Macintosh®

Virtual User General Reference

Beta Draft

August 13, 1990

Contents

Welcome to Virtual User

It is time to test. You have installed the latest system on each of the hardware configurations in your lab. Now you must take an important application through its paces on each CPU. The task before you is quite clear. You've done it many times before. There are many other tests to run after this one, and still more to design before the test coverage will be complete. Surrounded on all sides by computers, you might wonder if all this technology could be applied towards freeing you from the chore of repetitive testing.

Virtual User™ (VU) is designed to do just that. VU is a system in which tests are described in a high-level scripting language and executed on a target system remotely over AppleTalk. By specifying a test in a scripting language, VU overcomes many of the limitations of early automation schemes. VU scripts are editable with any text editor, unlike keyboard macros or literal recording mechanisms.

In the VU Scripting Language, the on-screen features you wish to have VU work with are described in abstract terms. VU locates the desired feature at runtime based on your description, allowing it to execute the same command on different screen configurations and on different revisions of the software under test. This represents a significant improvement over literal recording tools, for which you must re-record portions of a test as the layout of windows and their contents change from monitor to monitor and version to version .

The VU Scripting Language is a procedural language in which automated tests can be written with internal logic and

control flow that is inaccessible from macro or recording-driven systems. The language combines traditional programming language features with a set of commands that direct the interpreter to manipulate the user interface of a number of Macintoshes, each called a target, over an AppleTalk network. By separating the target software from the great bulk of the test driving software, intrusion upon the target system is minimized to a small piece of resident code called Agent VU, which patches no traps.

The core of the VU environment is available for use now. This release is in the form of an MPW shell tool with Commando support. It can run multiple scripts on multiple targets in one test session. In the

interest of getting the tool out to you, we are delivering a subset of VU's ultimate functionality. It is capable of manipulating windows, menus, menuItems (including hierarchical and scrolling items) and the standard controls (buttons, checkBoxes, radioButtons, and scrollBars). It also provides you with a means of moving the mouse to a particular location, clicking, double clicking and typing keystrokes. To help compensate for the features it lacks in this release, VU can control a sister application called VU Recorder under script control. VU Recorder is a literal recording tool. It is the predecessor to the script-driven VU.

Future releases of VU will extend the utility and sophistication of the system in many ways. VU will become an application environment with an interactive target picker and extensive features to monitor tests and debug scripts. Our underlying metaphor of actors reading scripts will emerge, with the test engineer taking the role of playwright and director. Currently, in VU , each actor is assigned a script to read and most often these actors have a target machine to test. Actors can send messages to one another, and thus synchronize and influence each other's efforts. VU has been designed to be a cooperative tool. In the future, external programs will be able to interact with VU actors, thereby becoming VU actors themselves. These external actors might be test result validation tools such as screen comparison tools. Or, an external tool may play the role of a test generator, feeding scripts to VU.

Chapter 1 Introduction

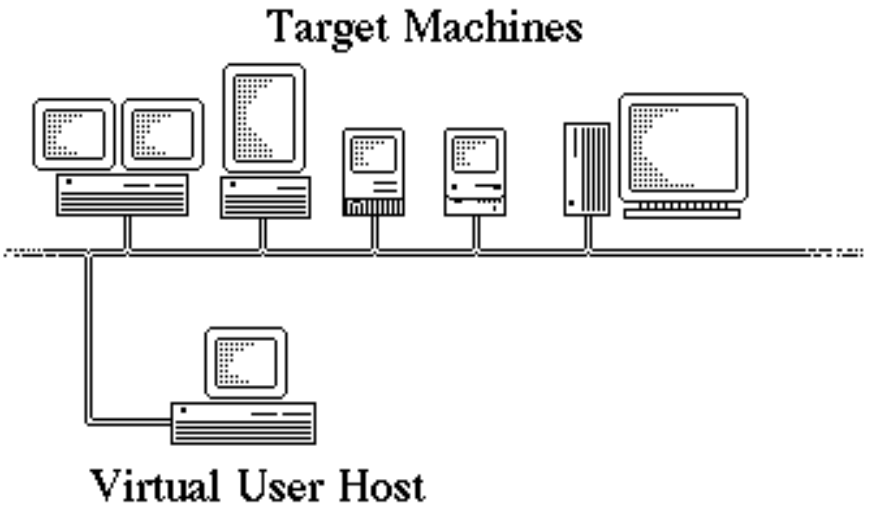
This chapter gives an overview of the VU architecture. It also discusses the value of VU in combatting what has been called the "Combinatorial Explosion" of possible hardware and software configurations.

The VU Architecture

Virtual User can emulate a number of users operating their respective Macintoshes each in a different way. This means that VU can operate against multiple targets with each target being assigned a different script. It is recommended that, when staging a test using Virtual User, a private network is used.

One machine on the network is configured as the VU host (a.k.a "VU Machine") and, for now, all others will be test machines hereafter referred to as target machines (a.k.a. "targets").

Figure 1-1 The Basic VU Hardware Configuration


Target Machines
Virtual User Host

The VU host currently needs MPW in which VU runs as a tool. In each target machine resides Agent VU, which is loaded as an init.

The overall architecture of VU is based on a theater metaphor. There are actors in the Virtual User system which play different roles and there is a director which stages, oversees, and directs the test execution. A VU actor is any entity that plays a role in the testing process in the Virtual User system. The most prominent of these actors are the ones that play the role of the emulated (virtual) Macintosh users during the test. Other actors which will be supported in the future include the human tester or any external tools which communicate with the director or virtual users.

The Virtual User architecture resembles that of a multi-tasking system. Actors in the VU system are different processes or threads, each of which is governed by the script associated with it. The ability to have multiple actors simultaneously running multiple scripts makes VU a multi-tasking system. The actors are the process handlers, reading and executing scripts in their own runtime world.

Each VU process handler (actor) runs independent of the rest of the actors. One actor cannot inadvertently interfere with another actor. To allow for the coordination of actions between different users in a testing process, the VU architecture provides a message passing scheme for actors. This message passing can be used for various purposes such as the coordination of actions between actors. Or, it could be used to create a client-server relationship between actors where one actor (script) serves (computes) for other client actors (scripts).

Combatting the Combinatorial Explosion

The motivation behind the Virtual User concept arose from observing the compatibility testing process. Every time a single piece in a configuration of software or hardware is modified, the entire configuration must be tested. With the increasing complexity in the configurations of work environments, the problem of testing to maintain quality is growing exponentially. Virtual User helps address this problem while preserving the sanity of the tester.

Once you develop your test suites (in the form of VU scripts and libraries), the same test suite can be run against any number of configurations simultaneously. You can set up a group of test machines consisting of 1) various hardware platforms running a given version of system software or , 2) the same hardware platform running different versions of the system software or, 3) a

combination of both. You can even test against localized international versions of software using a localized version of the test suite developed for the US versions. Clearly, the potential time savings resulting from automating compatibility testing using VU is enormous.

Apart from its value in compatibility testing, VU has other general advantages. VU's virtual users will test for days and nights without breaks or requests for salary increases. VU has also been proven useful in performing reliability testing of hardware platforms in heat laboratories. More importantly, VU scripts can be developed in parallel with the software development from the software specification. This helps you start testing early, catching problems at an early stage in the development cycle.

Chapter 2 Getting Started

This chapter outlines what you need to do to start using VU. Setting up to use Virtual User is quite simple. Two classes of machines must be configured, the target machines and VU host machine.

Hardware Needs

The hardware needed to run VU varies with the extent of testing desired. The greatest benefit comes with running one host with many targets, each being either a different CPU model or configured differently in some way. You can use any Macintosh in Apple's current product line as a VU target machine. Currently, this includes any Macintosh from the Mac Plus to the Macintosh IIfx (and including the Macintosh Portable).

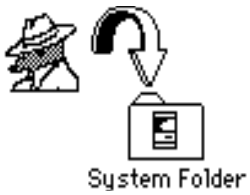
The machine designated as the VU machine must have at least 1MB of memory to run MPW 3.0 or higher.

Beyond that, you simply need to network this host machine with each of the machines you wish to simulate user interaction upon. The physical medium of networking them together isn't important as long as AppleTalk™ is supported. We recommend that you run your tests over a private network

for performance reasons. Obviously, a faster physical medium such as Ethernet cabling will improve your test times.

Target Setup

Figure 2-1 Installing Agent VU



To set up a target for use with VU, you must first copy an Agent VU into each target machine's system folder. When you reboot each machine, the agent will adopt the name entered in the respective machine's Chooser desk accessory. It will then join the parade of icons that starts in the lower left corner of the screen at system startup. The Agent VU learns its name at startup time, so if you want to change its name, you must reboot the machine whose name has changed.

VU has been shown to run successfully on targets running versions of the system as early as System File Version 4.1. It will also run on targets running the very latest versions of the system, including System 7.0.

It is also highly recommended that you turn off key repeat on each target machine using the Keyboard CDEV. If this is not done, characters typed by VU on the target machine may be unintentionally repeated.

Host Setup

We offer two ways to install the host software needed to run VU. The first method involves the installation of VU Extension. VU Extension is a set of small utilities for VU users to enhance their script development environment. VU is presently an MPW tool, but is intended to be an application environment. To make up for the lack of a VU environment, users of VU at Apple have

developed some MPW scripts. Using the first installation scheme will make these features available to you. The second scheme will yield the minimum configuration for running VU.

Of course, there are many ways to configure MPW given your own preferences. These are just two quick schemes we have provided for you.

