ADVANCED COMMODORE 128™ GRAPHICS AND SOUND PROGRAMMING

Stan Krute

No. 2630
$21.95

ADVANCED COMMODORE 128™ GRAPHICS AND SOUND PROGRAMMING

Stan Krute

TAB BOOKS Inc.
Blue Ridge Summit, PA

This One's For Sharron & Neil & Jason & Benjamin & Phred
With Love

FIRST EDITION

FIRST PRINTING

Printed in the United States of America

Library of Congress Cataloging in Publication Data

Krute, Stan.
Advanced Commodore 128 graphics and sound programming / by Stan Krute.
p. cm.
Includes index.
ISBN 0-8306-0930-X ISBN 0-8306-8630-4 (pbk.)
1. Commodore 128 (Computer)—Programming. 2. Computer graphics.
3. Computer sound processing. I. Title. II. Title: Advanced Commodore One-twenty-eight graphics and sound programming.
QA76.8.C645K78 1987 87-29028
006.6′865—dc19 CIP

Questions regarding the content of this book should be addressed to:

Reader Inquiry Branch
TAB BOOKS Inc.
Blue Ridge Summit, PA 17294-0214

Contents

List of Programs

Preface

The folks from Pennsylvania have given the C-64 a worthy successor, the *Commodore 128.* Ignore the snobs who look at the low price and laugh. It's got great graphics and sound chips, an elegant memory management system, a sophisticated BASIC interpreter, perfect C-64 emulation, a well designed keyboard, powerful CP/M capabilities, and plenty of hooks for imaginative programmers. The hardware's there; now it's up to all you clever coders to push the 128 to its limits and beyond. My aim in this book is to give you a few tools that will ease the journey.

This book will show you some sophisticated C-128 graphics and sound programming techniques. It's packed with clear explanations and obsessively commented programs. The programs are written in BASIC 7.0 and 6502 assembly language. I've hacked away on the 128 for over a year now, and made a number of interesting discoveries. This book contains many of them.

Many wonderful people helped me get this information out to you. The book would not be here without their kind assistance. I'd like to acknowledge a few of them here; the rest know who they are. Bruce Hammond and the Starpoint Software crew let me use their 128 for a couple of months that turned into ten, and sweetened the deal with software, shared enthusiasm, and gallons of diet soda. Roger Wagner and Glen Bredon trusted me with a prepublication copy of Glen's rock-solid Merlin 128 assembler. Dan Weston and Leslie Kay kept the flame burning. Diane Le Bold and Jim Gracely of Commodore went out of their way to provide useful system documentation on short notice. Larry Jackel, Ray Collins, Kevin Burton, Bob Ostrander, and Ron Powers of TAB supplied faith, beyond-infinite patience, motivational techniques, and an efficient publishing operation. Jim, Laura, and Saylor Flett helped keep the author warm, cool, electrified,

mobile, and well fed. Richard and Karen Perez supplied continued inspiration, both spiritual and gastro-intestinal. The Wizard came through with miraculous gifts in the pinches. Scott Statton appreciated the Plywood Palace. Levi Thomas shared love, intelligence, and encouragement. Dana Andrews provided a home away from home. John and Anita Pryor appeared near the end with their remarkable energy and friendship. Ruth and Irving Krute came up with their usual abundance of love and support. Steve, Cynthia, Raisin, and Rags Fink provided last-minute philosophical discussions, archetypical accommodations, love, more diet soda, and genre deconstruction. Steve Jasik made a last-minute loan of time and equipment. The Plywood Palace cat friends maintained equilibrium and family values at all times.

As I've mentioned in other books, sight and sound are two of the widest channels into the human heart and mind. That's why I'm partial to machines that excel in communicating with those two senses. The Commodore 128 is one of them. It's a sturdy, sophisticated instrument. Please use it well.

