Splat!

Version 1.0

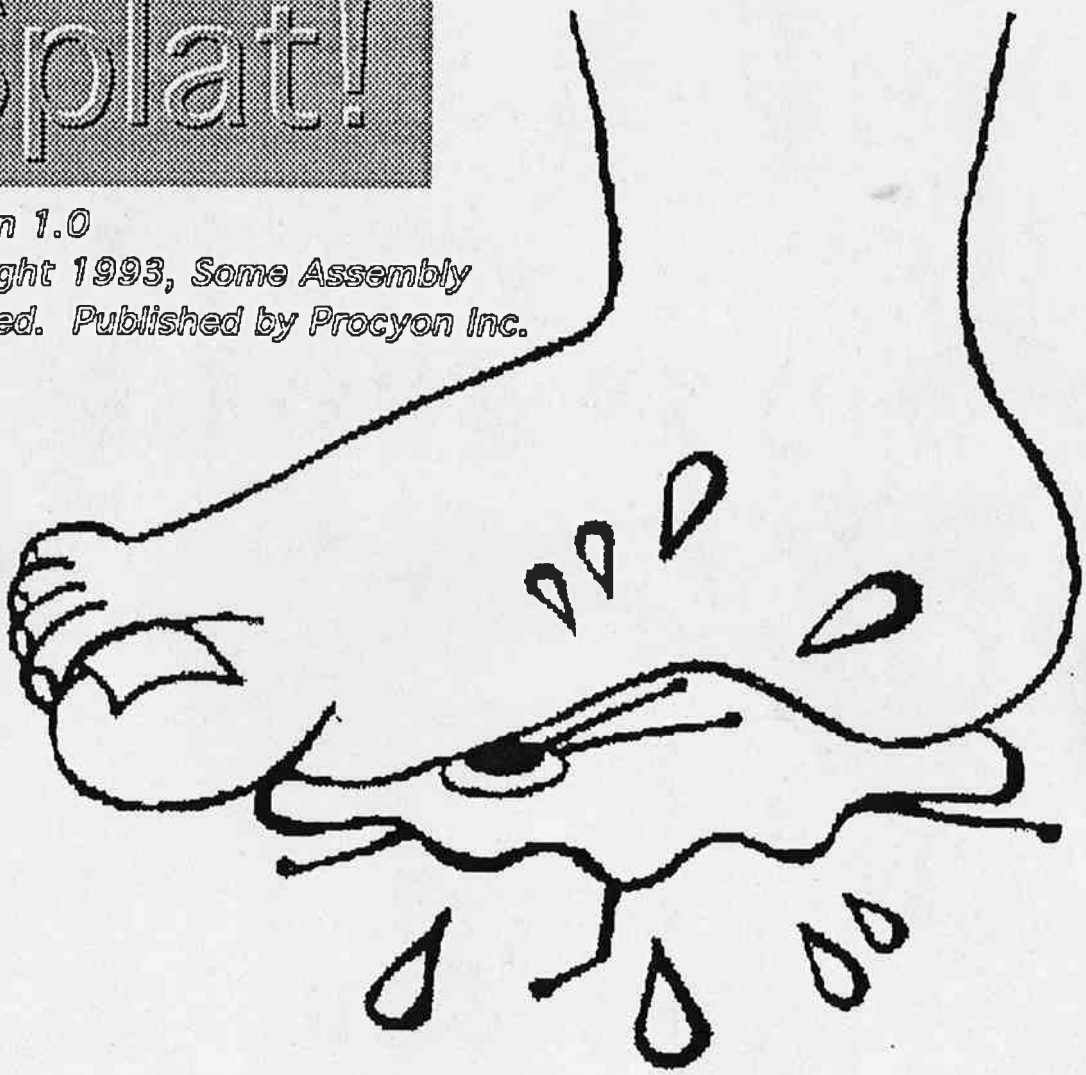

TABLE OF CONTENTS

Chapter 1: INTRODUCTION

While working on another program I was writing in ORCA/C for the IIgs, I was finding it increasingly difficult to debug this large C program with a machine-level debugger such as **GSBug**. I really needed a source-level debugger, but the only one available, Prizm, was, at the time, terribly buggy and not very good. So, **Splat!** was created to answer this need for a good source-level debugger for the IIgs. Originally, I was just going to whip together something simple for myself, but when Mike Westerfield at The ByteWorks showed interest in the program (though he eventually went with his own), I decided to go all out and make something truly impressive. This is when the text-based desktop-like interface for the program was conceived.

The complications of coordinating a desktop-based debugger working with another desktop program at the same time led me to decide that a text-based debugger would be easier to write. Also, I thought this would allow me to debug text-based programs more easily. However, the final product (or rather the one you have in your hands, as I don't think this program will ever really be *done*), is far from being a simple program. There are over 30,000 lines of code here, *all* of which is assembly code and macros. Actually, quite an extensive macro system was created (which help me greatly reduce the number of lines of code) to extend the assembler and fill in some of the important features missing in the ORCA/M assembler.

Splat! works with any language that supports the ORCA debugging information format, such as ORCA/C or ORCA/Pascal. The ORCA/M assembler, does not support source-level debugging, and **Splat!** does not (yet) handle machine-level debugging. However, it can be used in conjunction with Apple Computer's machine-level debugger, GSBug, to provide you with the best of both worlds - source-level and machine-level control.

Splat! requires roughly 128K or more memory to run comfortably, depending on the size of the program you are debugging, above what you would normally need. Also, you should be running System Software 5.0.4 or later (System Software 6.0 or later for use with GNO/ME).

This manual assumes that the reader has at least a good working knowledge of how to use their IIgs system, and that you know, or at least have a good book on, the programming language you are using. This manual does not make any attempt to teach you a programming language, nor does it discuss how to use your chosen operating shell, apart from describing how to invoke the debugger and how to prepare your programs for use with the debugger. Any technical details given are provided only for those who need them or just want to know. If you do not understand these details, or do not expect you will need to know about them, feel free to ignore them.

Also, those of you who are familiar with the graphical "desktop" interface used on the IIgs and Macintosh should be able to get into the program very quickly, and can probably just skim over a good deal of the manual. **Splat!**, though it runs on the text screen, simulates the desktop interface using the special MouseText characters available on the IIgs. With the exception of a menu bar, and the lack of mouse control, this interface is virtually identical to that available through the standard Toolbox functions. Additionally, all of the program's functions are accessible from the keyboard for those of you who prefer this method of input. At this time, this is the only form of input available, though I hope to add mouse control as well in the future.

There are a great many features this debugger *could* have had in it and many features that I would like to have included, but priorities had to be set and some features had to be set aside for possible inclusion in future versions. As I have said, this program will likely continue to grow over time, and I am always interested in hearing users' comments, criticisms and suggestions to help me to improve **Splat!** and to help me to decide which new features should be added first.

Included on your program disk are several addendum files that include up-to-date information about the debugger. The "Release.Notes" file contains information on new features added since this manual was printed, as well as anything additional that we forgot to put into this manual in time. This file will also alert you to known problems with the supported compilers and environments. Please read this over before you install or run the program. The "Bugs" file lists a history of all of the known bugs and tells you when they were fixed or if they are still outstanding. Finally, the "Future.Thoughts" file lists some of the possible features that might be added to the debugger in the future. In your feedback, you can put in your votes for the things you would most like to see, or add new ideas to the list if what you want is not listed there.

You can contact me at the address and phone number listed below, or via electronic-mail from the Internet, America Online, or AppleLink at the address also listed below. If this fails, you can also get a message to me through Procyon, whose address is also given below, following mine.

Finally, I'd like to thank Jawaid Bazyar for helping me get this program to you and for providing lots of helpful information along the way. Also, David Empson for directing me on how to use SANE, and Mike Westerfield for getting me started on the project. Finally, a *big* thanks to all of the beta testers for this project, who helped me track down many errors quickly and who provided me with plenty of good ideas on improving the program. Thanks for your time and patience.

Splat! was written by:
Michael Hackett
Some Assembly Required
102-234 Oakdale Avenue
St. Catharines, ON L2P 2K6
Canada
(905) 687-1657

email:
Internet, America Online,
AppleLink:
`Mike.Hackett@uwo.ca`

and is published by:

Procyon Enterprises, Inc.
PO Box 620334
Littleton, CO 80162-0334 USA

(303) 781-3273

email:
Internet: `bazyar@cs.uiuc.edu`
America Online: `GNO Jawaid`
GEnie: `Procyon.Inc`
Delphi: `JawaidB`

Manual written by Mike Hackett, edited by Jawaid Bazyar
Cover art of one dead bug and program icons by Lynn Hackett

Chapter 2: INSTALLATION

There are two complete versions of the debugger included in this package: a shell version and an INIT version. Both are functionally identical but each version has its advantages and disadvantages. To avoid placing any unnecessary limitations on the use of the debugger, I created both. The INIT version will likely be the more popular of the two, but the shell version is there if your situation warrants it.

The main advantage of using the INIT version is that once it is placed in the "System.Setup" folder of your boot disk or partition, it is always available no matter where you are. The shell version, as the name suggests, can only be invoked from within the ORCA, APW, or GNO shells (or some other compatible shell). Since most program development will normally take place within one of these shells, this is not often such a great inconvenience, but using the INIT version will prevent you from making the very common error of executing a program that has been compiled with debugging codes without one of the versions of the debugger running. Doing so will invariably cause the program to crash and force you to reboot the machine. In addition, with the INIT version, you can debug other INITs (both permanent and temporary), as well as CDAs, NDAs, and Control Panels (CDevs).

On the other hand, if you are somewhat cramped for memory or use the debugger only occasionally, using the shell version will leave some extra memory free that would otherwise be permanently occupied by the debugger code, even when dormant (The debugger is about 80K in size, plus 1.5K stack space). The shell version still requires the same amount of memory be available, but will only need it when you are actually using the debugger. You will just need to be careful that you do not forget to execute all programs containing debugging information explicitly through the debugger shell command, to avoid crashing the computer. Note that programs without debugging information can be run through the debugger command with no ill effects (other than that there will be less memory available), though they will not cause the debugger to become active. If the INIT version is already installed when the shell version is run, the shell version will abort and report the error. You can then reexecute the program you wish to debug *without* the debugger shell command.

Now to the details of installation. There are Installer scripts included for those of you who prefer to use them, but we will also give you details here on how to install the necessary files by hand. This may be a handy reference even to those who use the Installer scripts, just to let you know what the Installer actually did.

Firstly, as always, you should make a backup copy of the program disk, even if you are just going to be installing the program onto a hard drive and then putting the floppy away. There are so many ways for accidents to happen, so it is better to take this extra step and be safe than to damage your original program disk. Then, store the original disk in a different location from the backup disk, to protect it from stray magnets, coffee/pop spills, and other such hazards.