◆ *Note:* When using a Mac Plus as a host, you must first copy the AppleTalk 52.0 INIT into the system folder of the Mac Plus. This INIT is included in the VU package.

Easy Installation Method

Installation of VU Extension for MPW is extremely simple. All you have to do is copy a file named "UserStartup•VU" into the folder that contains the MPW shell application. Then copy the folder, Virtual User, to anywhere on your hard disk. The Virtual User folder contains the VU tool. Once you have done this, your VU scripting environment is ready to be used next time you launch the MPW Shell. You can launch MPW by double-clicking on the VU tool. This will set your working directory to Virtual User. You should always set your working directory to the folder which contains your VU scripts. If you create another folder for your VU scripts then use the MPW menu item "Set Directory" or the "Directory" command to switch to the correct working directory.

Detailed information on how to use VU Extension is given in Chapter 4, VU Extension.

Minimum Installation Method

If you'd rather not use the extensions provided, you can do a minimum installation as follows:

1. Install MPW 3.0 or higher onto your hard disk
2. Place the VU tool in one of the following two places:
   - within the Tools folder within the MPW folder (i.e. "{MPW}:Tools:")
   - a folder of your own which will be your working directory.
3. To avoid typing the full pathname to your scripts, set the working directory to the directory containing your scripts. If you haven't put the VU tool in the {MPW}Tools folder, the tool should be in this same working directory.

Chapter 3

Running VU as an MPW Tool

VU runs as an MPW tool. This chapter describes the details involved in running VU from MPW. The tool may be launched from the command line or via MPW's Commando facility. Commando allows the user to interactively specify VU's parameters from a dialog.

Using the Command Line

To run a script using the VU tool under MPW, you have to supply some information to VU in the form of command line options. Here we describe the arguments that VU accepts. A simple example of a VU command line follows:

```
VU -a "Actor1" -t "*:TargetMac" -s Script.vu
```

An argument has two parts:

1. argument specifier
2. argument value

The argument specifier is one or more letters preceded by a hyphen. All VU arguments except "-vers", "-dt" , "-c" and "-cs" require a value to follow the specifier .

- **-t** <**target address**> This specifies the target on which the test is to be performed. The target name is specified by giving its network path name, namely, <zone>:<target's Chooser User Name>. If your target is in the same network zone as the VU machine, then the meta-character "*" (asterisk), meaning "this zone"may be used. For example, consider a target having a Chooser User Name of "BOB". If the target is in the same zone as the VU host machine, you could specify the target address as -t "*:BOB". Always enclose the target name within single or double quotes.

- **-s <script file name>** This specifies the name of the script file to be run. Enclose the file name in single or double quotes if the name has any blanks in it. In the absence of this argument, the script will be read from standard input. In this case, you can type the script in any open window, select the whole script and press enter followed by command-enter . It's recommended that you use the extension ".vu" in names of VU script files to make them distinct and easily identifiable.

- **-a <actor name>** This specifies the name of an actor. An actor may be assigned a script and a target via the command line with the "-s" and "-t" options. If you do not specify an actor name from the command line, VU creates an actor anyway to run the script on the specified target. The actor's name is assigned to the target address, i.e. "*:TargetMac". If you haven't specified a target from the command line, then you must specify an actor.

- **-c** This indicates compile only. This is used if you just want to test the syntax of your script, without running it. If you use this option you need not specify the target.

- **-l <log file name>** This specifies the name of the log file. The form of specification is the same as that of the -s option. If you use this option, VU tool will write some important information about the test along with any error/warning messages in the specified file. Tracing also goes to this file.

- **-dt** This indicates that diagnostic trace is to be enabled. This is used only if you have specified the log file. This option is used to trace the execution of the script for reporting bugs and debugging purposes. Every statement will be written into the log file along with any effects due to that statement on the script variables.

- **-o <output file name>** This is used to redirect the output of print and println statements from the standard output to another file. If only one script is being run, this has the same effect as redirection of output using ">" in MPW. This option is of greatest utility when running multiple actors or targets.

- **-m <mouse speed>** This is used to set the speed of mouse movement on the target. The mouse always moves in steps. The speed specifies the number of pixels you want the mouse to move in every step. Hence it should be a positive integer. If you want the mouse to make all movements in just one step, set the speed to 0. The default setting is 50.

- **-k <keystroke rate>** This is used to set the maximum rate of keystrokes on the target in characters per second. The rate must be a positive integer. Note that the rate specified is used as a maximum limit. Due to uncertainty in network transaction timing, VU cannot maintain a constant rate of keystrokes. The default setting is 20.

- **-p; <patience setting>** This is used to set the "patience" of VU for the target. Patience must be a positive integer. VU's actions get slower (increased pauses between actions) as patience increases. The default setting is 1.

- **-vers** This is used to get the version information of VU. The following command line will give you Virtual User version information

  VU -vers

  This option directs VU to tell you the version of Virtual User (the package), VU (the tool), and the version of Agent VU which is compatible with the tool VU. Please use this option to obtain version info when reporting problems. Note this will not tell you the version of the Agent VU you are actually using. It will tell you the version of Agent VU you **should** be using with the VU tool you are running. The log file will tell you the currently executing version of a particular target's Agent VU.

- **-fail <n>** This is used to set the maximum number of command failures (e.g. couldn't drag a window in the specified manner) you want to allow in your script execution. VU will abort if the number of failures exceeds this number. During script development, you might want to set this to 1. The default setting is infinite. That is, VU will not stop until all scripts have been completed or the user aborts execution.

- **-timeout <t>** This is used to set the maximum time for VU to wait for a transaction with Agent VU to complete. If the transaction fails to complete within this time period then the transaction is considered to have failed. The default setting is 20. For busy networks, this setting should be increased.

- **-retries <r>** This sets the maximum number of times VU retries a transaction with the Agent VU. The default setting is 3. For busy networks, this setting should be increased. If the transaction fails to complete despite this number of retries, then VU gives up. In such situations VU suspects target failure and tries to re-acquire the target.

- **-cs** This is used to make VU's string matching case sensitive. By default, any string matching done by VU is not case sensitive.

- **-libs <search path>** This is used to specify the search path for VU to access task libraries. VU always looks for the declared libraries in the current directory first. If not found, it makes use of the search path provided on the command line (if any) and finally it checks the MPW variable "VULibraries" for a search path. By default this variable will be set to your "…:Virtual User:Libraries" folder by the provided "UserStartup•VU" MPW script (if you are using it).

◆ *Note:* Make sure that you have exported the MPW variable "VULibraries" using the MPW command "Export". Otherwise, VU will not be able to access the path defined by this variable. (The "UserStartup•VU" MPW script will do this for you.)

Example:

```
VU      -a1 'Actor1' -s1 'TestScript1.vu' -l1 'TestScript1.log' ∂
        -o1 'TestScript1.out' ∂
        -a2 'Actor2' -s2 'TestScript2.vu' -l2 'TestScript2.log' ∂
        -o2 'TestScript2.out' ∂
        -libs 'HD:Virtual User:MyScripts:Libraries:'
```

In the above command line either or both of the scripts can be using libraries. The same search path is used for all scripts. Note that you are specifying only a search path to locate the libraries. Some of your libraries may reside in this specified directory and some may reside in your current directory and some may even reside in another directory whose path name is specified by setting the MPW shell variable "VULibraries".

Note that to support running multiple actors from one VU interpreter, each of the specifiers (expect "-libs") can be repeated. Repeated specifiers are followed by numbers which tell VU how the different specifiers match up. If you use a specifier in numbered form, then its lower numbered forms must precede its higher numbered forms.

In cases where all of the actors are being run by the same script, it is not necessary to assign the script to each actor. Just specify the one script with "-s" and all actors will be run by that script. The script will only be compiled once; no extra memory will be consumed by mentioning the script more than once. With the exception of the "-t", "-l", and "-o" options, any specifier given without a number is applied to all actors. It is an error to use a specifier in numbered and unnumbered form in the same command line.

Example:

```
VU     -a1 "Admin" ∂
       -t1 "*:gozer" ∂
       -s1 "AdminTest.vu" ∂
       -o1 "AdminTest.out" ∂
∂
       -a2 "BusyTester" ∂
       -t2 "*:chuck" ∂
       -s2 "BusyWorkstation.vu" ∂
       -o2 "BusyWS.out" ∂
       -m2 1 ∂
       -k2 50 ∂
∂
       -a3 "LazyTester" ∂
       -t3 "Farside:Twiddledee" ∂
       -s2 "LazyWorkstation.vu" ∂
       -o3 "LazyWS.out" ∂
       -m3 1 ∂
       -k3 5
```

◆ *Note:* the line continuation character "∂" (option-d) is only necessary if the command is broken into multiple lines.

In this example there are three actors, each running different scripts. Actor 1 is “Admin”, who is running a test of the AppleShare™ Administration software on the target “gozer”. Actor 2 is “BusyTester”, who is playing the role of a busy AppleShare workstation user, with mouse and keyboard speed set to very high values. The third actor is “LazyTester”, operating on “Twiddledee” in the “Farside” zone. “LazyTester” is playing the role of the lazy workstation user and has mouse and keyboard rates set to a very low level.

Another example illustrates the use of -s without a number after it to specify the one script which will be run on all of the targets:

```
VU    -t1 "*:gozer" ∂
      -m1 10 ∂
∂
      -t2 "*:chuck" ∂
      -m2 5 ∂
∂
      -t3 "Farside:Twiddledee" ∂
      -m3 1 ∂
∂
      -s "Chooser QuickLook.vu" ∂
      -k 30
```

Here, the quicklook test for the Chooser is being run on three targets. Each target has a different mouse speed, but all the targets have a maximum keystroke rate of thirty characters per second.