Introduction

Welcome to a slightly different kind of programming book. It's for a slightly different kind of programmer working on a different kind of hardware/software system and programming a different kind of user interface. This programming relies heavily on graphics and sound techniques to open a broad band of communication with computer users.

Computer hardware is getting more powerful all the time, as is the built-in systems software and the specific applications programs— all working towards the goal of creating cybernetic tools that normal human beings can use. Machines like the Macintosh, Amiga, Atari ST, Apple IIGS, and the IBM PS/2s are leading the trend. But they all cost too much for the average user. The C-128 goes for a paltry couple of hundred dollars, yet has the hardware muscle to pull off a lot of modern tricks. All it needs is a little extra smart software magic to bring its powers to light. You're the one who'll provide that magic, and this book will show you how.

Books that teach this new style of programming are more difficult to organize. The kinds of programs that get these machines dancing are not short and sweet. The interactions with the hardware and systems software are intricate. Even brief examples aren't brief anymore. The growing sophistication of the systems and the programmers demands for more information that's better organized, with a sweep that goes from grand overview to tiny details—information that gives you a pile of useful tools.

Ultimately, at the rate of current developments, that's just a couple of years—such sophisticated programming tutorials or toolboxes will sit on some kind of laser-storage medium and run interactively in multiple windows on your screen. You need all kinds of cross-indexed, readily-accessible information when you're solving programming tasks: language references; system information; system interface examples; user interface examples; syntax and structure diagrams; pseudo-code generators; and charts, graphs and pictures.

We don't all have that laser technology now. But it's time to get ready. And so we get to the organization of this book, which is a bit unlike most other programming texts—not just to be different, but because different needs demand different forms.

The goal is to get you dancing around the C-128 system, able to solve any sound or graphics programming challenge that arises. Stealing from the more advanced graduate schools, I've used a project-oriented, case study method. This lets me touch on both theoretical and practical issues. The book contains two major programming projects. I've divided each project into eight sections. Each section forms a chapter of the book.

A project's first section discusses its Human Interface. This section tells what the project's programs do, and how to use them. General design and ease-of-use issues come to life in the specific C-128 context.

A project's second section discusses its System Interface. This section tells about the project's programs' interactions with the C-128 hardware and ROMs. Some of this material clarifies information available elsewhere. Much of it is new, information I've dug out by playing with the system.

Commodore is known for the byzantine nature of its systems, and the C-128 doesn't break that trend. What I do is show you how to dance around the C-128, tap-dancing on the good ROM stuff, jitter-bugging on the good hardware stuff, pirouetting over the muck.

The third section for each project is called Program Notes. That's where I discuss some of the important structural features of the project's programs. I also point out some of the more interesting language usages.

Fourth in each project's array of tools is a section called Stretching. This is where I suggest things you can do to extend the project. For example, though the 80-column graphics routines I supply are faster than the ROM's 40-column routines, there's still room for a two-to-four times speed optimization. So, in that project's Stretching section, I lay out some speedup techniques.

Fifth in each project comes a section of Calling Structure Diagrams. These are pictures that show the essential architecture of the project's programs. I think you'll find them invaluable as you learn to put together huge, bug-free suites of programs. More specific detail on these diagrams can be found in Appendix B.

The sixth project section is called Subroutine Line Starts. Each routine in a program is listed with the line in the source code where it begins. This'll help you find routines that are referred to in the other sections.

Seventh is a section of Selected Algorithms. Using a C/Pascal-like pseudo-code, these algorithms show the work-horse intelligence of the project's programs, unmarred by the realities of BASIC 7.0 or 6502 assembly language. There's some interesting code here, including a Macintosh-like interrupt-driven cursor and a complete pseudo-code implementation of an optimized Bresenham line drawing algorithm. Appendix C details the Pinhead Pseudo-Code (PPC) that I use. If you've got a background in structured programming, you should find it pretty transparent.

