command was executed. To set the AppleWorks pathname use < .setdisk > (See Chapter 4).

```
<sa-A>:<all :
disk :                        //From the file
$3 = "From the Try file            " + $0 :
oa-Q :                        //Desktop Index
disk :                        //Could have used oa-&
rtn :                         //Back to the file we were in
$1 = "From Desktop Index (oa-Q)   " + $0 :
oa-Q :
oa-A :                        //Shortcut to Add Files menu from oa-Q
rtn :
oa-& :                        //Same as path
$2 = "From Add Files menu          " + $0 :
.Cls 1 :                      //Clear the screen
.WriteStr 0,8,"Note that there is no filename from any of the above" :
.WriteStr 0,10,$1 :
.WriteStr 0,11,$2 :
.WriteStr 0,12,$3 :
msg ' Key ' :
A = key :
| : esc :
>!
```

< Path >
< oa-* >
Reads the current volume name or subdirectory name and the currently highlighted file name into string 0 (zero) when the Main Menu, Add Files list of files is displayed.

If the Desktop Index menu < oa-Q >, is displayed then the currently highlighted file name is NOT stored into string 0. In effect, < path > becomes < disk > from this menu. See the example.

A brief flash at the top left of the screen indicates that the command was executed. A real bummer is that if you are defining a keyboard macro somebody gets lost when you type < oa-X > W < oa-Q : oa-A : rtn : oa-* : esc : esc : oa-X >. The macro gets defined just fine, but the bottom right of the screen will continue with "Recording W" (or whatever letter you were defining).

```
<sa-A>:<all :
oa-Q :                        //Desktop Index
```

```
path :                      //Could have used oa-*
rtn :                       //Back to the file we were in
$1 = "From Desktop Index (oa-Q)  " + $0 : //Save it
oa-Q :
oa-A :
rtn :
oa-* :                      //Same as path
$2 = "From Add Files menu        " + $0 :
.Cls 1 :                    //Clear the screen
.WriteStr 0,8,"Note that there is no filename from the oa-Q menu."
.WriteStr 0,10,$1 :
.WriteStr 0,11,$2 :
msg ' Key ' :
A = key :
| : esc :
>!
```

< Cell >
< oa-- >
Reads the contents of the current Spreadsheet cell, Data Base category or Word Processor line into string 0 (zero). Move the cursor to the cell and use the command. When using the keyboard version < sa-- > you will see a brief flash at the top left of the screen indicates that it was executed. (See < sa-0 > for a use of the oa-- command.)

NOTE: You will not see the flash if < sa-- > is part of a keyboard macro. There have been times when < sa-- > appeared to be ignored and recording of the keyboard macro aborted with no audible or visible cue to the user.

In rechecking this moments ago, it worked fine to use < sa-- > to capture a DB category's contents, oa-Q to a WP file and dump the contents out using < sa-0 > . So, I guess the official word is, it works in keyboard macros. If it doesn't work for you I'd appreciate a note on the reproducible steps you took to cause it to malfunction so we can update the manual.

The current layout and display settings do not affect < cell > . ln the Data Base, < cell > uses the full category entry as shown in the single record layout. ln the Spreadsheet it uses the actual value of the Spreadsheet cell, not just the displayed value. For example, if cell contained 3.512 but the formatting was dollars with 2 decimal places it would appear as $3.51, but < cell > would return 3.512 so that no precision would be lost.

```
<cell>                      //by itself sets string 0.
$1 = cell :                 //Sets only string 1
```

Dire Warning:
If you perform a < cell > on a AWP line that contains one or more tab characters and move that data to a DB via .SetCat, you will make the DB goofy. As near as I can tell, < cell > does not pick up any of the printer formating stuff i.e., bold, underline, etc., etc. If you think that there is a possibility that your macro can pick up a tab then consider:

```
<sa-A>:<all :
// Set up code
$1 = cell :
$1 = .SubChar $1,9,9,32 :   //Tabs to spaces
// Get on with your life code
>!
```

//The following might be the more prudent i.e., can someone come
//up with another case where cell picks up a control character?

```
<sa-A>:<all :
// Set up code
$1 = cell :
$1 = .SubChar $1,0,31,32 :                    //All control characters to spaces
// Get on with your life code
>!
```

From within a macro, use <cell> as part of any string definition:

```
<sa-A>:<asp :                //Copy a cell
$3 = cell :
down :print $3 :
>!

<sa-A>:<awp :
$0 = "$1 = cell does not change $0" :
oa-9 :                       //Bottom of file
print "Some text" :
$1 = cell :                  //Pick up the text we just wrote
msg $0 + "      $1 = " + $1 :
.Spacebar :
$1 = "Note that a plain 'cell' sets $0" :
cell :
msg $0 + "      $1 = " + $1 :
.SpaceBar :
msg '' :
>!
```

<Recall>
This command was removed for AppleWorks 5.0. See the dot commands .SaveVar and .LoadVar for an alternate way to proceed.

Also, the keyboard <*oa->*> equivalent of this command has been changed to be identical with oa-.

In pre UltraMacros 4.x it sets macro 0 (zero) equal to the text stored by the <store> command. See the <store> command for details on how to simulate this command.

<Store>
Stores the current contents of macro 0 (zero), up to 15 characters, in a special unused area of a Word Processor (15), Data Base (13), or Spreadsheet (14). Since these locations vary depending on which type of file you are in, you must be in the same type of file when you call <recall>.)

This command was removed for AppleWorks 5.0. See the dot commands <.SaveVar> and <.LoadVar> for an alternate way to proceed.

Also, the keyboard <*oa-<*> equivalent of this command has been changed to be identical with oa-,

You can (painfully) simulate the original store command by writing three macros to save/restore the contents of $0 to/from memory locations associated with the type of file. Displaying what was saved at the bottom right of the screen is left as an exercise for the reader.

Note that $0 is silently truncated if too long. This is in keeping with the way store/recall worked in pre UltraMacros 4.x. Further note that <.SaveVar>/<.LoadVar> does no truncating.

```
<sa-A>:<all :
```

```
ba-A :                       //Do constants
if B > D :
  $0 = left $0,D :           //String too long. Truncate it
endif :
.PokeStr $0,C :              //Save $0
.LoadVar "TempVars",255 :    //Restore all variables
>!

<sa-B>:<all :
ba-A :                       //Do constants
$0 = .PeekStr C :            //$0 restored
.LoadVar "TempVars",0 :      //Restore A(0) through Z(0)
>!

<ba-A>:<asr :

//On exit:
//C = location to store/retrieve the $0 string
//D = maximum number of characters that can be store in location C

.SaveVar "TempVars" :        //Save all variables on disk
A = peek $0C6B :             //Get file information
A = .AndBits A,3 :           //Remove all except file type info
B = len $0 :                 //See how much to store. Used by sa-A
if A = 1 C = $9200 :         //DB
  D = 13 : endmacro :
endif :
if A = 2 : C = $7D00 :       //WP
  D = 15 :
  endmacro :
endif :
C = $7FF3 : D = 14 :         //Must be a SS
>!
```

< Bell >
< oa-Ctrl-G >
Sounds the AppleWorks error bell once. It's handy for getting someone's attention.

< NoSleep >
< oa-Ctrl-N >
Domain: All Modules
Cancels the currently defined "sleeping" macro, if any. See the description of < wake > for more information and examples.

< Debug >
< oa-Ctrl-X >
Domain: All Modules
Debug replaces the old UltraMacros Clear as the "live" oa-Control-X command. See the new < clear > command later in this file. Debug turns on the screen display and runs a debugger if one is present. See Chapter 2 for the extensive documentation on Mark Munz's handy debugger.

IIgs users can press oa-Clear; the Clear key on the numeric keypad is the same as Control-X.

Special UltraMacros Tokens

The following tokens are for use within macro definitions only. None of them are keyboard commands, and they can not be recorded using the oa-X command. They require no parameters.

< SaveScr >
Domain: All Modules
Copies the current text screen into a storage area where it can be restored by < restscr > . This allows you to put messages on the screen, draw boxes, or anything else, and then restore the screen later.

I've never found a need for < SaveScr > / < RestScr > so I asked Bud Simrin when it is needed. His reply:

"The most common use is for a screen displaying a user menu. Suppose you want your macroset to perform a bunch of operations, changing the screen display many times, but return to the very menu with the same option hilighted after all is done. BAR (| : esc > , will not restore the menu. Also, it is possible your macroset may wish to perform a certain set of operations from 2 different menus. Only SaveScr-RestScr can restore the correct menu. "

```
<sa-A>:<all :
SaveScr :
.Cls 0 :
$1 = .GetString "Enter your birthdate: ","11/18/67",8:

if Z > 0 and Z < 155 then :
msg ' You came into this world on ' + $1 + '  Key ' :
else msg str$ Z + ' Key ': endif :
A = key :
RestScr :
// At this point the character under the cursor will appear to have
// become a space. It is not gone. To get it back you can do:
// | : esc : or oa-Q : rtn :
// Ignoring the fact that it is gone works cause it really isn't.
// Not using .Cls also keeps the character in place under the cursor
| : esc :
>!
```

< RestScr >
Domain: All Modules
Restores the text screen stored by SaveScr.

See SaveScr for what at first glance appears to be a problem, but isn't.

< Input >
Domain: All Modules
Allows you to enter text or OA commands until Return is pressed (the Return is not passed on to AppleWorks). To exit this command without pressing Return, press Control-@. The macro will be stopped.

Any text typed (except the ending Return), is given to AppleWorks and so is entered in the current WP, SS, or DB. If at a menu you can move around using the arrow or numeric keys.

//Here we present the user with the Desktop Index menu and ask them
//to select a file. Not really needed cause oa-Q works pretty much
//this way anyway. But it does demo the command so don't get pushy.

< sa-A > : < all : oa-Q :
msg 'Choose a WP file from this Desktop or tab to another' :

```
input :
rtn :   //Needed cause the user's rtn wasn't passed to AW
>!

//Here we might use input to move the cursor about the screen
//and prompting for input on a form, etc.

<sa-A>:<all :
oa-9 : //Get below any user text
print "<TEXT>" :
rtn :
msg 'Type something and end with Return' :
input :
msg '' ://Erase message
rtn :   //Line for TEND
print "<TEND>" :
>!
```

<Key>
Domain: All Modules
Pauses until a key is pressed. The keypress is not passed along to AppleWorks. In an equation, key returns the value of the key pressed. For example:

```
<sa-A>:<all :
msg 'Press any key. Esc to quit' :
A = key :
if A = 27 then :             //esc
  msg '' :                   //Clean up the screen
  endmacro :
endif :
if A < 128 then :
  msg  "You typed: " + chr$ A :
else :
  msg 'You pressed oa-key' :
endif :
wait 2000 :
msg '' :
goto sa-A :
>!
```

If the user presses Return, A will = 13, and if the user holds down oa- while pressing the key, 128 will be added to the key value. This example will only print the keystroke if oa- is not pressed.

<Begin>
Domain: All Modules
This does nothing unless used with <rpt>. It marks the restarting point for repeating part of a macro instead of repeating from the beginning.

In the following macro, the begin must be used so that <rpt> does not go all the way to the start of the macro and reset X every time:

```
// Funny thing happened on the way to printing. The original of
// this macro had a line: print X :
// which worked fine except the characters were mushed together giving
// 010203040506070 instead of the wanted 0 10 20 30 40 50 60 70

// Changing said line to: print X + " " : got us a syntax error for
// our troubles. So, while you might see : "print X" : in an example
// and it would work fine, consider making it : "print str$ X" :
// just "cause."

<sa-A>:<all :
X = 0 :
Begin :
  print str$ X + " " :
  X = X + 10 :
  if X < 80 then :
Rpt :
>!
```

< Rpt >
Domain: All Modules
Repeats part or all of the current macro by searching backwards from the < rpt > token until a < Begin > is found, or until the beginning of the macro is reached.

No commands after < rpt > will be executed unless they are part of a IF-THEN-ELSE statement OR the "exit : rpt" trick is used. A conditional command must be used to exit the macro or it will run continuously. For example:

This would drive you crazy; Press Escape to exit. Note no Begin is present so rpt goes to the beginning of the macro.

```
<sa-A>:<all : bell : rpt>!
```

Prints a message, then beeps like crazy. Same as above, except this one has a Begin to keep the print statement from printing over and over.

```
< sa-A > : < all :
oa-9 :  //To bottom of Try file
print "This part executes once" :
Begin :
  bell ://Indentation shows scope of Begin/Rpt
Rpt :
> !
```

The *Exit/Rpt trick.* It is quite simple. The exit command scans downward through the macro looking for a Rpt statement. If found, execution continues with the macro command immediately after the Rpt. If Rpt is NOT found, then the macro exits as you would expect.

The important thing to notice here is the line:

```
if $0 ="" exit : endif :
```

A null line causes UM to scan downwards through the macro looking for a Rpt token. If found, the next token is executed (the second A = peek $15 in this example). If none found, the macro is exited al la < endmacro > . See Chapter 4, < .GetValue > for another example.

```
<sa-A>:<awp :
A = peek $15 :              //Cursor vertical (line) position
if A > 2 then :             //Assure at the top of the screen
  oa-up :
endif :

for I = 2 to 20 :           //Look for a blank line on the screen
```

```
  cell :
  if $0 = "" exit : endif : //Blank line found. Goto after Rpt token
  down :
next I :                    //Fall through if no blank line

oa-down :                   //Next page
oa-up :                     //To our expected place
endmacro :                  //No blank lines on the screen

Rpt :                       //exit : Rpt trick. Never want to Rpt

A = peek $15 :              //Are we at the bottom of the screen (21)?

ifnot A = 21 then :
  oa-down :                 //Go to bottom of page
  A = A - 1  :
  (down) A :
  oa-up :                   //Top of page
  endmacro :
endif :
oa-down :                   //Next page
oa-up :                     //Cursor to top of screen
>!
```

< Exit >
Domain: All Modules
Exits a Begin/Rpt loop or the current macro if no rpt is found.

ln the following example (taken from Ultra to the max cause its a good example and because of the name Erika, a neat name), the macro will allow any user input until Escape is pressed. At that point the Begin/Rpt loop is exited and the macro puts a message on the screen:

```
<sa-A>:<all : //Lotsa room for comments here; Hi Lester!

Begin :                     //restart here after rpt
  X = key :                 //get a key for Will
  If X = 27 then exit :     //leave if we got Escape
  else print chrs X :       //print character for Erika
Rpt :                       //repeat the loop for Steve
msg 'The macro is over.  Key ' :
#Key2Stop :
>!
```

No room for more of Will's friends or relatives

< Endmacro >
Domain: All Modules
ImmedIately exits the current macro no matter what follows. If the macro was called from another i.e., < sa-X > : < all : sa-A : more stuff > ! < endmacro > in sa-A will return to "more stuff" in macro sa-X. For example:

```
<sa-A>:<all :
//Setup stuff...
if A = 1 : then
  X = 1 :
  endmacro :
endif :
```

```
//More macro stuff...
>!
```

< Stop >
Domain: All Modules
Stops all macro activity immediately. Use it to stop a nested macro from returning to the calling macro, or to get out of a < rpt > situation

```
<sa-A>:<all :
$0 = "Fred" :                //Filename
Z = 200 :                    //Exact match
oa-Q : oa-1 : find :         //Find the file on the desktop
if Z = 0 msg 'Cannot find the file ' + $0 + ' on the desktop.  Key' :
A = key : msg '' : stop : endif :
rtn :                        //Accept if found
>!
```

Parameters for UltraMacros Tokens

The next group of tokens require parameters. Most parameters involve variables, so a description of Ultras′ variables follows.

Macro Parameters

Here are some of the possible parameters for UltraMacros tokens:

MACRO
a macro name such as sa-B or ba-D

NUM (number)
a literal decimal number from 0 to 65535
a literal hexadecimal number from $0 to $FFFF
a variable name from A to Z (the value of the variable is used)
an UltraMacros label (see Chapter 2 ″Address Table″ for supported labels)

NUM VAR (numeric variable)
a variable name from A(0) to Z(9) (the value of the variable is used)

NUM2 (number format 2)
a NUM (see above)
a NUM VAR (see above)
tokens: < key > , < peek > , < peekword > , < len > , < val > , < asc > token

NUM EXP (numeric expression)
a NUM (see above)

a NUM2 (see above) if the compiler gives an error, NUM2 is unavailable for this particular command definition.

a NUM or NUM2 equation; NUM′s must be connected by +, -, /, or *; equations are evaluated from left to right only. Parenthesis are not supported to control order of evaluation.

If order of evaluation is important i.e., X = A + (B * (C - D)) then you will have to preform this particular operation in three steps:

```
<sa-A>:<all :
//Evaluate A + (B * (C - D))  Best come out 162 ;-)
```

```
A = 2 :
B = 10 :
C = 22 :
D = 6 :
E = C - D :
F = B * E :
X = A + F :
msg str$ X + " " + %J% + " Key " :
#Key2Stop :
>!
```

The range of values is 0 to 65535. Numbers "wrap around" if the range is exceeded. For example, 0 - 1 = 65535 and 65534 + 3 = 1. Since only integer numbers are allowed, division will return the quotient. See Chapter 4, the .SubChar command for a really elegant use of this wrap around feature.

STRING
a literal string surrounded by quotes

STRING VAR (string variable)
a string name from $0 to $99

STRING EXP (string expression)
A STRING (see above)

Tokens: < chr$ > , < str$ > , < date > , < date2 > , < time > , < time24 > , < cell > , < screen >
< getstr > < a ton of dot commands > token

A STRING Expression: STRING EXP's must be connected by " + "; the equation is evaluated from left to right until the maximum length of 80 characters is reached.

UltraMacros Tokens with parameters

< () NUM >
Domain: All Modules
This new repeat command executes anything in parentheses the number of times specified by NUM.

```
<sa-A>:<awp :                 //A special line in Try awaits you ;-)
oa-l :                        //Get to top o screen
(down) 4 : oa-Y :             //Moves to 5th line and erases it
>!

<sa-A>:<all :
oa-9 :                        //Get to the bottom away from stuff
X = 10 :                      //Repetitions can be absolute or num var
(sa-Z : rtn) X :              //Execute macro Z and rtn token ten times
>!

<sa-Z>:<all :
(print "-=") 30 :             //Prints a 60-character line
>!
```

< Asc STRING VAR >
Domain: All Modules
Converts the first character of a string to its ASCII value.
Example shows how to increment an ASCII character. (Ed: No idea what a real world example would be since I've never used this command in any of my macro packages.)

```
<sa-A>:<all :               //a has an ASCII value of 97
$0 = "a" : X = asc $0 :
X = X + 2 :                 //ASCII value of "c"
msg  $0 + " becomes " + chr$ X + "  " + %J% + " Key " + %K% :
#Key2Stop :
>!
```

< Chr$ NUM >
Domain: All Modules
Prints the ASCII value of a variable or a number. See the file KeyChart for a complete list. Both of the following macros will print the number 1 :

```
<sa-A>:<all : X = 49 : print chr$ X>!

<sa-A>:<all : print chr$ 49>!
```

As you can see on the chart, 49 is the code for a 1. if X was equal to 177, an oa-1 command would be executed. This is handy for sending special codes to your printer along with the < pr# > token i.e., < pr# > selects the printer and you then send it the special codes you desire.

Here is a snippet from the included macro package "Mac.DoubleSide."

```
<sa-A>:<all :
//Setup stuff...
if W = 1 and T = 1 then :   //If last page is odd eject it
  pr# D :                   //Select printer
  print chr$ 27 + "&l1H" :  //Feed from tray (lower case L1H)
  print chr$ 27 + "&l0H" :  //Now eject it   (Lower case L0H)
  pr# 0 :                   //Back to AppleWorks
endif :
// More stuff
>!
```

< Clear NUM >
Domain: All Modules
Clears numeric variables to 0 and string variables to null strings based on the value of the numeric (new for UltraMacros 4) option. In UltraMacros 3 < clear > was the same as < cler 255 > in UltraMacros 4. String 0 (macro 0) is not affected because it is a special case. Use < $0 = "" > to clear it.

```
Option    Result
================================
0-9       clear a numeric array- <clear 6> clears A(6) thru Z(6)
50        clears all 260 numeric variables A(0) through Z(9)
100-190   clear ten strings- <clear 180> clears strings $80 thru $89
          (clear 100 clears strings $1 through $9, sparing $0)
200       clear strings 1 thru 99
255       clear all numeric and string variables (Except $0)
```

< oa-Ctrl-X > and *< oa-Clear > IIgs* were the same as *< clear >* in UltraMacros 3. Now these key operations have been moved to the *< debug >* option.

< Display NUM >
Domain: All Modules
Allows you to turn the screen display off so macros don't make you blind as the screens wildly flash by. We kind of like to see macros hard at work, but some users are intimidated by the action.

If the numeric expression is 0, the display is turned off; if the expression is 1 the display springs to life. if a macro runs amok after turning off the display, just press <oa-Ctrl-X> or <oa-Clear> on the IIgs, followed by <esc> to turn the display back on.

See <go> command for another way to turn the display back on.

NOTE: While the display is off, <esc> can't be used to abort a macro. To Ignore <esc> at Other times as well, use <poke $11AC,$1A>. To reenable it, use <poke $11AC,$lB>. Ed: The current common way to disable Esc is: <poke $11AC,0>.

Actually, you can't disable <esc>. You can change the key that is used to esc. The <poke $11AC,$1A> changes it from <esc> (Control-[) to Control-Z. The current <poke $11AC,0) changes it to Control-@ which is much less likely to be hit accidentally by the user. Give the following a go in the Try file.

```
<sa-A>:<all :                 //Change the commented out
poke $11AC,0 :                //Control-@
//poke $11AC,$1A :            //Control-Z
A = 0 :
(A = A + 1 : sa-B) 50 :
msg "A returned  " + str$ A + '  Key  ':
poke $11AC,27 :               //Decimal 27 us hex $1B = Control-[
#Key2Stop :
>!

<sa-B>:<asr :
oa-Q :
rtn :
>!
```

<|>
Domain: All Modules
This is a strange sort of token that doesn't fit in anywhere else so because of its association with <display>, I'm putting it here.

Its macro name is "BAR" but you cannot use <BAR> in a macro, only <|>.

Any time you see <|> it will be in the form of: <| : esc>. What these two do is update the screen with less flicker than <esc : esc>. We will see why the <esc> is crucial in a moment. <|> is shorthand for:

```
<sa-A>:<all : display #off : oa-Q : esc : display #on>!
```

Take the above to Try and give it a shot. You will notice that the screen looks fine, but the cursor is at the bottom of the screen, blinking just before the word "Column." If you use the up &/or down arrow keys an inverse menu bar will float across the middle of the screen.

The problem is that you are really at the Main Menu with the Try file's screen showing. What is needed at this point is an <esc> to flip you back to the Try file. Now you know.

<$1 = GetStr NUM>
Domain: All Modules
Presents a ">" prompt on the bottom line of the screen, allowing up to 60 characters to be entered. NUM sets the maximum number of characters users can enter. (oa-0 does a getstr 60 from the keyboard so you can define macro 0 "live").

```
<sa-A>:<all :                 //Read in up to 8 characters and print
```

```
oa-9 :                      //Bottom of the Try file
$1 = getstr 8 :
print $1 :
>!
```

< GoTo MACRO >
Domain: All Modules
Sends control to the specified macro name. The called macro will NOT return. Goto just jumps to the named macro and keeps on going. No nesting occurs when goto is used.

IMPORTANT: The goto'd macro type can NOT be < asr > . At a minimum it has to be the same type as the current file i.e., if in a word processing file the goto'd macro must be < awp > . Generally, folks make them < all > . Too bad they work that way since making them something other than < asr > opens the possibility that a user will inadvertently fumble-finger a macro they should not call directly.

Actually, you can goto them if you know the secret. You can learn this secret if you study the default macros that came with AW 5.x. Once you find it, you need to understand why this is not a generic solution and needs to be expanded a bit to make sure you are not inadvertently stranding a macro. Happy hunting ;-)

Note that sa-B returns to sa-A if something other than 4 was typed. The goto from sa-C could be changed to a sa-D and nothing would change because sa-D executes a stop token.

```
<sa-A>:<all :
msg 'Type either 4 or anything else and hit return' :
$4 = getstr 1 :             //Allow 1 key press
sa-B :
$0 = "You did not type 4" : goto sa-D :
>!

//If user typed 4 then sa-A is not returned to because sa-D invokes
stop
<sa-B>:<all :
if $4 = "4" then goto sa-C : endif :
>!

<sa-C>:<all :
$0 = "You typed 4" :
goto sa-D :
>!

<sa-D>:<all :
msg %J% + $0 + "  Hit a key" + %K% :
A = key :                   //Real world would call #Key2Stop
msg '' :                    //which does the same thing in my packages
stop :
>!
```

< Hilight L,T,R,B >
Domain: All Modules
Allows you to invert (hilight), any portion of the AppleWorks screen.

L = the left column (1-80)
T = the top row (1-24)
R = the right column (1-80)
B = bottom row (1-24)

L, T, R, and B are all numeric expressions.

This example will invert the entire screen:

```
<sa-A>:<all :
L = 1 :
hilight 1,L,80,24 :
>!
```

This example will hilight starting at column 9 on line L for the number of characters in string variable S(A) plus 10 more, and end on line L + 6. After a key press it will then unhilight a portion of the lines. Another keypress puts everything back to normal.

```
<sa-A>:<all :
A = 0 :
L = 2 :
$0 = "Keep your eye on The tab line at the top of the screen" :
hilight 9,L,len $(A) + 15,L + 6 :
msg %J% + $0 + "  Key" + %K% :
A = key :

hilight 0,L + 2 ,len $(A) + 15,L + 4 :
msg 'Erasing part of the highlighted lines.  Key' :
A = key :
hilight 0,L,len $(A) + 15,L + 6 :
msg '' :
>!
```

If the first parameter is 0, the specified rows will change to normal. The right column value is ignored if the left column is 0.

< Keyto NUM >
Domain: All Modules
Similar to input, but allows you to specify the ending key by ASCII value. lt traps Esc and returns variable Z set to 0 if Esc was pressed otherwise Z is set to the ASCII value of the last key. Note: You cannot < poke $11AC,0 > to turn off Esc since $1B is looked for internally to < keyto > .