Note that you can achieve a variety of effects by including and excluding various command options. If you specify an actor (-a), a target (-t), or both without a script name, VU allows you to write a script “on the fly” in an open window. This is useful for trying very small scripts or single-command scripts.

```
VU -a “*:The_N_MAN”
```

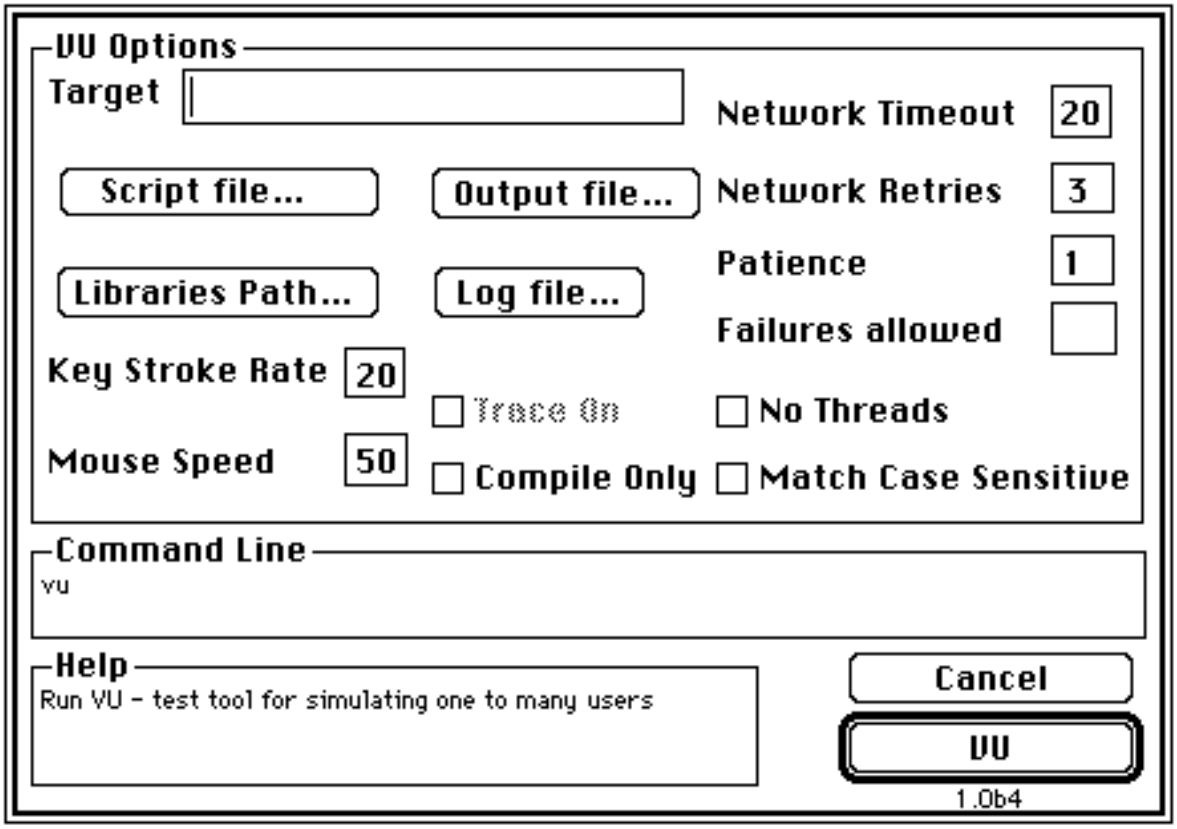
You may also specify an actor and a script without specifying a target. If you then want the actor to operate on a target, you can acquire the target “on the fly” within the script using the VU Scripting Language. (Refer to the Virtual User Language Reference Manual for details.)

```
VU -a “*:The_N_MAN” -s “*:SuperScript”
```

Or, you can specify a target and a script without specifying an actor name. VU will create an actor for you to run the script (see the explanation for “-a” above). You may then change the name of the actor within the script, if desired, using the VU Scripting Language.

```
VU -t “*:gozer” -s “*:SuperScript”
```

Figure 3-1 The VU Commando dialog



VU provides you with a Commando interface to make it easier to build your command line to run your tests. To invoke Commando, just type “VU” and press option-enter (alternatively, type “Commando VU” and press enter). This brings up a dialog window as shown in the above figure. All options are visible in this window, and help text for each option can be instantly accessed by holding the mouse down over the option. The help information is displayed in the standard help window at the bottom of the Commando dialog. Some of the options come with default settings, which you can modify.

Setting the Options

- **Target:** This edit text box comes with no default value. You should type in the network address of the target (<zone>:<user name>) in this box. If the target is on the same zone as the VU machine, then you can use the meta-character "*" (asterisk) for your zone name.

- **Script file… :** You can select your script file to be run by clicking the mouse on this button. The standard file dialog will pop up, and you may select the requisite script file, then select OK or just double-click on the selection. Note that initially only the files in the working directory with the ".vu" extension in their names are listed. Selecting the radiobutton "All text files" will cause all of the files in your working directory to be listed.

- **Libraries Path… :** You can select the search path directory for libraries by clicking the mouse on this button. The standard file dialog will appear, and you may select the requisite directory and then select the Directory button.

- **Log file… :** You can select the name of the file into which you want the test information to be logged by clicking on this button. This will bring up the standard file dialog, allowing you to type in the name of the file. You can select OK after you enter the name, or press return.

- **Trace On:** If you specified the "Log file..." option, this checkbox will become active. You can click on this checkbox if you want to trace your script. The trace information goes into the log file.

- **Mouse Speed:** Type the desired mouse speed in this box.

- **Key Stroke Rate:** Type the desired keystroke rate in this box.

- **Match Case Sensitive:** This checkbox may be selected if you want VU's string matching to be case sensitive.

- **Network Timeout:** Type the desired network timeout value in this box.

- **Network Retries:** Type the desired number of network retries in this box.

- **Patience:** Type the desired patience level in this box.

- **Failures Allowed:** If you want to set a limit to the number of failures allowed by VU before it aborts, type your new limit in this box.

Running the Test

Commando constructs the command line as you set the options. The Command Line box displays the command line resulting from the options you select from the dialog box. You can copy all or part of this command line using Command-c or the Edit menu. But you cannot type into this window directly.

Pressing Enter or clicking on the VU button (also known as the Do It button), passes the command line to the MPW shell for execution. If you change your mind and want to exit the Commando dialog without running the script, select the Cancel button.

You can get these special results by pressing the Option and/or Command keys while selecting the VU button:

- **Option key.** The command line is also written to standard error. This means that the command is executed and is echoed to the active window.

- **Command key.** The command line is not passed to the shell. Nothing is executed.

- **Option-Command keys.** The command line is written to the active window without being executed.

VU Runtime Execution Control

The VU tool also provides some limited real-time execution control. You can ask VU to suspend execution of scripts while they are running by using Command-S. To resume execution while in suspended mode use Command-R. Note that it may take a while before VU suspends execution, since it has to complete any pending AppleTalk transactions. You can abort the execution of VU like any other MPW tool with a Command-period. VU might take some time before it really aborts after you type in a Command-period, as it has to abort all pending network transactions and do some other necessary clean up before exiting.

Chapter 4 VU Extension

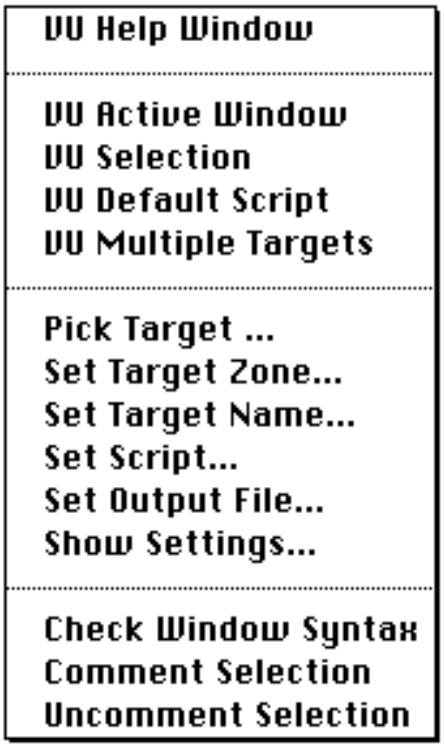
This chapter describes VU Extension. This is available to you if you have used the **Easy Installation Method** described above. The extensions are provided by way of MPW scripts and are set up at the time the MPW shell is launched. A VU Menu is set up by the "UserStartup•VU" script which makes the extensions available to the user.

These extensions will help you avoid a cluttered worksheet as you go through session after session of script development along with many test execution cycles. In addition, they will provide features that come in handy.

Using the VU Menu

This section provides a detailed guide to using the VU Menu. Once VU Extension is installed into your MPW system, a VU Menu is appended to your MPW shell application's menu bar every time you launch it. Under that menu, you will find menu items that will invoke useful, VU-related MPW scripts and tools.

Figure 4-1 The VU Menu



VU Help Window Item

VU Help Window, when selected, will bring up a new window containing the VU Help information. This is the same help text which would be obtained by the command “help VU” entered from the MPW worksheet. The help text gives a brief description of the VU command line and associated options.

VU Execution Menu Items

VU Active Window, when selected, will execute the contents of the currently active window, which should more than likely be the script file you have been working on. It runs the script in the active window against a predefined target in a predefined zone, and redirects errors and standard output to a predefined file. These predefined variables can be set using other VU Menu items, described below in **Setting Predefined Variables**.