If you want to use the Installer scripts, insert your backup of the Splat! program disk and launch the Installer program. Click on the "Customize" button at the opening window. You will then be presented with a list of installation options. Which ones you will select depends on which version you wish to install and which shell environment you are using, if any. The chart below lists which script or scripts you need to use for your particular setup.

INIT version with no shell:
    Splat INIT

INIT version with ORCA shell:
    Splat INIT
    Splat INIT utilities for ORCA

INIT version with GNO/ME shell:
    Splat INIT
    Splat INIT utilities for GNO

Shell version with ORCA shell:
    Splat shell version for ORCA

Shell version with GNO/ME shell:
    Splat shell version for GNO

The destination for installation will vary depending on the script, so you should install only one script at a time, just to be safe. Each script tells you what the destination should be in its "Help" window - select the script and click on the "Help" button to view this.

The "Splat INIT" script should be run with your startup volume as the destination, while all of the ORCA-specific scripts should be run with your ORCA Utilities directory as the destination, and the destination directory for the GNO-specific scripts should be any directory that is in your GNO/ME search path, most likely the one in which other shell utilities are kept.

With some of the shell-specific scripts, there may be additional work that you will have to perform yourself, as the Installer program is fairly limited in what it can do. For example, it does not have the capability to modify an existing file, which is what is often needed here. When this is necessary, the script will always display its "Help" window whenever you try to "Install" it, where the additional instructions will be given. Write them down so that you will remember exactly what to do when you quit the Installer. Please make sure that you complete these instructions, as the debugger or some support utility may fail to function correctly if you do not.

You can, of course, install more than one version and choosing both the ORCA and GNO scripts for either version, even if you install them in the same directory, will not have any harmful effects. In fact the scripts for the two are very similar, but have been set up in this manner to make it easier for the user. For details on what each script actually does, continue reading the sections below. If you don't really care, at least at the moment, and you have followed all of the instructions in the Installer help windows for the scripts you have installed, then you can skip the remainder of this section and begin using the program.

If you prefer to do things by hand, or just want to know what is happening behind the scenes, you can read one or both of the following subsections for the two forms of the debugger. If you have problems after the installation, the following information may also help you to diagnose the problem and to find a cure.

INIT Version:

To install the INIT version, first copy the "**Splat.INIT**" file from the (backup) program disk into the "System.Setup" folder inside the "System" folder on your boot disk or partition. The only consideration you should make here is that the debugger must be loaded before any INITs that you might want to debug with it. So, it is probably safest to make sure it is one of the first files in the "System.Setup" folder, *after* "Tool.Setup". It is very important that "Tool.Setup" be the first INIT loaded during bootup. If you are also using GSBug in conjunction with Splat!, you can install them in any relative order, but again with GSBug, it should be one of the first INITs loaded. If you are unsure how to rearrange the order of the files in a subdirectory (folder), one way is to move all of the files out to another area and then move them back in one-by-one. Note, that using the Finder to move the files to the desktop and then back will *not* work! You must move the files to another folder or disk to remove the file's entry from the original directory.

If at any time you wish to temporarily disable the loading of Splat!, you can use the Finder (or some other file utility) to check the Inactive box (in the "Icon Info" dialog) for the "Splat.INIT" file. To remove the debugger permanently, simply delete the "Splat.INIT" file, or use the "Remove" operation in the Installer. If Splat! is still in memory at this point, you should restart (reboot) your machine as soon as possible to completely purge the program from your machine.

Additionally, there is a small shell utility and a CDA (Classic Desk Accessory) that allow you to enable and disable the debugger during a session. The shell utility, "**DebugSet**", must be placed in your ORCA/APW Utilities directory, or anywhere in your search path if you are using GNO/ME. Also, if you are using ORCA or APW, you must add the following line to your "SYSCMND" file (see your ORCA or APW manual for information about this file).

```
DEBUGSET          *U
```

Important: This line must be added whether or not you have used the Installer scripts! There is currently no way for an Installer script to modify an existing file, and since each user may have configured this file differently, we cannot simply replace the existing file.

The syntax of this command is:

```
debugset [on|off]
```

That is, to turn off the debugger, you issue the command "`debugset on`"; to turn it off, use "`debugset off`". Simply issuing "`debugset`" with no parameters will display the current setting of the debugger flag. If the INIT version is not installed, the program will simply report that and exit without any further action. Provided your shell allows aliases to be set up (and all of the ORCA, APW, and GNO/ME shells do), you can easily create aliases for turning the debugger on and off, if you like.

To use the CDA, copy the file "DebugSet.CDA" to the "Desk.Accs" folder inside the "System" folder on your boot disk or partition. It will show up as "Splat! DebugSet" in the CDA menu. (To access the CDA menu, press Apple-Control-Esc at any time - it will immediately appear unless an important operation is in progress, in which case, the menu will appear when the operation is finished.) To open the CDA, move the cursor bar to "Splat! DebugSet" with the up- and down-arrow keys, and then hit <Return>. A message will appear on the screen informing you of the current setting of the debugger flag, and then it will ask you if you wish to change the setting. Hit <Return> to accept the change, or press <Esc> to cancel the operation. If the Splat! INIT is not installed at the time you open the CDA,

the situation will be reported and you will be asked to press any key to return to the menu.

When Splat! is turned off, programs with debugging codes will not cause Splat! to become "active" - that is, the full debugging environment will not be started up, saving a great deal of overhead and allowing your program to execute much faster; faster than even the debugger's fastest execution mode. Splat! is still interpreting the debugging codes, but just skips over the information, allowing your programs to execute normally, whether or not they have been compiled with debugger information included.

Finally, there is one additional file, "**FixSplat**", that GNO/ME v1.0 users need in order to make the debugger operate correctly under this environment. This small program reconnects the debugger to the necessary system vectors after the GNO/ME kernel has disconnected it during startup. Copy this file into your utilities directory with "**DebugSet**", or into some other directory in your search path. If you are using GNO/ME v2.0 or later, you will not need this program.

You will also likely want to place an additional line that executes "FixSplat" somewhere in your "gshrc" file so that the debugger is reconnected automatically every time you start GNO/ME. FixSplat works fine even when the INIT version has not been loaded. It first checks to see if the Splat! INIT has actually been loaded before taking any actions, and does nothing if it does not find the debugger. When it is run, it normally prints a one line message indicating its action, however this can be disabled by using "FixSplat -s" if desired. "FixSplat" need only be run once after each time GNO/ME has started up, and should never be run from outside GNO/ME.

(For those of you interested in the technical details, the debugger INIT posts a MessageCenter message for use by these utilities. This is what FixSplat and the other utilities check for and use. The format of this message is not fixed and may change in future versions so do not count on its format. This message is for our use only!)

Remember, you have to reboot the system before any changes to files in the system folder will take proper effect. In this case, this means that installing the debugger INIT file does not cause the debugger to be loaded immediately. It will be loaded only during GS/OS startup. Similarly, deleting the INIT does not remove the debugger from memory until a reboot. You *can* use an Apple utility called "IR" (for "Init Restarter"), which is a Finder Extension that allows you to load and remove INITs and DAs without resetting the computer, to install the debugger from the Finder, however, it is not safe to remove Splat! with this program, and Splat! will report an error message if you attempt to load it when it is already installed.

Shell Version:

Installing the shell version of Splat! requires only slightly more effort. First you must copy the "**Splat**" program file from the program disk to your shell's utilities folder. Where this is depends on which shell you are using and how you have your setup configured.

If you are using the ORCA or APW shells, this would normally be the "Utilities" subdirectory in the same directory as the shell program itself. However, you can change the location at which the shell looks for utilities by changing the utilities prefix (17 for ORCA 2.0 and greater; 6 for APW and earlier ORCA shell versions). Also, you must add the following new line to your "SYSCMND" file (see your shell reference manual for more information about this file):

```
SPLAT              *U
```

If you are using GNO/ME, you can place the program file anywhere in your search path. However, if you have both ORCA and GNO/ME installed on the same volume and your GNO/ME search path includes the ORCA Utilities directory, then this directory would be the most logical choice for the Splat file. For more information on search paths, see the GNO/ME Shell Reference.

Chapter 3: PREPARING PROGRAMS FOR USE WITH SPLAT!

To prepare a program for use with Splat!, you compile the program as you usually would, except that you must pass a flag to the compiler indicating that you want it to create debugging information along with the code. For ORCA and APW shells, this is done with the "compile" command, and the addition of the "+d" flag. This may also be true for GNO/ME users, if you are using a "compile" utility with the same syntax as the standard ORCA/APW version. Here are some example commands:

```
compile +d mysource.c keep=$
cmpl +d -m onefile.pas keep=Test
compile +d -p +e hello.c keep=Obj/$
```

That is all there is to it. Now each source module that you have compiled with the "+d" will have debugging information included, provided, of course, that you compiler supports source-level debugging (see below). If you are using Prizm as a development environment, you should ensure that the "Generate debug code" check box in the "Compile..." dialog is checked when compiling programs for Splat!. Also, Prizm users can set breakpoints and auto-go ranges in source files with the Prizm editor and Splat! will recognize these and act accordingly. However, the Prizm markers will not be shown in the Splat! Source window. Also see the descriptions of the "Set/Clear Breakpoint" and "Set Auto-go Range" commands in Section 5 of this manual.

Note that is quite acceptable to have some source modules compiled without debugging information and some with. As soon as the program reaches a subroutine that has been compiled with debugging information, the debugger will be activated and you can begin tracing there. Any subroutines that this code calls that have not been compiled with the required information will not be traceable. These will just be executed at full machine-speed as if they were, for example, a system library call.

There are no other standard shells or ways of compiling programs at the time of this writing. However, should future new shells or new compilation commands appear, you will have to check the manual or help files of those shells for information on how to properly prepare your programs for debugging.

At this time, the only languages that provide the information necessary for source-level debugging are ORCA/C and ORCA/Pascal. Check the release notes that came on your program disk for updates to this list.

Splat! allocates and uses its own stack space, so there is no need to allocate extra stack space for it. This also allows Splat! to debug CDAs, for example, which are limited to 1 page of stack space of their own, unless they allocate more.

Important: Do not try to debug code in dynamic segments under Splat!. The debugger keeps pointers to certain structures embedded in your program code and it cannot handle having these structures moved to another location (or being removed completely) in mid-program.