: keyto 13 + 128 : would act just like < input > except oa-return would be needed to end it instead of just Return

The following macro will accept input until the user denied key is pressed. if the user presses Esc the macro will stop at that point; otherwise, it will continue by executing macro B.

```
<sa-A>:<all :
oa-9 :                      //Get to bottom so we don't clobber stuff
msg 'Give key that will stop <KeyTo>' :
A = key :
$1 = chr$ A :
msg "Type your input to <KeyTo>. End with  " + %J% + $1 + %K% + "  " :
keyto asc $1 :
if Z = 0 then :
  msg 'Esc pressed' :
  #Key2Stop :
else goto sa-B :
endif :
>!
```

```
<sa-B>:<all :               //Must be all for goto to work
msg 'Got to sa-B. Key ' :
#Key2Stop :
>!
```

< Left STRING VAR,NUM >
Domain: All Modules
Extracts the leftmost NUM characters from STRING. If NUM is larger than the length of the string then the entire string is returned i.e., it doesn't cause a crash.

```
<sa-A>:<all :               //Prints Beagle on message line
$1 = 'Beagle Bros' :
$2 = left $1,6 :
msg $2 + "   " + %J% + ' Key to Continue ' :
#Key2Stop :
>!
```

< Len STRING VAR >
Domain: All Modules
Returns the length of the specified string as part of a variable equation. For example:

```
<sa-A>:<all:
$0 = "Short little thing" :
$1 = "Big" :
A = len $0 :
B = len $1 :
If A > B then :
  msg $0 + "is longer than " + $1 + "  " + %J% + " Key " :
else :
  msg $1 + "is longer than " + $0 + "  " + %J% + " Key " :
endif :
#Key2Stop :
>!
```

< Mid STRING VAR,NUM1,NUM2 >
Domain: All Modules
Extracts any part of a string. The first numeric expression (NUM1), is the offset into the string, and the second (NUM2), is the number of characters to extract.

```
//O = Offset, L = Length
<sa-A>:<all :
$3 = "UltraMacros" :
Savescr :
.Cls 0 :                    //Clear the screen
msgxy 1,19 :
msg "Note that at times L exceeds the number of chars from Offset" :
msgxy 1,20 :
msg "to the end of the string and yet UltraMacros does not crash." :
msgxy 0,128 :
for O = 1 to len $3 : step 3 :
for L = 2 to len $3 : step 2 :
  $1 = mid $3,O,L :
  msg "O = " + str$ O + " L = " + str$ L + " $1 = " + %J% + $1 + %K% +
"  Key" :
  A = key :
next L :
next O :
```

```
Restscr :
#AllDone :
>!
```

< Msg STRING >
Domain: All Modules
Prints a message on the screen immediately below the current data window (i.e., on the dash "---" or underline dividing line). The command syntax is identical to < print > . Messages are normal text unless the message string starts with single quotation marks. (See %J% and %K% later for more inverse/normal text.)

Whenever a message is displayed, the remainder of the line is filled with the second to last character that was already on the line (usually "-" or " "). This automatically erases the remainder of any previous message. As the example shows, a null message erases the entire line. A message expression must always be followed by a ":" or " > ".

Messages can be placed anywhere on the screen < msgxy > and can include some special codes. To make those codes easier to use, msg strings can be specified as follows:

Normal:	<msg "text">	Prints "text" in normal letters
inverse:	<msg 'text'>	Prints 'text' in inverse letters
MouseText:	<msg &@A&>	Displays open and solid apple
Control Chars:	<msg %A%>	Clears to end of line

For the technically minded, the MouseText characters have the high bit set.

The control characters have the three highest bits clear. Here are some control codes which can be used, albeit with great caution:

1	Ctrl-A (%A%)	Clear to end of line
2	Ctrl-8 (%B%)	CLear current line
4	Ctrl-D (%D%)	Clear to end of page
10	Ctrl-J (%J%)	Invert text
11	Ctrl-K (%K%)	Normal text

MouseText can also be included in messages by using chr$ codes from 192 ($C0) to 223 ($DF). This is the only example for msg since there are so many embedded in this and other chapters.

```
<sa-A>:<all :
Clear 255 :
oa-1 :                       //Keep cursor out of display
for I = 64 to 95 :           //Set up ASCII keys for MouseText
  $1 = $1 + chr$ I + " ":
next I :
for I = 192 to 223 :         //Set up the MouseText
  $2 = $2 + chr$ I + " " :
next I :
Savescr :
.Cls 1 :
msgxy 0,10 :                 //ASCII Line
msg $1 :                     //Don't have to use msgxy. Mix & match
.WriteStr 0,11,$2 :          //MouseText line
msgxy 0,128 :                //Reset msg
msg 'Key to Continue' :
A = Key :
RestScr :
| : esc :
```

```
>!
```

< Msgxy Horiz,Vert >
Domain: All Modules
Sets the Horizontal and Vertical coordinates for succeeding < msg > commands.

If the first number (Horiz), is 255, the message will be centered .

Use msgxy 0,128 to reset < msg > to normal. (Actually, anything greater than 128 through 255 works as well.)

This command is just for some UltraMacros 3.x compatibility. Use < .WriteStr > (Chapter 4) for displaying messages anywhere on the screen.

No examples for this command since there are many examples in this reference that a quick oa-F will find.

< Onerr OPTION >
Domain: All Modules
Allows you some control over what happens if an error occurs. An error is defined as a keystroke that causes AppleWorks or a TimeOut application to ring the error bell. Normally a macro continues on wIthout regard to the error (the error bell is silenced as well). There are five onerr options: stop, endmacro, exit, off and goto . When any of the options stop, endmacro, exit and goto are executed an < onerr off > condition is set by the macro.

The onerr status is always reset to < onerr off > when a sequence of macros is done executing.

< Onerr stop >
Domain: All Modules
Stops macros after an error This was changed in Ultra 4.0 to stop everything. To stop only the current macro, use < onerr endmacro > .

```
<sa-1>:<all :
onerr stop :
>!
```

< Onerr endmacro >
Domain: All Modules
Stop macro after an error, If the macro was called from another macro, control returns to the calling macro. This does not shut down all macros; only the current macro is ended.

```
<sa-2>:<all :
onerr endmacro :
>!
```

< Onerr exit >
Domain: All Modules
Exits begin/rpt loop. Can also be used with the : exit : Rpt : trick. See < rpt > for details on said "trick."

```
<sa-3>:<all :
onerr exit :
>!
```

< Onerr off >
Domain: All Modules

Ignore all errors Resets the onerr status to normal, so macros ignore the errors, for better or for worse.

< sa-4 > : < all :
onerr off :
> !

< Onerr goto >
Domain: All Modules
On any error, execute the named macro (sa-B in the example) and then return to the calling macro (sa-5 in this example), one command BEYOND the command that caused the error. If the called macro (sa-B) executes a < stop > command then all macro activity ceases.

A confusion factor here is the word "goto" which in all other Ultramacro contexts means to go to the named macro without putting a return address on the stack. Not in this case. In this case it is exactly like a normal call of another macro. My "guess" is that the 'goto' string allowed Randy to discern the syntax for onerr sa-? and thus to launch the following sa- macro in the normal manner i.e., it will come back unless the destination executes a "stop," command.

In Ultra 3.x there was no < onerr exit > option so folks had to be creative in order to get out of a loop and allow the macro to continue. The following example is based on GEnie posts in 1992 by *Steve Beville* and *Randy Brandt.*

```
<sa-5>:<all :
onerr goto sa-B :            //See comments at Begin :

oa-Q :
esc>5<rtn:                   //Get to "Other Activities"
>2<rtn :                     //Now "File Activities"
rtn :                        //Now List files

B = 0 :                      //A counter for your edification

Begin:
// With the following onerr commented out we go to sa-B on the first
// error and then no more because the first error essentially
// does an <onerr off>

// Removing the comments will cause onerr to be set up each time
// through the loop and you will get a display of just what is
// going on. Note that A never changes once the error occurs.

// The way to "fix" the problem is to reorder the command thusly:

//   A = B :
//   B = B + 1
//   down :                  //Still the error causer
// Rpt :
         //Now we return here which is out of the loop
//>!

// Comment onerr command out and recompile and run again. Hit ESC and
// look in the debugger to see that A is now getting updated each time
// through the loop which shows you that onerr got set to off after
// the first error.

  onerr goto sa-B :
```

```
  down :                      //Error occurs on this command
  A = B :                     //onerr skips this command
  B = B + 1 :                 //we increment B
Rpt :                         //Repeat forever
>!

<sa-B>:<all :
msg "A = " + str$ A + "  B = " + str$ B + %J% + "  ESC to quit " :
>!
```

Steve Beville had a cute bit of code if you could not rearrange the code or perhaps you were not in a Begin/Rpt loop.

```
<sa-A>:<all :
Z = 0 :
oa-9 :
onerr goto sa-B :
B = 1 :
C = 2 :
oa-down :                     //Error occurs on this command
B = C:                        //onerr skips this command
C = C + 1 :                   //we increment C
if Z > 0 then :
  msg 'We have an error Houston. Key' :
  #Key2Stop :
endif :
>!

<sa-B>:<all :
Z = 1 :                       //Flag that error has occurred
>!
```

Note: The above "works" but there is some disturbing guano left in the Try file. It appears that onerr goto does NOT cause the next instruction to be skipped. It is executed AND "A=B" is printed to the file.

I then changed <A = B> to <rpt> since it was going to be skipped and perhaps the intent was not to skip complex expressions, only a simple no parameter command.

This was much worse. A Control-C was inserted into the file (center text), an upper-case T was printed, and a "Where do you want to Print this file" menu popped up.

At ths point I'd recommend that <onerr goto macro> not be used. <onerr endmacro> worked fine.

<Posn VAR1,VAR2>
Domain: All Modules
Assigns the current cursor position to the two variables following the token. Here how it works for each AppleWorks application:

	VAR1	VAR2
Word Processor:	column	line
Data Base:	category	record
Spreadsheet:	column	row

If the cursor is not in one of these three applications, both variables will be set to zero. **Ed:** I do not find this to be true. I find that the variables get set to somewhat random values if I'm at the Main Menu, Desktop Index, or Add Files menus. The values change if I enter a desktop file, exit to the Main Menu and call a macro that contains <posn>. Entering another file and immediately to the Main Menu where the macro is called will yield different values. Reentering/exiting the original file will result in the same values for the two variables.

<posn> is compatible with Timeout applications that use AppleWorks applications. For example, Timeout Graph works in the Spreadsheet, so <posn> can be used with it.

In the following, if on line 120 and column 2 in the Word Processor, X will be 2 and Y will be 120.

```
<sa-A>:<awp :
posn X,Y :
>!
```

<Pr# NUM EXP>
Domain: All Modules
Determines where the <print> command sends its information.

<pr# 0> sends all <print> characters to AppleWorks. This is the normal state of affairs.

<pr# 1> sends the characters to the first printer in your AppleWorks printer list. Because AppleWorks 4 and 5 have a limit is 5 printers, the <pr#> limit is also 5. For AppleWorks 3 there is a limit of 3 printers which is enforced by UltraMacros 3.x. Unfortunately, if UltraMacros 4.3 is used with AppleWorks 3 it thinks the limit is now 5. No idea what happens if you try and send data to a printer 4 or 5 in AppleWorks 3.

You must use <pr# 0> to reset the <print> command when you're done.

Following is from the AppleWorks 4 and 5 /EXTRAS/MACROS/MACRO.SAMPLES. Not clear to me what the "^T to printer" comment is all about since the three chr$ codes are: Control-N, =, and Control-M

```
<sa-A>:<all :                //^T to printer
pr# 1 :                      //Text to first printer on list
//         ^N        =           ^M = Return
print chr$ 14 + chr$ 61 + chr$ 13 :
pr# 0 :                      //Put it back to screen
>!
```

<pr#> may not work with all interface cards. lt does work with the printer and modem ports on the IIgs. (**Ed:** See <chr$> for an example. >

NOTE: Don't define chr$ 0 in a string to use with the <pr#> command. Use any of the strings $1 through $99 and not $0 with chr$ 0 - 0 string 0 is actually macro 0, and will ignore any characters past the first chr$ 0.

<Print NUM,STRING,ALMOST ANYTHING>
Domain: All Modules
Print has the most variations of any single UltraMacros command. The compiler will be happy to point out any errors You might make, but studying this section will make you much less error prone

Printing Literal Text
<print> allows a literal text string to be printed. You may use either double or single quotes around the text. The limit is 80 characters of text at a time.

```
<sa-A>:<all :                //prints "Literal text <rtn>"
oa-9 :                       //Bottom of Try file
print "Literal text <rtn>" :
>!
```

The < rtn > is not converted to an actual Return so the cursor is left at the end of the line.

```
<sa-A>:<all :
oa-9 :                       //Bottom of Try file
print '"double" quotes inside  "single" quotes' :
rtn :
>!
```

Printing Numeric Variables
< print > can be used to display the value of any numeric variable. For example, if variable Q holds the desktop number of a specific file, this macro sequence would return you to that file:

```
<sa-A>:<all :
oa-Q :                       //Bring up the desktop index
print Q :                    //Highlight the entry
rtn :                        //Select it
>!
```

When printing numeric variables, a "$" immediately after the print statement will cause the variable's hexadecimal value to be displayed In either two or four characters.

```
<sa-A>:<all :
A = $FF :
B = 255 :
C = $FFF :
X = 61453 :
oa-9 :                       //Bottom of Try file
print$ A :
rtn :
print$ B :
rtn :
print$ A + B :
rtn :
print$ C :
rtn :
print$ X :                   //A little hex humor from Randy
>!
```

Numeric variables can also be printed as characters rather than numbers. The < chr$ > token converts the numeric value to the equivalent keyboard command. See the file KeyChart in this distribution for a complete list. Here is a sample:

The KeyChart file shows that 185 is an oa-9 and that $41 is an upper case "A". This sample will jump to the end of the file and then print an "A".

```
<sa-A>:<awp :
X = 185 :                    //Decimal 185 = oa-9
Y = $41 :                    //Hex $41 = decimal 65 = A
print chr$ X :               //To the bottom of the Try file
print chr$ Y :               //Print the character A
>!
```

You can print more than one numeric variable with a single < print > token. You have to separate them by a plus sign < + > . The same goes for printing a combination of numeric and string variables.

```
<sa-A>:<awp :
X = 185 :                    //Decimal 185 = oa-9
Y = $41 :                    //Hex $41 = decimal 65 = A
print chr$ X + chr$ Y :      //To the bottom of Try and print A
>!
```

The one hundred string variables can be printed by themselves, or with numeric variables &/or literal text. You do have to separate them with a plus < + > symbol.

```
<sa-A>:<all :
oa-9 :                       //Get to the bottom of the Try file
X = 100 :
Y = 200 :
print "Numeric variable X = " + str$ X + "  and Y = " + str$ Y :
rtn :   //leave this out and cursor is left up a line

// Not too sensible, but this shows that string variables, literal
// text and numeric variables can all be printed by one print

$1 = "Numeric variable X = " :
$2 = "and Y = " + str$ Y :
$3 = $1 + str$ X + "  " + $2 :          //Or gather together in one
print $3 :
rtn :
print $1 + str$ X + "  " + $2 :         //Left rtn token off this one
>!
```

NOTE: ALL PRINT STATEMENTS MUST BE FOLLOWED BY ":" OR ' > ". Other tokens can be followed by spaces and then another token, but < print > is an exception.

This does NOT print two slashes followed by a return. In fact, it will not compile because the "//" is seen as the beginning of a comment so the tail end of the macro is lost. If you move stuff around, it still won't compile. Just give up trying to print two slashes with the same print statement.

```
<sa-A>:<all : print "//" : rtn>!
```

This DOES print two slashes followed by a return.

```
<sa-A>:<all : print "/" : print "/" : rtn>!
```

Preceding isn't sensible, but it isn't going to get changed at this stage of the game so just live with it.

Going on a bit more, this compiles, but doesn't yield what you think it should i.e., "//," rather, you get ": > "

```
<sa-A>:<all>//<rtn :
>!
```

Changing it a bit will give you a syntax error. Go figure.

```
<sa-A>:<all>//<rtn>!
```

Finally, from the UM 3.x Beagle and 4.0 Jem manuals we have the following which I seem to be too dense to see how these examples show me anything different than assigning from some string other than $0 to $2 differs from what I see when I compile and run these macros.

I've played around with this quite a bit and I don't make the connection. Please let me know if you do so I can clarify this for others like myself.

These strings can contain text or command keystrokes. To define a string with commands instead of text, just define macro 0 (zero), the same as $0 and then use a macro like this: (**Ed:** I've tried defining a macro 0 i.e., < sa-0 > : < all : oa-1 > ! and compiled it with no error messages (which, it turns out, is different than successfully compiling it). However, when I call sa-0 I get the current contents of string $0 printed to the file I'm currently in so I don't know what the phrase "just define macro 0 the same as $0 means.)

```
<sa-A>:<all : $2 = $0 : print $2>! execute macro 0 (zero)
```

Because macro 0 (zero) and $0 are the same thing

```
<sa-A>:<all : print $0>! is exactly the same as: <sa-A>:<all : sa-0>!
```

< Right STRING VAR,NUM >
Domain: All Modules
Extracts the rightmost NUM characters from STRING

```
<sa-A>:<all :               //Prints 'Bros'
$1 = "Beagle Bros" :
$2 = right $1,4 :
print $2 :
>!
```

< Screen NUM EXP,NUM EXP,NUM EXP >
Domain: All Modules
Reads any part of the AppleWorks screen into a string variable. lt is used like this:

Read current file name from top line

```
<sa-A>:<all :
$1 = screen 7,1,15 :
msg $1 + "   Key" :
#Key2Stop :
>!
```

The first parameter is the left column (1-80) The second parameter is the line (1-24)The third parameter is the length (1-80

Screen usually treats all characters as normal text, regardless of how they appear on the screen. To get literal screen bytes, add 128 to the second parameter. The line will be in memory starting at $900. Normal text will have the high bit set, and MouseText will range from $40 to $5F.

Ed NOTE: I've not been able to see any difference between an inverse and non inverse line at $900. Also, I see normal text with the high bit = 0 and MouseText with it set i.e., with MouseText enabled by ^T and pressing @ yields the solid Apple = └ = 192.

Randy NOTE: Don't try using < msg > with a string made this way.The bytes at $900 do not contain standard text characters. Printing them could clobber AppleWorks.

< Str$ VAR NAME >

Domain: All Modules
Converts numeric variable to a decimal character string. It must be used as part of an equation. Here are some examples:

```
 < sa-A > : < all :
A = 4 :
$3 = " A = " + str$ A + " " :
print $3 :
 > !
```

Prints "255" because 255 is the decimal equivalent of the hexadecimal $FF assigned to B.

```
<sa-A>:< all :
B = $FF :
$1 = str$ B :
print $1 :
>!
```

< Val STRING VAR >
Domain: All Modules
< val > is the opposite of < str$ > . It converts a string variable to a numeric value and must also be used as part of an equation. If the specified string starts with a non-numeric character, the value will always be zero. If the first character is a number, this number (and other numbers following immediately after it) will be converted to a numeric value. Here are some examples:

```
<sa-A>:<all :
oa-1 :                      //Get cursor away from printing
SaveScr :
.Cls 1 :
$1 = "test4" : A = val $1 : //A = 0
$2 = "48612" : B = val $2 : //B = 48612
$3 = "280ZX" : C = val $3 : //C = 280
$4 = "65535" : D = val $4 : //D = 65535
$5 = "65537" : E = val $5 : //V = 1
.WriteStr 0,10,$1 + "   " + str$ A :
.WriteStr 0,11,$2 + "   " + str$ B :
.WriteStr 0,12,$3 + "   " + str$ C :
.WriteStr 0,13,$4 + "   " + str$ D :
.WriteStr 0,14,$5 + "   " + str$ E :
msg ' Key to continue ' :
K = key :
RestScr :
| : esc :
sa-F>sa-A<rtn>N!
```

The last value is because the maximum a variable can hold is 65535. Add 1 = 0, add another = 1, which is the result here.

< Wait NUM EXP >
Domain: All Modules
Delays a macro for a set amount of time, or until a key is pressed . The actual delay will vary depending on your computer. Experiment to find the approximate delay needed for a second or a minute on your computer. Here's a macro you can use to calculate delay values. Calibrate wait by timing bell interval.

```
<sa-A>:<all :
msg "Enter a delay value.  ESC to exit. " :
```

```
$0 = getstr 5 :
if $0 = "" then exit : endif :
D = val $0 :
bell :
wait D :
bell :
Rpt :
>!
```

ln UltraMacros, < wait > lets you set a default keystroke value. The default vale is 0 (NOTE). If the < wait > time expires, the default is returned, but if the user presses a key, that key is returned. Poke #waitkey with the high ASCII value. Peek it after the wait to get the result. This macro loops after wait, unless the user presses something other than Return.

```
<sa-A>:<all :
msg 'Return to continue, any other key to stop' :
poke #WaitKey,141 :          //Set default to Return 13+128
wait 500 :
A = peek #WaitKey :
poke #WaitKey,0 :            //Put default back
ifnot A = 141 then :
  msg '' :
  stop :                     // non-Return pressed
else Rpt : endif :           //User pressed Return or waited
>!
```

NOTE: Once the default in #WaitKey is changed it remains changed for the rest of the AppleWorks session unless it is changed by a subsequent poke #WaitKey,0.

< Wake MACRO at NUM EXP:NUM EXP >
Domain: All Modules
Puts a macro to *sleep* and wakes it at a designated 24-hour time. (24 hour time is important i.e., 9:04 AM is 9:04. However, 9:04 PM is 21:04. After a < wake > command has been issued, you can work normally using macros and any UltraMacros or AppleWorks commands. When the clock's time matches the sleeping macro's time, it springs to life. Use it to set alarms, automatically save a file every few minutes, or shut everything down at 5:00 PM so you can go home!

The following example will start macro "B" at noon. Then when macro "B' wakes up, it will set macro "C ' to wake up at 5:00 pm.

```
<sa-A>:<all :
wake sa-B at 12:00 :
>!

<sa-B>:<all :
bell : bell :
msg "It's lunch time!" :
wake sa-C at 17:00 :
>!

<sa-C>:<all :
Bell : bell :
msg "Quitting time  " + %J% + "Key" :
#Key2Stop :
>!
```

Only one macro can be ″sleeping ′ at a time, but as shown in the previous example, each macro that ″wakes up″ can put another macro to ″sleep″. The time must be given in 24-hour format (0:00 to 23:59); variables may be used:

When sa-A is pressed, the hour and minute variables are set to the current time. The current file is saved, and macro ba-A is set to wake and then save the current file every ten minutes unless the < nosleep > command is used to deactivate it. (In this example the time has been changed to 1 minute so you can see it work in a reasonable time frame. Ten minutes or longer is a more realistic cycle time.)

```
<sa-A>:<all :
$0 = time24 :                 //5:22, not 05:22
$1 = right $0,2 :             //Get the minutes
M = val $1 :
$1 = left $0,2 :              //Might be 5:
H = val $1 :                  //val stops on first non digit
goto ba-A :                   //And don't ever come back
>!

<ba-A>:<all :
oa-S :                        //Save the current file
M = M + 1 :                   //Wait 1 minute
ifnot M < 60 then :           //Check if crossed 60 minute boundary
  M = 0 :                     //Back to zip for you my good man
  H = H + 1 :                 //New hour
endif :
ifnot H < 25 then :
  H = 0 :                     //Its midnight boys and girls!
endif :
wake ba-A at H:M :
| : esc :                     //Update screen time
msg str$ H + ":" + str$ M :
>!
```

NOTE 1: Of course, this command won't be too useful if You don't have a clock.

NOTE 2: Suppose that you invoke wake in this fashion:

```
<sa-A>:<all :
wake ba-A at H:M :
Link sa-B in "TextStuff" :
>!
```

Further suppose that sa-B in the "TextStuff" package does not execute an unlink before the wake times out. In that case the wake will look for a ba-A macro in "TextStuff" and execute it if found. If not, it just goes away.

Macro Sets and Task Files

UltraMacros′ macro sets differ from UltraMacros task files in that they may be cached in desktop memory as well as being stored on disk. Each time a task file is loaded, UltraMacros attempts to store it on the desktop so that when any macros in it are called it is available Instantly without any disk access.

< Launch STRING VAR >
Domain: All Modules
Launches the specified Task file and executes the second macro in that macro set. To restore your default macros, launch SEG.UM. Task files which are not in memory or on the AppleWorks startup disk can be launched by specifying the complete pathname. For example, launch "/HD/TASKS/UPDATE.CHECKS".

```
<sa-L>:<all : launch "UM4.0.SYSTEM">!    //Restore defaults AW 4.0

<ba-L>:<all : launch "SEG.UM" >!       //Restore defaults AW 5.x
```

Note that the standard has changed from sa-L to ba-L. It is my guess that this was chosen because folks are less likely to fumble finger ba-L than sa-L since most folks don't write ba- keyboard accessible macros. They use them as asr type macros.

Here is my ba-L macro from my default macro package. It is possible that there might be times when you call this macro that it will not bring up a list of launchable task files because Z was set to a value that caused < find > to "not see" the string in $0. See < find > for details. You can elect to slip in : Z = 0 : just prior to the < find > and that will solve the "problem." When a < find > fails Z is set to 0 to indicate this to the caller so simply calling the macro a second time also works.

```
// Brings up the list of launchable task files.

<ba-L>:<all :
$0 = "Ultra Options" :
oa-ESC :                    //Timeout menu
find :                      //Find Ultra Options
if Z = 0 then endmacro : endif :     //End here if not found
rtn :                       //Select Ultra Options
$0 = "Launch a Task" :
  find :                    //Find it
  if Z = 0 then endmacro :  //No way this should fail.
  endif :
rtn :                       //Select Launch a Task

//If one or more previously launched from disk, then all not shown
// unless you ask.
$0 = "Launch Task from disk" :
  find :                    //Make all applications available
  if Z = 0 then endmacro :
  endif :
rtn :                       //Found. Gives user list of UM task files.
>!

====

Here is my ba-L macro from my task files:

Return to the default macros for AW 5.x
<ba-L>:<all : launch "SEG.UM">!
```

< Call MACRO in STRING VAR >
Domain: All Modules
Loads the macro set and then executes the specified macro instead the second one in the set. As soon as that macro (or series of macros is done executing, UltraMacros reloads the original macro set and continues with the calling macro. The called macro (or series of macros), may NOT < call > another macro in yet another task file. See < link > for a mind numbing work around.

The following macro loads a task file, calls macro M, reloads the original macros and goes to macro sa-X.

```
<sa-A>:<all : call sa-M in  "MENU.MACROS"  : goto sa-X>!
```

Something you can do in the called macro package:

```
#BoxDraw = ba-9             //Sets up the label #BoxDraw

#BoxDraw :                  //Calls ba-9 in Mac.Common from
                            //within Mac.Common

<#BoxDraw>:<all : stuff>!   //Definition of #BoxDraw (ba-9)
```

There are a number of ways things can be set up in the calling task file.

First Example sets up this label in the calling task file:

```
#BoxDraw = call ba-9 in "Mac.Common"
```

Later in the program a line:

```
Stuff:
#BoxDraw :                  //Makes #BoxDraw look like a local.
More Stuff :
```

Second example:

These show that the macro name within the calling program can be different or the same as in the called.

```
<sa-A>:<all : call ba-9 in "Mac.Common">!
        OR
<ba-9>:<all : call ba-9 in "Mac.Common">!

Third example:

#BoxDraw = ba-9             //Set up label

<#BoxDraw>:<all : call #BoxDraw in "Mac.Common">!
```

Finally, from Will Nelken's "Ultra to the max," we learn that $0 is not preserved when a CALLed macro returns to the CALLing macro. All others are preserved. As you might have already guessed, (based on the myriad of times you have been told), $0 is not a string variable. It is a macro. Most times you can treat it as a string variable and all will be well. There are these few situations where it gets clobbered that the ever alert TAPL programmer must watch for.

Some folks avoid using $0 for this uncertainty reason. I use it at the slightest provocation across several lines of macro code since I know that no reasonable person stores long term data in it so I don't have to worry about clobbering a calling macro's precious string.

< Link MACRO in String VAR >
Domain: All Modules
Loads the macro set and then executes the specified macro instead of the second one in the set. When that macro (or series of macros) is done executing, UltraMacros stops, leaving the new macro set active.