VU Selection executes only those lines in the active window that are highlighted instead of executing the contents of the entire window. Again, it uses the predefined variables indicating zone, target name, and output file.

VU Default Script executes a VU script from a predefined script file. The predefined target and output file are again used.

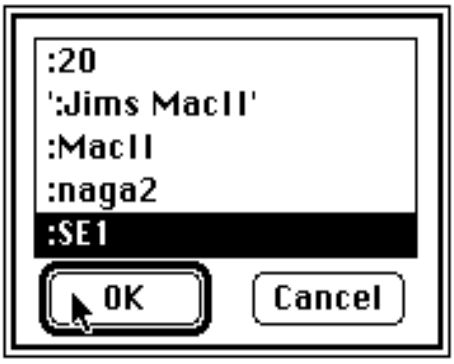
VU Multiple Targets allows you to set up a multiple-target test using the Commando interface. MPW commands for running VU against multiple targets can become very long. One can make mistakes very often in constructing these commands. So we have provided this menu item which actually constructs the command for you and outputs it to the worksheet. You can then select the command and hit the enter key to run it. You can use this item as a way to invoke Commando for a single target as well.

Setting Predefined Variables

The next six menu items set or show the predefined variables that dictate how the **VU Execution Menu Items** will behave. There are four variables that concern us here. They define the zone in which the target CPU is located, the Chooser name of the target, the name of the default VU script file, and the name of file to which VU output is redirected. When you set these variables they are saved in the "Settings_vu" file and used as defaults until you change them again. Since the values are saved in a file, they will not be lost when you quit MPW.

Pick Target… brings up a dialog box as shown below listing all the available target machines on the network. You can pick the desired target by selecting it and hitting the OK button. The target address is saved in the "Settings_vu" file and will be used as the default. Note that on networks without zones, the dialog does not show any zone name. In these cases the zone name is set to the meta-character "*" .

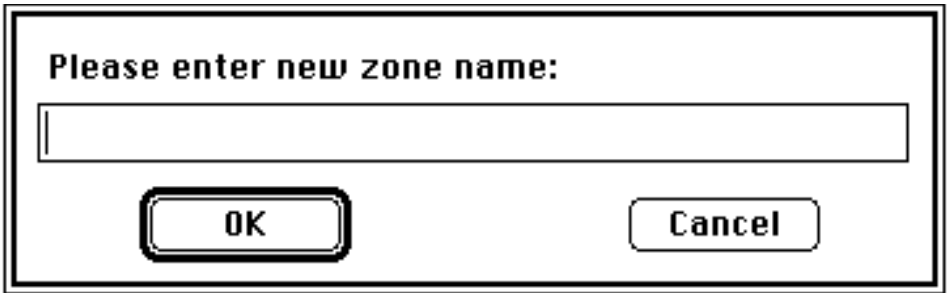
Figure 4-2 The Pick Target dialog box



You may also set the zone and target name separately using the following two items.

Set Target Zone... brings up a dialog box as shown below and asks you to enter the name of the zone. Simply type in the zone name and click the OK button or press the return key. Clicking on Cancel causes the MPW script to abort, leaving the variable unchanged. Enter the meta-character "*" to indicate that the local zone should be used.

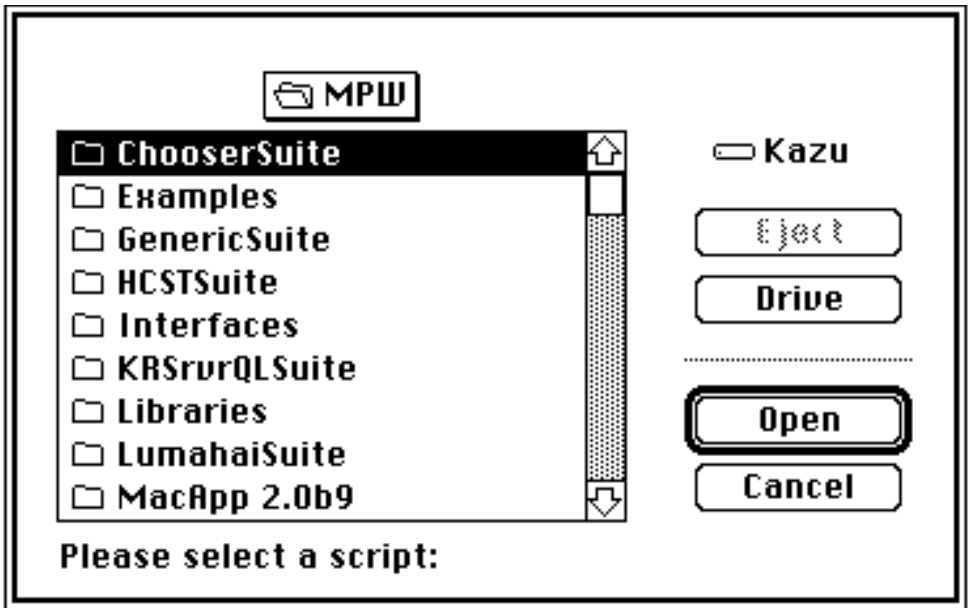
Figure 4-3 Set Target Zone dialog box



The menu item Set Target Name... brings up a dialog analogous to that resulting from selection of the Set Target Zone... item. The dialog allows the user to set the name of the target CPU to be used for repetitive executions during script development.

Selecting Set Script... presents a standard file dialog box (as shown below) which can be used to select a script file that you will be using repetitively. The file you select in this dialog is executed every time you select VU Default Script.

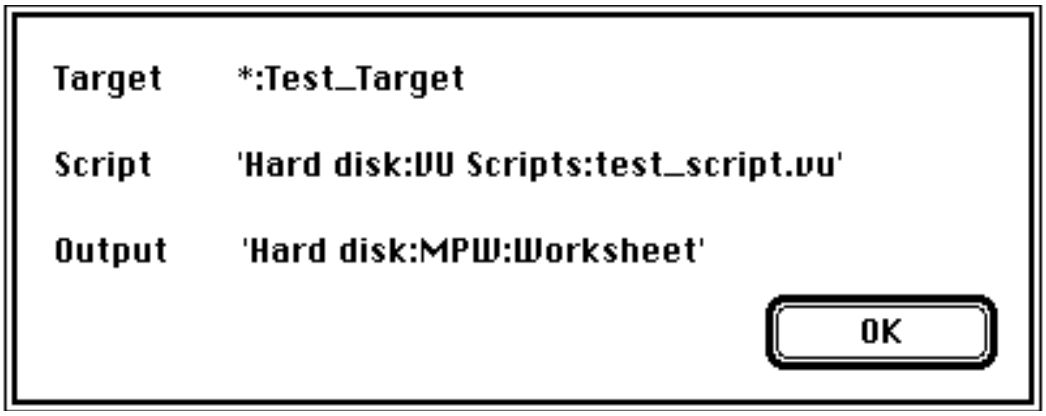
Figure 4-4 The Set Script dialog box



Set Output File... will let you set the name of the file to which you want all VU output redirected. You may want to set it to the Worksheet as a default.

To see your current variable settings, select Show Settings.... A dialog box will display the settings that will govern how the items in the VU Menu will be executed. An example of such a dialog is shown below.

Figure 4-5 The Show Settings dialog box

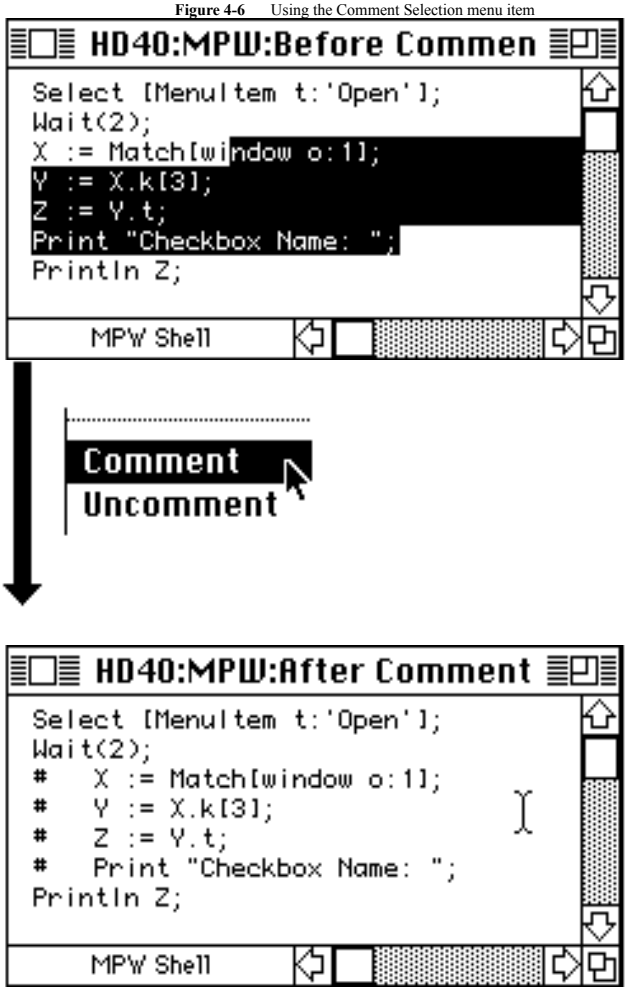


Other Items

Check Window Syntax checks the syntax of the VU script in the active window and informs you of any errors in your script by printing error messages to the predefined Output file. For this command, you probably want to set the predefined Output file to be the Worksheet or another file for which a window is currently open.