For finer control over which parts of your program are to be debugged, you can use bit 1 of the options code for the ORCA/C "`#pragma debug`" command to enable and disable the creation of debugging information within a source file. If set, debugging information for Splat! will be generated from that point on, up until another "`#pragma debug`" command clears the flag. If cleared, the code following will not containing debugger codes. This can be tremendously handy when used to avoid routines that would take the debugger a very long time to execute, such as code with long loops. Once this code has been debugged, it is then safe to disable debugging code generation for the routine,

allowing it to be executed at full speed with no debugger intervention, as if it were a standard library routine. It is important to note that in the debugger command descriptions in Section 5, "full speed" refers to full debugging speed - this is *not* full machine speed. There is a significant amount of overhead involved in interpreting the debugging instructions embedded in the program's code which must be handled regardless of whether the debugger's interface screen is being updated or not. As a result, program execution speed is *significantly* reduced whenever debugging instructions are included in the code. Therefore, disabling debugging code generation can greatly improve the speed at which your program runs and thus reduce your development time.

Depending on the other options selected, the "`#pragma debug`" directive can also cause additional code to be created to do various run-time checks on your code. You can use this directive in conjunction with the "compile" command "+d" option, but you can also use these features without compiling the code for source-level debugging (i.e. without the "+d" flag). However, if you turn on bit 1, you must run the program through a source level debugger, otherwise it will crash upon reaching the first debugging information block. Note that you cannot change "`#pragma debug`" options in the middle of a subroutine.

Some of the other debugging features provided by the ORCA/C and ORCA/Pascal compilers can still be quite useful even within Splat!. For ORCA/C, bit 0 of the "`#pragma debug`" parameter controls the generation of range checking code. In ORCA/Pascal range checking can be enabled with the "`{$RangeCheck}`" directive. This is one of the features that Splat! cannot adequately provide, as the compilers do not generate enough information for debuggers to handle range checking themselves. The code generated by the C compiler (at the time of this writing) is limited to checking, at the start of each function, that there is enough room left on the program's run-time stack for all of the function's local variables. As such, it is of somewhat limited value, but still can be an important test. On the other hand, the Pascal compiler generates a multitude of checks, from the same stack overflow checking described above, to array bounds checking, math overflow errors, the use of NIL pointers, and more. See you ORCA/Pascal manual for details.

Another feature useful in debugging C programs is the stack frame checking code enabled by bit 4 of the "`#pragma debug`" parameter. The code generated here will check that the correct amount of data is passed to a subroutine and report an error if too little or too much is passed. C programs are particularly prone to these types of errors for two reasons. The first has been somewhat solved with the ANSI C specification which provides for function prototypes which describe exactly the size and types of parameters expected by a subroutine. If you are programming in ANSI C and are using prototypes, you can use the C compiler's "`#pragma lint`" directive to have the compiler check function parameters for you. However, many older programs do not use prototypes and are prone to frame errors. The other place where problems can occur is in functions which accept a variable number of parameters (e.g. `printf`). A very common error is to pass a long integer value to printf when the format string specifies that a short integer is expected, or vice-versa. Pascal, as it has much more stringent subroutine parameter type checking and does not allow subroutines with a variable number of parameters, has no need for such checking and there is no equivalent directive for that compiler.

Less useful, when working under Splat!, is the trace-back feature that both compilers provide. In C it is enabled by setting bit 3 of the "`#pragma debug`" parameter, and in Pascal you would use the "`{$Names}`" directive to enable the feature. The trace-back code keeps track of the subroutine execution stack and provides this information to the previously described run-time error checking routines to allow them to describe where the error occurred and the sequence of calls leading up to error. Under Splat! this feature is not really needed, as you can see which line is executing in the debugger's Source window and you can examine the program's call stack with the Subroutine Call Stack command. However, if you are debugging programs outside of Splat!, this is an extremely valuable feature.

Finally, it should be mentioned that the profiler flag in the C compiler's "`#pragma debug`" parameter (bit 2) does not need to be set to do profiling under Splat!. Profiling information is always kept by Splat! with no real performance penalty, so this flag should never be used when preparing programs for use with Splat!.

For more information about the ORCA/C "`#pragma debug`" directive, see the chapter on "The Preprocessor" in you ORCA/C manual. Information on the ORCA/Pascal directives mentioned here can be found in the two "Compiler Directive" chapters of the ORCA/Pascal manual.

Chapter 4: RUNNING SPLAT!

How you go about invoking Splat! very much depends on which version you are using, however, either way it is very simple. If you are running GNO/ME v1.0 and the INIT version of Splat!, you must first execute the "FixSplat" utility to rehook the debugger to the necessary places. If you followed the instructions in the Installation section and added this command to your "gshrc" file so that it would be executed when the shell starts up, there is no need to run this again. However, if unsure, it is perfectly safe to run the program any number of times. It checks first to see if the debugger is properly installed, and only if there is a problem will it go through the reinstallation process.

Regardless of which shell you are running, if you are using the INIT version, you must first make sure that the debugger is set to become active when required. By default it is, but if you want to check, and you are running any of the supported shells, run "DebugSet" with no parameters. It will report the current setting of the debugger flag. If it is "on", Splat! will become active as soon as it detects a debugger instruction being executed. If it is "off", Splat! will still interpret debugger instructions, to allow programs to run without crashing, but will not become active. This mode is useful when you want to run your program a little more quickly than what is possible even under the debugger's Go mode, but you do not want to recompile your entire program to remove all of the debugging instructions.

If the debugger is set to be "quiet", you can enable it by entering the command "`debugset on`" at the command line of your shell. To disable it again, enter "`debugset off`".

Alternatively, whether you have the ORCA or GNO/ME shell or not, you can use the DebugSet CDA to enable and disable Splat!, or determine its status, at any time. Any changes will not take effect until the next time the debugger is invoked, so you cannot use the CDA to turn off the debugger while it's in the middle of running a program.

Once you are sure that the debugger is installed and enabled, just execute your program (prepared as described in Chapter 3) normally, either from a shell or any program launcher, such as the Finder. As soon as the first debugging code is reached in the program, Splat! will become active and the first program line will be highlighted. If your program runs but the debugger does not become active, either the debugger is disabled or you have not prepared your program correctly. If the machine crashes, you have probably not installed the debugger correctly. Assuming everything is alright, you are now ready to run your program under Splat! - proceed to Section 5, the debugger Command Reference.

If you are using the shell version of Splat!, there is no set up required. Just get into your shell program of choice and launch your program as you usually would except that you must prefix the command with the name of the Splat! shell-version program file, normally "`Splat`". So, for example, to run a program called "dummy" with a couple of command line arguments, you might type:

```
splat dummy -a -o test.obj
```

You can also redirect input or output to the program being debugged - just add these indications to the command line as usual. Running the debugger on a remote terminal under GNO may be possible at some future time, but for now the debugger always takes its input from the local keyboard and uses the local screen for output, so all redirections will be taken with respect to the program being debugged, not the debugger.

When using the shell version, you must be very careful not to forget to prefix your execution command with the debugger's command name (i.e. "`Splat`", unless renamed by the user). If omitted,

the machine will almost invariably crash immediately. That is the main reason why the INIT version is recommended over the shell version unless your situation prohibits its use.

Chapter 5: COMMAND REFERENCE

This section describes all of Splat's commands in detail, grouped by the type of function they perform. A command summary page is given in Appendix A which would make a handy quick reference sheet as well, although the program's help window also contains a complete list of commands. The summary, though, also acts as a index of sorts, as the commands are listed in the order that they are described in this section.

Before I get into the commands, however, an overview of the program's interface and a description of some of the common features that you will encounter is in order.

Program interface

Splat! uses a text-based desktop-like user interface, using MouseText characters on the 80-column screen. If you are familiar with Apple's standard graphical interface for the GS, then you should have no problem figuring out the debugger's interface. You cannot, at this time, however, use the mouse to manipulate the debugger. This may change in future, but you will always be able to use every feature of the debugger from the keyboard.

There are normally three windows on the main debugger screen, and any commands which require further input or which provide you with information use standard dialog boxes with buttons, check boxes, list boxes, and line edit items. Most of what you know about the desktop interface will also be applicable in this program, although there are a few things missing. Most notable are the lack of mouse control and menu bar, and the fact that windows cannot be moved or re-sized. In its favour, however, Splat! allows you to access all of its features from the keyboard.

The debugger's main three windows, and their functions, are as follows:

- a **Source** window, displaying the current file, normally with the next line to be executed centered in the window. It can also be used to view any other text or source file you wish.
- a **Variables** window, listing the current subroutine's local variables, as well as those global variables accessed by the routine, and their values. Pointers to simple variables also have the values they point to following the pointer's value.
- an **Output** window, allowing you to view the program's text output without flipping back and forth between the debugger display and the 80-column text screen.

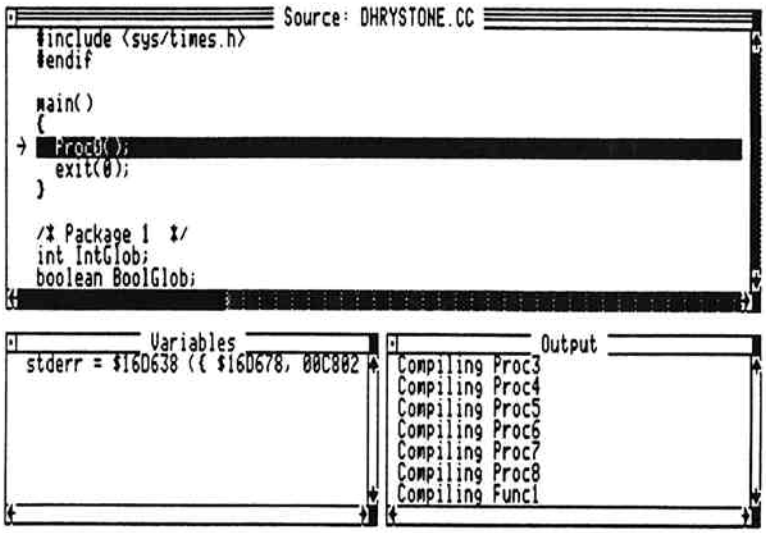

Figure 1 - Splat! main screen

As described later, it is possible to remove the Output window to allow for a wider Variables window, or two separate variables windows; one for local variables and one for global.

Data value formats

When Splat! displays the contents of a variable, either in the Variables window, in the structure view dialog, or when placing a default value in an edit box, it uses certain standard formats. Also, when accepting input, it expects values to be entered in certain formats. There are several types of data that the debugger recognizes (integers, pointers, floating-point values, characters, and strings), and the input and output formats for each of these types are described below.

For output, Splat! prints all integer values in decimal format, and all pointer values in hexadecimal format. However, when accepting these values, the debugger recognizes several special character prefixes to indicate a non-decimal base for the value. These symbols are the same ones used by the ORCA/M assembler, which are '$' for hexadecimal, '@' for octal, and '%' for binary. If there is no prefix, the number is assumed to be a decimal (base 10) number.