The following macro loads a task file, calls macro M and stays in MENU.MACROS until < unlink > is used. At that time return is made to sa-A and the "goto sa-X" is executed.

```
<sa-A>:<all link sa-R in  "MENU.MACROS"  : goto sa-X>
```

According to Page 5.5 in "Ultra to the max!" the "work around" to not being able to < call > a macro from a < called > task file is to < link > to the fist macro, which then can < call > a macro in yet another task file. (No more < linking > in that macro set though ;-)

Unfortunately, this is not true as Page 15.1 points out later:

" < Call > and < Link > both store the name of the calling task file in the same Desktop memory location. So, < Unlink > will return to a calling macro set, even if the secondary macros set was called with < Call > ."

If they both store the return address in the same location there is no way to get back to the original task file that did the < link > to the secondary task file that does a < call > to a tertiary task file.

As far as my experiments have shown so far, UltraMacros does not crash if more than one < call > or < link > is made. The problem is getting back to the original task file at the point where it can continue its job (or terminate it if complete).

To see one solution on handling multiple < Link > commands without any < Unlink > commands, load, compile (ba-C), save as task file (sa-Ctrl-T), the three files TF1, TF2, and TF3. Launch TF1 and call sa-A. Follow the bouncing ball from that point to the end.

< Unlink >
Domain: All Modules
Returns from a < link > macro set back to the original set. This works like < call > except the return to the original macros is triggered by a specific < unlink > command instead of by the macro ending.

If-Then-Else Logic

One of UltraMacros' best features is its true conditional capability utilIzing if-then-else logic. UltraMacros allows a full range of conditional commands using the numeric and string variables. Seven tokens are involved with conditional logic: if, ifnot, and, or, then, else, endif. All of these commands work just as they did in UltraMacros 3.x.

< If >
< Ifnot >
Domain: All Modules
The key to a conditional macro: < if > and < ifnot > are always followed by a numeric or string variable:

< sa-A > : < all : if A

which is followed by an operator (greater than > , less than < , or equals =)

< sa-A > : < all : if A operator

which is followed by the expression to be evaluated and more tokens

< sa-A > : < all : if A > 8 then A(3) = len $90 > !

If the statement is true, the macro continues normally. If the statement is not true, the macro ends (unless an < else > is present later in the macro).

< ifnot > works the same as < if > except that the statement must be false for the macro to continue normally.

All numeric conditionals must start with one of these six formats:

if A = or ifnot A =
if B > or ifnot D >
if C < or Ifnot C <

The equation is completed with any valid numeric expression such as:

< sa-A > : < all : if A = C + 4 then print A : else stop > !

All string conditionals must start with one of these six formats:

```
If $0 =  or   Ifnot $3 =
if $1 >  or   Ifnot $4 >
if $2 <  or   Ifnot $5 <
```

The equation is completed with a string expression, which could another string variable name, a literal text string, or one of the legal string definition tokens such as < date > , < time > or < screen >):

```
<sa-T>:<all : ifnot $6 > time then goto sa-T>!
```

NOTE: There exists a really confusing "quirk" concerning < if > .
If you have the following:

```
<sa-A>:<all :
if Z < 1 or > 2 then :
  A = 1 :
endif :
>!
```

The above will compile even though it is syntactically wrong. What it will do is start printing your macro file from The "2" to the end of the file into whatever file or menu your cursor currently resides. Naturally, it is a WP file type that really gets messed up.

Here is the corrected macro in case you didn't spot the error in the above (it took me a bit and I initially wrote it ;-)

```
<sa-A>:<all :
if Z < 1 or Z > 2 then :    //Simple change, dramatic effect!
  A = 1 :
endif :
>!
```

See "Defining Numeric Variables" and "Defining String Variables" section earlier in this chapter where use of tokens in an < if > or < ifnot > equation is beat to death.

< And >
< Or >
Domain: All Modules
Used with < if > and < ifnot > to test multiple equations. This macro prints true if both expressions are true:

```
<sa-A>:<all : if X > 8 and Y = 2 then print "true">!
```

This macro will print "true" if one of the expressions is true:

```
<sa-A>:<all : if X > 8 or Y = 2 then print "true">!
```

With all <or>'s, if any equation is true, the result will be true.

With all <and>'s, if any equation is false, the result will be false.

Don't forget that <ifnot> simply reverses the result.

Both <and> and <or> may be used in the same <if-then-else> sequence. The equations are evaluated from left to right, with the cumulative result being <and>'d or <or>'d with the current result. This means that the rightmost evaluation has priority. A final true <or> will make the result true regardless of what precedes it, and a final false <and> will make the result false.

In this example the E = 4 makes the <if> true. If E did not contain 4 then either A or B and C and D must be true i.e., C = 2 and D = 3 is false unless A or B is true.

```
<sa-A>:<all :
clear 255 :                  //All numeric variables = 0
E = 4 :
if A = 1 or B = 1 and C = 2 and D = 3 or E = 4 then :
  msg 'Made it through the maze.  Key ' :
  #Key2Stop :
endif :
>!
```

<Then>
Domain: All Modules
Does absolutely nothing but take up one byte of space. Randy (at one time at least), thought it useless. I, on the other, hand simply adore it. Do keep in mind who wrote UltraMacros and who simply is writing about it when deciding on whether to use it ;-)

lt's used to make macro if-then-else logic more readable:

```
<sa-A>:<all : if A > 4 then C = 3>!   //is clearer than:
<sa-A>:<all : if A > 4 C = 3>!
```

<Else>
Domain: All Modules
The <else> reverses the true-false condition of the <if-then-else> logic. If the statement is true, then execute the first part, if the statement is false, then execute the second part which follows the <else> token either to the end of the macro or an <endif> token, whichever occurs first.

The <else> can really bite you if you are not careful. The problem is that in UltraMacros the <if> and <else> do **not** follow the rules you might have learned with other languages, such as C. The following example should make the problem clear.

The indentation here implies that the else will only be executed when the first <if> is true and the second false. That is not the way it turns out. If either the first or second <if> is false the <else> is executed.

```
<sa-A>:<all :
oa-1 :
SaveScr :
.Cls 1 :
A = 5 :
```

```
B = 7 :
if A = 4 then :
  .WriteStr 0,10,"First if true" :
  if B = 7 then :
     .WriteStr 0,11,"Second if true" :
  else :
     .WriteStr 0,12,"First else true" :
endif :
msg 'Key to continue' :
K = key :
oa-9 :
RestScr :
| : esc :
>!
```

Many times the simplest thing to do is to call another macro where the 2nd <if> is performed.

Everything can be done in the same macro through a series of carefully placed <if>, <ifnot> and <else> statements. Keep in mind that in the real world the first <if> might perform a number of actions prior to calling sa-B.

```
<sa-A>:<all :
oa-1 :
SaveScr :
.Cls 1 :
A = 5 :
B = 7 :
if A = 5 then :
  .WriteStr 0,11,"first if true" :
  sa-B :
endif :
msg 'Key to continue' :
K = key :
oa-9 :
RestScr :
| : esc :
>!

<sa-B>:<all :
if B = 6 then :
.WriteStr 0,12,"2nd if true" :
else :
  .WriteStr 0,12,"2nd if false. Did else" :
endif :                    //endif not needed since end of macro but
                           //style does count!
>!
```

This single macro should give the same results of the two above. I think you will find it a bit harder to understand. Do watch those <or> and <and> tokens. They have to be in the order that makes sense for the test you are performing.

As a test I compiled the two macro solution and the following single macro solution without any <.WriteStr> lines. The two macro solution is 16 bytes less. Of course, there is the cost of another macro name.

```
<sa-A>:<all :
oa-1 :
```

```
SaveScr :
.Cls 1 :
A = 4 :
B = 6 :
if A = 4 then :
  .WriteStr 0,10,"Do A = 4 stuff" :
endif :
if A = 4 and B = 7 then :
  .WriteStr 0,11,"Do A and B are true" :
endif :
if B < 7 or B > 7 and A = 4 then :
  .WriteStr 0,12,"Do A true and B false" :
endif :
msg 'Key to continue' :
K = key :
oa-9 :
RestScr :
| : esc :
>!
```

<Endif>
Domain: All Modules
Does nothing unless used with either an <if> or <ifnot> (see above). its purpose is to cancel the conditional status of a macro and cause any commands following the "endif" to be executed regardless of any preceding <if> conditions. <elseoff> is no longer available which is no loss since <endif> has the same meaning as <elseoff>.

```
<sa-A>:<all :
//Setup stuff
if A = 4 then :
  sa-B :
  if Q = 9 then :
     sa-C :
     if R = 4 then :
        sa-D :
endif :                    //One endif closes all preceding if's
//More stuff
>!
```

For-Next LOOPS

One of UltraMacros's handy new features is the ability to use for-next loops to execute commands a set number of times.

<For VAR = NUM to NUM>
Domain: All Modules
The <for> command begins a for-next loop by specifying the variable to use as a counter, and the range of values to cycle through.

```
<sa-A>:<awp :
for I = 1 to 10 :
  print 1 :
  rtn :
next I :
>!
```

<Next VAR>
Domain: All Modules
The <next> command indicates the end of a for-next loop. UltraMacros updates the specified variable and checks to see if the loop is finished, or if it should keep executing. The specific variable must be specified, since for-next loops may be nested.

```
<sa-A>:<all :
oa-9 :                      //Get beyond last in file for print
for I = 1 to 9 :
  rtn :
  for J = 1 to 5 :
     print I :
     spc :
     print J :
     (spc) 6 :              //So cool repeat command
  next J :
next I :
>!
```

< Step VAR >
Domain: All Modules
The < step > command allows you to determine what happens to the counter variable when < next > updates it. Steps may be positive or negative.

```
<sa-A>:<awp :               //Count down
oa-9 :                      //Bottom of Try file
for I = 10 to 1 step - 1 :
  print I :
  spc :
next I :
>!
```

```
<sa-A>:<awp :               //Print even numbers
oa-9 :                      //Bottom of Try file
for I = 0 to 8 step 2 :
  print I :
  spc :
next I :
>!
```

For Advanced UltraMacros Users Only

The following UltraMacros tokens are very specialized and shouldn't be used unless you understand exactly what you want them to do. They were included because we thought there might still be a few hackers out there who like to deal directly with their Apples. Besides that, we'll use them to write some pretty powerful macros ourselves

See the file Macros Special on the UltraMacros disk for examples.

< Jsr >
Domain: All Modules
The < jsr > token is used to run machine language subroutines. < call > has been changed into a completely different command. See < call > earlier in this chapter.

Surprisingly, < jsr > simply does a jsr to the address specified . lt is up to you to make sure that the address is valid and that the routine will return to the macro via the machine language instruction "rts" with all bank switches set properly.

A good place to poke in machine language subroutines is the AppleWorks temporary work buffer from $800 to $9FF.

CAUTION: The buffer is destroyed by AppleWorks disk access and by a few UltraMacros commands. Be careful.

When a macro is operating, the alternate zero page is active, as well as the second bank of $D000 memory. Page 1 of the 80-column display is active. If you change any of these, they must be restored before your routine returns control to UltraMacros. Otherwise, AppleWorks will surely die.

The < jsr > command is a bonus feature and must be used carefully by experts only. Know what you are doing before you use it!

< Poke >
Domain: All Modules
< poke > and < pokeword > are handy, albeit dangerous, commands. Use them to build machine language subroutines for use with < jsr > or to change special locations. < poke > stores a single byte at the specified memory address.

```
<sa-A>:<all : poke $10F1,1>! force overstrike
```

The insert cursor continues to blink until another key is pressed, then you see that the cursor is changed. To make it instantly change add an invalid key like this:

Ed: Did you "invalid key" fans ever wonder why and invalid key isn't a syntax error? Well, I believe that I've figured it out. The invalid key isn't invalid to UltraMacros, it is invalid to AppleWorks. Normally this would cause the error bell to ring, however, unless < onerr > is set the error is ignored by UltraMacros and the error bell is suppressed. Control D,E,O, etc., are other invalid key combination that you can use.

```
<sa-A>:<all : poke $10F1,1 : ctrl-x>!
```

< PokeWord >
Domain: All Modules
< pokeword > stores a two-byte value in two consecutive bytes starting at the specified memory address. For example:

```
<sa-A>:<all :
pokeword $300,$201 :        //$201 = 513 decimal
A = peek $300 :
B = peek $301 :
msg  "A = " + str$ A + "  B = " + str$ B  + "  " + %J% + "Key" :
#Key2Stop :
>!
```

NOTE1: A single byte can contain a number from 0 to 255, for a total of 256 possible values per byte. To represent larger numbers, we can use two bytes. The second (high) byte is multiplied by 256, and the result is added to the first (low) byte to form a number ranging from 0 to 65535. UltraMacros variables are two bytes, hence the 65535 upper limit. High byte (256 * 255) = 65280 + Low byte 255 = 65535.

< Peek >

Domain: All Modules
< peek > returns the value found at a specified address. See < PokeWord > for an example plus a myriad of places throughout this chapter.

< PeekWord >
Domain: All Modules
< peekword > returns a two-byte value at the address. lt multiplies the second byte by 256 and then adds the first byte to the result. See preceding NOTE1 in < PokeWord > .

This example from the Macros Ultra file uses < peek > to determine the current file number:

Leave ″1″ file; go to main menu

```
<sa-L>:<all : Q = peek #openfile : oa-Q : esc>!
```

Return ″2″ the file we left
```
<sa-2>:<all : oa-Q :print Q : rtn>!
```

The following example of peekword gets the data base record count:

```
<sa-A>:<all : R = peekword #dbrecs>!  //R = total record
```

NOTE: peek will automatically use main memory if page 0 is Peeked. No tricks are needed to run special AppleWorks zero page locations.

External Dot Commands

External commands are probably the most significant new feature of Ultra 4.0. Called "dot" commands because their names all start with a period, over 500 of these commands may be added to Ultra 4.0. There were 45 dot commands included with Ultra 4.0, with over 40 more on the Ultra Extras disk.

Others have written a large number of dot commands which are available from several sources.

Dot commands are stored in init files and added to Ultra 4.x at bootup. Adding new commands is as easy as adding a file to AW.INITS subdirectory.

Prior to Ultra 4.x's dot commands, UltraMacros 3.x offered a few extra ampersand "&" commands. They were limited in that they couldn't be part of macro equations, and there weren't very many. In any case, they have been dropped from UltraMacros 4.x

There are three different types of dot commands:

Stand-alone commands

Stand-alone commands do not return a result, so are never part of an equation. For example < sa-A > : < all : .say "Hello" > ! will print the message "Hello" on the bottom line of the screen, wait for a key press, and then restore the bottom line.

String commands

Nomenclature alert! What is a "string?" A string is a series of ASCII characters. Since a series is one after another you can see where the word string comes from i.e., characters "strung along."

Normally we think of a string as a series of printable characters, however, it can (in theory and reality), contain non-printable characters. Now I suppose you want to know what the heck is a non-printable character? Here is my shot at the definition.

On an Apple][keyboard there are four meta keys, Control, Shift (Caps Lock is shift with a brick on it), Open-Apple, and Closed-Apple. Each of these keys has the ability to redefine the meaning of the other keys on the keyboard i.e., a-z to A-Z by shift.

The Control key treats the characters from @ to _ as having a value 64 less than their normal value i.e., A = 65, Control-A = 1. (See file KeyChart for visual understanding).

The Open-Apple key modifies the character by adding 128 i.e, high ASCII value so A = 65 and oa-A = 193.

We have not been told just what the Solid-Apple key, Solid-Apple-Control, Both-Apple, and Both-Apple-Control key do to modify the UltraMacros perception of a key. Suffice to say that each modifies the key such that it can be percieved as unique by UltraMacros.

String commands either return a string result to define strings or they require a string input. For example, < sa-A > : < all : $50 = .peekstr $B5F : msg $50 > ! will define $50 as "OA-? for Help" and display that message. ($B5F works for AW 4 or 5. The original Jem Manual specified $AF1. That probably works with a leading space, however, try $AF2 if problems arise with AW 3.)

Numeric Commands

Numeric commands return numeric results so they are used to define numeric variables &/or they require a numric input. For example, < sa-A > : < all : X = .eof > ! sets X equal to the last Word Processor line, last Data Base record, or the last Spreadsheet row containing data. By the way, "eof," stands for "End of File," so as you can see, these command mnemonics are not created in a vacuum (or an ASCII blender ;-)

Try out the New Dot Commands

For AW 4 or 5 look in the /Extras/UltraMacros folder and you will see a number of files that begin with the word, "Dot." Load them and try out the new dot commands.

Dot Command Reference

The following summary introduces the included commands. When variables are required by a command, variable names may be used to make the description more readable. For example, instead of:

STRING VAR = .peekstr NUM, its given as $1 = .peekstr Address. Of course you may use string variable names other than $1. Address is a NUM so it may be a literal number or a variable.

STRING may be a literal string i.e., "Hello World" or a string variable that contains the literal string.

ProDOS uses *class 0 strings*, where the first byte in the string space represents the length of the string (not including the length byte) and is followed by the ASCII-encoded bytes representing the characters. (I believe this used to be called a P-string, a shorthand for Pascal-string.)

In the following, when an address points to a string, those strings must be a Class 0 strings e.g., on reading the length byte will be used to determine the length of the string and on writing the first byte will be the length of the string.

From the above it follows that the string space must be the length of the string plus one byte.

Default Dot Commands

< .AskYN $1 >
Domain: All Modules
This returns consistent results no matter how a user may have patched their Yes/No order.

Useful Returns:
Z = 0 Esc pressed
Z = 1 = N
Z = 2 = Y

More (Useful?) Returns:
Z = 147 = oa-Ctrl-S
Z = 155 = oa-Esc
Z = 209 = oa-Q
Z = 211 = oa-S

No Longer Exist Returns
Z = 191 = oa-/ or oa-? pressed

```
<sa-A>:<all :
$1 = 'Esc = exit macro, N = No, Y = Yes, oa-Esc, oa-Q' :
```

```
.AskYN $1 :
if Z = 0 then :
   msg 'Esc pressed. I quit. Key' :
   #Key2Stop :
endif :
if Z = 1 then :
   msg 'N pressed. Key' :
   A = key :
   msg '' :
   rpt :       //Rpt with no Begin goes to top of macro
endif :
if Z = 2 then :
   msg 'Y pressed. Key' :
   A = key :
   msg '' :
   rpt :
endif :
if Z = 155 then :
   msg 'oa-Esc pressed. Key' :
   A = key :
   msg '' :
   rpt :
endif :
if Z = 209 then :
   msg 'I asked for Y/N and you hit oa-Q?!? Key.' :
   A = key :
   msg '' :
   rpt :
endif :
if Z = 147 then :
   msg 'I asked for Y/N and you hit oa-Ctrl-S?!? Key.' :
   A = key :
   msg '' :
   rpt :
endif :
if Z = 211 then :
   msg 'I asked for Y/N and you hit oa-S?!? Key.' :
   A = key :
   msg '' :
   rpt :
endif :

msg 'Unknown Z ' + str$ Z  + ' Key ' :
A = key :
msg '' :
rpt :
>!
```

< $1 = .AwPath >
Domain: All Modules
This command returns the path where AppleWorks was launched from. For AW 5.0 it returned a slash "/" as the trailing character. This was removed for AW 5.1 i.e., 5.0 = /H1/AW/, 5.1 = /H1/AW

```
<sa-A>:<all :
$1 = .awpath :
msg $1 :
```

```
#Key2Stop :
>!
```

< .Box X,Y,W,L,T >
Domain: All Modules
Draws a box at the location specified. User can then use .WriteStr to fill the box.

X = The left column number (0-77) or 255 if box is to be centered
Y = The upper line number (0-20)
W = The width in columns (1-78)
L = length in lines (1-22)
T = Type of box lines: 0 = plain text, 1 = MouseText.

A bit of realities here:

A box that begins in column 77 isn't going to contain all that much information i.e., none.

If Y = 20 then L cannot exceed 2 and T must = 0.
To draw the same minuscule box with T = 1 then Y must = 19.

An L value that causes the box to exceed column 78 will result in a null right side of the box.

```
<sa-A>:<all:
savescr    // save the screen
.Cls 0 :   //Clear the screen

.Box 255,7,40,5,0:
.WriteStr 255,10,"This is a centered plain text box" :
.SpaceBar :

.Box 255,7,40,5,1 :
.WriteStr 255,10,"This is a centered MouseText box" :
.SpaceBar :

.Box 0,0,40,5,1 :
.WriteStr 6,3,"This box is in the upper left" :
.SpaceBar :

.Box 38,15,40,5,1 :
.WriteStr 45,18,"This box is in lower right" :
.SpaceBar :

.Box 0,19,30,3,0 :
.WriteStr 5,20,"This is a tiny box" :
.SpaceBar :

.Box 0,19,30,3,1 :
.WriteStr 5,21,"MT Loses bottom line" :
.SpaceBar :
restscr        // restore the original screen
>!
```

< .Beep Duration,Pitch >
Domain: All Modules
Duration is a number (0-255) that defines the duration of the beep. The smaller the number the shorter the duration.

Pitch is a number (0-255) that defines the pitch of the beep. The smaller the number the higher the pitch with 50 being a realistic lowest number.

```
<sa-A>:<all :
msg '0-255 for Duration. Esc to quit' :
$1 = getstr 3 :
if $1 = "" then :
   msg 'Esc Hit. Key' :
   #Key2Stop :
endif :
D = val $1 :
msg '0-255 for Pitch. Esc to quit' :
$1 = getstr 3 :
if $1 = "" then :
   msg 'Esc Hit. Key' :
   #Key2Stop :
endif :
P = val $1 :
.Beep D,P :
rpt :      //No Begin so go to top of macro
>!
```

< $1 = .Caps $2 >
Domain: All Modules
Capitalizes the first character of each word in $2 and places the result in $1.

```
<sa-A>:<all :
$1 = "this is a string with no leading caps" :
$2 = .Caps $1 :
oa-1 :     //Get cursor out of the way
.Cls 1 :
.WriteStr 0,10,$1 :
.WriteStr 0,11,$2 :
msg 'Key' :
K = key :
oa-9 :     //Get cursor to the bottom
| : esc :
msg '' :
>!
```

< $1 = .Case ″Case String″,″Work String″ >
Domain: All Modules
This command lets you change the case of a Work String based on the case of a Case String. The Case String consists of up to 26 characters. If the Nth character of the Case String is upper case then the associated characters in the work string will be converted to upper case i.e., AAAAAAAAAAAAAAAAAAAAAAAAAA is the same as ABCDEFGHIJKLMNOPQRSTUVWXYZ and aaaaaaaaaaaaaaaaaaaaaaaaaa is the same as abcdefghijklmnopqrstuvwxyz.

If the case string defines N characters, where N is less than 26 (i.e., 16), then those characters beyond Nth character (Q-Z), are not affected.

//Make every work string occurrence of a, b, and c upper case
< sa-A > : < all :
$2 = ″AAA″ : //case string
$1 = .case $2, ″abcdefghiJKLMNOPQRS″ : //work string

msg $1 :   //Show result
#Key2Stop :
>!

< .CacheList FirstStr >
Domain: All Modules
This command will return the names of the currently cached macro sets in the strings starting with FirstStr. It sets Z equal to the number of currently cached files. If you set FirstStr to 0, it will just set Z and not define any strings.

```
<sa-A>:<all :
//Store cached macro set names starting with $1
.CacheList 1 :
msg str$ Z + " Macro sets in the cache. Use Debugger to see names" :
#Key2Stop :
>!
```

< $1 = .Choose $2,N >
Domain: All Modules
Yields the Nth item from a comma separated list.

$1 = The result item

$2 = The source list. Does not have to be a string variable. It can be a "list,of,items" In either case, the list is limited to 80 characters.

N = The offset (index) into the list to choose the item. N can range from 1 through the number of items in the list.

Z = 0 = N is out of range
Z = 1 = N is in range.

See the .SubChar command for another keen example of .Choose usage.

```
<sa-A>:<all :
// Here we are setting N to 0 through 5 even though the only
// "legal" values are 1 through 3. Note that Z will tell you
// when the index is (1) or is not (0) in range
// You might think that a check of $1 for being null would
// suffice. However, note the second item in the list is null and
// yet N is still in range.

$2 = "Index 1,,Index 3" :  //Three items in list, 2nd null
.Cls 1 :
.WriteStr 0,10,"The input string: " + $2 :
for N = 0 to 5 :   //Check for 5 items in list
   $1 = .Choose $2,N :
   msg "N = " + str$ N + "  Z = " + str$ Z + "  $1 = " + $1 :
   .SpaceBar :
next N :
msg '' :   //Erase message line
| : esc :
>!
```

A really neat example from /extras disk.

```
<sa-A>:<all :
$1 = "Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday" :
X = .weekday 0,0,0 //Today gets filled in for 0 parms
$2 = "Today is " + .Choose $1,X :  //X = 1 through 7. Sunday is 1
msg $2 + "  " + %J% + " Key " + %K% :
#Key2Stop :
>!
```

< .DeskCount X >
Domain: All Modules
The .DeskCount command sets Z equal to the number of current desktop files (active desktop only) of the type specified. Use 1 for DB, 2 for WP, 4 for SS, or any combination of the three.

Geek Alert!

```
S  D  W
S  B  P
--------
4  2  1  Binary Values
1  1  1  Binary Bits
```

Inspection of the two lines above shows that the three binary bits, if set to 1, have a combined total of 7. In the following example .DeskCount 3 is used which is the total for the first two binary bits. The SS and WP would use a value of 5 (4 + 1).

```
<sa-A>:<all :
.DeskCount 3 :
msg "There are " + str$ Z + " DB and WP files" :
#Key2Stop :
>!
```

< .Dropdir >
Domain: All Modules
This command drops the last subdirectory from the current pathname.

```
<sa-A>:<all :
$1 = .GetFPath :   //Get this file's path
.Cls 1 :
oa-Q :
msg 'Watch the upper left of your screen. Follow space instructions at
bottom' :
.setdisk "/four/three/two/one" :
.spacebar :
.dropdir :
.spacebar :
.dropdir :
.spacebar :
.dropdir :
msg 'Note that you cannot drop the last level of a pathname' :
.spacebar :
.dropdir :
msg 'See, four is still there even though we tried to drop it' :
.spacebar :
.SetDisk $1 :
rtn :
>!
```

< $1 = .Embedded >
Domain: All Modules
The < .embedded > command returns the bottom line string from the Word Processor (Line 15, Column 12 or **Boldface Begin**, Line 99 Column 57, etc.) even if the screen display is off.

In the following example, if no printer options are found (by .findpo), the cursor will be on the last line in the file.

```
<sa-A>:<all :
// display 0 : //See Note:
.findpo :
$1 = .embedded :
if Z = 0 then :
   $1 = "Did not find a printer option. Go to Chapter 4 and try
again":
else :
   $1 = "We found " + $1 :
endif :
msg $1 + "  " + %J% + "Key" :
#Key2Stop :
>!

// If you uncomment the <display 0> the macro will still find the
// printr option (if one exists), however, the screen will not
// be updated. Note that your macro could look at $1 and base its
// actions on that

<sa-A>:<all :
display 0 :
.findpo :
$1 = .embedded :
if Z = 0 then :
   $1 = "Did not find a printer option. Go to Chapter 4 and try
again":
else :
   $1 = "We found " + $1 + " Now hand delete yadda, yadda, yadda" :
endif :
print " yadda, yadda, yadda " :
display 1 :
msg $1 + "  " + %J% + "Key" :
#Key2Stop :
>!
```

< X = .Eof >
Domain: All Modules
Returns the end of file value: last line number (AWP), last record (DB), or last row which contains data (ASP). Use it to check if you have reached the end.