Comment Selection can be used to comment out one or more lines of a VU script. First, select the lines that you wish commented. Then select this menu item. All script lines within the current selection will be preceded by a pound sign (#) and a tab. It is not necessary to select an entire line. As shown below, selecting any part of the line will do.

Figure 4-6 Using the Comment Selection menu item



Uncomment Selection works exactly the same way, except it removes the pound sign and any white space characters at the beginning of each selected line.

The VU Menu is easily configurable to your liking. As you become more acquainted with VU, you might want to add some more useful commands to this menu. Or you might just want to add keyboard equivalents of your own to these items. To be able to make any such changes you must learn the MPW command "AddMenu". The VU Menu is set up by the Startup file in your Virtual User folder. You can edit this file to your liking in MPW.

Chapter 5

Augmenting VU with VU Recorder

VU has many of the facilities needed to automate manipulation of the standard Macintosh user interface elements. What it lacks in specific commands can usually be built up from its primitive mouse movement and keyboard commands. Common tasks such as selecting a tool from a tools palette can be made into a reusable task without an extraordinary amount of effort. Freehand drawing, however, does not lend itself well to scripting in VU.

VU is not yet complete. One of the planned additions to the system is the ability to record a user's activity so that actions otherwise indescribable in the scripting language can be included in test scripts. Until we reach that point, VU can fall back on its predecessor, VU Recorder, to reproduce literal mouse movement and keyboard activities. This chapter describes the use of VU Recorder with VU. Refer to the VU Recorder Reference Manual for more information on VU Recorder.

Invoking VU Recorder From VU

VU Recorder can reproduce input with high fidelity, but sharing the CPU with VU and the MPW Shell tends to cause playback to be slower than normal. VU Recorder is much more sensitive than VU to changes in the screen layout. You should try to minimize your dependence on VU Recorder, as its recordings are more delicate than VU scripts.

The VU Scripting Language includes a statement which directs VU Recorder to play back one of its recordings on the target VU is using. This statement is called, for historical reasons and in the interest of backward compatibility, **callpp**. It takes two arguments, one of which is the name of the VU Recorder (p: for player) and the other is the file name containing the recording created with VU Recorder. For example, the following call tells VU Recorder to playback the recording called "freehand draw".

```
callpp p:"VU Recorder" f:":freehand draw";
```

The **f:** argument specifies the name of a VU Recorder recording file. This can be a full or partial path name.

The **p:** argument names the VU Recorder program that the VU should contact. VU Recorders adopt their application name for the purpose of announcing their availability for remote control. For arcane reasons, the VU Recorder must be running on the same machine as the VU interpreter.

Setting Up

Running VU and VU Recorder together requires a machine with enough memory to run both VU Recorder and the MPW Shell under MultiFinder. So 2 megabytes is probably the low end of the workable range.

VU and VU Recorder will be communicating with each other via a special feature of AppleTalk called intra-node delivery. Intra-node delivery allows two programs to talk to one another as if they were running on different machines on the network. This feature is normally disabled. It can be activated by installing the SetSelfSend CDEV in the System Folder of the machine on which you intend to run VU and VU Recorder. Once SetSelfSend is in the System Folder, it can be reached through the Control Panel and used to activate intra-node delivery.

Script Development

A grave weakness of VU Recorder is that playback of its recordings doesn't have the desired effect if anything significant on the target's screen changes position between the time the recording was made and the time it is replayed. VU can be used to

overcome many of these problems by aligning and sizing windows in the precise manner required for successful playback. Getting this to work right takes a little practice and patience, but it isn't difficult.

Useful Preface

A common practice in the world of MPW Shell scripts is to put names of files and directories into variables, so that as the script is run on different machines, the values of the variables can change, but the script need not change. This practice carries over into VU programming, especially in the case of scripts which use the **callpp** command. **Callpp** makes references to files and to VU Recorder applications, both of which may change from file system to file system. VU variables can be used to control all the aspects of the script that are likely to change from machine to machine. One possible header to a VU script is the following:

```
# The following VU code could be included near the top of any
# VU script which uses callpp. You would substitute the values
# for these variables that describe your machine.

# Recordings_path is the path to where the recordings
# for this test are held.

Recordings_path := "HardDisk:TestProjectFolder:VURecordings:";

# VU_recorder_name is the name of the VU Recorder application (as seen
# in theFinder) with which this script should work.

VU_recorder_name := "VU Recorder";
```

These variables can be incorporated into the **callpp** command anywhere in the script following their definition or in tasks which use the global declarations. The values of these variables can be changed elsewhere in the script, but it is probably better to define other variables for the different directory paths and VU Recorders you want to reference.

Depending on the organization of your folders, it may be possible to set “Recordings_path” to a partial path name. A partial path name describes the series of folders you open to find a file, starting from the one you are currently in. Using partial paths make moving a collection of test files from one machine to another relatively easy in that so long as you keep the folders containing the recordings in the same positions relative to one another, then “Recordings_path” does not need to be changed.

The General Process

Start off by writing the VU portion of the script, leaving comments in the places where you intend to call VU Recorder. With the script handy, step the target application through the process until you reach a point where VU Recorder needs to be called. This can be done manually or by executing the script piecewise. At this point, it is desirable to have VU position the windows such that their placement can be recreated when the script is run.

You might want to decide on a point on the screen where you want the top, left corner of all your windows to line up. A good point might be just below the menu bar near the Apple menu, at x = 5, y = 25. In VU, a window can be moved to this point with the command:

```
# VU code to position a document window for work with VU Recorder
drag [window] a:{ 5,25 };
```

Now that the windows are set up just as they will be when VU is running the test, run VU Recorder and record whatever actions you want it to perform. When the recording is finished, save it and continue stepping through the test to make whatever other recordings you need to support your VU script.

Test Execution

Running a test with VU and VU Recorder is much easier than writing and recording one. With intra-node delivery (SetSelfSend) on, launch VU Recorder and the MPW Shell. Bring VU Recorder into the foreground and choose Remote Control from the Control menu. VU Recorder will hide its Target Picker window and the message “Awaiting control from afar” will appear in the box at the top of its main window. That's all you need to do to with VU Recorder. Now switch back to MPW.

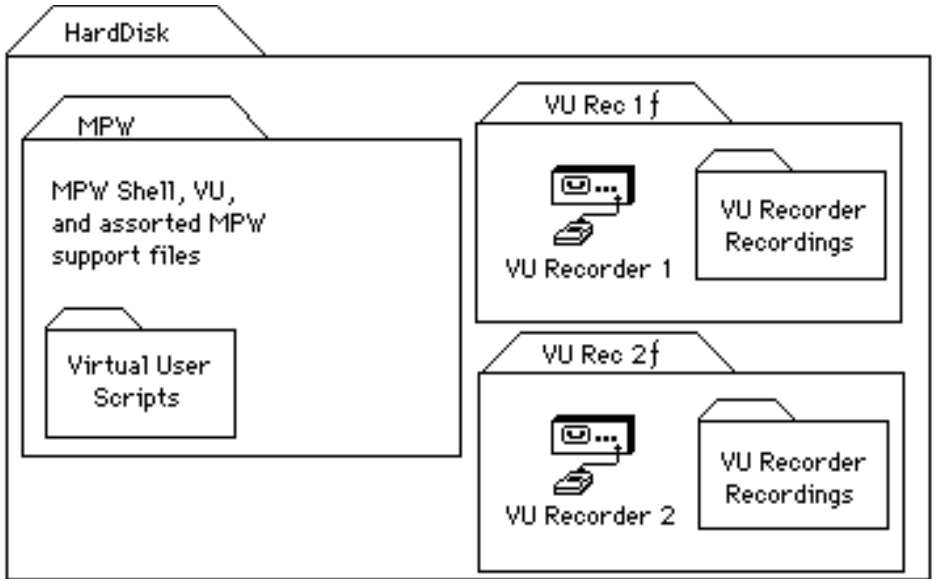
You are now ready to use VU with VU Recorder. Run the VU script as you would any

other. VU will direct VU Recorder to run the recorded portions of the test and will relay any errors VU Recorder encounters to the MPW worksheet. VU Recorder errors will also be routed to the VU log file if you requested logging on the MPW command line.

Multiple VU Recorders

For multiple target tests where at least two assigned scripts require VU Recorder playback via **callpp**, a uniquely named VU Recorder needs to be executing for each such script. To help VU tell the VU Recorders apart, each copy of the VU Recorder application should be renamed in the Finder and each needs to be placed in a separate folder (so they can each write their log files). A simple naming scheme like “VU Recorder 1”, “VU Recorder 2”, “VU Recorder 3”, etc. will work fine. Each of those VU Recorders will need a complete set of the recordings that are to be used in the test. For a two-VU Recorder test, the directory setup might look like the following diagram.

Figure 5-1 A sample VU/VU Recorder folder arrangement



It should be stressed that this is not the required arrangement. It simply meets the requirements of having VU Recorders with different names in different directories, each with its own copy of the recordings.