Finally, each project closes with its program's source code. There's a lot of it, and it's probably the most heavily commented you'll ever find. I can never figure out in the bright light of morning what the heck I cranked out in the wee hours the night before.

So I write lots of comments, and polish the code organization for readability. There's a lot of code here. It's highly modular and filled with software tools you might find useful in other contexts.

So, that's the layout of chapters for each project. You can profit from this book without running the programs, but I think you're better off if you run them and play with modifications. Type in the code if you're long on time and short on money. But a couple of hundred pages is a lot of typing to do. You can send for disks that contain all the book's listings; see the order form at the back of the book.

If you do type in the programs, you'll need an assembler. I recommend Glen Bredon's *Merlin-128 System,* available from:

Roger Wagner Publishing
10761 Woodside Avenue Suite E
Post Office Box 582
Santee, California 92071
(619) 562-3670

Several appendices appear at the back of the book. I suggest you read the first four of them now. Appendix A: Useful Conventions, describes abbreviations and jargon (minimal) that I've used in the book. Appendix B: Calling Structure Diagrams, describes the above-mentioned diagrams and their iconography. Appendix C: Pinhead Pseudo-Code, documents the PPC. And Appendix D: System Interface Summary, steers you toward the lines of code in the book's programs that contain interesting system-related operations.

There are a number of other appendices. They're there so I can get all these loose scraps of paper off my walls and desks. I hope they do the same for you.

Chapter 1: Human Interface

The heart of this first programming project is a set of 80-column graphics routines. These routines can be called from BASIC 7.0 or from assembly language. The assembly language object file Grafix 80 contains the complete graphics package (routines and interface code).

Prepare a disk that contains the compiled object code for G80 Install and for Grafix 80. See Figs. 8-1 and 8-2 for their assembly language source code.

G80 Install loads Grafix 80, adjusts the C-128 environment as necessary, and hooks in the new graphics routines. Working from the C-128's direct mode, not from within a running program, give these two commands to run G80 Install:

```
BLOAD "G80 INSTALL"
SYS 7592
```

You now have six new graphics commands. They're designed to work like the 40-column graphics commands that are part of BASIC 7.0. Take a look at the first few pages of the Grafix 80 program listing, Fig. 8-2, for detailed manual entries for the commands. Briefly, the commands are as follows:

G80Box —draws outlined and filled boxes on the 80-column graphics screen.

G80Color —sets the foreground and background colors for the 80-column graphics screen.

G80Draw	—draws points, lines, and connected series of lines on the 80-column graphics screen.
G80Graphic	—puts the 80-column display into text or graphics mode, with optional screen clearing.
G80Scat	—removes the 80-column graphics commands.

After running G80 Install, you can play with the commands from BASIC's direct mode. Then you can try the BASIC 7.0 program G80 Test Suite. Figure 8-3 lists its source code. G80 Test Suite tests the performance of the 80-column routines. Make sure you've run G80 Install, as shown above. Then run G80 Test Suite with this command:

```
RUN "G80 TEST SUITE"
```

G80 Test Suite contains numerous examples of calling the 80-column graphics routines from within a BASIC 7.0 program. If you want to compare the performance of the 80-column graphics routines to the C-128's built-in 40-column BASIC 7.0 graphics commands, get rid of the 80-column package, and run the program G40 Test Suite:

```
G80SCAT
RUN "G40 TEST SUITE"
```

The 80-column graphics routines can also be called from assembly language programs. See Chapter 4: Stretching.

Chapter 2: System Interface

Remember, refer to Appendix D:System Interface Summary to locate instances of the following items in the program code.

2.1 NEW COMMAND FROM ASSEMBLY LANGUAGE

BASIC 7.0's NEW command clears out any BASIC programs currently in memory, and sets a number of BASIC and system variables so a new program can get going. It comes in quite handy when you've futzed with the system and want to clean up any unforeseen side effects of the futzing before letting BASIC come back into play.