Floating point values are always output with 6 decimal places of accuracy, but input values can be of any precision and will be rounded by the conversion routine if the value cannot accurately be stored in the destination variable. You can also specify a power of 10 multiplier in the standard 'E' representation (e.g. 1.23e10 = 12300000000) in an input value.

For characters and strings, Splat! uses the standard C escape sequence conventions to indicate control characters, both in input and output strings. For those of you not familiar with the C conventions, the table below lists all of the special escape sequences. Each sequence begins with the backslash character ('\'), which is then followed by a single letter from the table, or, if no letter is assigned to the code, you can encode a one to three digit octal number or an 'x' followed by a one or two digit hexadecimal number. Any other character following the backslash will be included as is. Note that to code a backslash character into a string, you must put two backslashes together ("\\"). This is used in output strings so do not be alarmed if you see backslashes doubled in a variables listed value - there is really only one there in memory. Note that, to avoid getting caught by an un-terminated or very long string, Splat! only shows the first 40 characters of a C-style string. Other string formats that include a length byte or word (such as Pascal strings or GS/OS Class 1 strings) are not trimmed in this manner; however, the limits of a window or control may prevent the entire string from being seen.

Sequence code	ASCII code	Meaning
\a	7	alarm bell
\b	8	backspace
\t	9	tab
\n	10	line feed (newline)
\v	11	vertical tab
\f	12	form feed (clear screen)
\n	13	carriage return

All of the debugger's commands are now described, listed in what I felt was a fairly logical order, divided into several groups of similar commands. Each command description begins with a line printed in bold that lists, on the left, the key that invokes the command and, on the right edge, the name of the command. The detailed description of the command is then given in the paragraph(s) that follow. In the cases where the command is invoked with a Apple key sequence (e.g. Apple-A), you can always substitute the Control key for the Apple key (e.g. use Control-A instead). This was done to allow more flexibility to users in selecting "pass-through modifier keys" (see Set Preferences below). If you need to use the Control key as a pass-through modifier, you can still invoke these commands using the Apple key combination, and vice-versa.

Window manipulation commands

The following commands allow you to make full use of the debugger's desktop-like interface.

<Tab> **Select Next Window**

The currently active window is indicated by having its title bar highlighted. This command cycles the active window indicator forward (in left-to-right, top-to-bottom order) through the windows. Specifically, in a standard configuration, if the Source window is active, the Variables window will be made active by this command; if the Variables window is active, the Output window will be activated; and, finally, if the Output window is active, the Source window will be reactivated.

Apple-<Tab> **Select Previous Window**

This command performs the opposite function to the previously described command, changing the active window in the reverse direction. That is, Source to Output, Output to Variables, and Variables to Source.

Up-arrow — **Move Up**
Down-arrow — **Move Down**

The exact function these keys provide varies depending on the currently active window. If the Output window is active, the up-arrow key scrolls the window's view of its contents up one line, and the down-arrow scrolls the window down one line, to the limits of the data being viewed (which, in this case, is the size of the 80-column screen - 80 by 24 characters). For any of the other windows, these commands move the selection arrow (shown on the left edge of the window) up or down one line, respectively, scrolling the window's contents if necessary.

Left-arrow — **Move Left**
Right-arrow — **Move Right**

In all windows, these two keys perform the same function: to scroll the window's view one column to the left or right respectively, up to the edges of the data shown.

Apple-Up-arrow — **Page Up**
Apple-Down-arrow — **Page Down**

As with the Move Up and Move Down commands, this pair of commands function differently for the Source and Variables windows than for the Output window. In either of the first two, Page Up will move the cursor to the top line of the window, leaving the contents unchanged, unless the cursor is already on the top line, in which case the contents of the window will be scrolled down a windowful and the cursor left on the new top line. The effect of the Page Down command is the same, in the downward direction, moving the cursor to the bottom line or scrolling the data up one windowful.

In the Output window, the paging commands always scrolls the window's contents one windowful, up to the limits of the data displayed.

Apple-Left-arrow — **Page Left**
Apple-Right-arrow — **Page Right**

Again, as with the Move Left and Move Right commands, these two keys perform the same function in all windows. They scroll the window's view one windowful to the left or right respectively, up to the edges of the data shown.

Apple-1 to Apple-9 — **Jump To Part**

Pressing the Apple (or Control) key with any of the number keys (either those across the top of the keyboard or those on the keypad) causes the active window to "jump" to the start of that numbered part of that window's contents. The contents are divided into 8 parts, so **Apple-1** jumps to the start of the contents, **Apple-2** jumps to the start of the second division, and so on up to **Apple-8** which jumps to the start of the eighth and final part. **Apple-9** jumps to the end of the window's contents. This set of

commands will likely only be really useful in the Source window, however, the **Apple-1** and **Apple-9** commands, to jump to the start and end of a window's contents respectively, may be useful in any window.

J — Jump To Line

This command presents you with a dialog which requests that you enter the number of the line you wish to jump to. To abort the command and remain where you are, hit the "Cancel" button (selected by hitting the **<Esc>** key). To jump to a specific line, enter a new number and hit "OK" (selected by hitting **<Return>**). The first line of any window's contents is numbered 1. Again, I expect this command to be most useful when used with the Source window, as you will be able to use the line numbers from your source file editor to get to those same lines while in the debugger. Note that there is one difference between the line numbering used in the ORCA/Editor versus what the ORCA compilers, and hence Splat!, use. Form feed characters (ASCII 12, or Ctrl-L), often used in C programs to help break up paper printouts, are counted by the compilers and Splat! as ending a line, and thus incrementing the line number counter. The ORCA/Editor (as of version 2.0) does not. Be aware of this discrepancy when trying to match line numbers between the debugger and the ORCA/Editor.

This command can also be used to determine the number of the line which is currently selected in the active window. When the dialog is brought up, the current line number is placed in the editing box as a default. You can then check the line number and hit "Cancel" to abort the jump.

Apple-V — Change Debugger Window Configuration

This command changes the debugger's window configuration. It cycles through three available configurations: the standard one (shown above), and two variations of the variables window. The first of these is a Variables window that takes up the whole bottom of the screen; the Output window disappears. The second is two variables windows, one for locals and one for globals. The Source window remains the same size in all these configurations. Note that the globals window does not always work appropriately due to bugs in the compilers.

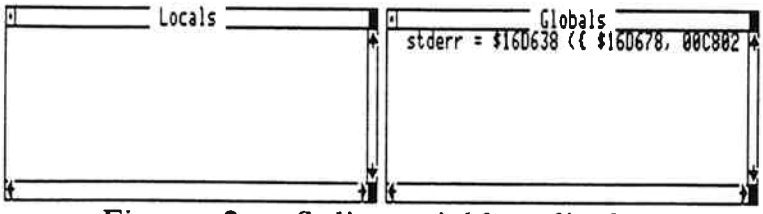

Figure 2 - Split variables display

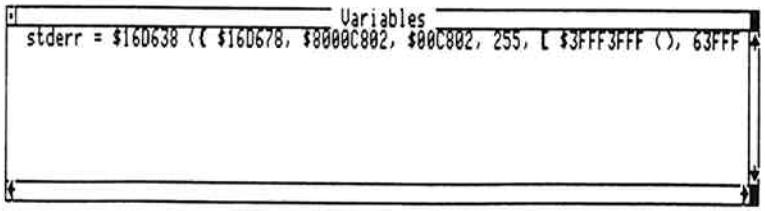

Figure 3 - Variables only

Execution commands

<Space> **Step (using default stepping mode)**

Once the default stepping mode has been set to your liking, using the space bar to step through the program is usually the easiest way of following important parts of the code. Initially, the default mode is to step into all subroutines encountered, whenever that subroutine has also been compiled with the necessary debugging codes. To check or to change the stepping mode, use the "Set Preferences" (**Apple-P**) command (described later under "Miscellaneous commands").

See the descriptions of the following two commands for the differences between the two stepping modes. Note that if the line to be executed has no traceable function calls, all of the Step commands perform identically. (A traceable function is one that has been compiled with the necessary debugging information included.)

I **Step Into Subroutines**

This command is used to force the debugger to follow execution into any subroutines encountered on the next line of code. That is, if the next line of code executed (the one highlighted in the Source window) contains any calls to other subroutines also compiled with debugging code generation turned on, execution will halt again at the first line of that subroutine, allowing you to trace through it. If the subroutine called was not compiled with debugging information included, such as for a standard library call, it will not be traced.

If the line contains more than one function call, all function calls are traced. However, upon entering any subroutine that you are not interested in following, you can use the "Return" (**R**) command (see below) to execute the function at full speed until it returns to the caller.

O **Step Over Subroutines**

If you do not wish to trace the subroutine calls that the next line makes, use this command to force the debugger to execute all subroutines at full speed. At any point during their execution, however, you can still interrupt them to see what is happening. This can be particularly handy if a subroutine requires rescuing after it enters an endless loop or encounters some other problem.

If the line contains more than a single subroutine call, all are executed at full speed - you cannot selectively step over only some of the calls.

T **Trace**

Trace mode automatically steps through the program a line at a time, using the current stepping mode (that is, it uses the stepping mode to determine whether to follow subroutine calls or not). Pressing **<ESC>** pauses execution again.

One good application for this command is to use it to skip ahead just a few lines without having to repeatedly use one of the Step commands or setting a breakpoint and using the Go command. Tracing moves slowly enough that you should be able to stop execution where you want to just by watching the executed source lines being highlighted on the screen. It might also be desirable to follow a loop using Trace mode while you watch the values of variables being updated.

G **Go**

Using the Go command sets the debugger to begin executing code at its fastest rate, by switching to the program's current video mode. Even if you return to the debugger screen, the Source and Variables windows will not be updated while the code is executing. Full speed execution will continue until the **<ESC>** key is hit or a breakpoint is reached (see the Set Breakpoint command below).

The video mode selected will be the last one that was active before the debugger screen was last turned on. If this is not the correct mode, you can switch to another mode with the appropriate key (see later section on display mode commands).

The speed of execution under the Go command is still nowhere near full machine execution speed. The debugger still has to do a great deal of work to keep track of the executing program's activities, regardless of whether it reports these activities immediately or not. However, because the Go command suspends the maintenance of the debugger's interface, execution speed is significantly better than under Trace mode.

R **Return**

The Return command causes the rest of the current subroutine to be executed at full speed, in a fashion similar to the Go command except that the debugger display remains visible, though the Source and Variables windows still do not get updated. Execution halts again at the next line in the subroutine that called the current one, unless a breakpoint is encountered or the **<ESC>** key is used to pause execution before this point is reached.