The only tricky part of this whole arrangement is when there are multiple actors reading the same script (there is a metaphorical actor reading the script for each target). Each of these actors must use a different VU Recorder when it comes time to execute the **callpp** statement. The only way for them to distinguish themselves from one another is to use the target's name, which is embedded in the target descriptor. The script can then test the name with a cascading if - else - if until it runs out of known targets, at which points the script can report the error and shut down the test for this target (alternately, a global variable could be set which would tell the script to skip the portions of the test that depend on VU Recorder).

```
match[target t:?target_name];
if target_name ~= /≈Target 1/ begin
   recorder_name := "VU Recorder 1";
end;
else begin
   if target_name ~= /≈Target 2/ begin
      recorder_name := "VU Recorder 2";
   end;
   else begin # can't determine VU Recorder for target
      println "I don∂'t know which Recorder to use for
   target:",target_name;
      exit; # terminate the run of this script against this target
   end;
end;
```

Using partial path names eliminates the need to modify the value assigned to “Recordings_path”, as it is the same in both cases.

```
Recordings_path := ":VU Recorder Recordings:";
```

If the set of recordings you want to use also depends on the target, you could add an assignment to the variable “Recordings_path” at the places where recorder_name is assigned. The variables “recorder_name” and “Recordings_path” can be used in the **callpp** statement like so:

```
callpp p:recorder_name f:"{Recordings_path}{recording_name}";
```

Shutting Down

When the test is completed, you can quit from both programs in whatever order suits you. VU Recorder may complain that its target isn't responding. This is because VU released its agent without telling VU Recorder. Agent VU's unwillingness to take orders confuses VU Recorder, so it raves about possible target crashes and network failures. In general, this is not indicative of any problem in your VU/VU Recorder session.

Chapter 6 Running VU Against Applications Built With MacApp

In order to find the location of items in windows for VU, Agent VU peruses the content list of windows and the item list of dialogs. For applications which are not built with MacApp, these data structures can be accessed via global variables by Agent VU. For MacApp applications, however, items in windows are subclasses of the class TView, which are not available to the Agent via global data structures. As a result, Agent VU can't provide any information about these items to VU in the normal way.

This chapter describes a way in which an application built with MacApp can assist VU so that full VU functionality is available for that application.

The Agent VU Assistance Hook for MacApp Applications

Agent VU provides a hook facility which allows an application to assist Agent VU by providing information about itself. We've provided such an assistance hook for MacApp applications. A user can build a MacApp application with the Agent VU assistance hook for MacApp applications; and then test the application with VU just like any other application.

The hook exists as a unit called UVUAssist. The unit consists of four files: VUAssist.p, VUAssist.inc1.p, VUAssist.inc2.p, and VUAssist.a.

You must hook this unit into your MacApp program in the following places:

In MyApplication.IApplication, you must execute the following three statements:

```
NEW(gVUAssist);
FailNil(gVUAssist);
gVUAssist.IVUAssist;
```

In MyApplication.AboutToLoseControl you must execute this:

```
gVUAssist.SuspendMole;
```

In MyApplication.RegainControl you must execute this:

```
gVUAssist.ResumeMole;
```

See the assistance hook source code provided with the VU package for more detailed comments regarding the use of the hook.

Running VU Against MacApp Applications Under System 7.0

The default behavior for MacApp 2.0 applications running under System 7.0 is to call WaitNextEvent in the main event loop with an unlimited sleep time. Due to changes in MultiFinder for System 7.0, Agent VU does not get execution time when it needs it in this case. (See Appendix A, Making Applications "VU-Friendly".) As a result, VU will fail to run against a target machine with this configuration. You can work around this problem by overriding the default MacApp 2.0 behavior so that WaitNextEvent is always called with a finite sleep time such as 1 second.

Chapter 7 Scripting Hints

VU scripts, like most programs, can be written in ways which help improve the performance of the script execution. This chapter is dedicated to providing you with some techniques to make your scripts run in less time and within smaller memory configurations. Also included in this chapter are some hints on debugging scripts. You may need to refer to the Virtual User Language Reference Manual to fully understand the information given.

Reducing Execution Time

Script execution goes through the following two steps:

1. **Parsing and Loading**: VU reads the script from the script file, checks for syntax and builds a script representation in memory.

2. **Interpreting**: Each statement in the script is interpreted one after another.

The Parsing and Loading step is done in the beginning and takes a reasonably short period of time. This time period increases linearly with the size of the script. Therefore, your script file should be only as large as it is really needed to be. You might want to make sure that your script file does not contain unused task definitions.

Script interpretation consumes most of your total execution time. A VU script consists of two types of statements :

1. Statements whose interpretation is done without communicating with Agent VU
2. Statements whose interpretation involves talking to Agent VU

The interpretation of the first type of statement takes less time in general than the second type of statement. The parts of VU scripts with just the first type of statement are like programs written in any other programming language. You can improve the performance of these parts by the usual mechanisms such as the use of faster algorithms.

The second type of statements are more critical to improving the time performance. Statements which fall into this critical category are:

1. Any command which asks for some action to be performed on the target machine. For example, Selections (Select [window/menu/menuitem/…]), Drag/Scroll/Zoom commands, etc.
2. Any statement which uses an explicit "match""or "collect" on an element from the target. For example,

```
match[window t:?w_title o:1];
all_file_items := collect[menuItem m:[menu t:'File']];
while (not match[window s:dialog o:1]!);
```

Interpretation of most of these statements involve two steps:

1. find the appropriate target element
2. perform the operation asked for

VU cannot really do any time-saving in the second step since the script writer obviously wants the operation to be done to completion. But a script writer can make the first step easier for VU and hence save some time. VU uses the description provided by the script (in the form of a descriptor) to find the corresponding target element. You as a script writer can speed up the process by constructing these descriptors wisely. The descriptors should be as specific as possible. This will reduce the search space. The following guidelines will help you decide how specific to be.

Guidelines for Creating Descriptors

Specifying Trait Values

1. **String valued traits:** Provide complete strings if possible. If you do not know the complete strings then use regular expressions. Avoid incomplete (non-regular expression) strings.

   For example, if you know a window title is 'First Window' don't just indicate 'First W' even though it might do the job for you in your context. You can use a regular expression like /First W≈/ instead.

2. **Numerically specified traits:** Do not specify these traits unless you are certain of their values. On the other hand whenever you are certain of their values, do specify them. The "o:" trait is an example of a numerically specified trait.

3. **Owner traits:** Whenever applicable, always try to specify the owner trait for a descriptor. Some instances of descriptors with owner traits are menuItems, buttons etc. For example:

   Use

   ```
   select [menuitem t:'Monaco' o:7 m:[menu t:'Font' o:4]]!;
   ```

   instead of

   ```
   select [menuitem t:'Monaco' o:7]!;
   ```

   and

   ```
   select [button t:'Cancel' w:[window o:1]]!;
   ```

   instead of

```
Select [button t:'Cancel']!;
```

4. **Trait values which have descriptors (excluding owner traits):** Traits such as the content list in a window, items list in a menu or mouse trait in a target descriptor are at least partially built with descriptors. They should not be specified unless absolutely necessary. When it does become necessary to specify them (in cases when that is the only information you have) then specify as few such embedded descriptors as possible. For example, use

```
match [menu t:?m_title o:4];
```

instead of

```
match [menu t:?m_title o:4 i:{[menuItem t:'Chicago'],
                              [menuItem t:'Courier']}];
```

If you do not the know the rank of the menu, then you have to specify the "i" trait. In such a case specify as few items as required to make it unique.

```
match [menu t:?m_title i:{[menuItem t:'Chicago']}];
```

Similarly, use

```
match [window t:/Find≈/ o:2 ];
```

instead of

```
match [window t:/Find≈/ o:2 k:{[button t:'Find Next'],
                              [radiobutton t:'Whole word']}];
```

Use of the Perfect Match Operator

Whenever you are certain of your description, make use of the bang (**!**) operator. On the other hand, never use the **!** operator after a descriptor if you are not certain of the description you have created. Never use the **!** operator if any of the traits has an incomplete value specified.

For example, if you know for sure that the front-most window has a radio button called "Partial Word" then use:

```
select [radiobutton t:'Partial Word' w:[window o:1]]!;
```

But if you are providing a partial title (maybe you are unsure) then do not use "**!**"

```
select [radiobutton t:'Partial' w:[window o:1]];
```

Similarly, if you want to select the window with the radio button "Partial Word", but you are not sure of other controls present in that window then do not use perfect match:

```
select [window k:{[radiobutton t:'Partial Word']}];
```

Altering Execution Time with System Tasks

The system tasks **patience**, **mouseSpeed**, **typeSpeed** and **wait** can be used in a script to modify the speed of execution. As you may have observed, VU's actions are generally quite swift. Hence more often than not these tasks will be used to decrease the pace of VU's actions. Overuse of these tasks is detrimental to the script's performance.

Patience can be thought of as a measure of "intra-command" delay time. The **wait** statement, on the other hand, can be thought of as an "inter-command" delay. A single VU command statement such as menu selection may involve several distinct actions against the target machine (i.e moving to the menu, dropping the menu bar, moving to the item, etc.). **Patience** is a relative measure of the delay time in-between each of these actions. The **wait** statement, however, only affects delay time in-between whole VU statements. In general, **patience** is more effective in dealing with systems which are responding slowly due to the particular hardware and software combination.

Before you use any of these tasks, however, make sure that you really need it. Once you know that you have to slow down VU (probably to accommodate slower responses from the test applications), try to localize the portion of the script which needs to be run at the slower speed. Do not slow down the whole script when the intention is to slow down just two statements. Once you have determined the portions of the script which have to be slowed down, try to find the optimal combination of mouse and keyboard speed, **patience**, and **wait** statements required. After this you can wrap the proper portions of the scripts with their corresponding settings. Do not forget to reset the changes made to any of these settings after the slower part of the script is finished.