That's why it's used in G80 Install and Grafix 80. But in both instances I invoke the NEW command from assembly language. Surprisingly, this turns out to be a simple task. It's done with a JSR call to a documented vector, JmpNEW, located at address $AF84 in the ROM. A few preliminaries are required before the call: First, get the machine into a bank 15 memory configuration. Next, be sure the byte just before the start of BASIC's text area is zeroed (see section 2.2). Finally, set the zero flag of the 8502's processor to 1.

2.2 SETTING BASIC PROGRAM TEXT STARTING ADDRESS

Memory locations $002D-$002E, known as TxtTab, point to the start of a BASIC program's text. Normally, the pointer value is $1C01. G80 Install moves BASIC text two pages up in memory to make room for the Grafix 80 routines. It resets the TxtTab pointer value to $1E01 to accomplish the move.

Notice the convention: the pointer points one byte into a memory page, and the byte just before this start of BASIC text must be set to 0. Don't ask me why, it's a convention that's hung on from the earliest days of Pet BASIC, but it must be done. So, in G80 Install, we set $1E00 to 0. And, when the user removes the Grafix 80 routines, we move BASIC's text start back down to $1C01 and zero out $1C00.

2.3 WARM START FROM ASSEMBLY LANGUAGE

This is another part of the magic ritual to follow after fiddling with BASIC. A warm start, also known as a soft reset, takes care of BASIC and system variables that a NEW call doesn't hit. Once again, the Commodore designers give us a nice documented entry point, SoftReset, located at memory location $4003 in the ROM.

To review, here's what to do if you want to fiddle with BASIC, then bring it back to life safely: do the fiddling. Then get into a bank 15 memory configuration. Make sure there's a zero just before the start of BASIC text. Prime the processor's zero flag, setting it to 1. Call on JmpNew ($AF84). Call on SoftReset ($4003). This is what I do in G80 Install before loading the Grafix 80 routines, and in Grafix 80 when the user chooses to remove the new routines.

2.4 KERNEL ROUTINE LOAD ($FFD5)

The kernel's LOAD routine lets you load disk files when you're working in assembly language. There are some preliminaries. First, a call to SetBnk to tell the system what memory configurations to use (see Section 2.5). Second, a call to SetNam to set a file name and any control characters (see Section 2.7). Third, a call to SetLFS to set its three parameters (see Section 2.6).

After the preliminaries, it's time to call the load routine. It requires one parameter, and can take up to three, all passed via the 8502's main registers. A function selector, passed in A, is required. If you want to truly load a file into memory, put a zero into the A register. If you just want to check an area of memory against a file, put a non-zero value into A. That tells Load to perform a verify operation. If you want to load the file into memory at an address different than what the file header bytes indicate, X gets the lo-byte of the load address, and Y gets the hi-byte. Also, the secondary address/command set up by the preliminary SetLFS call must be zero for this relocation to take effect.

In G80 Install, Load is used to pull in the Grafix 80 code. Study that code for a textbook example of this vital routine's use.

2.5 KERNEL ROUTINE SETBNK ($FF68)

SetBnk is used to prepare for I/O operations. It's usually called along with SetLFS and SetNam before calls to Load, Open, and/or Save. It sets the memory banks to be used with the upcoming operation. The first bank it sets indicates where data will be Saved from orLoaded to. This is done by passing a logical bank number (0..15) in the A register. It also sets the memory bank in which the file name string is living. This is done by passing a logical bank number (0..15) in the X register. After these two registers are set, you call on SetBnk with a JSR.

2.6 KERNEL ROUTINE SETLFS ($FFBA)

This kernel routine is called prior to using various kernel input/output routines. It sets up a file's logical file number, device number, and secondary address/command. This is analogous to the numbers supplied when you use BASIC 7.0's **OPEN** command. The system stores the logical file number in the system global LA ($00B8), the device number in the system global FA ($00BA), and the secondary address/command in the system global SA ($00B9).