Program control commands

L **Load Source File**

The source window is not limited to displaying only the file in which the current line appears. Using this command, you can load any text or source file into the Source window for viewing. However, as soon as you use one of the execution commands (see above), the current source file is reloaded into the window.

To select the file to load, a dialog similar to the graphical Standard File (SF) dialog used in most GS/OS-based programs, with which you should be acquainted, is presented. The current path is shown at the top of the dialog, as well as the list of selectable files and the usual buttons. Only text and source files (types TXT ($04) and SRC ($B0)) and subdirectories are listed, the latter being marked with a folder icon to distinguish them. One notable difference between this dialog and the SF dialog is that Splat!'s dialog does not keep checking AppleShare volumes and does not notice when ejectable media (such as 3.5" disks) have been changed.

Otherwise, this dialog is virtually identical functionally to the Standard File dialog. The up- and down-arrows move the selection bar up and down, and additionally, holding the Apple key down with the arrow key either moves the selection bar to the top or bottom of the displayed list or causes the list to scroll one windowful in the appropriate direction. Only one file can be selected at a time.

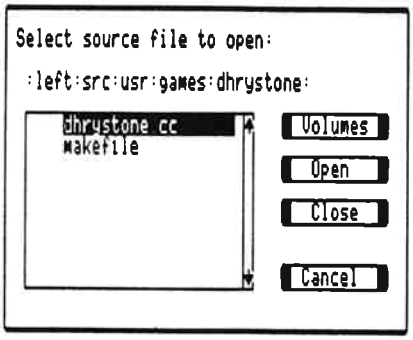

Figure 4 - Standard File Dialog

Working from top to bottom, the buttons and their keyboard equivalents are as follows:

The **Volumes** button (selected by pressing the **<Tab>** key) brings up the current list of volumes available online. You can then select a new volume from the list. Note that this listing does not get updated when you insert a new disk, unlike the SF dialog. However, you can hit the Volumes button again to manually force an update.

The **Open** button (**<Return>** key) performs one of two functions depending on the type of the file selected. If the file selected is a subdirectory or volume name, the name of the selected directory or volume is added to the current directory pathname and the new directory is listed. If the selected file is a text or source file, that file is loaded and displayed in the Source window.

The **Close** button (**<Delete>** key) removes the last subdirectory from the current directory pathname and displays the contents of the parent directory. If the directory path consisted only of a volume name, closing will bring up the Volumes list described above.

Finally, the **Cancel** button (**<Esc>** key) aborts the load command and removes the dialog without further action.

The load commands use prefix 8 to hold the current directory pathname.

A **Set Auto-go Range**

If there is a section of your code that you want to repeatedly run at full speed, you may find it convenient to set up an auto-go range for this section. This would be primarily useful on a piece of code inside a larger loop; for blocks that will be executed only once or twice, it would be simpler to just set a breakpoint at the end of the block. This command is only available when the Source window is active.

The initial position of the cursor when the command is given is used as the anchor point for one end of the range, and you can set the other end using the up- and down-arrow keys (with the Apple key for "page" moves). As you set the range, the selected lines are marked by a highlighting bar in the window border to the left of the text.

Hitting **<Return>** accepts the range, while pressing **<Esc>** cancels the operation. If the new range overlaps or lies adjacent to another range, the ranges are merged into one. The only visible implication of this is that you will be unable to remove just the new range - you will instead remove the entire combined range.

Lines which are marked as part of an auto-go range will have a vertical bar in the left-hand margin of the Source window beside the line. You can easily see the extent of the range by the extent of the continuous vertical margin line.

When execution reaches any line in an auto-go range, the line is executed at full speed (see the description of the Go and Return commands above) regardless of the current execution mode. You can, of course, interrupt execution with the **<ESC>** key while the program is in an auto-go range.

Note, also, that you can set a maximum of 16 auto-go ranges in each source file. However, because of their limited usefulness, this is not likely to be a serious limitation.

For Prizm users: During execution, Splat! does recognize lines marked in the Prizm editor as auto-go and will execute them as such, however, the lines will not be marked in any special way in the Source window. In fact, using Prizm's auto-go and breakpoint markers may actually be a more convenient way of setting up these markers as you will not have to reset them every time you reenter the debugger.

Apple-A **Remove Auto-go Range**

To remove an auto-go range, simply move the cursor somewhere into the range and select this command. This command is also only available when the Source window is active.

B **Set/Clear Breakpoint**

Often, when debugging a large or fairly large program, you will want to quickly get to a certain point in the program to test a new piece of code or to locate the source of an error. Setting a breakpoint at or just before the important section of code will let you do this very easily.

To set a breakpoint, simply move the cursor to the desired line, loading the appropriate file into the Source window first, if necessary, and select the Set/Clear Breakpoint command. You will see that the

line is now marked with a diamond in the left margin. To remove the breakpoint, place the cursor back on the same line and choose the Set/Clear Breakpoint command again. As with auto-go ranges, you may have a maximum of 16 breakpoints set in any one source file.

Whenever the executing program reaches a line marked with a breakpoint, regardless of execution mode, the debugger will halt execution and sound a beep. Note that breakpoints have precedence over auto-go ranges so any breakpoints hit will still cause execution to halt. As a reminder of this, the diamonds that mark lines with breakpoints set on them show through the vertical line that marks an auto-go range.

For Prizm users: During execution, Splat! does recognize lines marked in the Prizm editor as breakpoints and will stop upon reaching the line just as if the breakpoint had been set within Splat!. However, the lines will not be marked in any special way in the Source window. In fact, using Prizm's auto-go and breakpoint markers may actually be a more convenient way of setting up these markers as you will not have to reset them every time you reenter the debugger.

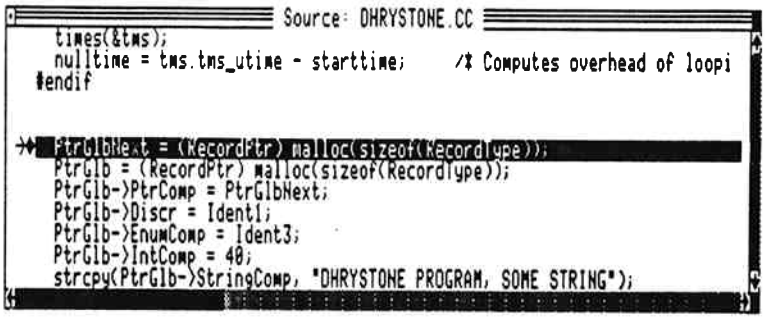

Figure 5 - Debug breakpoint marker

S **List Subroutine Call Stack**

This command brings up a list of the active function invocations in order from most recent to least recent; in other words, the program stack. Each line lists the subroutine's name as well as the source file and the line number from which the call was made. Then, if you want to view one of these calling lines, you can move the selection bar to the desired call and select the "Open" button ('**O**' key). This will load the appropriate source file into the Source window and display the line from which the call was made. To remove the list without changing your current position in the Source window, use the "OK" button (**<Return>** key).

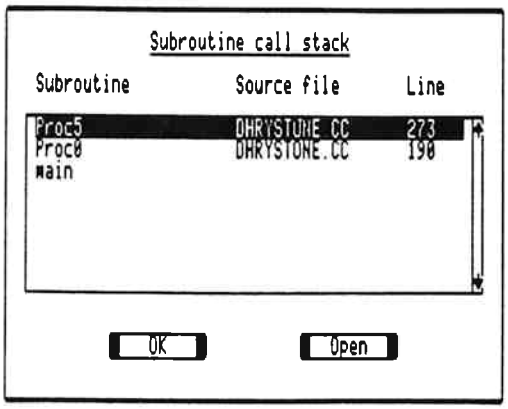

Figure 6 - Subroutine call stack window

= **Assign Value**

This command allows you to change the value of any simple variable (that is, one that is not a structure or an array). This command is only available when one of the variables windows is active and the selection arrow is pointing to a simple variable. The current value is placed in the editing box and you can use all of the standard line edit features to edit the value. See the information on editing and on value formats at the start of the "Command Reference" section. A word of caution: when changing the value of strings, be careful not to overflow the space allocated for the string - no check is made by the debugger.

To change the value of a structure field or array element, you must first expand the structure or array with the View Structure command - see below.

<Return> **View Structure**

The View Structure command allows you to view a structure or array variable in an expanded format, as opposed to the single line, condensed form shown in the variables window. A dialog is brought up which includes a large list box and four buttons along the bottom. In the list box, each field or array element (which I will hereafter group together as an element) is displayed on a separate line, with the field name or element number on the left and the element's contents on the right. You can use the standard list box keys to scroll the contents if there are more elements than can be displayed at once.

When you first invoke the command on a variable, its first (topmost) "level" of elements will be displayed. Any elements which are also structures or arrays will be displayed as in the Variables window, and can be further expanded by moving the selection bar to that element's line and clicking the "Expand" button (selected by pressing the **<Return>** key). To back up to a previous (higher) level of the structure, use the "Backup" button (**<Delete>** key). To finish viewing a structure, select the "Done" button (**<Esc>** key). Notice the similarity to the file selection dialog - the buttons have different names, but the keys perform similar functions in both situations.

The other button (the third in the row) is the "Assign" button ('=' key) which performs identically to the Assignment command described above, but allows you to change the value of any *simple* structure field or array element. You simply move the selection bar to the appropriate line and select assign, which brings up the same dialog box as described in the Assignment command above (please refer to that description for details). Non-simple elements must be further expanded before they can be changed.

Structure and array expansions can be intermixed freely, so you can expand the elements of an array of records, for example, and structures can have fields which are arrays of any number of dimensions, etc. Pointers (or pointers to pointers, etc.) to structures or arrays can be expanded as well, being automatically dereferenced to get to the structure data. Objects, too, can be viewed in this manner. Only simple variables and pointers to simple types are non-expandable, as they are already in their most basic form.

After you have assigned a template to a pointer variable (see the Template commands section below), you can also then use the View Structure command to view the data at the address pointed to by the variable in the format given in the structure template. As before, the Variables window must be active

and the currently selected variable must be a pointer, and you must have already assigned a template to the variable. Also, the template you assigned must be currently loaded into memory. Finally, the current value of the pointer must represent a possibly valid value, and any values out of the valid memory range of the IIgs or which point to "dangerous" areas of memory, such as the I/O softswitches, will be refused. This final restriction also applies in the above case where pointers to structures are expanded. No expansions will take place if the debugger determines that data may overlap an invalid memory region. However, the debugger cannot always determine the size of a structure or variable, and so may expand something even though it is partially in a dangerous area. The effect varies, but may cause the machine to crash, so remember to use caution when expanding pointer variables.