For example, if increased **patience** is required in opening and closing the Chooser, the script segment could be written as follows:

```
patience_for_chooser := 3;    # pre-determined patience setting for
                              # selecting Chooser menuitem
save_patience := patience(patience_for_chooser);# set patience to new
                                    #value
select[menuItem t:'Chooser' m:[menu o:1]]!;     # select Chooser with
                                    # increased patience
close [window t:'Chooser' o:1]!;    # close Chooser window with
                         # increased patience
patience(save_patience);      # reset patience to original
                  # value
```

Summary :

- Have as few (if any) **wait**s in your script as possible.
- Change **patience**, **mouseSpeed** or **typeSpeed** only if absolutely necessary.
- Localize the changes to any of these settings to where needed and then reset back.
- Try to avoid setting **patience**, **mouseSpeed** or **typeSpeed** from the command line.
- Try increasing **patience** instead of **wait** if the intent is to slow the pace.

Memory Efficiency

Sometimes when running long scripts and/or multiple scripts, VU might run out of memory. In this case, MPW reports "Unable to swap segment". Of course, you can always try giving the MPW shell a bigger partition of memory. But since this is not always possible, we have to find ways to make the script memory efficient as well.

The VU Scripting Language provides two compound data types, descriptors and lists. Both of these structures can grow quite large at times. Hence they have to be used carefully. The scripting language does not provide any explicit means of *freeing* these structures when not in use. VU frees these structures for you when no variable in the script refers to them. So when you are done using a list or descriptor, reassign the variable containing the list or descriptor value to their respective null values (i.e. { } and [ ]). This will indicate to VU that you are not interested in keeping the list or descriptor around anymore. One rule of thumb is to avoid carrying large lists or descriptors around as variable values through large parts of the script.

In the case of descriptors, most often your interest lies in particular trait values. In such cases, use unification in the first place to bind these trait values to variables. For example, consider the following script:

```
front_window := match[window o:1]!;
if (front_window.style = dialog)    #if a dialog window
begin
   if (match [button t:'OK' w:front_window])    #if OK button exists
      select [button t:'OK' w:front_window]!;
   else if (match [button t:'Cancel' w:front_window]) # if Cancel
                                          # button
      select [button t:'Cancel' w:front_window]!;
end;
```

In this script fragment, the complete description of the window,

```
   [window t:'window1' s:dialog o:1 z:false g:false k:{[button t:'OK',
```

```
                                                [button t:'Cancel']}]
```

is carried on through all the statements as the value of the variable “front_window”. This not only uses up memory for a longer duration, it also slows down the matching since the description contains the ‘k’ trait, which could be a very long list of descriptors.

Since the interest is only in checking if the window is a dialog, we can unify the style trait instead of assigning the complete descriptor to a variable. You can also unify the title trait to use in later matchings.

The same script fragment can be rewritten as:

```
match[window o:1 s:?front_style t:?front_title]!;
if (front_style = dialog)     #if a dialog window
begin
   if (match [button t:'OK' w:[window t:front_title o:1]]!)
      #if OK button
      select [button t:'OK' w:[window t:front_title o:1]]!;
   else if (match [button t:‘Cancel’ w:[window t:front_title o:1]]!)          #if
cancel button
      select [button t:‘Cancel’ w:[window t:front_title o:1]]!;
end;
```

As a last resort, if you cannot get your script running due to less memory, try running the tool with the command line option “- noThreads”. With this option VU might give a slower performance but will be less memory intensive.

Debugging Scripts

The tracing of VU script execution in log files has been proven to be very useful for debugging or monitoring scripts. Setting the trace on allows you to get a verbose description of the script execution in the associated log file. Obviously, it's necessary to specify a log file on the command line to be able to use this feature. Remember that you can open this log file up within MPW and view the trace as the script is executing. For multiple target tests, you can keep all the log files open and the windows tiled while running the tests.

The command-line designator is '-dt '. For example:

```
VU -t1 "*:SE1"  -s1 "Script1" -l1 "Logfile1" -dt1  ∂
   -t2 "*:SE2"  -s2 "Script2" -l2 "Logfile2" -dt2
```

You can also turn debugging trace on and off within a script. Use the system task, **trace**, to turn debug trace on/off within a script. The default trace value is false. You can continue to set trace on from the command line in MPW.

Chapter 8

Troubleshooting Guide

This chapter is dedicated to helping the VU user diagnose problems that may be encountered in using VU. The problems have been divided into three categories. The first category of problems involves Agent VU installations on target machines. The second category is test staging problems. These are the problems that occur in specifying VU's parameters in the command line or the Commando dialog . The third category of problems involves problems that may be encountered once VU begins running.

Target Installation Problems

Troubleshooting Agent VU Installation

To prevent the embarrassment of accidentally taking over the wrong machine with VU, Agent VU will refuse to install itself if the Chooser name is not unique in the zone. In this case Agent VU's icon will appear with a

sign saying “Name Clash” plastered over its face. There can only be one Agent VU with an empty name in a given zone so remember to give your target machines names.

Three other error conditions can prevent Agent VU from successfully initializing on the target machine. If there isn’t enough memory in the target’s system heap, Agent VU will back out, and the message “Memory’” replaces the agent’s mug.

This should be a very rare condition. If the test permits it, you might try removing other startup documents to make room for Agent VU. The remaining installation error occurs if it cannot link up with the AppleTalk driver. In this case, the abbreviation “A-Talk’” is displayed.

The recourse in this case is to make sure that AppleTalk is activated in the Chooser. If the “A-Talk” message persists, then reinstallation of the system may be in order.

Agent VU installs a driver at startup time. This driver is installed in a slot in the driver table normally reserved for desk accessories which are specific to applications. Agent VU tries to find an unused slot in this portion of the driver table to prevent conflicts with other drivers. However, it is still possible that a driver slot table conflict may cause Agent VU or another driver to fail. If you suspect that this is happening, you might try changing the DRVR resource ID of the driver with a resource editor or removing the driver altogether if it isn't necessary to the test.

If there is a failure to open Agent VU's driver due to a numbering conflict or any other problem, the following icon will appear on the desktop.

Test Staging Problems

This section documents some common problems users have in staging a Virtual User test session using the MPW command line interface. Common error messages with associated explanations are also given.

Some basic things to remember when dealing with command line problems are as follows:

- Always make sure that the command line has blanks at the appropriate places as delimiters.

- Delimit the strings with single quotes, whenever possible. If your string contains a single quote then use double quotes as a delimiter. For example, -t "Bob's Mac".

- If your command line is long and you want to distribute it over multiple lines then make sure you escape each end of line (return key) with the character "∂" (Option-d).

  For example,

```
VU -t 'target1' ∂
-s 'script1' ∂
-o 'output'
```

To support logging and redirected output for multiple targets it may prove necessary to increase the number of open files allowed by the operating system. This number is normally set to a relatively small number such as ten. To change it, you need to use a disk and file editor to edit the maximum number of open files, which is stored in the boot blocks of the volume. A conservative estimate of the number of open files you will need is (3 * number-of-actors) + 4.

While executing the VU tool if you make an error on the command line, you might get one or more error messages of the form described below:

```
ill-formed command line, cannot proceed
```

This message is usually accompanied by some other specific message which tells you more specifically what was wrong on the command line.

```
Improper <argument> <specified value> (a positive integer expected)
```

Messages of this generic form such as:

```
Improper Mouse speed five (a positive integer expected)
```

are largely self-explanatory. In this case, a positive integer was expected but something else was encountered. The "something else" is also printed in the message to give you a better idea of what you should fix.

```
<switch> option already used
```

Or, more specifically,

```
'-dt' option already used
```

This message indicates that your command line contains a repetition of switches. If you are running multiple targets and want to specify different switches for each target then follow the switch with the target index (i.e. “-dt1”, “-dt2”, etc.).

```
Maximum number of targets allowed is 12
```

This message indicates that either you've really tried to run against more than 12 targets or you've given an incorrect target index to one of the switches. For example “-t13” or “-s-l” with no delimiting blank between -s and -l.

```
<switch> option repeated as <specified string>
```

Or more specifically,

```
'-dt' option repeated as -dt2
```

In multiple target configurations, you can omit the switch index if you want that switch to apply to all the targets. But after doing so, it is not acceptable to re-specify that switch with an index for a particular target.

```
parameter not used --> <switch>
```

Or, more specifically,

```
parameter not used --> -l
```

This message is given for switches which follow earlier uses of the same switch with faulty arguments. The remaining switches are processed, ignoring the one with the problem.

```
Improper Target Specification: <specified string>
```

The target can be specified in two forms. The more commonly used form is:

<zone name>:<Target’s Chooser name>

An alternate form uses the network node and socket numbers,

#<network>:<node>:<socket>

Other error messages which could result from an improperly specified target are:

```
if preceded by '#' then the target should be specified as
'#network:node:socket'
```

```
target should be specified zone:name ( default zone is '*' )
```

```
Improper target (discontinuity/repeat in target numbers):  <string>
```

Or, specifically,

```
Improper target (discontinuity/repeat in target numbers):  t5
```

If you are specifying multiple targets, then the switch index has to go up in increments of one starting with 1. This message indicates that you are not doing so. You might have specified "-t1" and then gone on to "-t5", omitting t2, t3 and t4.

```
<argument value> must follow <switch>
```

Or,

```
<switch> should be followed by <value indicator>
```

For example,

```
target name must follow '-t'
```

Or,

```
'-t2' should be followed by a target specification
```

This indicates that you have forgotten to specify a value after a switch.

```
illegal switch  -- ignored
```

This message indicates that there was some switch on the command line which could not be interpreted. Such switches are ignored.

```
Target not found: <address>
```

This message indicates that VU was unable to find a machine with an Agent VU on the network at the target address specified.

Runtime Problems

The following is a list of some problems that may be encountered during VU execution. Explanations follow.