Example shows a way to Get rid of hyphenated words when at the end of a line. The command progresses downward from cursor.

```
<sa-A>:<awp :
Begin :
   E = .eof :  //Last line in file at this time
   oa-. : posn A,B :   //End of line
   ifnot E > B endmacro : endif :  //End of file
   if A > 2 left : endif : //Any characters
   A = peek #curschar :    //Get last char
   ifnot A = 173 down :    //If not dash (hyphen)
```

```
       Rpt :
   endif :
   down : up : //Show next line
   msg 'Kill or Concatenate Y/N/C/Esc' :
   C = key : msg '' :
   if C > 96 and C < 123 then C = C - 32 : endif : //Force upper case
  if C = 27 stop : endif : //ESC key
   if C = 67 then right : oa-DEL : endif : //C so keep hyphen
   if C = 89 then oa-DEL : oa-DEL : up : endif : //Y so nuke hyphen
   down :  //Next line
Rpt :
>!
```

< $1 = .FDate Format >
Domain: All Modules
This command returns today's date in a variety of possible formats as shown below:

```
Format =  1 - 09/21/92
Format =  2 - 09.21.92
Format =  3 - 09-21-92
Format =  4 - 21/09/92
Format =  5 - 21.09.92
Format =  6 - 21-09-92
Format =  7 - 92/09/21
Format =  8 - 92-09-21
Format =  9 - 92.09.21
Format = 10 - 920921
Format = 11 - Sep 21 92
Format = 12 - 21 Sep 92
Format = 13 - September 21, 1992
Format = 14 - 21 September 1992
```

```
// This macro shows all the format options for use by .fdate & .fdate2

<sa-A>:<all :
   SaveScr :   //Save screen
   .Cls 1 :    //Clear the screen
   .writeStr 15,3,".FDate                        .FDate2"
   for A = 1 to 14 //Loop thru
       Y  = 4 + A :
       $0 = str$ A + " - "+ .FDate A :
       if A < 10 then $0 = " " + $0: endif :
       .writeStr 14,Y,$0:
       $0 = str$ A + " - "+ .FDate2 11,18,1776,A :
       if A < 10 then $0 = " " + $0 : endif :
       .writeStr 44,Y,$0 :
   next A:
   .SpaceBar:
   RestScr:    //Restore screen
>!
```

< $1 = .FDate2 Month,Day,Year,Format >
Domain: All Modules
This command works like .fdate except that you must specify a date. Note that Year needs to be 1992, not 92. (See .FDate for examples of all 14 formats.)

This command returns today's date in a variety of possible formats as shown below:

```
Format =  1 - 09/21/92
Format =  2 - 09.21.92
Format =  3 - 09-21-92
Format =  4 - 21/09/92
Format =  5 - 21.09.92
Format =  6 - 21-09-92
Format =  7 - 92/09/21
Format =  8 - 92-09-21
Format =  9 - 92.09.21
Format = 10 - 920921
Format = 11 - Sep 21 92
Format = 12 - 21 Sep 92
Format = 13 - September 21, 1992
Format = 14 - 21 September 1992
```

```
<sa-A>:<all :
$1 = .FDate2 8,26,1960,5 :
msg "Randy was born " + $1 :
#Key2Stop :
>!
```

< .FindPO >
Domain: Word Processor
Moves the cursor to the next character based printer option e.g., Boldface Begin/End (┌), Superscript Begin/End (╥), Subscript Begin/End (╨), Underline Begin/End (_), Print page number (▌), Print Date or Time (├), Enter from Keyboard (─), Mail Merge (█), Special Code (^), Apple (^) (Note 2), Sticky Space ( ) (Note 3). (The MouseText characters will print differently on all printers except for IW II.)

< .findpo > sets Z to 0 if it fails (none found) or 1 if successful.

See < .embedded > command for an example using < .findpo >

Note 1: The screen symbol for each of the above printer options is shown in the parentheses accompanying each character's name.

Note 2: The Apple is entered from the keyboard by typing Control-A. It looks like a carat, but is called Apple.

Note 3: The Sticky Space is entered from the keyboard by typing OA-Spacebar.

Note 4: The OA-F (built-in AppleWorks command), has provisions for finding any of the above plus all the other printer options that .findpo ignores.

< $1 = .GetFpath >
Domain: All Modules
Returns the pathname where the current file was loaded from (e.g., original directory).

< X = .ID >
Domain: Within a TimeOut application
Replaces the < id# > token. Returns the current TimeOut applications's ID.

The good news is that this command is handy for including variables in a message.

The bad news is, many times this particular example doesn't work because the TimeOut application doesn't allow variable I to be set at the crucial time. (My guess.)

```
<sa-A>:<all :
oa-esc :   //Bring up TO menu
>8<rtn :   //8th = DB Replace in my TimeOut Menu
I = .id :  //Ultra Compiler seems to work too
$1 = "TimeOut ID# " + str$ I :
esc :      //Get away from TO menu
msg $1 :
>! Show id number
```

< .Line Horiz,Vert,Length,Char >
Domain: All Modules
The .line command draws a horizontal line at the specified coordinates, using the specified character for the number of characters specified by Length. If you set the high bit as in the example, MouseText will be used. You may use 255 for the Horizontal value to center the line.

Type Ctrl-T and one of the characters below to get associated MouseText
```
@ 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 [ \ ] ^
└ ┴ ┬ ├ ┼ ┼ ╞ ╟ ╙ ╓ ╨ ╥ ╟ ═ ╫ ╧ ╨ ╤ ╥ ╙ ╘ ╒ ╓ ╫ ╪ ┘ ┌ █ ▄ ▌ ▌ ▀
```

```
<sa-A>:<all :
.cls 1 :
msgxy 255,20 :
msg 'Watch the top line of equal signs and the bottom of dashes' :
.SpaceBar :
.line 0,1,79,#"S"  //top line
.line 0,22,79,#"S" //bottom line
.SpaceBar :
msgxy 0,128 :
oa-Q : rtn :
>!
```

< $1 = .Lower STRING >
Domain: All Modules
Makes each character in STRING lower case and puts the result in $1.

I couldn't come up with an example as good as Will's so I "borrowed" his. The only addition is that I've added some code to make sure the cursor is the same (insert/overstrike), when the macro exits as when entered.

```
<sa-A>:<all :
$4 = "COPYRIGHT BY WILL NELKEN" :
$1 = .Lower mid $4,11,2 :  //Lower the 2nd word
oa-9 :     //Bottom of Try file
C = peek $10F1 :   //0 = insert, 1 = overstrike
insert :   //UM command assures insert
print $4 : //Insert the all caps line
oa-E :     //Toggle to overstrike cursor
(oa-left) 3 :  //Back up three words
print $1 : //Overstrike BY with by
(oa-right) 3 : //Cursor back to end of line
poke $10F1,C : //Put cursor to input state
>!
```

< .Msay String >
Domain: All Modules

<.msay> is like <.say>, except that it doesn't save or restore the bottom line, making it useful inside of loops since the screen would flicker if .say was used, since it restores the bottom line whenever it exits. Use .say for "one-shot" messages, such as the end of a macro.

```
//Copy this to the Try file, compile and run it.
//Next, comment out the <.MSay line and uncomment the <.say> line
// Compile and run again.

<sa-A>:<all :
for I = 1 to 6 //Loop six times
   zoom :
   bell :
   .msay "Notice how .msay message is solid as a rock.  Any key" :
// .say "Notice how .say message flashes.  Any key" :
   wait 500 :
 next I :
| : esc :  //Clean up screen from .msay
>!
```

<.NewFile "FileName",Type>
Domain: All Modules
This command creates a new file, giving it the name "FileName" as specified. The type indicates which type of file to create:

type = 1 : WP, 2 : DB, 3 : SS

Z returns the file number of the newly created file. If Z = 0, then the file could not be created because you've already have 12 files on a desktop. You can move to another of the three desktops that has less than 12 files and create it there.

If you try and create an illegal filename (illegal characters or null), AppleWorks will prompt you to provide a filename from the keyboard. Unfortunately, this confuses .NewFile and the file number returned in Z is one less than it should be. So, keep those filenames legal and non-null.

There is no warning/prompt if you create a filename of a file that is already on the desktop.

Why is this command here (Most people use a macro anyway)? Well, it handles the actions with or without display and Expert Mode leaving AppleWorks in the same state as you left it.

With a DB file, you still need to specify category names. Your macro will have to take care of that or you can go do it manually after the DB file is created.

```
<sa-A>:<all :
$1 = "New.WP.file" : X = 1 :   // New WP file
ba-A :

$1 = "New.DB.file" : X = 2 :   // New DB file
ba-A :

$1 = "New.SS.file" : X = 3 :   // New SS file
ba-A :
msg '' :   //Erase message
>!

<ba-A>:<asr :
.NewFile $1,X :
if Z = 0 msg 'Desktop full. Cannot create. Key' :
   #Key2Stop :
endif :
$0 =  "File number for  '" + $1 + "'  is " + str$ Z  + ".    Hit a
key" :
msg  %J% + $0 +%K% :
X = key :
>!
```

< .Online STRING >
Domain: All Modules
STRING may be a filename (Randy), partial pathname (Ultra/Randy), or a full pathname (/HD1/Ultra/Randy).

If STRING is a partial pathname or a filename the current disk is checked for the file specified. if STRIng is a full pathname then that disk is checked for the filename.

```
<sa-A>:<all :
$1 = "/H2/UM.MANUAL/MANUAL/CH01" : //Change to something on your HD
.OnLine $1 :   //Go look
if Z > 0 then :
   $0 = "File " + $1 + " is available" :
else :
   $0 = "Cannot find " + $1 :
endif :
msg $0 + "  " + %J% + " Key " :
A = key :
msg '' :
>!
```

< $1 = .Peekstr Address >
Domain: All Modules
Returns the string stored in memory starting at Address.
Address must point to a Class 0 string. See "Dot Command Reference," at the beginning of this chapter for the *Class 0 strings* story.

Delete the current file from Disk. Note that this hummer isn't my standard < sa-A > . This sucker is dangerous so I made it almost fumble finger proof. It deletes the file from the disk! (Oh, by the way, there is a .PeekStr on the second line of the macro ;-)

The < oa-F > on the fifth line is NOT doing a find. < oa-F > from the oa-Q menu takes you to the "File Activities" menu. Try it by hand, you'll like it!

```
<sa-Ctrl-D>:<all :
$1 = .peekstr $0C56 :  //Get this file's name
$6 = .getfpath :   //Get the path this file loaded from
oa-Q :     //Get main menu
oa-F>6<oa-rtn : up : rtn : //Select a ProDOS path
oa-Y :     //Wipe out default
print $6 : rtn :   //The file's path
$0 = $1 :
Z = 200 :  //exact match required
find : //Find it
if Z = 0 then :
   $2 = $1 + " File not found. Hit a key" :
   msg %J% + $2 :  //Give message
   #Key2Stop :
endif :
```

```
oa-rtn :   //Remove it. No questions with oa-rtn
$0 = $1 :  //File user started from
Z = 200 :  //Exact match
oa-Q :
find :
rtn :
>!
```

<.PeekVar Address,Start,Count,Size>
Domain: All Modules
The <.PeekVar> command is used to read Count consecutive bytes at Address starting with variable Start. Size (0 or 1), determines if single bytes (peek) are read, or two-byte words (peekword) are used. Unlike loops which use arrays, such as C(2), C(3), C(4) for three bytes, <.peekvar> would use C(2), D(2), E(2). This allows up to 520 bytes to be read in one pass - 260 words from A(0) to Z(9). (Each variable can hold two bytes.)

1. Move this text to the "Try" file and compile.

2. Press sa-A to see a word printed on the screen.

3. Are you as excited as I am?

```
<sa-A>:<all :
.peekvar $2234,C(2),5,1    //Read five words at $2234
oa-9 :
print chr$ C(2) :
print chr$ D(2) :
print chr$ E(2) :
print chr$ F(2) :
print chr$ G(2) :
>!
```

<.PokeStr STRING, Address>
Domain: All Modules
Pokes STRING into memory as a class 0 string starting at Address. See "Dot Command Reference," at the beginning of this chapter for the Class 0 string story. This command is very **dangerous** and should not be used without through understanding of what is being poked and where.

In the following example "Mr. I count real good" put an E on the end of the string, making it a 16 character filename. One more than is legal for ProDOS. The bad news is that when I attempted to save the file "Try," AppleWorks locked up because that extra bit had clobbered the pointer to the desktop file's original path.

The worse news is that it changed the name of Chapter 1 (a file that was not on the desktop), made it unreadable, and somehow lost 19 bytes of HD memory.

The good news is, ProSel 16 soon set the world right and my daily backups paid off big time cause I have not changed Chapter 1 in some days.

So, the following now works fine and the warning above that this is a very dangerous command has hit home with me.

```
<sa-A>:<all :
$1 = "A0123456789ABCD" :   //15 char filename
.PokeStr $1,$0C56 :    //Change this file's name
| : esc :  //Let the flickers begin
msg 'Note on top line that filename has changed.  Key' :
```

```
A = key :
.PokeStr "Try",$0C56 : //Put it back daddy
| : esc :  //Let the flickers begin
msg 'Note that filename is now back to the original.  Key' :
A = key :
msg '' :   //Erase message line
>!
```

<.PokeVar Address,Start,Count,Size>
Domain: All Modules
The short story: <.PokeVar> works just like <.PeekVar>, except that it stores the variable values in memory instead of reading memory into the variables.

The <.PokeVar> command is used to write Count consecutive bytes at Address starting with variable Start. Size (0 or 1), determines if single bytes (peek) are written, or two-byte words (peekword) are used. Unlike loops which use arrays, such as C(2), C(3), C(4) for three bytes, <.peekvar> would use C(2), D(2), E(2). This allows up to 520 bytes to be read in one pass - 260 words from A(0) to Z(9). (Each variable can hold two bytes.)

This example writes H E L L O ! on the top line of the screen in between the file name and the REVIEW/ADD/CHANGE header. The reason the message shows up with spaces between the letters is an artifact of the way Apple text memory is displayed. To put the characters together we would have to write one byte to bank 0 and the next to bank 1, etc. I know of no way to do this with UM and its variant of poke.

Note that the Apple puts the low byte in memory first so HE is put into C(2) as EH. In E(2) we see !O. Trust me, the LL in D(2) is reversed also ;-)

```
<sa-A>:<all :
//     E H
C(2) = $C5C8 :
//     L L
D(2) = $CCCC :
//     ! O
E(2) = $A1CF :
.pokevar $407,C(2),3,2 :
.SpaceBar :
| : esc :
>!
```

<X = .PeekWordZP Address>
Domain: All Modules
This command is used to peek a pair of bytes in auxiliary zero page memory. If you don't know what that means, ignore this command. To peek a pair of main zero page bytes, just use plain old <peekword>.

Probably an unfortunate choice of names here since it reads from aux memory while <.PokeWordZP> writes to main memory.

```
<sa-A>:<all :
X = .PeekWordZP $22 :
msg X :
>!
```

<.PokeWordZP Address,Value>
Domain: All Modules

< .PokeWordZP > works just like < .PokeZP > except that it stores two bytes in memory instead of one. The values are stored in main memory.

// No meaningful example yet; maybe some day

< .PokeZP Address, Value >
Domain: All Modules
Pokes Value into the main memory zero page at Address. Strictly for the "techie" types. Very **dangerous** if you don't understand what you are doing.

// No meaningful example yet; maybe some day

< .Pop X >
Domain: All Modules
Pops return address from the subroutine stack. In English, this means that when a macro calls another macro (not a goto), the second macro can do a < .pop 1 > to avoid returning to the caller.

X is the number of levels to pop off of the stack:
```
    X =  0, pop nothing. Return as usual.
  X =  1, through 15, pop that many return addresses off of the stack.
  X = 20, Clear the entire stack.
```

Macro subroutines can stack up to sixteen return addresses on the stack. Each call in the following adds another return address to the top of the stack. (Some view it as adding to the bottom of the stack. It all has to do with whether you are a big-endian or a little-endian. Pick the view that suits you.)

```
sa-A  = = >  sa-B  = = >  sa-C  = = >  sa-D
        sa-A       sa-B       sa-C
                sa-A       sa-B
                        sa-A
```

When a macro has finished its function and control returns to the calling macro by popping the top address off of the stack and going to that address. (One way to view this process is the plate holders at your local buffet where each time a plate is taken, a spring pushes another up for the next person until such time all plates have been taken.)

The example shows that sa-A calls sa-B which tells sa-C how many levels to pop. When sa-C pops 1 level the return isn't to sa-B, but to sa-A.

```
<sa-A>:<all :
X = 65535 :    //Number of levels to pop -1
sa-B :
msg 'Back to sa-A from sa-C, jumping over sa-B' :
.SpaceBar :
msg '' :
>!

<sa-B>:<all :
sa-C :
msg 'Return from sa-C to sa-B' :
.Spacebar :
goto sa-B :    //Loop forever
>!

<sa-C>:<all :
X = X + 1 :    //65535 + 1 = 0 in Ultra Land
```

```
msg 'sa-C is about to pop ' + str$ X + ' level(s) off of stack' :
.SpaceBar :
msg '' :
.pop X :
>!
```

< X = .Rightmost >
Domain: Spread Sheet
1. This command returns the rightmost spreadsheet column containing data i.e. 3 means column "C" is the rightmost column containing data while 13 means that column "M" is the rightmost column containing data. One exception, in a completely empty SS it returns 1 rather than 0.

2. For the AWP it returns the last allowed character position on the current word processor line + 1. In other words, it returns the last allowed cursor position. Type a single character in that column and you will find that you are moved to column 1 in a new line.

3. For the DB it returns the last column on the screen. For the DB, a column is a "slot" which is independent of the actual category contained in that slot. That is, you can rearrange the layout so the category order is scrambled, but slots always start at one and count up from left to right.

Compile the example in the Try file. I find it works as advertised in AWP and ASP. It just might work as intended in ADB, however, I fail to see what it is trying to prove. I *always* get a value of 115 from any DB file I test. YMMV. Let me know if it does.

```
< sa-A > : < all :
X = .Rightmost :
msg X :
.Spacebar :
msg '' :
> !
```

< X = .Search "IsHere",Start,End >
Domain: All Modules
This command will search up to 99 string variables ($1-$99) for "IsHere". The search is a "contains" search, not an equal search. This is a caseless search. Start and End specify the range of string variables to search through. If a string is found, its number is returned in X. X = 0 means failure. You cannot search $0 using this command. If you wish to search subsequent strings you must start after the latest match.

```
<sa-A>:<all :
SaveScr :
.Cls 1 :
$75 = "I'm told that case does not matter" :
$2 = "CaSe DoEs NoT MatTeR"
X = .Search $2,70,80 :
Ifnot X = 0 then :
   .WriteStr 0,10,"Found:  " +$(X) :
   .WriteStr 0,11,"Search: " + $2 :
   msg 'Match found in string ' + str$ X :
else  :
   msg $2 + "  " + %J% +"NOT FOUND" + %K% :
endif :
.SpaceBar :
RestScr :
| : esc :
>!
```

< .SetCol Column,Width >
Domain: Spread Sheet
The < .setcol > command changes the width of the specified spreadsheet column. No flicker, no pain. Note that a column cannot be wider than 70 columns. Further note that if you express a width greater than 70 the .GetValue dot command will not advance until you either give it a value less than 71 or hit ESC.

An interesting "quirk" of < .SetCol > is that it does not mark the SS file as changed when you change one or more column widths with the following macro.

```
<sa-A>:<asp :
C(1) = peek $10F1 :    //Cursor state
poke $10F1,1 : //Force overstrike cursor
$0 = .CellID : //$0 = Col and row info i.e., C14
$0 = .SubChar $0,#'0',#'9',#'*' : //Numbers to asterisks. $0 = C**
$0 = .ZapChar $0,#'*' :    //Nuke asterisks. $0 = C
$1 = .GetString 'Adjust which column? ',$0,2 :
if Z = 0 or Z > 100 then : //Client wants out
   poke $10F1,C(1) :   //Put back cursor state
   endmacro :
endif :
C = .Column $1 :
W = .GetValue "How wide? ",1,70,9 :
poke $10F1,C(1) :  //Put back cursor state
if Z = 0 or Z > 100 then : //Client wants to quit
   endmacro :
endif :
.SetCol C,W :
>!
```

< .SetDisk STRING >
Domain: All Modules
Sets the AppleWorks data directory to the path specified by STRING i.e., after executing the following macro the default location to load files from will be "/DATA2/FILES/AWP"

```
< sa-A > : < all :
.SetDisk "/DATA2/FILES/AWP" :
> !
```

This shows that you can use a label, string variable, and a quoted string either by themselves or in any combination.

```
<sa-A>:<all :
.SetDisk #BaseDir + $9  + "/Templates" :
>!
```

< .SetDisk > replaces the < & path > command.

< .SetFpath STRING >
Domain: All Modules
Sets the current file's directory path. No checking is done with respects to the validity of the path. It is simply set to the value of STRING. If the path is invalid (i.e., 1234), you will get an error only when you try to save the file i.e., Getting errors trying to save at 1234.

As shipped, AppleWorks is configured to save a file to the current path when OA-S is typed and to the file's original path when OA-CONTROL-S is typed. In this case location $10B1 will contain $D3 (oa-S, see KeyChart file).

Generally folks reverse this setting using Randy's Free Patcher (RFP) because it is more natural that way. In this case $10B1 will contain $93 (oa-Ctrl-S, see KeyChart file).

As you might guess, location $10B1 reflects which **internal** key to use when oa-S is pressed (and by implication which internal key to use when oa-Ctrl-S is pressed).

If your macro is going to be used by a number of people then consider the following code from Randy Brandt to assure that your macro will save to the path just set by < .setfpath > . (Another approach is to inform the user up-front that your macros always assume oa-S saves to a file's original path and if their system doesn't support this, your macro will not run correctly.)

```
// Memory location $10B1 contains the key stroke that will
// cause the current file to be saved to its original path.
// AppleWorks ships with $10B1 = $93 (oa-Ctrl-S)
// Randy's Free patcher changes it to $D3 (oa-S)
// Randy said, "You can't poke $10B1, because it's just a
// reflection of what is defined in the save segment,"

// However, you can execute it to cause the file to be saved
// to its original path, irrespective of $10B1 setting

// Compile this, move to some other folder ie., /DATA and
// load a file so that it becomes the current folder.
// Next execute this macro and see that Try gets saved at
// /DATA/Try and /DATA/Txt/Try. Delete these two files
// and then oa-S Try and see that the current path is Try's
// original.

<sa-A>:<all :
$1 = .GetFpath :   //Get the current path for Try
.SetFpath "/DATA/TXT" :    //Change to fit your system
A = peek $10B1 :   //Get the current Smart Save option

//$D3 + $93 = decimal 358. Subtracting current smart save key
//guarantees you the "save current path command in C
C = 358 - A :
print chr$ C :

//This guarantees that you save to the original path command in A
print chr$ A : //Original path or .SetFpath in this case
.SetFpath $1 : //Back to Try's real home
>!
```

< .Sort Start,End,Direction >
Domain: All Modules
The sort command sorts a range of strings from Start to End in ascending or descending order, as specified by Direction. See Chapter 2 "Testing String Relationships," for examples how NOT to test the relationship between two strings.

```
<sa-A>:<all :
// handy trick to build a short string series
$50 = "Randy,Joanna,Heather,Erika,Michael" :
$51 = "az,aq,aa,ay,ax" :
$52 = "10,1,15,9,2" :  //Note that this one doesn't sort "right"
$53 = "10,01,15,09,02" :   //This one does
$54 = "aa,AA,bb,BB,a" :
$55 = "10,1,fred,Fred,15"
```

```
for I = 50 to 55 :

   //Move each of the five items from a $(I) to strings $1 through $5
   X = 1 : ($(X) = .Choose $(I),X: X = X + 1) 5 :

   //Its chronological because Randy set $50 that way. Not because of
   //any macro magic.
   $8 = "Chronological order: " :
   ba-A :  //Display $1 through $5

   .sort 1,5,1 :   //Sort $1 - $5 in ascending order
   $8 = "Ascending sort order: " :
   ba-A :      //Display $1 through $5

   .sort 1,5,0 :   //Sort $1 - $5 in descending order
   $8 = "Descending sort order: " :
   ba-A :      //Display $1 through $5
next I :
msg 'How much fun can one person stand ;-?   Key' :
#Key2Stop :
>!

<ba-A>:<all :
X = 1 :        //Build display string in $8
($8 = $8 + $(X) + " ":
X = X + 1) 5 :
msg $8 :
.spacebar :
msg '' :
>!
```

< .Speed 1000 >
Domain: All Modules
This command, written at the request of J.S. Rowe of England, allows you to slow down Ultra 4.x. This can be handy for demos or for debugging. It's actually a delay command, since higher numbers make Ultra run slower. Highest number allowed is 65535 which is about a 30 second delay between commands on a 8Mhz IIgs. You may press a key during long delays to hurry the macro along. Use 0 to return to full speed.

```
<sa-A>:<all :
for X = 1000 to 4000 step 1000 :
   .Speed X :
   oa-Q :
   $0 =  'Speed ' + str$ X :
   msg %J% + $0 + %K% :
   esc :
   msg %J% + $0 + %K% :
   up :
   msg %J% + $0 + %K% :
   up :
   rtn :
   msg %J% + $0 + %K% :
   esc :
   msg %J% + $0 + %K% :
   esc :
next X :
.Speed 0 : //Important to get that sucker back
```

```
>!
```

< $1 = .SubChar Text, First, Last, NewChar >
Domain: All Modules
The < .subchar > command replaces a range of characters (First through Last), with the specified new character. First must have a lower ASCII value than Last. The ASCII values are used, but the compiler can convert text characters for you so you don't have to know the number values.

The first example shows using text characters. The second shows a combination of character values and text.

```
<sa-A>:<all :
$1 = "text"    //Starting string
$2 = .subchar $1,#'x',#'x',#'s'    //Replace x with s
msg "Replace the 'x' in " + $1 + " with 's' to get " + $2 :
>!
```

Many thanks to Randy Brandt for writing the following macro. I had a version that worked, however, it wasn't nearly as elegant or fast.