For convenience, character pointers and byte pointers have a default template assigned to them which allows you to view the value pointed to as a C-style (null terminated) string. Of course, this default template can be overridden with an actual template assignment.

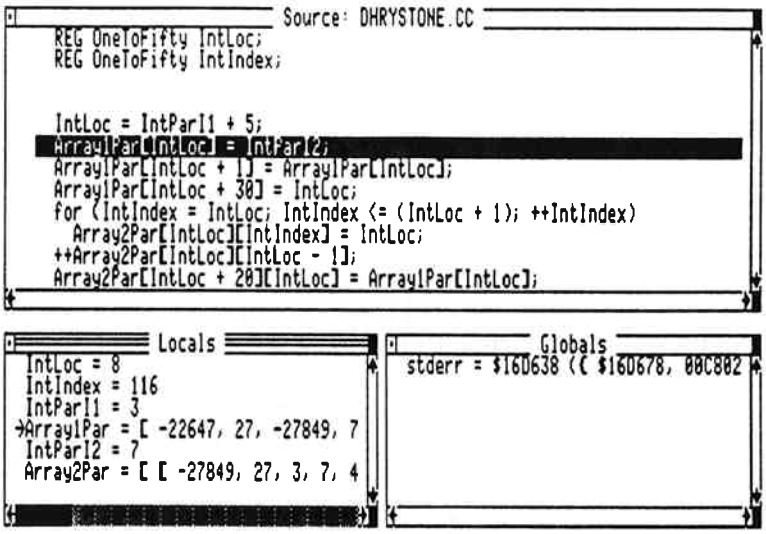

Figure 7 - Unexpanded array

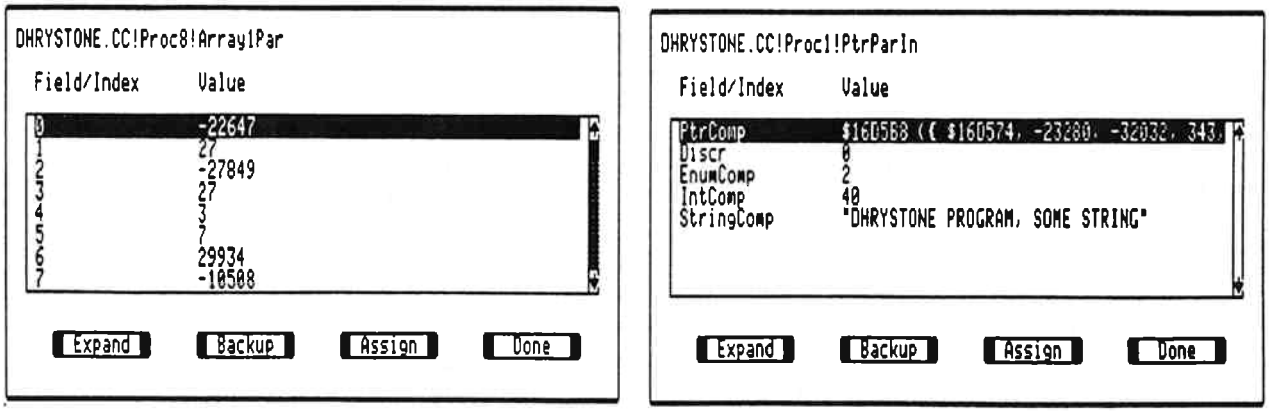

Figure 8 - Expanded array and structure

Template commands

Apple-L **Load Template File**

Selecting this command brings up a file selection dialog as described earlier under the "Load Source File" command. From this you can select a new template file to load. If you cancel the selection, nothing happens. However, if you Open a new file, all of the currently loaded templates are purged from memory to be replaced by the ones in the new file.

When Splat! starts running a new program it checks the program's directory (prefix 9) for two files, "**Splat.tmpl**" and "**Splat.asgn**", loading them automatically if they are found. "**Splat.tmpl**" should contain any structure templates that you will be commonly using with this program, and "**Splat.asgn**" should contain template-to-variable assignments. This latter file can be created using the Write Template Assignments command (described below) or manually using the format described in Appendix B.

Template files have been made practically obsolete by the addition of structure information in the debugging code generated by the newer ORCA languages with version numbers of 2.0 or greater. If you are still running a pre-2.0 version of an ORCA language, then templates may be a satisfactory alternative. However, the amount of effort involved in creating template and assignment files for each of your programs may be enough that buying the upgrades would be worthwhile. Templates can also be used to provide work-arounds for any problems or inadequacies still present in the new compilers, so they may still find use even for those with the upgraded compilers.

For more information on template files see the GSBug manual and also Appendix B in this manual describing our extensions to the format.

V Assign Template

Once you have a set of templates loaded, you can then assign the templates to variables to view the data stored there, or pointed to by that variable, in structure format. To use this command, the Variables window must be active and a pointer variable must be selected. When the command is invoked, a dialog window is presented containing the list of available templates from which you may choose the template to assign.

Along the top of the window, the variable's complete identifier is shown, which consists of the source file's name, followed by the function's name, followed by the variable's name all separated by exclamation points ('!' - e.g. "source.c!main!name"). This name is used to differentiate local variables with the same name but from different functions. When saving your assignments to disk, this name (with space separators instead of exclamation points) is stored in the file (e.g. "source.c main name"). Unless you intend to make your own assignments file by hand or write an automatic generation program, you need not really be concerned with this format. For global variables, the source and function name fields will be empty (e.g. "!!global") and these fields are replaced in the assignment file by an asterisk ('*') followed by just the variable name (e.g. '* global').

The list box is pretty standard, but you should note that the templates are listed in the order that they were loaded from the template file, instead of being sorted alphabetically. This was done so that you would have control over the order, thus allowing you to place your most frequently used templates at the start of the list. Also it allows you to group templates as you wish, such as is done in the GSBug standard Toolbox templates file, "GSBug.Templates", which, by the way, can be loaded and used by Splat!.

Figure 9 - Assign Template dialog

Once you have selected the desired template, select "OK" (**<Return>** key) to confirm the selection. To abort the selection, use the "Cancel" button (**<Esc>** key).

W **Write Template Assignments**

Once you have gone to all the trouble of setting up a bunch of template assignments, you will probably not want to have to go through the same procedure again every time you run the program, especially if you will be working on the project for a while. Well, you don't have to. Splat! allows you to write all of your currently defined template-to-variable assignments to a file, named "Splat.asgn", which will be loaded automatically every time you start the debugger from the directory holding the file. Also, you can edit this file by hand, though this is not recommended and is not likely to be necessary. However, should you feel the need, the file format is defined in Appendix B.

Apple-S **Change Variable Type to C-string Pointer**
Apple-C **Change Variable Type to Character Pointer**
Apple-B **Change Variable Type to Byte Pointer**

These commands allow you to change the way certain pointer variables are displayed in the variable windows (as opposed to templates which are used with the View Structure dialog). Because ORCA/C cannot determine whether you intended a variable defined as `"char **` to represent a pointer to a character or to a string, it always takes the safe road and calls them character pointers in the debugger's symbol table. As a result, even if the variable does point to a string, Splat! normally only displays the first character of this string in the variable windows. To work around this, Splat! provides two methods of viewing character (or byte) pointers as string pointers.

Firstly, there is in effect a default template assigned to all character and byte pointers. This allows you to view the string pointed to using the View Structure (<Return>) command, as described above. However, this is not very convenient when, for example, you want to watch the contents of a string change as you trace through the code. For those instances where you want to be able to see the contents of a string pointer in one of the variable windows, you can use the Apple-S command to change the variable's type, for the duration of the program's execution, to a C-string pointer.

To use these commands, select the variable to change in one of the variable windows, and then choose the appropriate command for the type of pointer you would like. To change the pointer to a C-string pointer, use Apple-S. To change the variable back to a character or byte pointer, use the Apple-C or

Apple-B commands, respectively. These commands are only available for simple byte, character, or C-string pointers (not for structure fields or array elements) and they do not affect the way the pointer's data is displayed in the View Structure dialog.

Display commands

D **Turn on Debugger Display**

Use this command to return to the debugger display or to redraw the display if it has been overwritten. This could happen if a program writes directly to the 80-column screen space instead of using one of the support I/O hooks (i.e. the Text Tools, the GS/OS Console Driver, or the ORCA shell ConsoleOut command), or perhaps if you discover a (gasp!) bug in the debugger.

1 **Display Hires Page 1**
2 **Display Hires Page 2**
3 **Display Double Hires Screen**
8 **Display 80-column Screen**
0 **Display Super-hires Screen**

These commands allow you to view the output on the respective video mode displays at a given moment or while the program executes. Note that program execution can automatically cause one of the other display modes to be displayed. However, this does not prevent you from changing the mode back and continuing execution. To return to the debugger display, use the Debugger Display command above.

Miscellaneous commands

P **View Program Profile**

Anyone who has been programming for any length of time, or has taken a course or read a good book on programming, will likely have come across the famous "10/90" rule of programming. Roughly, this rule states that about 90 percent of your program's running time will be spent in only about 10 percent of the code. What this means is that it is usually an unnecessary waste of time trying to perfect every piece of your program when you will not likely see any great improvement in running time in a good deal of your code. The trick is to determine where your program spends most of its time - to find the "hot spots". This is the function of a profiler.

While your program is running, Splat! is constantly keeping track of which subroutine is currently active and is watching the amount of time it takes to execute each line. At any time you can view the current time breakdown of your program by selecting the View Program Profile command. The dialog that is presented lists all of the traceable subroutines that have been invoked at some time during this execution, listed in descending order of total execution time. That is, the routine in which the most time has been spent will be listed first.

Each line shows the name of the subroutine, followed by the total execution time of the subroutine thus far (in milliseconds), and the percentage of the program's total running time that this subroutine has

occupied. This last figure is probably the most important one. It allows you to see, at a glance, which subroutines you should concentrate on.

A few words of caution must be given, however. First of all, although the timing method used in Splat!, which counts at a rate of about one "tick" every 127 microseconds, is of much greater resolution than, for example, Prizm, which only counts at 60 ticks per second, it is still not very precise. Also, program execution can be delayed by interrupts, especially if you are running under GNO. The debugger simply determines the time from when it started the line executing until that line finishes. If the current subroutine calls another traceable subroutine (one compiled with debugging information), even in Go mode, the time spent in this new subroutine will be accounted to it, not to the caller.

On the other hand, time spent in a library call, toolbox or GS/OS call, or in an untraceable subroutine, will be accounted to the caller. This includes prompted keyboard input, which can therefore seriously throw off the timing percentages. For this reason, if you wish to properly profile a program that uses standard keyboard input, it is recommended that you prepare an input file containing the proper input responses and then redirect that input file into the program using the appropriate shell command line syntax. This way, your program will not spend time waiting for keyboard input, producing a more accurate profile of the rest of the program.