- Loss of contact with Agent VU

A number of error messages, shown below, may be generated by VU due to this general problem.

```
Unable to construct a <???> model
```

The most commonly seen such message is:

```
Unable to construct a display model
```

This usually happens when VU somehow loses contact with its agent during initialization. This is the initialization performed by VU before script execution actually begins. In such a case this message will be followed by another message such as:

```
loss of target during initialization prevented execution of
script
```

VU will quit soon after this message.

Loss of contact with the agent running on a target may cause other messages to be given by VU during script execution. You may get a series of messages such as:

```
### Target failure suspected, will attempt reacquisition ###
### Target failure suspected, will attempt reacquisition ###
### Target failure suspected, will attempt reacquisition ###
### Target failure suspected, will attempt reacquisition ###
### Target failure suspected, will attempt reacquisition ###
### Target not responding ###
### Aborting Script at 4:06:25 PM ###
```

What's happening here is that VU actually tries to re-initialize communication with the agent five times before it gives up. This situation may be simulated on a busy network. On a very busy network, the default network timeout value used by VU for network transactions may not be sufficiently large. VU might be led to believe that the agent failed to respond when what actually happened was that the response from the agent took too long to reach VU. Since the traffic on any network is a constantly changing factor, in most cases VU is able to re-acquire after a few tries. The following message informs you of this re-acquisition:

```
### Target re-acquired: retrying execution###
```

However, you might want to increase the "-timeout" and "-retries" parameters on your command line when running on busy networks to avoid this completely.

One reason for a loss of contact with Agent VU is the failure of the target's foreground application to give drivers execution time at some point in the application. An easy way to verify this is to display the Alarm Clock DA such that it is visible when your application is active. If the time in the Alarm Clock is not changing at the suspect point in the application, then you've verified that the application is not giving drivers execution time at this point.

- Problems running VU Recorder with VU

When accessing a target which is in the same zone as the VU host machine in a multi-zoned network, the user can specify the zone from the command line with the "*" meta-character. However, in a network with zones, VU Recorder expects to see the full zone name in the target name. If VU passes the target name with an "*" to VU Recorder, VU Recorder will not find the target. So, when using VU Recorder with VU in such a network, always specify the full zone name in the VU command line. If you don't, you'll see the following error message.

```
callpp error VU Recorder is unable to grab the target's mole.
```

As indicated in Chapter 5, Augmenting VU With VU Recorder, to use VU Recorder with VU, you must have the CDEV SetSelfSend installed on the VU machine and turned ON. Otherwise, you'll get the error message:

```
callpp error unable to find specified VU Recorder.
```

You'd get the same error message if you did not have VU Recorder running on the VU machine. Failing to put VU Recorder in Remote Control mode would also generate the above error message.

Appendix A

Making Applications "VU-Friendly"

An application is "VU-friendly" if VU scripts can be run against it at all times without loss of VU functionality. In general, applications which adhere to Apple's Macintosh software compatibility guidelines get "VU-friendliness" for free. However, there are certain things in particular to watch out for to insure that an application is "VU-friendly".

- Make sure that there is no point in the execution of your application where drivers are "locked out". Agent VU is a driver which needs to be able to get periodic execution time to provide VU with information about the application and to perform various actions on the target machine.

  There are a couple of ways in which Agent VU might become "locked out".

  1. The application goes into an event loop where GetNextEvent is called to do event processing but SystemTask is not called.

  2. Due to changes to MultiFinder in System 7.0, if the application running under System 7.0 calls WaitNextEvent with an infinite sleep time, Agent VU will not get execution time with the front-most application's context switched in. Agent VU must get execution time with the front-most application's context switched in to get information about that application's windows and menus.

- The standard MDEF maintains the global variables MenuDisable and TopMenuItem (see Inside Macintosh vol. 5). If you use a custom MDEF, you should maintain these variables as well. Menu Disable is set to the menu item number each time a new item is highlighted. This is particularly useful to VU in finding and verifying menu item selections. TopMenuItem contains the global coordinates indicating where the first menu item is to be drawn. This is useful to VU in handling scrolling and popup menus.

- To get VU support of popup menus, use the popup CDEF resource ID 63 which is available in System 7.0 or in the Macintosh Communications Toolbox. With these implementations of a popup, VU can find and select a popup menu item much like any other menu item with the simple “select” command. Unfortunately, for technical reasons, VU is unable to support the popup menu implementation described in Inside Mac vol. 5.

Appendix B A Rendezvous with Agent VU

It is important to know how a test tool might itself impact the testing and potentially compromise the results. In the interest of supplying that knowledge, the authors have sought out the least understood, most maligned component of the Virtual User system, Agent VU, for an interview. We found it hanging around in the system heap, where it was relaxing between assignments.

VU: Sitting here with you, you seem to be a pleasant enough piece of code. How do you account for your somewhat sinister reputation, even among those who know you fairly well?

Agent VU: Part of it is that darn icon I've been saddled with since version 0.01d1. To borrow a line from a more famous 'toon character, "I'm not bad, I'm just drawn that way." Another big part of it is the environment in which I work. Personal computer users have an instinctive fear of an entity that allows anyone to reach across the network and take over their machine.

VU: You don't sound terribly sympathetic.

Agent VU: On the contrary, I am. What people fail to appreciate about me is that installing me is voluntary, removing me is as easy as pulling me out of the System Folder and rebooting, and that I do a good job as part of a testing system, which is all I am intended to do.

VU: Shifting gears a bit, how did you get into this business?

Agent VU: My file type is INIT, which means that if I'm in the System Folder at startup, the INIT 31 mechanism loads me in and turns me loose.

VU: What exactly is it that you do at startup?

Agent VU: Oh, lots of things. Roughly it goes like this:

I initialize the code that displays my icon.

I make sure that the .ATP driver is loaded, if that fails I back out and show my A-Talk icon.

Otherwise, I try to open my driver, which is the bulk of my code, into the System heap. If that fails, I back out and show my Driver icon.

Once my driver is in memory, I detach it and lock it down.

In the course of opening it, a locked block to hold my state variables and buffers is allocated. If that went well, I try to initialize my variables and register myself with NBP.

How, I ask you, can they think of me as sneaky if I post my existence on the network for all to see?!

VU: Doesn't seem terribly fair to us either. All this loading and allocating, how much space do your various pieces take?

Agent VU: The driver is about 10K. The variables are about 3K. The INIT is 4K, but it hardly counts since it's purged when it finishes.

VU: Is that it for start-up?

Agent VU: There's a bit more to it. If the rest of installation goes all right, I queue up a request with ATP. Then I show my icon and kick back until someone sends me something over the network.

VU: You're completely inactive if there's no message for you?

Agent VU: Well, my driver gets periodic time, but it doesn't do anything with it.

VU: How about when a message does come in?

Agent VU: When a message does come in, the ATP driver calls me as a completion routine. Depending on the message, I either execute it immediately or I put it in a buffer to be executed at SystemTask time by my driver. Once the task is completed I queue up another GetRequest and am ready for the next message.

VU: What sorts of things do you do immediately, and what do you defer?

Agent VU: Mouse and keyboard stuff I do right away. Same with offers of employment and firing notices. If I'm working for you, you can ask me about the contents of memory and I'll tell you, or I can plop a block you've sent into memory. The latter is something VU or VU recorder has never asked me to do. I'll also tell anyone who wants to know about the machine I'm on and who I'm working for. These are all things that are safe to do at interrupt time.

Some of my work involves peeking around the windows and dialogs. I don't change them, I just look, and at this point I'm restricted to the foreground layer under MultiFinder. It's better to look at these things when memory isn't being shuffled around.

VU: The 'mouse and keyboard stuff', could you be a little more specific?

Agent VU: Sure. I try to make my work indistinguishable from the actions of the real mouse and keyboard. To that end I set the same low memory locations that the mouse does and call the cursor task to pin the point. Once the point is set, I post mouse events if the new button state my boss has sent over disagrees with the current button state.

For the keyboard, I locate the driver in the ADB device table and call it with the same arguments that the ADB interrupt handler would pass it. The whole deal is wonderfully invisible. An application would really have to work to tell the difference between my input and what comes from a user manipulating one of the input devices.

VU: Mac Plus users might not agree. There is the issue of the mouse button…

Agent VU: (Growls.) Yeah, people gripe about that. Let me tell you the real story. The Mac Plus vertical blanking interrupt handler samples the state of the mouse button at approximately 1/60th of a second intervals. If the state of the button disagrees with a low memory variable, the variable is updated and a mouse event is posted. I also manipulate that variable and post events. The VBL handler and I used to fight over the state of the mouse button, causing a steady stream of mouse events. Now what I do (and this is only on the Mac Plus!) is copy the interrupt handler down out of ROM and "NOP-out" the part that checks the button. I swap this modified VBL handler in for the real one when VU sends me a message that it wants to use me. I swap the old one back in when VU sends me the lay-off notice.

VU: So it's a matter of necessity. You mentioned the ADB keyboard driver, but as we all know the Macintosh Plus has no such unified driver for you to call.

Agent VU: Tell me about it! For the Plus I wind up doing most of the work of the keyboard driver. VU sends me the ASCII and processed keycode for the keystroke, leaving me the work of posting the events.

VU: Gee, it seems like you're an industrious, upstanding piece of code after all!

Agent VU: I like to think so!

Appendix C Status Codes Returned to the MPW Shell

0 Normal Termination
1 Command Line Problem
2 Some Error Occurred
3 System Error/Resource not available
4 Compilation Error
5 Runtime Semantic Error
6 Problem Staging Test

Index