More thanks to Will Nelken for writing "Ultra to the Max" Ultramacros manual. Without it I'd of never understood what the hey Randy did here ;-)

The line that begins "$6 = right $5..." is a real killer and you might stare at it for days before figuring it out. Instead, here are Randy's words on the subject:

"Steve Beville came up with the clever concept of using Ultra's math wrap characteristic. That is, adding so a value exceeds 65535 ($FFFF) wraps back to 0, so to shorten a string by 3, you can add the existing length to 65533 (65536 - 3 [which in hex is $10000 - 3]) and you'll end up 3 less than you started with. That trick saves a couple of statements, since otherwise you'd need: Z = len $1 : Z = Z - 3 : $1 = right $1,Z"

Belaboring the obvious, A = $FFFF + 1 = $10000. Since UM math can only deal with the four least significant hex digits of a number, the 1 in $10000 goes into the bit bucket and the number in variable A is $0000

```
<sa-A>:<awp :
$1 = "  PUT FileName   ;Comment"
$2 = .subchar $1,9,32,#'*' //Convert spaces and tabs to *'s
$3 = .zapchar $2,#'*'  //Kill 'em all to pack the string
$4 = .subchar $3,#';',#';',#','    //Change the comment semi-colon
$5 = .choose $4,1  //Lose the comment
$6 = right $5,65533+len $5 //Lose the PUT

.cls 1 :
msgxy 0,10 :   //Should be .WriteStr. Here for example
msg 'Original: ' + %K% + $1 :
msgxy 0,11 :
msg 'SubChar:  ' + %K% + $2 :
msgxy 0,12 :
msg 'ZapChar:  ' + %K% + $3 :
msgxy 0,13 :
msg 'SubChar:  ' + %K% + $4 :
msgxy 0,14 :
msg 'Choose:   ' + %K% + $5 :
msgxy 0,15 :
msg 'Right:    ' + %K% + $6 :
```

```
.SpaceBar :
msgxy 0,128 :
| : esc :
>!
```

< X = .SubString $1,$2,Start >
Domain: All Modules
This command searches for "hat" in "That hat! Where is my Hat" starting at the string position specified by Start. Case does not matter. If found, the position found will be returned in the offset variable (X in this case). If not found the offset variable (still X), will be set to 0.

```
<sa-A>:<all :
F = 0 :        //Flag says no matches yet
S = 1 :        //Start point in string
$1 = "hat"
$2 = "That hat! Where is my hat?" :

Begin :
   X = .substring $1,$2,S :
   $0 = $1  + " starts " + str$ X + " chars into the string.  Key" :
   ifnot X = 0 then msg %J% + $0 + %K% :
  S = key :    //Eats key. A would have worked as well
   S = X + 1 : //Point 1 beyond last occurrence
   F = 1 : //Flag that at least one match
Rpt : endif :
if F = 0 msg %J% + $1 + ' Not found' :
else msg %J% + 'No more occurrences of ' + $1 :
endif :
#Key2Stop :
>!
```

< .TitleBox X,Y,W,L,T,$ >
Domain: All Modules
Draws a box onscreen with a title bar i.e., this is pretty much MouseText .Box with a title bar.

```
X = The left column number (0-77) or 255 if box is to be centered
Y = The upper line number (0-21)
W = The width in columns (1-78)
L = length in lines (1-23)
T = Type of title text or box type
   T = 1, Title is inverse
   T = 2, Title is normal
   T = 3, Box is shadow.
$ = A quoted string or string variable containing title bar text
```

A bit of realities here:
A box that begins in column 77 isn't going to contain all that much information i.e., none.

An L value that causes the box to exceed column 78 will result in a null right side of the box.

```
<sa-A>:<all:
savescr        // save the screen
.Cls 0 :   //Get rid of detractions. Not needed

.TitleBox 0,0,35,5,1,"This box has an inverse title" :
.WriteStr 5,4,"Stuff something in the box" :
.SpaceBar :
```

```
.TitleBox 40,7,35,5,2,"This box has a normal title" :
.WriteStr 45,11,"Stuff something in the box" :
.SpaceBar :

.TitleBox 0,15,35,5,3,'This is a shadow box' :
.WriteStr 5,19,"Stuff something in the box" :
.SpaceBar :

restscr        // restore the original screen
>!
```

< X = .TOinMem >
Domain: TimeOut Menu
This command is used at the TimeOut menu. It returns a non-zero value if the highlighted application has been loaded into memory or 0 if the application is still on disk.

```
<sa-A>:<all :
.cls 1 :   //Clear out background stuff
oa-esc :   //To the TO menu
msgxy 255,0 :  //Center the message
msg 'Arrow to move, ESC to quit' :
msgxy 0,128 :  //Put back the default message line
Begin :
   X = .TOinMem :  //X > 0 = highlighted app in memory
   msg ' ' + str$ X + ' ':
   A = key :
   if A = 27 then msg "" :
       esc :
       exit :
   else :
       print chr$ A :  //Give the key to AppleWorks
Rpt :      //Loop back
>!
```

< .UnCache $1 >
Domain: All Modules
.uncache "macroset" //uncache the named macro set
.uncache "*" //uncache all macro sets in the cache
.uncache "#" //uncache all sets EXCEPT UM4.0.SYSTEM

Ultra 4 caches macro sets whenever you use launch, call or link. This provides maximum speed for switching, but ties up memory which may be better used for other things. The .uncache command removes the named macro set from the cache, freeing up the memory. .uncache returns the number of files uncached in Z.

It appears that .UnCache "*" and .UnCache "#" options work identically in AW 5.1 because the name now is *SEG.UM* while the .UnCache dot command is looking for the name *UM4.0.SYSTEM.*

The macro examples here are somewhat more involved than others. Macro sa-A has a line that needs to be uncommented (.UnCache "*") and another (UnCache "#"), commented.

Macro sa-B shows how to delete a named macro set. For it to work you have to first compile the Try file, call sa-ctrl-T to save the Try macro set to disk. The first time you call sa-B no macro set will be removed. After that, each time you call sa-B, one macro set (Try) will be removed from the cache.

```
<sa-A>:<all :
//.UnCache "*" :   //UnCache all Macro Sets
```

```
.uncache "#" : //UnCache all except UM4.0.SYSTEM
sa-D :     //Go display
>!

<sa-B>:<all :
.UnCache "Try" //Named Macro Set
sa-D :     //Display
sa-C :     //Put Try back in cache
>!

<sa-C>:<asr :
$0 = "Ultra Options"
Z = 200 :  //Exact match
oa-ESC :   //TO menu
find :
rtn :rtn : up : rtn :  //Launch from disk
$0 = "Try" :
Z = 200 :  //Exact match again
find : rtn :
>!

<sa-D>:<asr :
msg ' ' + str$ Z + " sets were removed " :
.SpaceBar :
msg '' :
>!
```

< $1 = .Upper STRING >
Domain: All Modules
Defines $1 as the all upper case equivalent of STRING. STRING can be a string variable or a quoted string.

```
<sa-A>:<all : $1 = .Upper "LowER 4th" :
msg $1 + "    Key to continue" :
#Key2Stop :
>!
```

< .Vline Xpos, Ypos, Len, Char >
Domain: All Modules
Draws a vertical line onscreen with any character you specify.

Xpos is the column in which to draw (0-79 or 255 to center).
Ypos is befinning row for the line (0-23)
Len is the length of the line (1-24)
Char is the ASCII value of the character to use.

Do not try to specify a character under 32. Control characters will mess your world up no end. Note that you can specify the characters for T with either the numerical value or the #′X′. You cannot specify (AFAIN), high bit on except by numerical means.

```
<sa-A>:<all :
oa-Q :
T = 32 :   //Start with space
C = 0 :        //Start col 0
.Cls 0 :
Begin :
   .VLine C,0,23,T :
```

```
   T = T + 1 :
   if T = 255 then :
       .spacebar :
       | : esc :
       endmacro :
   endif :
   C = C + 2 :
   if C > 79 then :
       C = 0 :
       .Spacebar :
       .Cls 0 :
   endif :
Rpt :
>!
```

< X = .WeekDay Month,Day,Year >
Domain: All Modules
Returns a value (1 = Sun, 2 = Mon…7 = Sat) indicating the day of week for a specified date. Year has to be a full year (1992), not just the last two numbers.

If ALL of Month, Day, or Year are 0 then the current date will be used. (Some documentation says that if ANY of Month, Day, or Year is zero then the current date will be used. NOT TRUE. All must be zero for the current date.)

```
<sa-A>:<all :
A  = .WeekDay 11,18,1967 :
$1 = "Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday" :
$2 = .Choose $1,A :
A  = .WeekDay 0,0,0 :
$3 = .Choose $1,A :
msg "Mark was born on a " + $2 + "  " + %J% + " Key " :
A  = key :
msg "Today is " + $3 + "  " + %J% + " Key " :
#Key2Stop :
>!
```

< .ZapChar "Text", Char >
Domain: All Modules
The .zapchar command removes the specified character from a string. This is handy for stripping commas from numbers so the extended math commands work. See < .SubChar > for another example.

```
<sa-A>:<all :
$1 = ",,1,234,567.01,," :
$2 = .ZapChar $1,#',' :
.cls 1 :

.WriteStr 0,10,'Orig  ' + %K% + $1 :
.WriteStr 0,11,'Fixed ' + %K% + $2 :
.SpaceBar :
| : esc :  //Restore the screen
>!
```

< .ZoomIn > !
Domain: All Modules
Forces zoom in status. Shows all printer options, tabs, etc. Look around in Chapter 3 and 4 if you want an example.

Data Base Dot Commands

These dot commands are installed by the I.UM.AND.SS file. The commands in this section work only when a DB file is current.

< .GetNames Category, FirstStr, Number >
Domain: Data Base
Captures the names of a defined range of categories from a DB file:

Category is the number of the first category in the range
FirstStr is the number of the first string in the range i.e., 2 not $2 is specified.
Number is the number of strings in the range.

To read all category names into strings 1-30, use:

```
< sa-A > : < adb : .getnames 1,1,30 > !
```

< .getnames > stops at the last assigned name even if Number specifies a larger range.

NOTE: You can see what numbers are assigned to DB categories by typing OA-N, Return and then the up and down arrow keys to scroll through the list if there are more categories than will fit on the screen.

< .SetNames Cat, Record, FirstStr, Number >
Domain: Data Base
Sets the names of a defined range of categories for a DB file:

Cat is the number of the first category in the range
FirstStr is the number of the first string in the range i.e., 2 not $2 is specified.
Number is the number of strings in the range.

< .setnames > stops at the last assigned name even if Number specifies a larger range.

Consider the following extension of the < .getnames > example from above:

```
< sa-A > : < adb :
.getnames 1,1,30 ://Get the category names
$0 = "NewDB" :
Z = 0 :
oa-Q : find :       //Find the  new DB on the desktop
ifnot Z = 0 then :
rtn ://Enter the file
.setnames 1,1,30 :
> !
```

< .GetRec Category, Record, FirstStr, Number >
Domain: Data Base
Captures the contents of a range of categories within a specified record:

Category is the number of the first category in the range
Record is the number of the record to get the data from
FirstStr is the number of the string for the first string in the range
Number is the number of categories to capture.

The following example will store three categories in $9, $10, and $11, starting at the current category, in the current record.

```
// Compile the following in Try. Load TryDb
// Call sa-A from various categories, including the last

<sa-A>:<adb :
clear 255 :
posn C,R :
I = 4 :
.getrec C,R,I,15 :
.Cls 1 :
for I = I to 15 :
   .WriteStr 0,I," $" + str$ I + " = " + $(I) :
next I :
msg 'Key' :
A = key :
| : esc :
>!
```

< .SetRec Category, Record, FirstStr, Number >
Domain: Data Base
Sets the contents of a range of categories within a specified record:

Category is the number of the first category in the range
Record is the number of the record to get the data from
FirstStr is the number of the string for the first string in the range
Number is the number of categories to capture.

The following ″not ready for prime time″ macro copies the cursor category from the current file to the cursor category in ″NewDB.″

In the real world you would want to query the user for the destination file name, move down to the next record in both files, move to the same category name, regain the original file so another could be copied, etc., etc.

```
<sa-A>:<adb :
posn C,R : //Category & Record Number
F = 3 :        //Use string 3
N = 1 :        //One category
.getrec C,R,F,N    //Capture it
$0 = "NewDB" : //We magically know this :-)
sa-B :     //Go to NewDB
posn C,R : //Category and record might differ
.setrec C,R,F,N :  //Copy data over
| : esc :  //Redraw screen
>!

<sa-B>:<asr :
oa-Q : oa-1 : find :   //Find the new DB
if Z = 0 msg 'Cannot find ' + $0 + ' Key' :
A = key : msg '' : stop : endif :
rtn :      //Accept
>!
```

< $1 = .GetCat Category, Record >
Domain: Data Base
Returns the contents of the specified Category from the specified Record. See < .setcat > for example of use.

< .SetCat Category, Record, STRING >

Domain: Data Base
Sets the Category in Record with the contents of STRING. If the data is coming from a AWP via <cell> then see the warning with the <cell> command concerning tabs.

Slightly more realistic example in that we have a sub-macro to handle flipping between files on the desktop and we get back to the original file. One way to test this is to load TryDB, oa-N and change the name to "Fred", and once again reload TryDB. Set the cursor in TryDB to the destination category, go to Fred, type sa-A.

The details on obtaining the name of the destination file in a more realistic fashion is left as an exercise for the reader.

```
<sa-A>:<adb :
$2 = .peekstr $0C56 :  //This file's name
posn C,R : //Current Category & Record
$1 = .getcat C,R : //Got it.
$0 = "TryDB" : //We magically know this :-)
sa-B :     //Get to TryDB
posn C,R : //Category and record might differ
.setcat C,R,$1 :   //Copy data over
$0 = $2 :
sa-B : //Back to original file
>!

<sa-B>:<asr :
oa-Q : oa-1 : find :   //Find the new DB
if Z = 0 msg 'Cannot find ' + $0 + ' Key' :
A = key : msg '' : stop : endif :
rtn :      //Accept
>!
```

<$1 = .CatName CategoryNum>
Domain: Data Base
Yields the category name from the Category number.

Silly example, however, a realistic example would be quite a bit bigger.

```
<sa-A>:<adb :
msg 'Input category number' :
$0 = getstr 2 :    //Up to two characters
C = val $0 :
$1 = .catname C :  //Get its name
if $1 = "" $0 = "Invalid category number" :
   sa-B : stop :
endif :
$0 = "Category number "  + str$ C + " is named   " + $1 :
sa-B :
>!

<sa-B>:<asr : msg %J% + $0 + "       Hit a key" + %K% :
A = key : msg '' :
>!
```

Another example:

```
<sa-A>:<adb :
msg 'Input category name' :
```

```
$0 = getstr 30 :   //Up to thirty characters
C = .catnum $0 :
if C = 0 $0 = $0 + " is an invalid category name" :
   sa-B : stop :
endif :
$0 = "Category number "  + str$ C + " is named   " + $0 :
sa-B :
>!

<sa-B>:<asr : msg %J% + $0 + "   Hit a key" + %K% :
A = key : msg '' :
>!
```

Spreadsheet Dot Commands
These commands are installed by the I.UM.DB.AND.SS file.

< X = .Column STRING >
Domain: Spread Sheet
Sets X to the text column number given in STRING. For example,
X = .column "A" returns 1, X = .column "AA" returns 27, and
X = .column "DP" returns 120. DP is the last possible column in a spreadsheet.

Unfortunately, this is a really "quirky" dot command. X = .column "DQ" returns 121, X = .column "ZZ" returns 702 and X = .column "" returns 3234, etc., etc. As you can see, it is up to you to validate the output as is done below in the < .colwidth > example.

See < .colwidth > for an example

< X = .ColWidth STRING >
Domain: Spread Sheet
Set X to the width of the column specified by STRING.

In the face of the input errors as outlined for < .column >, < .colwidth > returns a width of 9 for columns that cannot exist so there doesn't seem to be a way to catch input errors other than checking them yourself.

```
<sa-A>:<asp :
msg 'Input column name' :
$0 = getstr 2 :    //Up to two characters
C = .column $0 :
W = .colwidth $0 :
if C > 120 or C = 0 then $0 = $0 + " is an invalid column name" :
   sa-B :
   stop :
endif :
$0 = "Column " +  $0 + " is column number " + str$ C + " & width is "
+ str$ W :
sa-B :
>!

<sa-B>:<asr :
msg %J% + $0 + "   Hit a key" + %K% :
#Key2Stop :
>!
```

< $1 = .GetCell Column, Row, Format >

Domain: Spread Sheet
Returns the contents of the specified Spreadsheet cell regardless of the cursor position. If Format is 0, < .getcell > returns the literal value regardless of the screen display, and if Format is non-zero, the exact screen format is used, including spaces, even if the screen is off. For example, if the cell holds 2.448 and if formatted for dollars, < .getcell > will return "$2.45" if Format is 1, or "2.448" if Format is 0.

```
<sa-A>:<asp :
$1 = .GetCell 2,2,1 :
$2 = .GetCell 2,12,0 :
SaveScr :
.Cls 1 :
.WriteStr 0,10,$1 :
.WriteStr 0,11,$2 :
msg 'Key' :
A = key :
RestScr :
>!
```

< .SetCell Column,Row,STRING >
Domain: Spread Sheet
Sets the cell at the intersection of the specified Column and Row to the contents of STRING. To enter a value, STRING is specified as a quoted string to < .setcell > . To enter a Label you have to use a string variable: < $1 = chr$ 34 + "Label text" > . You cannot create a "Repeated" label with .setcell. See < .cellid > for one method to create a Repeated label

```
<sa-A>:<asp :
C = .column "D" :  //C = D also works.
.SetCell C,1,"31.7"
$1 = chr$ 34 + date2 : //Enter as a label. 34 is a double quote
.SetCell C,2,$1 :
X = 34 :   //Another way
$1 = chr$ X + "Still a label" :
.SetCell C,3,$1 :
| : esc :  //Cause screen to be redrawn
>!
```

< $1 = .CellID >
Domain: Spread Sheet
Sets $1 to the text ID for the current Spreadsheet cell.

```
//This is a case where there are a number of ways to accomplish the
//job. Use C = peek $B0 to get the column number, and R = peekword
//$AE for the row.

//<posn C,R> does the same thing as the peeks and saves nine bytes.

//Since you are in the cell in question then the following .cellid,
//.column fills the bill since it allows cells to be of varying width.

<sa-A>:<asp :
$2 = .cellid : //$2 = Col and Row i.e, C 14

$0 = mid $2,2,1 :  //2nd character of cellid
A = val $0 :
B = 1 :        //Default to 1 Col character
if A = 0 then B = 2 : endif :
```

```
$2 = left $2,B :

//Here is an alternate (and better) way to do the previous 5 lines
//$2 = .SubChar $2,#'0',#'9',#'*' : //Numbers to asterisks. $2 = C**
//$2 = .ZapChar $2,#'*' :  //Nuke asterisks. $2 = C

C = .column $2 :
W = .colwidth C :  //Width of current column
$1 = chr$ 34 : //A double quote. $1 = "=" works too
($1 = $1 + "=") W :    //Width of column in ='s
print $1 : rtn :   //Create a Repeated label
| : esc :  //Cause screen to be redrawn
>!
```

< $1 = .LastCol >
Domain: Spread Sheet
Sets $1 to the text ID for the last Spreadsheet column containing data i.e., A, B, AA, etc. To get the numeric value for the last column: For AW 3 use X = peek $80FE. For AW 4 & 5 use X = peek $80FC to get the last column. In the example below sa-A in TrySS < .LastCol > = "T" and $80FC = 20, which is the same thing since "T" is the twentieth letter of the alphabet.

```
<sa-A>:<asp :
clear 255 :
$1 = .LastCol :
A = val $1 :
B = peek $80FC :
$2 = "$80FC = " + str$ B + "   key" :
msg '.LastCol = ' + $1 + %K% + "  " + %J% + $2 :
#Key2Stop :
>!
```

< $1 = .LastRow >
Domain: Spread Sheet
Sets $1 to the text ID for the last Spreadsheet row containing data i.e., 1, 2, 225, etc.

```
<sa-A>:<asp :
clear 255 :
$1 = .LastRow :
A = val $1 :
msg '.LastRow = ' + $1 + %K% + "  " + %J% + "Key" :
#Key2Stop :
>!
```

Menu Commands

< .AddMany X, Y, FirstStr, Count, Space >
Domain: All Modules
Adds **Count** numbered items to a vertical menu starting at column (0-79) **X**, line (0-23) **Y**. Strings starting with **FirstStr** are spaced according to **Space** (usually 1 or 2). None of the strings specified can be null or you will get a "Dot Command Error" when the macro is called.

This menu command does nothing useful, however, hang in there until we get to < .DoMenu > .

```
<sa-A>:<all :
clear 255 :    //Clear all variables
.Cls 1     //Clear the screen. See <.Cls> for details
$1 = "Menu Item 1" :
```

```
$2 = "Menu Item 2" :
$3 = "Menu Item 3" :
.addmany 10,10,1,2,2 :
.addmany 30,10,3,1,2 :
.spacebar : | : esc :
>!
```

< .AddMenu Xpos, Ypos, $1 >
Domain: All Modules
Adds a numbered item **$1** to a vertical menu at column (0-79) **Xpos**, line (0-23) **Ypos.**

```
<sa-A>:<all :
clear 255 :    //Clear all variables
.Cls 1 :   //Clear the screen. See <.Cls> for details
$1 = "Menu Item 1" :
$2 = "Menu Item 2" :
$3 = "Menu Item 3" :
$4 = "Menu Item 4" :
.addmenu 10,10,$1 :    //Add menu items #1
.addmenu 10,12,$2 :    //Add menu items #1
.addmenu 30,10,$3 :    //Add menu item  #3
.addmenu 30,12,$4 :    //Add menu item  #4
.spacebar : | : esc :
>!
```

< .Cls NUM >
Domain: All Modules
Clears the screen. NUM = 0 the entire screen will be cleared, if 1, the middle 20 lines will be cleared, and if 2 or more, the normal work area will be cleared. The work area varies with the application.

See the SaveScr command for a note on what can be a confusion factor if SaveScr and RestScr are used in conjunction with .Cls.

```
<sa-A>:<all :
.Cls 0 :
.writestr 255,10,".Cls 0 cleared the entire screen." :
.writestr 255,11,".spacebar will put message on line 23" :
.writestr 255,12,"We didn't lie with our entire screen statement" :
.spacebar : | : esc :
.Cls 1 :
.writestr 255,10,".Cls 1 cleared the middle 20 lines" :
.spacebar : | : esc :
.Cls 2 :
.writestr 255,10,".Cls 2 cleared the work area" :
.spacebar : | : esc :
>!
```

< .DoMenu X >
Domain: All Modules
Activates a menu designed with < .AddMany > &/or < .AddMenu > , highlighting the item specified by X. Z is returned with the user's choice where 0 = Abort/none, and greater than 0 is the menu item number.

```
<sa-A>:<all :
clear 255 :    //Clear all variables
.Cls 1 :
```

```
$1 = "Menu Item 1" :
$2 = "Menu Item 2" :
$3 = "Menu Item 3" :
$4 = "Menu Item 4" :
.addmany 10,10,1,2,2 : //Add menu items #1 and #2.
.addmany 30,10,3,1,2 : //Add menu item  #3
.addmenu 30,12,$4 :    //Add menu item  #4
I = 3 :        //Menu item to highlight
.domenu I :    //3 in place of I works just as well
if Z = 0 : //User hit escape
   .Cls 1 :
   .writestr 255,10,"You hit ESC. Try one of the menu numbers." :
   (.beep 100,100) 5 : //Wake user up
   wait 3000 :
   RPT :
endif :
.Cls 1 .writestr 255,10,'You chose #' + str$ Z:
wait 3000: esc esc :
>!
```

<.FCard NUM,STRING,TYPE>
Domain: All Modules
.FCard draws a stacked filecard, based on NUM (1-4), on the screen, using STRING for the title. TYPE = 0 says to use plain text to draw the filecard. TYPE = 1 says to use MouseText to draw the filecard.

Unlike .FileCard, there is range checking done so NUM's larger than 4 will result in a "Dot Command Error" on line 21 of the screen.

See <.popmenu> section for technique to back out of a card to a lower numbered one.

See the file: Dot MenuTools in the UltraMacros folder on the /extras disk for AW 4 or AW 5 for some more examples on how to use the command.

```
<sa-A>:<all :
A = 0 :        //Type of filecard
$1 = " using normal text" :
Begin :

   .Cls 1  //Clear the screen.
   .FCard 1,"Filecard 1",A :
   .writestr  20,12, ".FCard 1 drew this" + $1 :
   .spacebar :
   .FCard 2,"Filecard 2",A :
   .writestr  20,12, ".FCard 2 drew this" + $1 :
   .spacebar :
   .FCard 3,"Filecard 3",A :
   .writestr  20,12, ".FCard 3 drew this" + $1 :
   .spacebar :
   .FCard 4,"Filecard 4",A :
   .writestr  20,12, ".FCard 4 drew this" + $1 :
   .spacebar : esc :
   if A > 0 then exit : endif :
   A = A + 1 : //Do MouseText this time
   $1 = " using MouseText" :
Rpt :
>!
```

If you wish to draw a single filecard and NUM isn't 1 then there will be a couple of ugly lines, called sidelines, coming out of the top right and lower left of the card. I cannot find a work around for this as in < .FileCard > .

< .FileCard STRING, NUM >
Domain: All Modules
.FileCard draws a stacked filecard, based on NUM (1-4), on the screen, using STRING for the title. Be careful with NUM. There is no range checking done so < .FileCard STRING, 10 > is attempted with some really ugly results.

See < .popmenu > section for technique to back out of a card to a lower numbered one.

See the file: Dot MenuTools in the UltraMacros folder on the /extras disk for AW 4 or AW 5 for some more examples on how to use the command.

See SA-B below for explanation of "orphan" cards.

```
<sa-A>:<all :
.Cls 1     //Clear the screen. See <.Cls> for details
.FileCard "Filecard 1", 1 :
.writestr  25,12, ".FileCard 1 drew this" :
.spacebar :
.FileCard "Filecard 2", 2 :
.writestr  25,12, ".FileCard 2 drew this" :
.spacebar :
.FileCard "Filecard 3", 3 :
.writestr  25,12, ".FileCard 3 drew this" :
.spacebar :
.FileCard "Filecard 4", 4 :
.writestr  25,12, ".FileCard 4 drew this" :
.spacebar : esc :
>!
```

If you wish to draw a single filecard and NUM isn't 1 then there will be a couple of ugly lines, called sidelines, coming out of the top right and lower left of the card. The two pokes in SA-B show how to deal with the situation.

```
<sa-B>:<all :
.Cls 1 :
poke $C3D,$0   //Extra sidelines off
.FileCard "Single Card",2 :
poke $C3D,$80 :    //Put them back for someone later.
.spacebar : esc :
>!
```

< $1 = .GetInput X,Y,$2,$3,L >
Domain: All Modules
This command allows you get string input at a specific location on the screen with a given prompt and a default string. You can also specify the maximum length of the string (1-80 char). With this command, you can set up your own input forms if you'd like.

```
$1 = the result string
X  = The left column number for Prompt to start in.
Y  = The line number that Prompt begins in
$2 = The user prompt string
$3 = The default string defined by you (can be null).
```

The simplistic view of the return values in Z:

```
Z =   0 = Escape key pressed. (See Note:)
Z = 155 = oa-esc keys pressed
Z = 191 = oa-\ or oa-? pressed
Z = 209 = oa-Q was pressed
Z = 211 = oa-S was pressed
```

Note: If the user has entered some characters and then hits esc the entered characters are discarded and .GetInput awaits fresh input. A second esc will cause .GetInput to abort and report 0 via Z.

The reality view of the return values in Z:

If a control character (exceptions noted below) is pressed as the first character then its value will be reported as the number of characters typed, however, the result string ($1 in the example), will be null.

Exceptions: Control H, U, Y, Z, \, and _ are caught internally and are not reported back. Also, Control-T changes the typed cursor from normal to MouseText, Inverse, and back to normal. It too is not reported back.

If a non-control character and the oa- key are pressed as the first character its high bit ASCII value is reported as the number of characters typed but the result string is null i.e., oa-A = 193 and $1 = null. Note that lower case a-z values are reported as if upper case A-Z had been pressed. In addition, oa-Delete keeps its normal function of deleting the character under the cursor.

Probably a bug:

1. If Return (Ctrl-M) is typed as the first character Z will contain 2, not 13 as you would expect. The result string is null.

2. If oa-Return (oa-Ctrl-M) is typed as the first character Z will contain 3, not 141 as you would expect. The result string is null.

If one or more characters are typed and then all are deleted via the Delete key and then either Return or oa-Return is typed zero is reported, which is the correct value.