For the technically inclined: Splat! uses the most significant 8 bits of the vertical video counter as a subdivision of the 60 Hz tick counter to obtain its 7860 Hz timing counter. (If you are running on a 50 Hz system, the counter actually runs at 7800 Hz but the millisecond counter will still be accurate.) It installs a dummy Heartbeat Queue task to ensure that the tick counter is incremented even when the Event Manager is not active. Subroutine times are stored in long words, allowing the debugger to profile programs for up to about 40 hours! If the program runs longer than this time, however, the profiler will breakdown and its information will be useless.

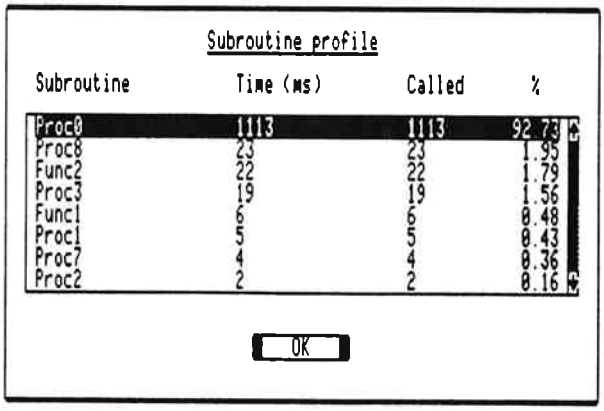

Figure 10 - Profile display

Apple-P **Set Preferences**

The name of this command is, I admit, a bit of a misnomer. However, the term "Preferences" is a fairly standard title in the Apple IIgs world, so I decided to stick with it. The function of the dialog presented by this command is to allow you to set some of the debugger's options to suit your requirements.

At this time, there are five settings for the user to adjust: the current stepping mode, how structures are displayd, the pass-through modifier keys, the default window configuration, and the size of the tabbing fields.

The current stepping mode is indicated by the presence or absence of a check beside the option. A check mark means that the default stepping mode for the Trace and the basic Step command is to follow the flow of the program into subroutine calls. If there is no check, subroutines will be executed at full speed and not traced. You can toggle this setting using the **'I'** key.

By default, structures and arrays are expanded to show their contents in the variable windows. You can disable this feature by turning off the "(E)xpand structures/arrays in variable windows" option. If this option is not checked, the right side of a structure or array variable's line will show only empty brackets; square ones for arrays, curly ones for structures. Turning off expansion reduces the amount of time needed to update the variable window(s), especially if there are a number of complex variables present, but you will then have to use the View Structures dialog to see the contents of these structures. This option is toggled using the 'E' key.

The list of pass-through modifiers shows the combination of modifier keys required to send a keystroke to the executing program's Event Manager queue, if the Event Manager is active. When text input is requested through the normal hooks, the debugger is suspended and all keystrokes are sent to the input routine. However, if GetNextEvent or TaskMaster are being used to obtain input, that input is never prompted for through those aforementioned channels. So, this mechanism was provided to allow you to test your program's keyboard handling functions as well. To send keystrokes to the program, you must hold down all of the checked modifier keys along with the keystroke that you wish to send to the program. Note that these modifiers are stripped off of the key code before it is passed to the event queue. You should keep this in mind, since you will not be able to use the Apple key as a pass-through modifier if you want to send menu item keyboard equivalents, as these codes usually include the Apple key.

The currently selected modifier keys are those with check marks beside the label. The key used to toggle each of these settings is the letter that is parenthesized in the label. Briefly, these are: **'A'** for the Apple key; **'O'** for the Option key; **'C'** for the Control key; **'S'** for the Shift key; and **'L'** for the Caps Lock key. The default setting is to use only the Option key to send keystrokes. If no modifiers are checked, keys are never passed to the Event Manager.

There are three windows configurations selectable through the Apple-V command. If you find you prefer to use one of the alternate configurations over the standard one, you can make that one your default configuration and it will be selected automatically every time Splat! starts up. Under "Default display configuration", choose the set-up that you prefer by entering the parenthesized letter. The check mark indicated which configuration is currently selected. Note that changing this value does not have any immediate effect on the display - it is only used when you start debugging a new program. If you would like this setting to remain intact after you shutdown the machine, you must make sure you save the settings, as described below.

The last setting allows you to set the spacing for tab stops in the Source window. When files with hard tabs are loaded into the Source window, this value is used to determine in which column the text following a tab character will be placed. This value can be set to any number from 1 to 99 by typing the value into the edit box. All editing keys and all numeric output will automatically go to the edit box as, at this time, all of the other controls in the dialog are buttons.

To accept the changes made, but only for this session, select "OK" (<Return> key). The settings will only remain in effect until the computer is rebooted, in the case of the INIT version, or only until the

current program finishes executing, for the shell version. To save the settings, choose "Save" (Apple-<Return>). This button causes the settings to be written to Splat!'s resource fork so that they can be retrieved the next time the debugger is loaded. Note that each version of Splat! (INIT and shell) has its own set of preferences, so changing the settings for one version will not affect the settings for the other. Finally, to restore the settings to the way they were before the dialog was invoked, select the "Cancel" button (<ESC> key).

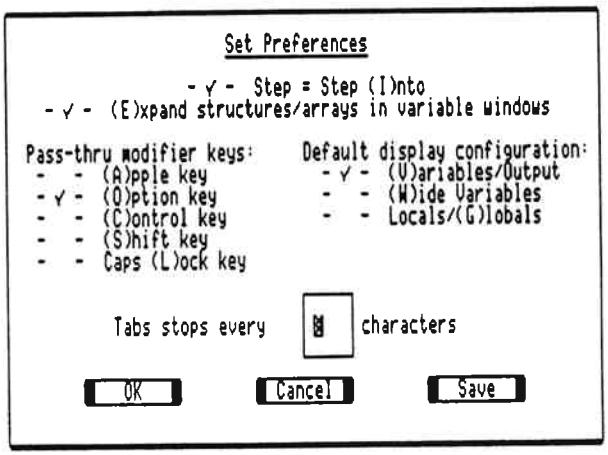

Figure 11 - Preferences Window

U Display Program Name and User ID

It may occasionally be useful to know the user ID assigned to the program you are debugging, perhaps for checking purposes or maybe for use with other programs, such as Nifty List (by David Lyons). This dialog displays the user ID of the code block from which execution was last interrupted. If all is going well, this will be part of your program's code. This user ID is then used to obtain the name of the owner of the block. If the name displayed is not that of the program being debugged, something has likely gone wrong with your program.

It is also possible, although not recommended, to debug two or more programs at once with Splat!. For example, you may want to determine why a particular NDA and application are not cooperating properly. This command can tell you which one of the programs is currently "active", if you are unsure.

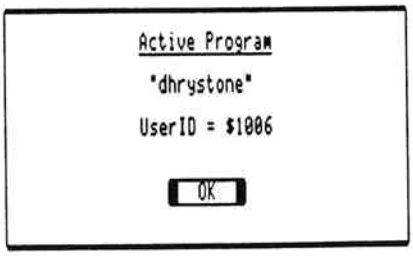

Figure 12 - Program Name and User ID display

N — Display Address of Next Program Line

This command helps you use the GSBug and NiftyList debugging utilities with Splat!. When you hit 'N', Splat! displays an address in hexadecimal notation. This address points to the first machine language instruction in the next line of code (the line the highlight bar is over in the source code). With this address, you can view the actual code with GSBug or NiftyList. This feature is useful to help view variable data in the rare case that Splat! cannot find it, or to help locate compiler errors in the output code. The address that is printed out points to the actual code, not the compilers debugger instructions that precede it.

Q — Quit Splat!

You can abort the execution of an active program at any time using the debugger's Quit command. I have attempted to safely shutdown the most common things an unfinished program may have left hanging, but there are too many cases to be able to take care of them all. Among the things the shutdown operation does handle are tasks like shutting down started tools, closing opened files, removing tool patches, and freeing allocated memory. This covers most of the most common conditions and should work for most programs. However, if you do anything "tricky", you may have problems aborting that program early.

For example, you should not abort a program that has changed any of the bank $E1 vectors either directly (which you should never do anyway) or with the SetVectors tool call. Any Out-of-memory (OOM queue), HeartBeat queue, Run queue, or GS/OS Notification handlers must be removed before shutting down a program. Also, new user IDs should be deleted, unless the program is intentionally leaving a piece of code or data permanently in memory. Some of these situations may be handled in a future version, but at the time of this writing they were not. Check the release notes on your program disk for updates.

If you are unsure whether a particular situation will be handled or not, feel free to ask (consult the Introduction for information on how to contact us), or, if you are brave enough, try it out. If it doesn't work, you will have to code an early exit option into your program. The above is by no means an exhaustive list of the possibilities, so if you have any further suggestions as to what should be included, please let us know.

? — Display Help Window

A complete list of commands and the keys tied to them is available at any time by hitting the '?' key (shifted or unshifted). Currently, the command information presented spans three "pages" (windows). To view the second and third (and possibly subsequent pages, in future versions), select the "Next Page" button (<Return> key). When you reach the last page, this button's label changes to "First Page". Selecting it again will return you to the first page of information. When you are finished viewing the dialog, select "Finished" (<ESC> key).

Chapter 6: COMMON PROGRAMMING ERRORS

This chapter is designed to give you a sampling of the most common programming errors that many programmers tend to make one or more (often many) times during their careers. If you are new to programming, the examples given will help you to learn what to look for when debugging your own programs. Experienced programmers should at least get a couple of good chuckles as they recall the times when they spent hours trying to track down some tiny little error, probably because they were looking for the wrong thing. Plus, they may even pick up a couple of tips from the experience of others.

This discussion is limited to errors that come up during run-time, as opposed to language syntax errors normally caught by the compiler. In that area, missing semicolons, brackets and comment delimiters, incompatible types in expressions, and misspelled names are the first things to look for. One nasty problem that is worth mentioning, in that it is not something most people will often think of checking, is that when creating constants or macros with the C language "`#define`" construct, be sure to *not* to put a semicolon (';') after the definition. If you do, then when you use the constant or macro in your code, the definition will be expanded, including the extraneous semicolon, generating a syntax error. This one I have seen bring several people to the point of practically giving up on a compiler, thinking that is where the error lies.

Beyond that, you are on your own. Your programming language text should give you information on some other tricky areas that might catch you up.