Prior to calling the routine with a simple **JSR**, the A register is loaded with the logical file number, the X register gets the device number, and the Y register gets a secondary address/command. The logical file number will be used in subsequent operations to refer to the file. The device number depends on the device. For example, disk drives are usually numbered 8 and 9, printers are 4 and 5, etc. The secondary address/command gives further device-specific information, and its use is optional. If not used, the Y register should be loaded with the value $FF (255).

In G80 Install, **SetLFS** is used to prepare for a **Load** command. This affects the parameters passed to **SetLFS** in the A, X, and Y registers. The logical file number, passed in A, is set to zero, since the **Load** command doesn't use a logical file number. The device number, passed in X, works in the usual way, taking the device number of the disk drive we want to **Load** from. In G80 Install we use the device number last used, plucked from memory location $00BA (FA). The secondary address/command, passed in Y, is set to 0 if we want the file to come in at an address different from what's stored in the file header. Otherwise, it's set to some non-zero value. This second option is used in G80 Install, since we want Grafix 80 to be loaded into its standard position. $FF is a nice non-zero value to use, since it's as non-zero as an 8502 gets.

2.7 KERNEL ROUTINE SETNAM ($FFBD)

When setting up for an input/output operation, you have the option of using a name. A name is used when dealing with true file devices, like disk drives and cassette recorders. When opening up a physical device, such as a printer or display screen, the name is omitted.

SetNam must be called to tell the system what you're doing about a name. If no name is to be used, just load the A register with zero and call **SetNam** with a **JSR**. If there is a name, the call preparation is different. The kernel routine **SetBnk** must be called to tell the system which memory configuration (0..15) should be set before looking for the name. Then the A register gets the length of the name, the X register gets the lo-byte of a pointer to the name's first character, and the Y register gets the hi-byte of this pointer. Then **SetName** is called with a **JSR**. The name itself must be followed by a zero byte in memory.

In this context 'name' is used in a general way to indicate the entire string we want to pass to the file system. In opening a disk file, for example, such a name may be a string as complex as the following, in which the actual file name is surrounded in the name string by various control elements:

"@:THE FILE NAME,S,W"

In G80 Install, we're using **SetNam** to prepare for a load command. The name string that gets set with a call to **SetNam** is contained in the G80 Install code. I stuff the A register with the length of the name string, X with the lo-byte of a pointer to the name string, and Y with the hi-byte of a pointer to the name string. When you examine this G80 Install code, be sure to note how I've used the assembler's labeling capabilities to make the source code independent of any changes to the name string.

2.8 MEMORY CONFIGURATION

From BASIC, you can configure C-128 memory with the **BANK** command. It's not much tougher to do the job from assembly language. The configuration register appears at $D500 when the I/O block is switched into memory, and at $FF00 at all times. I recommend you just use the $FF00 address, since it works just as well and is always available.

It's a good idea to restore memory to its previous state when you're done fooling around. Just save the current value of the configuration register, change it to a new value, do your fooling around, then set the configuration register back to its original value.

The one trick is to figure out how to set up a configuration byte. Commodore memory management has never been perfectly straightforward, and the C-128 continues that fine tradition. The C-128 Prg gives one of the better explanations, on pages 460-465. Or take a look at Fig. 2-1. If your head is a bit lazy, though, Fig. 2-2 will help. It shows the sixteen configuration bytes that correspond to bank setups 0 thru 15. Of course, other configurations are possible; again, refer to Fig. 2-1.

You can find a table like Fig. 2-2 right in the C-128's ROM. It runs from memory location $F7F0 through to $F7FF.