Appears you are best consulting the result string. If its null, nothing typed. Non null, soemthing was.

```
<sa-A>:<all:
.Cls 0 :
$1 = .GetInput 1,3,"Name:    ","",20:
$2 = .GetInput 1,4,"Address: ","",40 :
$3 = .GetInput 1,5,"City:    ","",30 :
$4 = .GetInput 1,6,"State:   ","",2 :
$5 = .GetInput 1,7,"Zip:     ","",10 :
msg 'Any key to quit this madness' :
A = key :
| : esc :
>!
```

< $1 = .GetString "Prompt","Default",MaxLen >
Domain: All Modules
This command is a mix between < .GetInput > and the UltraMacros < GetStr > command. With it, you can specify the Prompt for the user and the Default String as well as the maximum length. It is always displayed on the bottom of the screen.

When you're in the AppleWorks Data Base, the internal < getstr > command is used to define the contents of a category. Since .getstring, .getinput and .getvalue all use the same < getstr >, that can cause a few problems.

Here's a macro by Jim Parker which works around that. Press Enter for a null category, and use OA-Q to stop the macro.

```
<sa-A>:<adb :
msg ' Type oa-Q to quit ' :
$10 = .getstring "Enter: ","",60   //null default
if Z = 209 then :  //quit on OA-Q
   msg '' :
   stop :
endif :
spc :      //work around
esc :
oa-Y :     //Wipe out current category name
print $10 :
rtn :      //plug it in
goto sa-A :    //Repeat
>!

<sa-A>:<all :
.Cls 0 :
$1 = .GetString "Enter your birthdate: ","11/18/67",8:
if Z > 0 and Z < 155 then :
msg ' You came into this world on ' + $1 + '  Key ' :
else msg str$ Z + ' Key ': endif :
#Key2Stop :
>!
```

< X = .GetValue $1,L,H,D >
Domain: All Modules
Captures user numeric input at the bottom of the screen, with prompting, a minimum, and a maximum value limits.

X = The result value
$1 = The prompt string
L = The minimum acceptable value (0-65534)
H = The maximum acceptable value (1-65534)
D = The default value (defined by you)
Z = Command key values:
  Z = 0 = esc pressed (and X is set to 65535)
  Z = 155 = oa-esc was pressed
  Z = 191 = oa-/ or oa-? was pressed
  Z = 209 = oa-Q was pressed
  else Z = the length of the response string (1-5)

The weirdness outlined for < .GetInput > is present here with regards to control characters and oa-keys other than those given above, with the additional factor that the result is set to 65535. Thus, it appears that .GetValue is attempting to tell the programmer that the user has hit weird keys, however, the value of Z not being 0 tends to confuse the issue.

The stated maximum of 65534 has its problems too. If one types 65535, it is accepted and Z is set to 5. So a realistic maximum is 65533 with the knowledge that anytime 65535 is typed it will be accepted and Z set to 5.

This example is the first wholesale copying of an "Ultra to the max!" example. My only additions were to assure the cursor state was the same on exit as its state on input. In addition, I added the "or" check to check for some of the other values Z can return.

This is a really cool example by Will Nelken in that the Begin/Rpt loop is exited the second time without putting in an exit/Rpt trick or some such. Also, using the same code for two different things (Slot then Drive), is neat too.

```
<sa-A>:<all :
$7 = "Slot: " :    //Set prompt
Y = 0 :        //Init
A = 6 :        //Set first default
M = 7 :        //Set first max

// Get state of present cursor. 0 = insert, 1 = overstrike
C = peek $10F1 :
poke $10F1,1 : //Assure overstrike cursor

Begin :
   X = .GetValue $7,1,M,A :
   if Z = 0 or Z > 100 then :
       poke $10F1,C :  //Put cursor to input state
       endmacro :
   endif :
   if Y = 0 then :
       Y = X : //Save Slot
       $7 = "Drive: " :    //Change prompt
       A = 1 :     //Change default
       M = 2 : //Change max
       Rpt :   //Do 2nd loop
   endif :

msg ' Slot ' + str$ Y + ', Drive ' + str$ X + ' ' :
.SpaceBar :
poke $10F1,C : //Cursor state on input
msg '' :   //Clear msg line
>!

<sa-A>:<all :  //Version easier to play around with
B = peek $10F1 :
poke $10F1,1 :
Begin :
   msg '' :

   X = .GetValue "Number ",9,65533,0 :
   msg "X = " + str$ X + "  Z = " + str$ Z + "   Key" :
   A = Key :
   if A = 27 then exit : endif :
Rpt :
poke $10F1,B :
>!
```

< .List X,Y,W,L,S,E,$1 >
Domain: All Modules
This command displays a scrolling list in an optional titlebox.

X = Left column number for the box (0-77)

Y = The upper line number for the box (0-21)
W = The Width of the box in characters
L = The Length in lines of the box
S = The Starting string number (1 - (99 - L))
E = The number of the ending storage string (1-99)
$1 = The title string or null
$1 = null means do not draw a box around the list and there is
  no title for the list nor is the built-in SaveScr/RestScr
  activated
Z = Command key values
  Z = 0 = Esc was pressed
  Z = 155 = oa-Esc was pressed
  Z = 209 = oa-Q was pressed
  Z = 1 through (E - S) + 1 = the item number selected.

Use Down, OA-Down, Up, OA-Up, OA-1 and OA-9 to move quickly through the list.

Limits:
You can't use $0 as part of your list.
If $1 is too wide for the defined window it will appear be blank.

```
<sa-A>:<all :  //No title or box from .List
for I = 1 to 30 :
   $(I) = "This is Item #" + str$ I :
next i :
$5 = "Howdy = #5" :    //Vary the width a little
SaveScr :  //Save screen
.box 40,6,22,9,1 : //Draw a box around the list
.List 41,5,20,7,1,30,"" :  //null prompt means no box or savescr
RestScr :  //Restore the screen

if Z = 0 or Z > 30 then msg str$ Z : else :
msg "You picked Item #" + str$ Z :
.SpaceBar :
msg '' :
>!

<ba-A>:<all :  //Title and box from .List
.Cls 0 :   //.List built-in SaveScr will see blank
for I = 1 to 30 :
   $(I) = "This is Item #" + str$ i :
next I :

.List 41,5,20,7,1,30,"30 Item List":

if Z = 0 or Z > 30 then msg str$ Z :
else :
   msg "You picked Item #" + str$ Z :
endif :
.SpaceBar :
| : esc :  //Get the screen back
>!
```

< .MacroNames : goto sa-A >
Domain: All Modules
Displays a list of defined Macro Titles.

This is the "live" sa-Esc command which displays a list of Macro Titles. Choosing an entry runs that macro. The .MacroNames command MUST be followed by a goto "some macro" command. The macro doesn't even have to exist. The .MacroNames command will plug in the name of the macro that the user selects i.e., the dummy macro is a place holder.

See Chapter 2 for a discussion of Labels of all shapes and sizes. Macro Titles has its own section.

```
<sa-A>:<all :
.MacroNames :
goto sa-A :    //sa-A is a dummy macro name
>!
```

NOTE: If you enter the debugger < oa-Ctrl-X > , followed by < oa-D > to see dot commands, tab to where the screen titled MENUTOOLS2 is displayed. Look at the bottom of that screen and you will see .MacroNames displayed twice. The first is he one just discussed < sa-ESC > . The second is < ba-ESC > and does exactly the same thing as < sa-ESC > . There appears to have been some sort of plan to use this if MenuTools2 ever needed to be expanded.

< $1 = .MenuItem >
Domain: All Modules
This command reads the current menu item from any numbered menu list.

GOTCHA ALERT! Initially I used oa-Q to break out of the loop in this example macro. It appears that when you call < .MenuItem > and subsequently type oa-Q (209) to get away, < .MenuItem > reports esc (27) for some unknown (to me), reason. Esc is needed so one can back up in the menus. I guess that the writer of < .MenuItem > didn't want to deal with the user jumping to the Desktop Index.

To get around the oa-Q problem, the macro is changed to use oa-W to quit when you have played around enough. If you forget oa-W then oa-Ctrl-X to get into the debugger and then oa-N to stop the macro.

```
<sa-A>:<all :
oa-Q :     //Get to the main menu
esc :
msg "--> Cursor through the menus; oa-W to quit <--" :
.SpaceBar :

begin
   $1 = .MenuItem :    //Read the highlighted menu item
   msg $1 :    //Echo menu item
   X = key :   //Get a key
   if X = 215 then :   //Quit if user pressed oa-W
       msg '' :    //Shut off the message line
       stop :  //Stop all macro activity
   endif :
   print chr$ X :  //Send the character to AW
Rpt
>!
```

< .OnGosub Val,"abcdDefghijk" >
Domain: All Modules
This command lets you call another macro based on a value. Val is an index into the string that follows which contains a list of macro names. Lower case [a-z] represents < sa- > macros, and upper case [A-Z] represents < ba- > macros. All other characters [0-9!@#$%[˜, etc.] (in my experiments), represent < sa- > macros.

Considering the prototype given above, if Val = 4, then the command would call the macro < sa-D > , if 5 < ba-D > .

In the following example I intially tried feeding .OnGosub out-of-range values i.e., Z = 0 or Z = 15 just prior to the call of .OnGosub. It did nothing i.e., didn't crash, didn't try and call a non-existent macro, etc., etc.

I then put in the OnErr goto sa-N and sure enough, it trapped the out-of-range error. Commenting out one of the sub macros i.e., sa-C did NOT cause an error. We can only wish it did.

```
<sa-A>:<all :
Clear 255 :
OnErr goto sa-N :
$20 = "6cDfghijklm" :
for I = 1 to 11 :
   $(I) = "Do Job #" + str$ I :
next I :
$12 = "No macro #12" :

Begin :
   .List 41,5,20,7,1,12,"Job List":
   ifnot Z > 0 and Z < I then msg '' : endmacro : endif :
   .OnGosub  Z,$20 :
Rpt :
>!

<sa-6>:<asr : msg 'sa-6'>!
<sa-C>:<asr : msg 'sa-C'>!
<ba-D>:<asr : msg 'ba-D'>!
<sa-F>:<asr : msg 'sa-F'>!
<sa-G>:<asr : msg 'sa-G'>!
<sa-H>:<asr : msg 'sa-H'>!
<sa-I>:<asr : msg 'sa-I'>!
<sa-J>:<asr : msg 'sa-J'>!
<sa-K>:<asr : msg 'sa-K'>!
<sa-L>:<asr : msg 'sa-L'>!
<sa-M>:<asr : msg 'sa-M'>!

<sa-N>:<asr : msg ' OnErr to sa-N  Key ' : #Key2stop>!
```

< .LoadVar STRING, Option >
Domain: All Modules
Restores variables from file named by STRING. Also, see < .SaveVar > for the way the file was originally created.

1. If STRING is a fully-qualified pathname (starts with a "/" i.e., /H2/UMVars/filename), then the file is loaded from that path.

2. If STRING is simply a filename then it is loaded from the AW.INITS subdirectory i.e., /Pgms1/APW/FIVE.0/AW.INITS/filename.

3. If STRING is a partially-qualified pathname (VARS/filename) then it is prefixed with the path to the AW.INITS folder i.e., /Pgms1/APW/FIVE.0/AW.INITS/VARS/filename. (Naturally, you will have to create the folder "VARS" prior to running any macro that wants to save variables there.

Either 1 or 3 is the preferred method since it keeps non-INIT files from cluttering up your AW.INITS folder.

Option determines which variables are restored, using the same values as <clear>. One difference is that clear 100 or 200 or 255 will not clear $0, however, loadvar 100, 200, or 255 will load $0.

Option	Result
0-9	Loads a numeric array - 1 loads A(1) through Z(1), 7 loads A(7) through Z(7), etc.
50	Loads all 260 numeric variables
100-190	Loads ten strings - 180 loads 80 through 89, 100 loads strings 0 through 9.
200	Loads all strings 0 through 99

See <.savevar> for an example macro.

<.OnGoto Val,″abcdDefghijk″>
Domain: All Modules
This command lets you goto (not call) another macro based on a value. Val is an index into the string that follows which contains a list of macro names. Lower case represents SA macros, and upper case indicates BA macros i.e., .OnGoto 5,″abcdDefghijk″ would call ba-D while .OnGoto 4,″abcdDefghijk″ would call sa-D.

Note that the sub-macros have been changed from asr to awp or all. This is because Ultra's goto will not goto a macro marked asr.

Also, notice that they perform a goto to return to sa-A. A call of sa-A would work until a table (16 entries), fills up. Then UM would crash. Another solution would be for the sub-macros to <pop 1 : sa-A>.

In the following example I intially tried feeding .OnGosub out-of-range values i.e., Z = 0 or Z = 15 just prior to the call of .OnGosub. It did nothing i.e., didn't crash, didn't try and call a non-existent macro, etc., etc.

I then put in the OnErr goto sa-N and sure enough, it trapped the out-of-range error. Commenting out one of the sub macros i.e., sa-C did NOT cause an error. We can only wish it did.

```
<sa-A>:<all :
OnErr goto sa-N :
$20 = "6Bcdmlg7ijk" :
for I = 1 to 11 :
   $(I) = "Do Job #" + str$ I :
next I :
$12 = "Error. No #12" :

.List 41,5,20,7,1,12,"Job List":
ifnot Z > 0 and Z < I then msg '' : endmacro : endif :
.OnGoto  Z,$20 :
>!

// Note that these macros cannot be asr. I made them all. Awp would
// work too (see sa-6), unless you were not in a awp file when you
// called this macro. Try calling sa-A from the main menu. Call any
// except #1. Note that the menu stays on the screen. Call #1 and
// note that the macro silently aborts.
```

```
<sa-6>:<awp : msg 'sa-6' : goto sa-A>!
<ba-B>:<all : msg 'ba-B' : goto sa-A>!
<sa-C>:<all : msg 'sa-C' : goto sa-A>!
<sa-D>:<all : msg 'sa-D' : goto sa-A>!
<sa-M>:<all : msg 'sa-M' : goto sa-A>!
<sa-L>:<all : msg 'sa-L' : goto sa-A>!
<sa-G>:<all : msg 'sa-G' : goto sa-A>!
<sa-7>:<all : msg 'sa-7' : goto sa-A>!
<sa-I>:<all : msg 'sa-I' : goto sa-A>!
<sa-J>:<all : msg 'sa-J' : goto sa-A>!
<sa-K>:<all : msg 'sa-K' : goto sa-A>!

<sa-N>:<all : msg ' sa-N caught an error Key ' : #Key2Stop>!
```

< .MakeMenu X, Y, Frst, Count, Spc, Begin >
Domain: All Modules
This do-it-all command creates a vertical menu. It sets the location, item names and activates the menu.

X   is the starting column for the menu items
Y   is the starting line number for the menu items
Frst is the first storage string in a range
Count is the number of items in the menu
Spc is the spacing between menu items, normally 1 or 2
Begin is the number of the menu item to initially hilight.

Z is returned with the user's choice where 0 = Abort/none, and greater than 0 is the menu item number selected.

```
<sa-A>:<all :
// Define 4 strings
for X = 1 to 4: $(X) = str$ X: next X:

poke $1BC2,9:  //poke $1BC2,9: Allow Tab in menu
               //poke $1BC2,191: $BF:allow OA-?
               //poke $1BC2,141: $8D:allow OA-RTN

.FileCard "",1:    // Fake; but necessary
.Cls 1:    // optional:    // Hides FC
.makemenu 10,6,1,4,2,1:

K=0:           // initialize K (#key)
if Z = 0 then K = peek #key : endif :
if K = 9 : msg "Tab" : endif : // Tab was allowed & pressed
if K = 191 : msg "OA-? for help":  // OA-? was allowed
endif:

ifnot K = 9 then :
   msg str$ Z + " Key" : endif :   // Valid key was pressed

poke $1BC2,$BF:     // Restore the spare key; default OA-?

>!
```

The following examples are somewhat lengthy, however, menus tend to be a confusing subject so I think the space is well worth it.

The Custom Menu is based on a macro that Dan Crutcher gave me many years ago. The mousetext shading on the right and bottom is a favorite of Dan's.

It has been modified into a general purpose box drawer since it will draw the box based on input variables as opposed to a fixed location.

```
<sa-A>:<all :
clear 255 :    //Zap all variables
.FileCard "",1 :   //Assure oa-? is enabled
.Cls 1 :
I = 10 :   //Tell ba-A what string has header info
$10 = "Your Own Custom Menu Box" :
$11 = "Menu Item 1" :
$12 = "Menu Item 2" :
$13 = "Menu Item 3" :
.writestr 20,7,"Just a bare menu"
.makemenu 20,10,11,3,1,1 :
.Cls 1 :
.FileCard "In a filecard",1 :
.makemenu 20,10,11,3,1,1 :

.Cls 1 :

//StrtCol  StrtLine  FrstString  NumItems   Width    Height
   X = 20  : Y =  7 :   F = 11  :  N = 3 :   W = 33 : H = 7 :

ba-A :     //Draw the box

.makemenu X,Y,F,N,1,1 :    //ba-A adjusted X & Y to fit inside box
esc :
>!

<ba-A>:<asr :
// Inputs:
//                                   Header
// StrtCol  StrtLine  Width   Height  string #
//    X       Y         W       H        I

// Uses:
// Vars:    B, D, E
// Strings: $1, $2, $3, $4

// Save start col and line num
   B = X : D = Y :
   $1 = "" : $2 = "" : $3 = "" : $4 = "" :

// Set up the strings used to draw the box.
// $1 & $2 have normal text while $3 abd $4 have mousetext
   ($1 = $1 + "_" : $2 = $2 + " " : $3 = $3 + &L& : $4 = $4 + &S&) W :

   .WriteStr X, Y, $1 :

// Down a line and back a column
   Y = Y + 1 : X = X - 1 :

// Repeat for H lines
   (.WriteStr X,Y, &Z& + $2 + &N& : Y = Y + 1) H :
```

```
// Back up a line
   Y = Y - 1 :
   .WriteStr X, Y, &Z& + $1 + &N& :

   X = X + 2 : Y = Y + 1 :
   .WriteStr X,Y,$3 :

// Compute E = start col for heading
   E = len $(I)   : //Length of title (25 in this case)
   E = W - E / 2 : //Width - E =  33-25 = 8/2 = 4

// Back to top, over to E and down a line
   X = B + E : Y = D + 1 :
   .WriteStr X, Y, $(I)   :

// Top and down two lines for divider line
   X = B : Y = D + 2 :
   .WriteStr X, Y,$4 :

// For .makemenu to start placing menu items
   X = X + 1 : Y = Y + 1 :
>!
```

< .MenuBar PromptString, ItemString >
Domain: All Modules
Creates a horizontal menu on the bottom screen line (23), prompting the user with PromptString and using Itemstring to designate up to eight items separated by vertical bars "|" (pipes).

The user's choice is returned in variable Z, counting from left to right in the ItemString. If Z is zero, the user hit Escape. If Z is 155 then the user hit oa-ESC. This is usually tested for by asking if Z > 30 since that is larger than any reasonable menu size.

The user chooses items from the list by using the right and left arrow keys to hilight the wanted item and selecting it by pressing Return.

An alternate selection process is to press the first letter of the wanted item. This only works if the on-screen item is upper case (the user can press upper or lower case).

If two, or more, items have the same first upper case letter, the rightmost item is selected:

Pressing "A" when: Apple Amiga Atari
is displayed will select Atari.

Pressing "A" when   Apple Amiga atari
is displayed will select Amiga.

```
<sa-A>:<all :
.Cls 1 :
$1 = "Esc for next example    Choose" :
$2 = "Abc|Bcd|Cde|Def|Efg|Fgh|Ghi|Hij" :
.writestr 1,5,"You can use either the first letter of the item to
choose" :
.writestr 1,6,"or the right and left arrow keys to highlight and
Return to" :
.writestr 1,7,"chose. Try both methods." :
ba-A :
.Cls 1 :
```

```
$2 = "abc|bcd|cde|Def|efg|fgh|ghi|Hij" :
.writestr 1,5,"First letter selection works only if the item in the" :
.writestr 1,6,"list is UPPER case. You can type upper or lower." :
.writestr 1,7,"Only two items in this list can be chosen by typing
its" :
.writestr 1,8,"first letter." :
ba-A :
$2 = "A1|A2|A3|B1|B2|A4|b3" :
.Cls 1 :
.writestr 1,5,"When more than one item has the same first UPPER case"
:
.writestr 1,6,"letter the last one with that letter is the one chosen"
.writestr 1,8,"Note that b3 is not chosen when you type B, but A4 is
for A.":
.writestr 1,10,"Naturally, you can use the arrow keys and Return" :
.writestr 1,11,"to select any item in the list." :
$1 = "Esc to quit     Choose "
ba-A :
| : esc :  //Get display back.
>!

<ba-A>:<asr :
Begin :
   .menubar $1,$2 :
   if Z = 0 or Z > 30 then endmacro : endif :
   //Vertical bars to commas for <.choose>
   $3 = .subchar $2,#'|',#'|',#',' :
   $3 = .choose $3,Z :
   $3 = "You chose " + $3 :
   $3 = $3 + "   Any key to continue " :
   msg %J% + $3 + %K% : A = key : msg '' :
Rpt :
>!
```

<.MenuBar2 "Prompt","Choices",Delimiter>
Domain: All Modules
The .menubar2 command is identical to .menubar except that you specify the delimiter character following the two strings.

```
<sa-A>:<all :
.menubar2 "Pick","Arvada, Co\Somewhere else\London",#'\' :
if Z = 0 msg 'You hit ESC' :
   .SpaceBar :
   msg '' :
   endmacro :
endif :
if Z = 155 then msg 'You hit oa-ESC' :
   .SpaceBar :
   msg '' :
   endmacro :
endif :
msg 'You chose item ' + str$ Z :
.SpaceBar :
msg '' :
>!
```

<$90 = .Pick X,Y,W,L,Start,End,"Title">

Domain: All Modules
This command displays a scrolling list with strings specified by Start and End. **X,Y,W,L** specify the **X,Y** position on the screen for the upper left corner, **W** the Width of the box (in char) and **L** the # of Lines for the string display area. The "Title" is optional and if present it is drawn at the top of the box.

Limits: You can't use $0 as part of your list.

<.Pick> returns a string which indicates which items were picked. The user picks multiple items by a right arrow key to select (left to deselect) each item to be picked followed by Return to pick all selected items.

To select all items oa-right does the job.

To select a single item the user can simply highlight the wanted item and press Return.

To move through the list the user can use: down, oa-down, up, oa-up, oa-1, oa-9. Note that oa-2 through oa-8 are not supported.

Each character of the string is an item number that was picked. Use the length of the string to determine how many items were picked. The following macro demonstrates how to use this command:

X = Left column number for the box (0-77)
Y = The upper line number for the box (0-21)
W = The Width of the box in characters
L = The Length in lines of the box
S = The Starting string number (1 - (99 - L))
E = The number of the ending storage string (1-99)
$1 = The title string or null
$1 = null means do not draw a box around the list and there is
no title for the list

Z returns:
Z = 0 = Esc
Z = 155 = OA-Esc
Z = 209 = OA-Q
Z = Number of picked items

```
<sa-A>:<all:
SaveScr:

for I = 1 to 30 :
   $(I) = "This is Item #" + str$ I :
next I :
$90 = .Pick 41,5,20,7,1,30,"30 Item List":
RestScr :  //Get rid of the .Pick list
X = len $90:
ifnot X > 0 and X < I then : //I is list length + 1 after <for>
   msg str$ Z :    //Show nothing picked
   .SpaceBar :
   msg '' :
   stop :
endif :

$2 = "You picked " :
```

```
for I = 1 to Z :
   $1 = mid $90,I,1:
  X = asc $1:
  $2 = $2 + str$ X + " ":
next I :

msg $2 :
.SpaceBar :
msg '' :
>!
```

<.PopMenu>
Domain: All Modules
This command is useful only in conjunction with the .FileCard and .FCard commands. To see what it does compile the following example and step through it.

After you have loaded or created a file, the right side of the top line has the word
Escape: Somewhere, where Somewhere is where you will go if you press the escape key. Since it tells you where you will be going, it is called the "Escape Map."

The center of that line tells you where you currently "are," i.e., REVIEW/ADD/CHANGE and is called the "Title". (The left side has the name of the current file, however, that isn't relevant to this discussion.)

You start out in AppleWorks with Main Menu as the title and nothing in the Escape Map since you have not gone anywhere yet. If you add an AWP file (i.e., CH04), the title will become REVIEW/ADD/CHANGE and the Escape Map becomes Main Menu. If you now start adding filecards via the .FileCard command the title will change to the first filecard, Escape Map to CH04, etc., etc.

This information is kept in an Escape Stack which has a pointer to the second member of the stack (when such a member exists):

```
Main Menu   Chapter 4  Card 1      Card 2
          Main Menu  Chapter 4   Card 1      <-- Pointer
                               Chapter 4
                               Main Menu
```

The stack gets extended in the same fashion as you add cards 3 and 4.

There are two sets of macros below. The first is to use with <.FileCard) and the second with .FCard. IMHO .FCard with mousetext is the command of choice.