In any language, incorrectly set pointers are a fairly common occurrence. Whether you tried to use a pointer before it was set, after the memory it pointed to was freed, or when its value was NULL or NIL, the result is the same - wrong. Depending on where in memory you ended up accessing, you may even cause the computer to crash or at least behave unusually. On the IIgs, which has absolutely no memory protection scheme thus allowing any program to write to any location in memory, an erroneous memory write can easily corrupt the operating system software or some other resident piece of code. The existence of this kind of error may not be immediately evident, since the damaged piece of code may not be accessed until some later time, if at all during that session. Accessing a memory block that has already been deallocated is another error that may not show up often during testing. Use of dynamic memory allocation routines must be done carefully to make sure your program does not have unintended side effects, perhaps clogging or corrupting memory, and possibly bringing down the system.

Another thing to watch are loop conditions, especially in counted loops like "**for**" loops, or whatever the equivalent may be in the language you are using. A common mistake is to iterate the loop once too often or once too few, such as when your exit condition expression uses less than ('<') instead of less than or equal to ('<='), or vice-versa. Counted loop conditions are not the only places where errors can occur, though, so check all of those loop conditions carefully. Also, watch that the parts of the expression that might not be valid under some conditions are tested only after the required condition is found to be true. This is a function of the language's subexpression evaluation ordering. For example, suppose that you have a variable, **x**, that is used as a loop index on an array, **values**, of size **MAX**. To sum all of the elements up to the first non-zero value, you could use a block like this (given in C code):

```
x = sum = 0;
while (x < MAX  &&  values [x] != 0)
    sum += values [x];
```

Notice that in the condition for the 'while', the value of **x** was checked before accessing the array. We know this will always be checked first because the C language definition states that certain operators, such as '**&&**', 'll', and ',', always force the expression to the left of the operator to be evaluated first, then the right expression only if necessary. Since the logical AND is guaranteed to be false once the left expression is found to be false, the right expression is not evaluated. In contrast, the following statement is incorrect, although you might not ever notice any ill effects resulting from it:

```
while (values [x] != 0  &&  x < MAX)
```

If **x** is equal to or greater than MAX, then the loop will terminate as in the version above, but not before accessing the non-existent **MAX**'th element of the array.

When using the IIgs Toolbox, you should be careful not to assume that tool calls will always succeed, even if made correctly. Many Toolbox functions make hidden calls to other functions, which, for one reason or another, could fail, especially in the case of Memory Manager calls. Though it can seem tedious to check for errors after every call, failing to check often enough may result in a bug that seems unpredictable, or may never even occur on the programmer's machine. Generally, the description of the function in the appropriate Toolbox Reference Manual will tell you what errors are possible for a particular function, however, you should also check the Apple II Technical Notes for updates. In summary, it never hurts to check, except to slow down your program a tiny bit. If speed is an issue, you can remove the checks later after the program has been well tested.

Of course, there are many more errors that can crop up, but the best way to learn about them is to discover them yourself. Also take a look at **Chapter 3** for information on how the compilers can help you catch some common errors. As you gain more experience, you will learn what errors you are prone to make and be able train yourself to look for these first, or preferably to stop making them at all. Debugging is a skill (or, if you like, an art) that, like programming, must be developed over time. There is no substitute for experience. While related to programming, debugging is a separate skill, often neglected by otherwise good programmers. Everyone makes mistakes. (If they didn't, we wouldn't need debuggers.) However, a good debugger used by someone with good debugging skills can help you find those mistakes quickly and more easily, so you can spend more time working on the *real* features of the program.

Appendix A: Command Summary

Window manipulation commands:

<Tab>	Select next window (in left-to-right, top-to-bottom order)
Apple-<Tab>	Select previous window
Arrows	Move selection arrow or scroll window contents
Apple-Arrows	Page in selected direction
Apple-1 .. Apple-9	Jump To Part
J	Jump To Line
Apple-V	Change Debugger Window Configuration

Execution commands:

<Space>	Step using default stepping mode (see Set Preferences)
I	Step Into - step, follow execution into subroutines called
O	Step Over - step, skip over any subroutines called
T	Trace - continuous stepping using default stepping mode
G	Go - execute code at full debugging speed (no code tracing)
R	Return - execute at full speed until end of subroutine

Note: You can stop execution at any time by pressing <Space>.

Program control commands:

L	Load a source file into the source window
A	Set auto-go range
Apple-A	Remove auto-go range
B	Set or clear (i.e. toggles) a breakpoint on the selected line
S	List subroutine call stack
=	Assign a value to a variable (simple variables only)
<Return>	View the contents of the currently selected structure or array variable

Template commands:

Apple-L	Load a new set of templates from a template file (clears currently defined templates from memory)
V	Assign a loaded structure template to the selected pointer variable
W	Write the current list of structure template assignments to the "Splat.asgn" file.
Apple-S	Assign "C-string" template to pointer variable
Apple-C	Assign "Character pointer" template to pointer variable
Apple-B	Assign "Byte pointer" template to pointer variable

Display commands:

D	Turn on debugger display, or redraw display if already on
1	Display hires page 1
2	Display hires page 2
3	Display double hires screen
8	Display 80-column screen
0	Display super-hires screen

Miscellaneous commands:

P	View program profile
Apple-P	Set preferences for current stepping mode and pass-through modifier keys
U	Display the name and user ID of the program currently being debugged
Q	Quit debugger, aborting program being debugged (if not already terminated)
?	Display help window
N	Display the program name and memory manager user ID.

Appendix B: Splat! support files information

Template files

Splat!, like GSBug allows you to make use of templates to view data structures separated into their appropriate fields. These templates are loaded from a specially formatted text file, the format of which is identical to GSBug's, with the addition of a few more supported data types. If you are already familiar with the GSBug template format, then the only things you really need to know are the new data types and their declaration tokens.

Type	Meaning
float	single precision (4-byte) floating point value
double	double precision (8-byte) floating point value
extended	extended precision (10-byte) floating point value
truefalse	Boolean value (2-bytes)

If you are new to templates, and would like to create some of your own for use with your programs, then read on. The file format for templates is very simple. The file is created like any other text file and, while Splat! will recognize and load a file of any text or source file type, the recommended file type is TXT ($04). Hard tabs are handled and will simply be treated as field separators just as spaces are.

Each file can contain any number of templates, or at least up to the number that can be loaded into memory. You should try to keep your templates specific to each program you are working on to keep the number of templates in each file to a minimum. The more templates you have in a file, the longer it will take to load the file, and the more templates you will have to scroll through in the Assign Structure dialog's list. Also, you should put your most used structures near the top of the file, as the list is given in order of occurrence in the input file.

Each template begins with a line

_Start **<template_name>**

where **<template_name>** is the name of the template, as it will appear in the selection dialog. The "_Start" token, as with all of the defined tokens, can appear in any combination of uppercase and lowercase letters.

The body of the template consists of zero or more lines listing the names of the record fields in the structure, and assigning a type (and thus a size) and a repeat count.

<label> <type> <count>

The label is the name displayed for the field in the View Structures dialog, and must start at the left edge of the line (column 1). You can leave a field nameless by omitting the label and starting the line with one or more spaces or tabs.

The field's type must be chosen from one of the types in the list below. Only the first letter of the type is significant, as in GSBug, so you can abbreviate the type by giving only the first letter. Because of this I had to contrive another name for a Boolean variable ("truefalse") since

the "B" was already used for "byte". If you omit the type field, Splat! ignores the line and does not include it in its internal structure table. At this time, all integer fields are considered to be signed, as there is no provision for specifying that a field should be unsigned.

Type	Meaning
byte	single-byte integer
word	2-byte integer (e.g. C: "short int")
long	4-byte integer (e.g. C: "long int")
ascii	single ASCII character
cstring	null-terminated string (C language standard)
pstring	byte-prefixed string (Pascal language standard)
instring	word-prefixed string (GS/OS input string)
outstring	GS/OS output string (input string prefixed by a buffer size word)
float	single precision (4-byte) floating point value
double	double precision (8-byte) floating point value
extended	extended precision (10-byte) floating point value
truefalse	Boolean value (2-bytes)

In addition, you can create a 4-byte pointer to any of the above types by prefixing the type token with an asterix ('*') - for example, "*word" or "*cstring". There should not be a space between the asterix and the type name.

If you want to use your templates with GSBug and Nifty List (or any other program that does not expect our format extensions), to keep your template files compatible with these programs, you should choose only from fields listed in the first group and you should not use pointers.

The repeat count corresponds to the array size in a C or Pascal program, for example.

Finally, a line beginning with just

_End

marks the end of the structure definition.

Additionally, any line beginning with a semicolon (';') is a comment line and is ignored. Blank lines can be placed anywhere in the file and are also skipped. Template definitions without any accepted fields (lines with at least a type declared) are thrown out. The help templates included in the "GSBug.Templates" file that comes with GSBug fall into this category. These templates are not needed in Splat!, as all of the available template are displayed in a list box so you never have to remember exact template names.

Now that you have everything you need to make your own templates, lets look at a couple of sample templates, one taken from the "GSBug.Templates" file included with GSBug, containing templates for most of the IIgs Toolbox and GS/OS structures, and one of my own creation.

```
_START EventRecord
what            WORD
message         LONG
when            LONG
where           WORD    2
modifiers       WORD
_END

_START  MyStruct
intField        word
floatVal        float
emptyFlag       truefalse
stringPtr       *cstring
next            *byte
_END
```

Assignment files

Once you have assigned a number of structure templates to variables, you will want to save your efforts for continued use as you develop your program. **Splat!** provides a command, Write Template Assignments (described on page 26), that allows you to do just this. All of the current assignments are written to a file called "Splat.asgn" in the program's directory (prefix 9). This file is then read back in when **Splat!** starts debugging this program again, provided the program and the file remain in the same directory.

Using **Splat!**'s built-in template assignment commands should be the easiest, and thus preferred method of setting up template-to-variable assignments. However, for those of you who really want to create assignments by manually editing the file, or for someone who wants to write a program for automatically generating an assignments file from a source file, the following information will be useful.

Each line in the file contains a single assignment. The format of the line is

<filename> <sub_name> <variable_name> <template_name>

where **<filename>** is the name of the source file (the filename portion only) in which the subroutine appears, **<sub_name>** is the name of the subroutine in which the assignment is to apply, **<variable_name>** is the name of the variable, and **<template_name>** is the name of template being assigned to the variable.

As with template files, blank lines and lines beginning with a semicolon are ignored. Below is a sample line, as created by Splat!. Despite the debugger's output conventions, the interpreter is not case-sensitive.

SAMPLE.C main EVENTRECPTR EventRecord

Any number of assignments may be included in an assignments file, but no two lines should refer to the same variable. If this does happen, the first line will take precedence. This will get confusing if you should change the assignment though. When this happens, the first entry will be deleted and a new entry will be added to the end of the list, but the entry that will now be used will be the old *second* entry, now the first one remaining in the list.

INDEX