------------- Macros for use with .FileCard -----------------
```
<sa-A>:<all :
.Cls 1 :
ba-A :          //Draw the cards
.writestr 15,12,"The Escape Map says that ESC will get you back to
Card 3." :
.writestr 15,13,"That is true only if you program it into your macro."
.writestr 15,14,"Actually it will erase all cards and put you back to
the"
.writestr 15,15,"macro source file.  Hit esc to see that happen." :
sa-ctrl-A : esc : msg 'esc to go on' : sa-ctrl-A : .Cls 1 :
: ba-A :        //Get the cards back
.writestr 15,12,"Hit esc to erase Card 4 without using <.popmenu>" :
```

```
.writestr 15,14,"You erase by displaying a card above the one(s) you"
:
.writestr 15,15,"want to erase. Adding 128 to the card number tells" :
.writestr 15,16,".FileCard to NOT add the redrawn card's name to" :
.writestr 15,17,"the Escape Stack. Helps, but not whole solution." :
sa-ctrl-A :         //Get esc key from usr
.FileCard "",131 :     //Display Card 3
.writestr 15,12,"What the heck! Title for card 3 now says Card 4 and"
.writestr 15,13,"the Escape Map says esc will get you to Card 3."
.writestr 15,14,"What a mess!! Lets keep going. Hit esc again to zap
3" :
sa-ctrl-A : .FileCard "",130 :
.writestr 15,12,"esc to zap 2" :
sa-ctrl-A : .FileCard "",129 : //Card code for Card 1
.writestr 15,12,"Never did get any better did it? Lets do it over" :
.writestr 15,13,"with our good friend <.popmenu>  esc to go on" :
sa-ctrl-A : esc : .Cls 1 :
ba-A :         //Redraw all four boxes
For C = 131 to 129 step -1 :
.writestr 15,12,"Esc to erase the last filecard" :
.writestr 15,13,"Keep your eye on the Escape Menu it will name the" :
.writestr 15,14,"card under the present. The Title and the card name"
:
.writestr 15,15,"on the filecard tab will agree" :
sa-ctrl-A :
.popmenu :
.FileCard "",C :
next C :
sa-ctrl-A : esc :
>!

<ba-A>:<asr :
.FileCard "Card 1",1 :
.FileCard "Card 2",2 :
.FileCard "Card 3",3 :
.FileCard "Card 4",4 :
>!

<sa-ctrl-A>:<asr :
Begin : A = key : ifnot A = 27 bell : rpt : endif : msg ''>!

------------- Macros for use with .FCard -----------------

<sa-A>:<all :
T = 1 :            //Use mousetext
.Cls 1 :
ba-A :         //Draw the cards
.writestr 15,12,"The Escape Map says that ESC will get you back to
Card 3." :
.writestr 15,13,"That is true only if you program it into your macro."
.writestr 15,14,"Actually it will erase all cards and put you back to
the"
.writestr 15,15,"Main Menu.  Hit esc to see that happen." :
sa-ctrl-A : esc : msg 'esc to go on' : sa-ctrl-A : .Cls 1 :
: ba-A :       //Get the cards back
.writestr 15,12,"Hit esc to erase Card 4 without using <.popmenu>" :
```

```
.writestr 15,14,"You erase by displaying a card above the one(s) you"
:
.writestr 15,15,"want to erase. Adding 128 to the card number tells" :
.writestr 15,16,".FCard to NOT add the redrawn card's name to" :
.writestr 15,17,"the Escape Stack. Helps, but not whole solution." :
.writestr 15,19,"Note that the card names stay correct with .FCard":
sa-ctrl-A :        //Get esc key from usr
.FCard 131,"",T :       //Display Card 3
.writestr 15,12,"What the heck! Tab says Card 4, Title says Card 4
and"
.writestr 15,13,"the Escape Map says esc will get you to Card 3."
.writestr 15,14,"What a mess!! Lets keep going. Hit esc again to zap
3" :
sa-ctrl-A : .FCard 130,"",T :
.writestr 15,12,"esc to zap 2" :
sa-ctrl-A : .FCard 129,"",T :  //Card code for Card 1
.writestr 15,12,"Never did get any better did it? Lets do it over" :
.writestr 15,13,"with our good friend <.popmenu>  esc to go on" :
sa-ctrl-A : esc : .Cls 1 :
ba-A :         //Redraw all four boxes
For C = 131 to 129 step -1 :
.writestr 15,12,"Esc to erase the last filecard" :
.writestr 15,13,"Keep your eye on the Escape Menu it will name the" :
.writestr 15,14,"card under the present. The Title and the card name"
:
.writestr 15,15,"on the filecard tab will agree" :
sa-ctrl-A :
.popmenu :
.FCard C,"",T :
next C :
sa-ctrl-A : esc :
>!

<ba-A>:<all :
.FCard 1,"Card 1",T :
.FCard 2,"Card 2",T :
.FCard 3,"Card 3",T :
.FCard 4,"Card 4",T :
>!

<sa-ctrl-A>:<asr :
Begin : A = key : ifnot A = 27 bell : rpt : endif : msg ''>!
```

< .Qmenu STRING, Filetype >
Domain: All Modules
Displays a custom OA-Q menu with the title specified by STRING and listing the file types specified by Filetype.

```
Filetype = 1 for DB files
Filetype = 2 for WP files
Filetype = 4 for SS files
```

Adding Filetypes together will list combinations of filetypes:

```
Filetype = 5 for DB and SS files
Filetype = 6 for WP and SS files
```

```
<sa-A>:<all :
//Load several DB, WP, and SS files on the desktop before running
//this macro
.qmenu "DB files",1 : .spacebar :
.qmenu "WP files",2 : .spacebar :
.qmenu "SS files",4 : .spacebar :
.qmenu "DB & SS files",5 : .spacebar :
.qmenu "DB, WP, & SS files",7 : .spacebar :
>!
```

< .SaveVar STRING >
Domain: All Modules
Saves all the numeric and string variables in a file named by STRING.

1. If STRING is a fully-qualified pathname (starts with a "/" i.e., /H2/UMVars/filename), then the file is saved to that path.

2. If STRING is simply a filename then it is saved to the AW.INITS subdirectory i.e., /Pgms1/APW/FIVE.0/AW.INITS/filename.

3. If STRING is a partially-qualified pathname (VARS/filename) then it is prefixed with the path to the AW.INITS folder i.e., /Pgms1/APW/FIVE.0/AW.INITS/VARS/filename. (Naturally, you will have to create the folder "VARS" prior to running any macro that wants to save variables there.

Either 1 or 3 is the preferred method since it keeps non-INIT files from cluttering up your AW.INITS folder.

```
<sa-A>:<all :
.Cls 1 :
$0 = "Original Strings" : sa-ctrl-A :
$2 = .AwPath : //Path AppleWorks was booted from
$1 = $2 + "/AW.INITS/VARS/OrigSaveVars" :
.SetDisk $1 :
.DropDir : //Drop the filename off of the path
$3 = .PeekStr $0C56 :  //This files name
oa-Q : esc : rtn : rtn :
$2 = screen 15,12,7 :
if $2 = "Getting" then :
   $0 = $3 :   //Try file's name
   oa-Q :
   find :
   rtn :
   .Cls 1 :
   .WriteStr  0,10,"You do not have a 'VARS' folder in your AW.Inits
folder" :
   .WriteStr 0,11,"Fix this and retry this macro" :
   msg 'Key' :
   A = key :
   oa-Q : esc :    //Fix screen
   endmacro :
endif :
$0 = $3 :
oa-Q : find : rtn :    //Back to original Try file
.savevar $1 :      //Save everything
.Cls 1 :
```

```
.writestr 0,8,"Strings 10-19 set to 'Original Strings' & .SavedVar
saved them" :
ba-A :          //Display strings
.spacebar :
.Cls 1 :
 clear 110 :
.writestr 15,8,"Strings 10-19 Are cleared" :
ba-A :
.spacebar :
.writestr 15,8,"Strings 10-19 new contents" :
$0 = "New strings" : sa-ctrl-A : ba-A :
.spacebar :
.writestr 15,8,"Using <.LoadVar> restores Strings 10-19 'Original
contents'" :
.loadvar $1,110 :
ba-A :
.spacebar : | : esc :
>!

<ba-A>:<asr :       //Display strings 10-19
I = 10 :
(
   .writestr 15,I,"String #    =" :
   $0 = str$ I : .writestr 23,I,$0 :   //The string number
   .writestr 30,I,$(I) :       //The string value
   I = I + 1 :
) 10 :
>!

//Write contents $0 to strings 10-19
<sa-ctrl-A>:<asr : I = 10 : ($(I) = $0 : I = I + 1) 10>!
```

NOTE: Sometime ago I sent out a general request for aid concerning a problem I was having. A user reported that they and others experienced difficulties after using < .SaveVar > even though that wasn't my problem.

> There is very clearly some sort of very noxious problem with
> .savevar which I'll mention although you don't seem to use it. It was
> reported by one person and then denied. Since in my case it resulted in
> complete reworking of my whole directory system on the hard drive, I
> never had the courage to explore it in any depth. However the sequence of
> triggering it seemed to be that after a .savevar had been done,
> AppleWorks had been quit (back to GS/OS) and then at a later time
> re-launched, in some way, sometimes rather seriously files would be
> shifted around into different directories. If I cannot avoid .savevar's
> use (because it's in someone else's macro program), I try to remember to
> reboot the computer when I quit AppleWorks. I suspect this would never
> happen on a IIe, which suggests you might try running your macros on a
> IIe and see if there's something in that.

Recent discussion with author of the above we both agree that if one never boots GS/OS and then uses the Finder to boot AppleWorks, there will not be a problem with .SaveVar. Since I don't boot GS/OS and then boot AppleWorks (or any other 8 bit program, I've had no trouble with the < .SaveVar > or < .LoadVar > couplets through scads of testing of the above example.

< .Say STRING >
Domain: All Modules

Displays STRING on the bottom line and waits for any key to proceed. There is no built in prompt telling the user what to do so you might consider making part of the message, "Any key to continue."

```
<sa-A>:<all :
.say "Hello World.  Hit a key or sit here forever" :
$1 = "Hello back at you.  Key" :
.say $1 :
>!
```

< .SpaceBar >
Domain: All Modules
Displays "Press spacebar to continue," on line 23 (bottom) of screen.

Actually, it will accept any one of these keys and sets Z to indicate the key pressed: spacebar: Z = 32 ($20), Return: Z = 13 ($D), ESC: Z = 0 ($0), oa-Q: Z = 209 ($D1), oa-S: Z = 211 ($D3), or oa-Ctrl-S: Z = 147 ($93).

In addition, you can hit oa-E to toggle the cursor from insert/overstrike, oa-H to print current screen, or oa-T to modify the tab ruler and still have the prompt, "Press space bar to continue."

< $1 = .StripChar STRING, Char, Option >
Domain: All Modules
Strips all consecutive occurrences of Char (ASCII value) leading or trailing in STRING based on Option:

Option = 0 = Strip both ends
Option = 1 = Strip front end
Option = 2 = Strip rear end

NOTE: Option can be specified as: 42 (ASCII value of an asterisk) or #'*'
See macro example below.

```
<sa-A>:<all :
.Cls 1 :
$1 = "     ***Buried Text***     "
msg 'Spaces and Asterisks at both ends' + %K% + "    |" + $1 + "|" :
.spacebar : msg '' :
$2 = .stripchar $1,32,0 :
msg 'Spaces gone both ends' + %K% + "    |" + $2 + "|" :
$1 = $2 :.spacebar :
$2 = .stripchar $1,#'*',1 :
msg 'Front asterisks gone' + %K% + "     |" + $2 + "|" :
.spacebar : msg '' :
$1 = $2 :
$2 = .stripchar $1,42,2 :
msg 'Rear asterisks gone' + %K% + "      |" + $2 + "|" :
.spacebar : msg '' :
oa-Q : rtn :
>!
```

< .Therm X,Y,C,M >
Domain: All Modules
This command will draw a thermometer to track macro activity.

X = The left column number for the box
Y = The top line number for the box
C = The Current progress (0 to start)

M = The Maximum value

If C = 0, then an empty thermometer is drawn at X,Y and it is initialized with the M value. If > 0, then the thermometer is updated to reflect the current position.

.Therm only works when the screen stays the same so the simplest way to handle this is to: < SaveScr : .Cls 1 : .Therm stuff : RestScr > See SaveScr for a tiny problem with this approach.

```
<sa-A>:<all:
SaveScr : .Cls 1 : //Save and clear the screen

.TitleBox 255,9,34,2,2,"Fill er up" :
.therm 23,11,0,10 :
$15 = "Loop #" :

for I = 1 to 10 :
   wait 1000 :
   .therm 23,11,I,10 :
   .WriteStr 36,14,$15 + str$ I :
next I :
.SpaceBar :
RestScr :
>!
```

This one cures the "tiny" problem with SaveScr (doesn't use it ;-)

```
<sa-A>:<all:
.Cls 1 :   //Clear the screen

.TitleBox 255,9,34,2,2,"Fill er up" :
.therm 23,11,0,10 :
$15 = "Loop #" :

for I = 1 to 10 :
   wait 1000 :
   .therm 23,11,I,10 :
   .WriteStr 36,14,$15 + str$ I :
next I :
.SpaceBar :
| : esc :  //Get the screen back
>!

If you wish to keep the screen text in the background

<sa-Ctrl-A>:<all:
SaveScr :
.TitleBox 255,9,34,2,2,"Fill er up" :
.therm 23,11,0,10 :
$15 = "Loop #" :

for I = 1 to 10 :
   display 0 : //Turn off display
   wait 1000 :
   display 1 :
   .therm 23,11,I,10 :
   .WriteStr 36,14,$15 + str$ I :
next I :
```

```
display 1 :    //Assure display is on
.SpaceBar :
| : esc :  //Get the screen back
>!
```

< .Writestr Xpos, Ypos, STRING >
Displays message in STRING at column Xpos (0-79) and line Ypos (0-23). If Xpos = 255 then STRING will be centered on line Ypos.

Unlike < **msg** > , the quotes used for STRING have no effect on the display. See the example below for inverse text. You do not have to put the control codes in strings: < .... %J% + "Inverse Text" + %K% > works fine. DO NOT fail to specify the closing %K% or your screen will go bonkers per a note from Randy in several GEnie posts.

```
<sa-A>:<all :
.Cls 1 :
$1 = %J% + "Inverse" + %K% :
$2 = "Normal" :
$3 = "   " :
For I = 3 to 19 :
   .writestr 255,I,$1 + $3 + $2 :
   $4 = $1 : $1 = $2 : $2 = $4 :
next I :
msg 'Key' :
A = key :
| : esc :
>!
```

IIgs only commands

Stay away from these if you intend that your macros run on machines other than the IIgs. The commands in this section are contained in a init file named: I.UM.IIGS.CMDS which is NOT automatically copied to the AW.INITS folder by the INSTALL.ULTRA program. You will have to copy it yourself.

< .ExtKB >
Domain: All Modules
UltraMacros 4 automatically calls ba- macros when you press a key on the keypad or a function key on the extended keyboard (unless you "disconnect" this feature by setting: Other Activities/Select Standard Settings for AppleWorks/UltraMacros Options/Enable keypad macros to "No" A setting your kindly editor heartily recommends :-).

By defining ba- macros with the < .ExtKB > command, you can vary the function according to which keypress was used i.e., did the user call ba-1 by pressing Option, Command, and 1 or did they simply press the 1 key on the keypad?

In order to test this macro you must set: Other Activities/Select Standard Settings for AppleWorks/UltraMacros Options/Enable keypad macros to "Yes"

Your Editor recommends setting it back to "No" as soon as your are done testing.

```
<ba-1>:<all :

//This macro is entered when either the Option, Command, and 1 are
//pushed or simply the 1 on the IIgs keypad

.ExtKB :
```

```
if Z = 1 then msg 'You used the 1 on the keypad. Key ' :
   #Key2Stop :
endif :

msg 'You used the Option, Command, and 1 keys on the main keyboard.
Key ' :
#Key2Stop :
>!
```

< C = .GetColor Y >
Domain: All Modules
The < .getcolor > command allows you to read the border, background and text colors on a IIgs.

C is the result, indicating which color is displayed. See the table in the < .SetColor > command which follows.

Y is the area to be interrogated:

```
Y = 1, boarder
Y = 2, text
Y = 3, background

<sa-A>:<all :
O = .GetColor 1    //Border
T = .GetColor 2    //Text
B = .GetColor 3    //Background
msg 'Border: ' +str$ O +'  T: ' + str$ T +' Background: '+ str$ B + '
Key ' :
#Key2Stop :
>!
```

< .SetColor X,C >
Domain: All Modules
The .setcolor command allows you to change the border, background and text colors on a IIgs.

X is the area flag:
X = 1, border
X = 2, text
X = 3, background

C is the color

C = 0, black
C = 1, dark red
C = 2, dark blue
C = 3, purple
C = 4, dark green
C = 5, dark gray
C = 6, medium blue
C = 7, light blue
C = 8, brown
C = 9, orange
C = 10, light gray
C = 11, pink
C = 12, light green
C = 13, yellow

C = 14, aqua
C = 15, white

//This sample macro sets the border to 16 different colors in turn.
//Modify the "W = 1" to "W = 2" or "W = 3" to change the text or
// background colors in the sample.

```
<sa-A>:<all :
for W = 1 to 3 :        //1:Border, 2:Text, 3:Background
   S = .GetColor W :    //What's the starting W color?
   for C = 0 to 15 :    //Try the various colors
       .SetColor W,C :
       sa-B :  //Display just set colors.
       .SpaceBar : //Display settings, wait for spacebar
   next C :
   .SetColor W,S : //Restore original color
next W :
   msg "" :
>!

<sa-B>:<all :
O = .GetColor 1    //Border
T = .GetColor 2    //Text
B = .GetColor 3    //Background
msg 'Border: ' +str$ O +'  T: ' + str$ T +' Background: '+ str$ B + '
Key ' :
>!
```

< $1 = .TimeGS X >
Domain: All Modules
The .TimeGS command

```
X = the format flag:

   X = 1, time only
   X = 2, AM/PM if Control Panel set for it

<sa-A>:<all :
.cls 0 :   //Clear the entire screen
poke $11AC,0 : //Keep Esc from aborting the macro
.WriteStr 255,10,"Escape: Stop Macro"  : //Top message

Begin
   $1 = .timegs 1 :    //Grab time without AM/PM
   $2 = .TimeGS 2 :    //Grab time with AM/PM

   .WriteStr 255,0, $1 + & [& + $2 :   //Display both formats
   X = peek $C000 :    //Peek the key location
   ifnot X = 27 then : //Check for Escape

Rpt : endif :

//Clean things up
poke $11AC,27 :    //Restore Esc for stopping runaway macros
| : esc :  //Bring back screen
>!
```

MathTools

Mathtools contains extended math dot commands (they all begin with .x), and other dot commands that have nothing at all to do with math. The others are here cause that is where Randy and Mark wanted them. Do you have a problem with this? Good.

For more on extended math see the oa-X command under < debug > in Chapter 2.

This section is a combination of much of stuff from the AW 5.1 /EXTRAS/Dot.MathTools file, Will Nelken's, "Ultra to the Max," and some slight observations/experiments by myself.

As a reminder, there are three types of dot commands:

1. "Stand Alone," commands that perform a function, but do not return any text or numeric results, thus not needing an equation in their syntax.

2. "String Commands," that yield a text string result.

3. Numeric Commands that yield a numeric result.

The extended math commands allow 26 special Ultra 4 variables to handle numbers ranging from negative 21 million to positive 21 million with up to two decimal places. This allows you to calculate financial transactions without needing the spreadsheet, or to do any other math which is beyond the range of Ultra's normal 0-65535 limits.

One limitation is that the result of a multiplication or division can not exceed 214,748.36, at least if you want an accurate answer. Additions and subtractions work all the way up to the maximum value. Multiplications and divisions are limited because of the need to maintain two decimal place accuracy.

Ed: Because of this "limitation" for multiplication or division I would hesitate to use these extended math commands unless I knew for sure that there was no chance my data could exceed the limitation. Practice safe Hex, create a spreadsheet and do the calculations there.

Here are the ranges. (The second columns have commas to make the numbers more readable, but you can't use commas in your macros. You must write the numbers as they appear in the first columns of each group.)

	Extended Variables		Normal Ultra 4	
Lowest:	-21474836.47	-21,474,836.47	0	0
Highest:	21474836.47	21,474,836.47	65535	65,535

Technical trivia: Ultra variables are two bytes, hence the range from 0 to $FFFF. Extended variables are four bytes, with one bit reserved for the sign (pos or neg), hence the range is -7FFFFFFF to 7FFFFFFF.

Two commands do the majority of the extended variable work: .xMath handles the math operations, and .xStr makes results displayable. All extended variables are identified by a preceding grave accent "`" symbol, as in the following:

```
 < sa-A > : < all :
.xMath "`A = -12345678.90" : //Define a very small number
 > !
```

< X = .AndBits A,B >
Domain: All Modules
Returns the logical AND of the two values specified. If that doesn't make sense, you probably have no use for this command yet. For those eager to learn, an AND compares two numbers bit by bit and sets the corresponding bit in the result byte to true only if both comparison bits are true.

```
A    B  Result
-    -  ------
0    0    0 = False
0    1    0 = False
1    0    0 = False
1    1    1 = True
```

```
<sa-A>:<all :
X = .AndBits 3,18 :          //The answer is 2 which probably doesn't
msg "3 AND 18 = " + str$ X : //make much sense to you. See Note below.
#Key2Stop :
>!
```

Note: < .AndBits > is a binary (base 2) operation and you are feeding it decimal numbers which makes it hard to visualize just what is going on. Your best bet is if you have a calculator that can convert between the bases. If not, you will have to do it manually. For this example we have:

```
     7   6   5   4  3  2  1  0    Bit positions
    128  64  32  16 8  4  2  1    Binary bit values by position
    ---------------------------
 3 = 0   0   0   0  0  0  1  1     3 =  2 + 1
18 = 0   0   0   1  0  0  1  0    18 = 16 + 2
```

```
The only place we have a 1 in both numbers is in bit 1 which has a
value of 2 so the "answer" is 2.

A common use of this command is to detect if a bit (flag) is set or
not. This allows up to eight flags in a single byte (variable), which
is quite a saving if you are running short of variables.

Setting a flag is somewhat cumbersome because there is no .OrBits dot
command for UM. To set a flag you add up all the values for those
flags not being set i.e., to set bit 4 you would add 128 + 64 + 32 + 8
+ 4 + 2 + 1 = 239. A quicker way is to subtract 16 (value of the flag
to be set) from 255 (value of a byte if all bits = 1 i.e., 255 - 16 =
239. Such a value is commonly called a "mask."
```

```
A = 239 :                    //Bit 4 = 0 in mask
B = peek Memloc :            //Read byte from memory
C = .AndBits A,B :           //Force bit 4 to zero
C = C + 16                   //Set it
poke MemLoc,C :              //Store flags
```

Point: All references to variable C above *could* be changed to B and the result would be the same. Different variables were used so you could copy the above to the Try file and see results at every step of the way. Change Memloc to $800 for your practice runs.

< .GetBlock Ptr,Dest,Flag >
Domain: All Modules
This command allows you to grab the desktop memory block specified by Ptr. The block is moved to the Dest address. Flag determines the memory banks involved. If Flag is 1, the block will be placed

in main memory, and if Flag is 2, the block will be placed in auxiliary memory. **Ed:** This was a tough one to find an example that I understood. Thanks to Will Nelken for sending me the source to his macro to print on two sides of the page I finally figured it out. <.GetBlock> always reads from aux memory.

According to my friend Gary Welsh, "Stuff stored in auxmem is in a 'strange' format."

A pointer points to a "block." The block says how large the block is. The minimum block size is a function of how much memory the compter has and the contents of the "record," which is two bytes of info plus data. In the case of text it is the class 0 string (P-string).

So .GetBlock reads *class 0 strings* from aux memory to the destination which is always in main memory. See the file Mac.DoubleSide for another example of <.GetBlock>.

Z is set to the size of the block. It's up to you, the programmer, to make sure the Ptr is valid and the Dest is a safe place to load a block.

This macro just replicates the <cell> command, but it shows how <.GetPtr> and <.GetBlock> can be used. Be sure to copy the line that begins, "test line for" along with sa-A to the Try file.

test line for .GetBlock example -> <- Tabs Key to continue

```
<sa-A>:<all :
oa-1 :                      //Top of file
oa-F :                      //Find
rtn :                       //Text
oa-Y :                      //Wipe out any previous find stuff
print "test line" :         //Text to find
rtn :                       //Go do it

//Assume we found test line. No checking done, which isn't
//smart in the real world.
esc :

        //A points to line data held elsewhere in memory
A = PeekWord $B0            //Check WP line
P = .GetPtr A               //Grab memory pointer
.Cls 0 :
$1 = .HexWord A :
$2 = .HexWord P :
msgxy 0,10 :                //.WriteStr better. Need msgxy examples
msg "A = $" + $1 + "  P = $" + $2 :
if P > $CFFF then :
  msgxy 0,128 :             //Not text, maybe a carriage return
  msg "Non-text line. Key"  // or a printer option,
  #Key2Stop :               //So stop right here
endif :

// The line, with a two byte prefix code, is copied from where P
// points to $BB00. Z returns with the length of the string so
// by subtracting two and putting it into $BB01 the line is
// converted to a P-string in main memory i.e., something that
//.PeekString understands.

A = $BB00                   //IO Buffer for work area
.GetBlock P,A,1             //1 means normal (to main memory) GetBlock
Z = Z - 2                   //Length of text is block length minus 2
```

```
poke $BB01,Z                  //Set the string length
$1 = .PeekStr $BB01           //Grab the wp line

// Without this next line, <msg> would lock up trying to display
// the control characters which represent tabs, underlines, etc.

$2 = .subchar $1,0,31,#' ' ://replace all codes with spaces
msgxy 0,12 :
msg $2  :                     //display the line as a message
msgxy 0,128 :                 //Move the message back to default
                              //location
msg ' Key ' :
A = key :
| : esc :
>!
```

< Ptr = .GetPtr Adr >
Domain: All Modules
Use this command to set a memory pointer from an open AppleWorks file. Adr specifies an auxiliary memory address where the pointer is stored. This command could also be called .PeekAuxWord, since that′s what it does.

See .GetBlock above for an example.

< $1 = .Hex X >
Domain: All Modules
Convert a one-byte number to a hex string:

```
<sa-A>:<all :
$1 = .Hex 10 :
msg "  You're Canadian, " + $1 + "?  Lame joke by Randy :-)  Key " :
#Key2Stop :
>!
```

< X = .HexStr ″$A″ >
Domain: All Modules
Convert a hex string into a variable. Works with or without the leading ″$″.

```
<sa-A>:<all :
.Cls 0 :
M = .HexStr "ABCD" :
N = .HexStr "$ABCD" :
O = .HexStr "FFFG" :
P = .HexStr "FFF0" :
Q = .HexStr "FFF F" :

$0 = %J% + 'Note that the $ character is optional' + %K% :
.WriteStr 0,8,$0 :
.WriteStr 1,10,"ABCD = " + str$ M :
.WriteStr 0,11,"$ABCD = " + str$ N :
$0 = %J% + "Out of range characters (not 0-9, A-F), are taken as zero"
+ %K% :

.WriteStr 0,13,$0 :
.WriteStr 1,15,"FFFG = " + str$ O :
.WriteStr 1,16,"FFF0 = " + str$ P :
```

```
$0 = %J% + 'A space ends a hex number' + %K% :
.WriteStr 1,18,%J% + 'A space ends a hex number' + %K% :
.WriteStr 1,20,"FFF F = " + str$ Q :

msg 'Key to Continue' :
A = Key :
| : esc :
>!
```

< $1 = .HexWord X >
Domain: All Modules
Convert a decimal number with values between 0 to 65535 to a hex string. The result is always four characters (0000 to FFFF).

```
<sa-A>:<all :
$1 = .HexWord 61453 :
$2 = .HexWord 48879 :
.Cls 0 :
msg "I'd like some " + $1 + ". Preferably " + $2 + "  " + %J% + "Key"
+ %K% :
#Key2Stop :
>!
```

< X = .Mod A,B >
Domain: All Modules
This command returns the remainder of A divided by B.

```
<sa-A>:<all :
oa-Up :                     //Cursor to top of screen
J = 3 :                     //Start writing at line 3
.Cls 1 :
I = 20 :
A = 4001 :                  //You can play around with this number
Begin :
  X = .mod I,6 :
  $0 = str$ I + "/6 leaves a remainder of " + str$ X :
  .WriteStr 0,J,$0 :
  J = J + 1 :
  if J > 20 then :
     msg 'Key' :
     B = key :
     .Cls 1 :
     J = 3 :
     .Cls 1 :               //Reblank screen
  endif :
  K = I + A :
  if K < I or K > 65500 then :
     msg 'Key' :
     B = key :
     | : esc :
     endmacro :
  endif :
  I = K :
Rpt :
>!
```

< .PutBlock Ptr,Adr,Size,Flag >

Domain: All Modules
This command allows you to store a block on the desktop using the memory pointer specified by Ptr. Use 0 for Ptr if you want to allocate a new block. The block starts at Adr and is Size bytes long. Flag determines the memory banks involved. If Flag is 1, the block will be stored from main memory, and if Flag is 2, the block will be stored from auxiliary memory.

< .PutBlock > always writes to aux memory. It reads and writes *class 0 strings* from either main or aux memory to the destination block which is always in aux memory.

Z is set to the new pointer value. If 0 is returned, there was insufficient desktop memory to store the block.

Some users have complained that AppleWorks 4.0 only allows you to enter 22 characters in the pathname to the spelling dictionary. There are two good reasons for this: (1) it's cosmetically correct on the Standard Settings screen, and (2) locating the dictionaries in the MAIN directory of a disk greatly increases spell-checking efficiency.

However, if you still want to use a longer pathname for the dictionaries, it is very easy to set the dictionary path to anything you want, up to 48 characters. Simply incorporate this macro into your UltraMacros startup. If you use the default (built-in, "player") macro set, incorporate it into BA-K in SEG.AX so that it is < called > upon startup.

The "/" at the end of the pathname is essential. This will not save the setting to disk, but as long as this macro runs every time you start up, the spelling dictionaries will be found at that path whenever you spell check.

```
<sa-A>:<all :
$1="/ram/appleworks/dictionary/":     //Long path!
P = PeekWord $0AAD :                  //Ptr to dict
poke $800,0 :
.pokestr $1,$801:                     //Temp path
.PutBlock P,$800,50,1:                //Store it
msg "The dictionaries are located at " + $1 :
>!
```

< .PutPtr Adr, Ptr >
Domain: All Modules
Use this command to store a memory pointer in an open AppleWorks file, modifying a line, record or row. Adr specifies the auxiliary memory address where Ptr is stored. This command could also be called .pokeauxword.

< .RelBlock Ptr >
Domain: All Modules
This command releases a memory block back to the desktop. Death will result if Ptr is not a valid memory pointer.

I did not find a meaningful example for this command. Since the only command I can find that allocates aux memory (aka desktop memory), is < .PutBlock > , I assume that you need to save the Z output from that command in order to release the memory block here.

This command requires knowledge of AppleWorks internals that are reserved for the true guru.

Debug 2.6 and Extended Variables
oa-Ctrl-X brings up the Debug main screen. On the IIgs you can also use oa-Clear.

Press OA-X from the Debug main screen to access extended variables. The current values are displayed, along with the .xFixed and .xIntegers settings. You may edit the variables directly with Debug.

The are 26 extended variables. These variables are denoted by a preceding grave accent (`A to `Z). The extended variables can hold a negative or positive number up to 21 million with up to two decimal places.

NOTE: There is a **MAJOR GOTCHA** with the multiplication and division functions. "Ultra to the max," reported the maximum product or dividend is limited to 429,496.60 i.e., below half a million. Not too good for a system that is supposed to support up to 21 million.

In fact, the example on the /extras disk for the < .xFixed > command does not give the correct results for: 54321 * 240.37.
The correct answer is: 13,057,138.77
The example's answer is: 172,236.88

Bottom line here from my perspective, "Unless you know that all multiplication and division will remain below 400,000 - don't use these commands. Bite the bullet and do the calculations in a throw-away spreadsheet your macro creates on the fly."

< $1 = .xCompare A >
Domain: All Modules
The .xCompare command converts extended variables into strings which allow you to compare extended numbers using < if > or < ifnot > statements and have the results work correctly even when negative numbers and decimals are involved.

```
<sa-A>:<all :

//Copy the following two lines prior to the start and after
//Labels line. Remove the comment slashes
//#xA   = $1
//#xB   = $2

C = peek $10F1 :             //Get state of cursor. Insert/Overstrike
poke $10F1,1 :               //Force overstrike
Begin :
  .Cls 1 :                   //Clear middle 20 lines
  $3 = .GetString "Enter value for `A :","1", 12 : //get `A

  // we allow 12 characters so even -21012345.67 can be entered
  $4 = .GetString "Enter value for `B: ","2", 12 :  //get `B

  .xmath " `A = " + $3       //define A as extended var
  .xmath " `B = " + $4       //define B

  #xA = .xcompare A          //get comparison string A
  #xB = .xcompare B          //and B

// compare the numbers and indicate their relationship
// use a label to represent the strings to make it more readable

  if #xA > #xB then $0 = "`A:"  + $3 + " > `B:" + $4 : endif :
  if #xA < #xB then $0 =  "`A:" + $3 + " < `B:" + $4 : endif :
  if #xA = #xB then $0 =  "`A:" + $3 + " = `B:" + $4 : endif :
  .WriteStr 255,10,$0 :
  msg 'Esc to quit, any other to continue' :
```

```
  A = key :
  msg '' :                    //Erase message
  ifnot A = 27 then :
     Rpt :
  endif :
| : esc :                     //Freshen screen
poke $10F1,C :                //Put cursor to input state
>!
```

< .xFixed 128 >
Domain: All Modules
The .xFixed command sets the number of decimal places used when < .xStr > converts an extended variable to a displayable string. Legal values are 0, 1, 2, and 128. 128 defines appropriate (no trailing zeros). The default value is 2 until .xFixed is used.

```
<sa-A>:<all :
.xFixed 128 :                 //No trailing 0's in displayable result
X = 1000 :                    //Define a regular Ultra variable
.xMath "`Z = 240.37 * X" :  //Multiply Ultra variable by 240.37
$1 = .xStr Z :
msg $1 + " Correct " + %J% + "Key" + %K% :
A = key :
.xFixed 2 :                   //Restore the default setting
$1 = .xStr Z :
msg $1 + " Correct " + %J% + "Key" + %K% :
A = key :
X = 10000 :
.xMath "`Z = 240.37 * X" :  //Multiply Ultra variable by 240.37
$1 = .xStr Z :
msg $1 + " Incorrect " + %J% + "Key" + %K% :
A = key :
.xFixed 2 :                   //Restore the default setting
$1 = .xStr Z :
msg $1 + " Incorrect " + %J% + "Key" + %K% :
A = key :
| : esc :
>!
```

<.xIntegers TruthValue>
Domain: All Modules
The .xIntegers command determines how normal Ultra variables are used by the .xMath command. The default setting for "TruthValue" is 1 (#True) which means they are treated as integers, or whole numbers. If .xIntegers #False is used to change the setting, the Ultra variables are treated as hundredths. To simplify understanding this, think of integers as dollars and hundredths as pennies.

```
// Treat Ultra variables as integers (dollars) using .xIntegers
// default setting of #True

// Treat Ultra variables as hundredths (pennies) using .xIntegers
// non default setting of #False

// The /extras example of this had a non-extended variable of A(1)
// I've changed that to A to show that normal and extended are
// distinct even though they look alike and you have to keep
// them straight.
```

```
<sa-A>:<all:
A = 99 :                     //define good old Ultra variable
.xMath " `A = 1 + A" :       //`A is now 100

//A here is really `A but cannot be specified that way. Go figure.
$1 = .xStr A :

$1 = ".xIntegers = #True  when .xMath `A = 1 + A equals " + $1 :
.Cls 1 :
.WriteStr 0,9,"A = 99 :"
.WriteStr 0,10,$1 :          //display the result
// treat Ultra variables as hundredths (pennies)
.xIntegers #False :          //treat Ultra vars as hundredths
.xMath " `A = 1 + A" :       //`A is now 1.99
$1 = .xStr A :               //Really `A
$1 = ".xIntegers = #False when .xMath `A = 1 + A equals " + $1 :
.WriteStr 0,11,$1 :          //display the result
.xIntegers #True :           //restore default
msg 'Hit a key when done looking at the above' :
B = key :
| : esc :
>!
```

< .xMath " `A = B + C * 3.25 " >
Domain: All Modules
The .xMath command handles all extended variable definitions by evaluating the expression contained in the string which follows the command. The expression must start with a definition, such as " `A = ", and then you may add, subtract, divide or multiply extended variables, regular Ultra variables, or literal values. Only one definition may be entered for each .xMath command.

```
<sa-A>:<all:
C(2) = 12 :                  //define a regular Ultra variable
.xMath " `C = 10.3 * C(2) ": //Multiply 10.3 * 12

$1 = "10.3 * 12 = " + .xstr C :
$1 = $1 + "  " + %J% + " Key " :
msg $1 :
#Key2Stop :
>!
```

< $1 = .xStr A >
Domain: All Modules
The .xStr command is the extended variable equivalent to the < str$ > command used with normal variables. It converts an extended variable to a string which can then be displayed with < msg > or < print > .

The .xFixed setting determines how many decimal places, if any, are shown.

```
<sa-A>:<all :
.xFixed 128 :                //no trailing 0's in displayable result
X = 54321 :                  //define a regular Ultra variable
.xMath "`Z = 2.5 * X" :      //multiply Ultra variable by 2.5
$1 = "`Z = 2.5 * X = " + .xstr Z : //convert and display
.xFixed 2 :                  //restore default setting
msg $1 + "  " + %J% + " Key " :
#Key2Stop :
>!
```

File	Command Index	Page	Line #
CH03.TOC	&	46	513
CH03.TOC	()	62	1565
CH03.TOC	adb	43	332
CH04.TOC	.AddMany	119	1981
CH04.TOC	.AddMenu	120	2004
CH03.TOC	Ahead	47	598
CH03.TOC	all	43	329
CH03.TOC	And	82	2923
CH04.TOC	.AndBits	145	3613
CH03.TOC	aol	43	331
CH03.TOC	Asc	62	1587
CH04.TOC	.AskYN	90	128
CH03.TOC	asp	43	333
CH03.TOC	asr	43	334
CH03.TOC	awp	43	330
CH04.TOC	.AwPath	91	199
CH03.TOC	ba-Ctrl-C	41	197
CH03.TOC	ba-esc	50	755
CH03.TOC	ba-right	41	195
CH03.TOC	Back	48	638
CH04.TOC	.Beep	92	267
CH03.TOC	Begin	58	1321
CH03.TOC	Bell	56	1185
CH04.TOC	.Box	92	213
CH04.TOC	.CacheList	94	339
CH03.TOC	Call	79	2718
CH04.TOC	.Caps	93	296
CH04.TOC	.Case	93	317
CH04.TOC	.CatName	116	1774
CH03.TOC	Cell	54	1035
CH04.TOC	.CellID	118	1908
CH04.TOC	.Choose	94	355
CH03.TOC	Chr$	63	1603
CH03.TOC	Clear	63	1634
CH04.TOC	.Cls	120	2025
CH04.TOC	.Column	117	1821
CH04.TOC	.ColWidth	117	1835
CH03.TOC	Date	48	645
CH03.TOC	Date2	48	650
CH03.TOC	Debug	56	1196
CH03.TOC	dec	46	520
CH03.TOC	del	41	155
CH04.TOC	.DeskCount	95	409
CH03.TOC	Disk	53	967
CH03.TOC	Display	63	1656
CH04.TOC	.DoMenu	120	2050
CH03.TOC	down	41	184
CH04.TOC	.Dropdir	95	438
CH03.TOC	Else	83	2981
CH03.TOC	elseoff	46	510
CH04.TOC	.Embedded	95	466
CH03.TOC	end	39	51
CH03.TOC	Endif	85	3086
CH03.TOC	Endmacro	60	1452
CH04.TOC	.Eof	96	510
CH03.TOC	esc	41	156

File	Command Index	Page	Line #
CH03.TOC	Exit	60	1428
CH04.TOC	.ExtKB	142	3400
CH04.TOC	.FCard	121	2081
CH04.TOC	.FDate	97	542
CH04.TOC	.FDate2	97	583
CH04.TOC	.FileCard	122	2127
CH03.TOC	Find	48	664
CH04.TOC	.FindPO	98	614
CH03.TOC	First	52	900
CH03.TOC	For	85	3113
CH04.TOC	.GetBlock	146	3675
CH04.TOC	.GetCat	115	1733
CH04.TOC	.GetCell	117	1864
CH04.TOC	.GetColor	143	3436
CH04.TOC	.GetFpath	98	639
CH04.TOC	.GetInput	122	2173
CH04.TOC	.GetNames	114	1622
CH04.TOC	.GetPtr	148	3759
CH04.TOC	.GetRec	114	1665
CH03.TOC	GetStr	64	1720
CH04.TOC	.GetString	123	2250
CH04.TOC	.GetValue	124	2290
CH03.TOC	go	41	159
CH03.TOC	GoTo	65	1735
CH04.TOC	.Hex	148	3767
CH04.TOC	.HexStr	148	3779
CH04.TOC	.HexWord	149	3813
CH03.TOC	Hilight	65	1785
CH04.TOC	.ID	98	644
CH03.TOC	If	81	2844
CH03.TOC	ifkey	46	540
CH03.TOC	Ifnot	81	2845
CH03.TOC	inc	46	520
CH03.TOC	Input	57	1257
CH03.TOC	Insert	52	915
CH03.TOC	Jsr	86	3181
CH03.TOC	Key	58	1292
CH03.TOC	Keyto	66	1831
CH03.TOC	Labels	39	38
CH03.TOC	Last	52	908
CH04.TOC	.LastCol	119	1944
CH04.TOC	.LastRow	119	1964
CH03.TOC	Launch	78	2660
CH03.TOC	LC	52	895
CH03.TOC	left	41	181
CH03.TOC	Left	67	1866
CH03.TOC	Len	67	1881
CH04.TOC	.Line	99	666
CH03.TOC	Link	80	2790
CH04.TOC	.List	125	2373
CH04.TOC	.LoadVar	128	2547
CH04.TOC	.Lower	99	692
CH04.TOC	.MacroNames	126	2433
CH04.TOC	.MakeMenu	130	2647
CH04.TOC	.MenuBar	132	2778
CH04.TOC	.MenuBar2	133	2852
CH04.TOC	.MenuItem	127	2460

File	Command Index	Page	Line #
CH03.TOC	Mid	67	1901
CH04.TOC	.Mod	149	3829
CH04.TOC	.Msay	99	716
CH03.TOC	Msg	68	1931
CH03.TOC	Msgxy	69	1997
CH04.TOC	.NewFile	100	740
CH03.TOC	Next	86	3125
CH03.TOC	NoSleep	56	1190
CH03.TOC	oa-!	52	916
CH03.TOC	oa-#	50	769
CH03.TOC	oa-&	53	968
CH03.TOC	oa-*	53	999
CH03.TOC	oa-,	52	901
CH03.TOC	oa--	54	1036
CH03.TOC	oa-.	52	909
CH03.TOC	oa-0	51	856
CH03.TOC	oa-1	41	193
CH03.TOC	oa-:	52	891
CH03.TOC	oa-;	52	896
CH03.TOC	oa-<	52	902
CH03.TOC	oa->	52	911
CH03.TOC	oa->	55	1119
CH03.TOC	oa-@	53	962
CH03.TOC	oa-Clear	63	1652
CH03.TOC	oa-Ctrl-@	51	885
CH03.TOC	oa-Ctrl-G	56	1186
CH03.TOC	oa-Ctrl-N	56	1191
CH03.TOC	oa-Ctrl-X	56	1197
CH03.TOC	oa-X	51	850
CH03.TOC	oa-^	52	924
CH03.TOC	Onerr	69	2012
CH03.TOC	Onerr	69	2024
CH03.TOC	Onerr	69	2035
CH03.TOC	Onerr	69	2047
CH03.TOC	Onerr	69	2058
CH03.TOC	Onerr	70	2067
CH04.TOC	.OnGosub	127	2495
CH04.TOC	.OnGoto	129	2588
CH04.TOC	.Online	101	793
CH03.TOC	Or	82	2924
CH03.TOC	Path	53	998
CH03.TOC	Peek	87	3254
CH04.TOC	.Peekstr	101	817
CH04.TOC	.PeekVar	102	856
CH03.TOC	PeekWord	88	3259
CH04.TOC	.PeekWordZP	103	957
CH04.TOC	.Pick	133	2876
CH03.TOC	Poke	87	3206
CH04.TOC	.PokeStr	102	885
CH04.TOC	.PokeVar	103	922
CH03.TOC	PokeWord	87	3232
CH04.TOC	.PokeWordZP	103	971
CH04.TOC	.PokeZP	104	978
CH04.TOC	.Pop	104	985
CH04.TOC	.PopMenu	135	2949
CH03.TOC	Posn	71	2172
CH03.TOC	Pr#	72	2207

File	Command Index	Page	Line #
CH03.TOC	Print	72	2243
CH04.TOC	.PutBlock	149	3864
CH04.TOC	.PutPtr	150	3907
CH04.TOC	.Qmenu	137	3098
CH03.TOC	Read	52	923
CH03.TOC	Recall	55	1115
CH04.TOC	.RelBlock	150	3914
CH03.TOC	rem	46	516
CH03.TOC	RestScr	57	1251
CH03.TOC	right	41	182
CH03.TOC	Right	75	2422
CH04.TOC	.Rightmost	105	1041
CH03.TOC	Rpt	59	1350
CH03.TOC	rtn	41	157
CH03.TOC	sa-B	41	194
CH03.TOC	sa-Ctrl-C	41	196
CH03.TOC	sa-esc	49	754
CH03.TOC	SaveScr	57	1214
CH04.TOC	.SaveVar	138	3127
CH04.TOC	.Say	139	3234
CH03.TOC	Screen	75	2434
CH04.TOC	.Search	105	1072
CH04.TOC	.SetCat	115	1738
CH04.TOC	.SetCell	118	1887
CH04.TOC	.SetCol	106	1101
CH04.TOC	.SetColor	143	3463
CH04.TOC	.SetDisk	106	1134
CH04.TOC	.SetFpath	106	1156
CH04.TOC	.SetNames	114	1641
CH04.TOC	.SetRec	115	1696
CH04.TOC	.Sort	107	1215
CH04.TOC	.SpaceBar	140	3248
CH03.TOC	spc	41	185
CH04.TOC	.Speed	108	1264
CH03.TOC	start	39	47
CH03.TOC	Step	86	3147
CH03.TOC	Stop	61	1469
CH03.TOC	Store	55	1126
CH03.TOC	Str$	75	2466
CH04.TOC	.StripChar	140	3260
CH04.TOC	.SubChar	109	1295
CH04.TOC	.SubString	110	1362
CH03.TOC	tab	41	180
CH03.TOC	Then	83	2967
CH04.TOC	.Therm	140	3292
CH03.TOC	Time	48	655
CH03.TOC	Time24	48	659
CH04.TOC	.TimeGS	144	3519
CH04.TOC	.TitleBox	110	1391
CH04.TOC	.TOinMem	111	1436
CH03.TOC	UC	52	890
CH04.TOC	.UnCache	111	1462
CH03.TOC	Unlink	81	2829
CH03.TOC	up	41	183
CH04.TOC	.Upper	112	1518
CH03.TOC	Val	76	2489
CH04.TOC	.Vline	112	1530

File	Command Index	Page	Line #
CH03.TOC	Wait	76	2523
CH03.TOC	Wake	77	2568
CH04.TOC	.WeekDay	113	1568
CH04.TOC	.Writestr	142	3367
CH04.TOC	.xCompare	151	3955
CH04.TOC	.xFixed	152	4002
CH04.TOC	.xIntegers	152	4034
CH04.TOC	.xMath	153	4079
CH04.TOC	.xStr	153	4099
CH04.TOC	.ZapChar	113	1592
CH03.TOC	Zoom	53	961
CH04.TOC	.ZoomIn	113	1611
CH03.TOC	|	64	1696

File General Index Page

File	General Index	Page
CH03.TOC	<Clear NUM>	63
CH04.TOC	<.Cls NUM>	120
CH04.TOC	<X = .Column STRING>	117
CH04.TOC	<X = .ColWidth STRING>	117
CH01.TOC	Compile current WP file macro	12
CH02.TOC	Compiler Labels	21
CH02.TOC	Create a Task File	18
CH01.TOC	Creating a Macro File	10
CH01.TOC	Creating Custom Macros	10
CH01.TOC	Creating your very own UltraMacro	11
CH02.TOC	Cursor blink A	19
CH02.TOC	Cursor blink B	19
CH04.TOC	Data Base Dot Commands	114
CH03.TOC	<Date> sa-'	48
CH03.TOC	<Date2> sa-"	48
CH02.TOC	Debug	25
CH03.TOC	<Debug>	56
CH03.TOC	<Debug>	63
CH04.TOC	Debug 2.6 and Extended Variables	150
CH01.TOC	debugger	11
CH02.TOC	debugger	24
CH03.TOC	<dec>	46
CH04.TOC	Default Dot Commands	90
CH01.TOC	Default Macros	10
CH01.TOC	default set	14
CH02.TOC	Defining Numeric variables	31
CH02.TOC	Defining String Variables	33
CH03.TOC	<del> Delete key	41
CH04.TOC	<.DeskCount X>	95
CH02.TOC	Desktop Utils	20
CH03.TOC	Dire Warning:	54
CH01.TOC	disable/enable one key macros:	7
CH03.TOC	<Disk>	53
CH03.TOC	<Display NUM>	63
CH02.TOC	Display Items	25
CH04.TOC	<.DoMenu X>	120
CH04.TOC	Dot Command Reference	90
CH03.TOC	<down> Down-Arrow key	41
CH04.TOC	<.Dropdir>	95
CH03.TOC	Dropped UM 3.x Commands	46
CH03.TOC	<Else>	83
CH03.TOC	<elseoff>	46
CH04.TOC	<$1 = .Embedded>	95
CH03.TOC	<end>	39
CH02.TOC	<end>	22
CH03.TOC	<Endif>	85
CH03.TOC	<Endmacro>	60
CH04.TOC	<X = .Eof>	96
CH03.TOC	<esc> Escape key	41
CH01.TOC	Escape Map	8
CH03.TOC	<Exit>	60
CH01.TOC	exit/rpt trick	13
CH03.TOC	exit/rpt trick	59
CH02.TOC	Extended Numeric Variables	31
CH04.TOC	External Dot Commands	89
CH04.TOC	<.ExtKB>	142
CH04.TOC	<.FCard NUM,STRING,TYPE>	121

File	General Index	Page
CH04.TOC	<$1 = .FDate Format>	97
CH04.TOC	<$1 = .FDate2 Month,Day,Year,Format>	97
CH04.TOC	<.FileCard STRING, NUM>	122
CH03.TOC	<Find> sa-Return	48
CH04.TOC	<.FindPO>	98
CH03.TOC	<First>	52
CH03.TOC	First A Lot Of Background	39
CH03.TOC	<For VAR = NUM to NUM>	85
CH03.TOC	For Advanced UltraMacros Users Only	86
CH03.TOC	For-Next LOOPS	85
CH04.TOC	<.GetBlock Ptr,Dest,Flag>	146
CH04.TOC	<$1 = .GetCat Category, Record>	115
CH04.TOC	<$1 = .GetCell Column, Row, Format>	117
CH04.TOC	<C = .GetColor Y>	143
CH04.TOC	<$1 = .GetFpath>	98
CH04.TOC	<$1 = .GetInput X,Y,$2,$3,L>	122
CH04.TOC	<.GetNames Category, FirstStr, Number>	114
CH04.TOC	<Ptr = .GetPtr Adr>	148
CH04.TOC	<.GetRec Category, Record, FirstStr, Number>	114
CH03.TOC	<$1 = GetStr NUM>	64
CH04.TOC	<$1 = .GetString "Prompt","Default",MaxLen>	123
CH04.TOC	<X = .GetValue $1,L,H,D>	124
CH03.TOC	<go> Return key	41
CH03.TOC	<GoTo MACRO>	65
CH04.TOC	<$1 = .Hex X>	148
CH04.TOC	<X = .HexStr "$A">	148
CH04.TOC	<$1 = .HexWord X>	149
CH03.TOC	<Hilight L,T,R,B>	65
CH04.TOC	<X = .ID>	98
CH03.TOC	<If>	81
CH03.TOC	If-Then-Else Logic	81
CH03.TOC	<ifkey>	46
CH03.TOC	<Ifnot>	81
CH01.TOC	IIgs One-Key Macros in AppleWorks 3	7
CH01.TOC	IIgs One-Key Macros in AppleWorks 4 and 5.	7
CH04.TOC	IIgs only commands	142
CH03.TOC	<inc>	46
CH03.TOC	<Input>	57
CH03.TOC	<Insert>	52
CH01.TOC	Installing UltraMacros	6
CH01.TOC	invisible task file.	14
CH03.TOC	<Jsr>	86
CH03.TOC	<Key>	58
CH02.TOC	Key click on: Yes	19
CH03.TOC	<Keyto NUM>	66
CH03.TOC	Label Definitions	39
CH03.TOC	<Labels>	39
CH02.TOC	Labels must be on one line only.	21
CH03.TOC	<Last>	52
CH04.TOC	<$1 = .LastCol>	119
CH04.TOC	<$1 = .LastRow>	119
CH03.TOC	<Launch STRING VAR>	78
CH01.TOC	<Launch STRING VAR>	14
CH02.TOC	Launch a Task File	18
CH03.TOC	<LC>	52
CH03.TOC	<left> Left-Arrow key	41
CH03.TOC	<Left STRING VAR,NUM>	67

File	General Index	Page
CH03.TOC	<oa-;>	52
CH03.TOC	<oa-<>	52
CH03.TOC	<oa-<>	55
CH03.TOC	<oa->>	52
CH03.TOC	<oa->>	55
CH03.TOC	<oa-@>	53
CH02.TOC	oa-B Break Options	29
CH03.TOC	<oa-Clear> IIgs	63
CH03.TOC	<oa-Ctrl-@>	51
CH03.TOC	<oa-Ctrl-G>	56
CH03.TOC	<oa-Ctrl-N>	56
CH03.TOC	<oa-Ctrl-X>	56
CH03.TOC	<oa-Ctrl-X>	63
CH02.TOC	oa-D show Dot commands	26
CH02.TOC	oa-M show Macro names	26
CH02.TOC	oa-N End Macro	28
CH02.TOC	oa-P display Peek values	27
CH02.TOC	oa-S Options	30
CH02.TOC	oa-T set Trace options	28
CH02.TOC	oa-V View Screen	29
CH02.TOC	oa-W Walk Through	30
CH03.TOC	<oa-X>	51
CH02.TOC	oa-X Extended Numeric Variables	28
CH03.TOC	<oa-^>	52
CH03.TOC	<Onerr OPTION>	69
CH02.TOC	<Onerr OPTION>	25
CH03.TOC	<Onerr stop>	69
CH03.TOC	<Onerr endmacro>	69
CH03.TOC	<Onerr exit>	69
CH03.TOC	<Onerr off>	69
CH03.TOC	<Onerr goto>	70
CH04.TOC	<.OnGosub Val,"abcdDefghijk">	127
CH04.TOC	<.OnGoto Val,"abcdDefghijk">	129
CH04.TOC	<.Online STRING>	101
CH03.TOC	Open-Apple Commands	50
CH03.TOC	Optional Comments	39
CH03.TOC	Optional Comments	42
CH03.TOC	<Or>	82
CH03.TOC	Parameters for UltraMacros Tokens	61
CH03.TOC	<Path>	53
CH03.TOC	<Peek>	87
CH04.TOC	<$1 = .Peekstr Address>	101
CH04.TOC	<.PeekVar Address,Start,Count,Size>	102
CH03.TOC	<PeekWord>	88
CH04.TOC	<X = .PeekWordZP Address>	103
CH04.TOC	<$90 = .Pick X,Y,W,L,Start,End,"Title">	133
CH03.TOC	<Poke>	87
CH04.TOC	<.PokeStr STRING, Address>	102
CH04.TOC	<.PokeVar Address,Start,Count,Size>	103
CH03.TOC	<PokeWord>	87
CH04.TOC	<.PokeWordZP Address,Value>	103
CH04.TOC	<.PokeZP Address, Value>	104
CH04.TOC	<.Pop X>	104
CH04.TOC	<.PopMenu>	135
CH03.TOC	<Posn VAR1,VAR2>	71
CH03.TOC	<Pr# NUM EXP>	72
CH03.TOC	<Print NUM,STRING,ALMOST ANYTHING>	72