There are some examples of memory configuration sprinkled throughout the various 80-column graphics programs. In G80 Install, I want to get into a bank 15 configura-

<table>
<tr><th colspan="8">The Bits In A Memory Configuration Byte</th></tr>
<tr><td>bit 7</td><td>bit 6</td><td>bit 5</td><td>bit 4</td><td>bit 3</td><td>bit 2</td><td>bit 1</td><td>bit 0</td></tr>
<tr><td colspan="2">Bank Select</td><td colspan="2">Hi
Space
$C000-$FFFF</td><td colspan="2">Mid
Space
$8000-$BFFF</td><td>Lo
Space
$4000-
$7FFF</td><td>I/O
Space
$D000-
$DFFF</td></tr>
<tr><td colspan="2">00=ram bank 0
01=ram bank 1
10=expansion bank 2
11=expansion bank 3</td><td colspan="2">00=kernel rom
01=internal function rom
10=external function rom
11=ram</td><td colspan="2">00=basic rom hi
01=internal function rom
10=external function rom
11=ram</td><td>0=basic rom lo
1=ram</td><td>0=I/O
1=ram/ rom</td></tr>
</table>

Fig. 2-1. Each bit in a memory configuration byte determines which physical memory locations will show up in a particular logical address space.

Bank	Configuration Byte In ...		
	Binary	Hexadecimal	Decimal
0	%00111111	$3F	63
1	%01111111	$7F	127
2	%10111111	$BF	191
3	%11111111	$FF	255
4	%00010110	$16	22
5	%01010110	$56	86
6	%10010110	$96	150
7	%11010110	$D6	214
8	%00101010	$2A	42
9	%01101010	$6A	106
10	%10101010	$AA	170
11	%11101010	$EA	234
12	%00000110	$06	6
13	%00001010	$0A	10
14	%00000001	$01	1
15	%00000000	$00	0

Fig. 2-2. The C-128 has 16 standard memory configurations. For each one, this figure shows its configuration byte in binary, hexadecimal, and decimal.

tion before fiddling around with BASIC. Bank 15 is particularly useful because it lets you get at the major system resources: kernel and BASIC ROMS, system globals located in RAM 0, and the I/O block. You can get into any memory configuration from assembly language by stuffing the appropriate configuration byte into the configuration register that always exists at $FF00. Also, note how the previous configuration is restored when you're done. In Grafix 80, I also fiddle with memory configurations. Right before carrying out one of the new 80-column graphics commands, I put the machine into bank 15, and restore memory to its previous configuration when the command carrying out's done. Besides providing access to the major system resources mentioned above, bank 15 lets me get at the current BASIC program's source code and the Grafix 80 object code.

2.9 CRUISING THRU COMMAS IN BASIC PROGRAM TEXT

When Grafix 80 hits one of its 80-column graphics commands, it has to look for parameters. As with built-in BASIC 7.0 commands, the 80-column graphics commands expect their parameters to be separated by commas. It's nice to have a routine that makes sure a required comma is in the line, then cruises past the comma and picks up the next element in the BASIC line. (By the way, by BASIC line element I mean a constant, variable, command token, or vital piece of punctuation—anything other than a space.) There's such a comma cruising routine built into the BASIC ROM, but it's undocumented. So I wrote a similar little gem of my own, CommaCruz, for Grafix 80. It calls the low-memory routine IndTxt (see Section 2.12) with an index of zero to grab the current byte-of-interest from the BASIC line being parsed. If it's a comma, CommaCruz finishes with a jump to ChrGet, which moves the BASIC line parser along and grabs the first byte of the next element in the line, then returns to whoever called on CommaCruz. If there's no comma, CommaCruz finishes with a call to the system's error handler, which will send out the popular "SYNTAX ERROR" message and bring things to a crashing halt.

And someone told you the art of interpreter parsing was complex? Job protection fog, folks.

2.10 LOW-MEMORY ROUTINE CHRGET ($0380)

This is one of the BASIC interpreter's workhorse routines. Essentially, it moves BASIC's text pointer along to the next BASIC line element, skipping through spaces, and grabs the element's first byte. The byte, or character, fetched is returned in the processor's A register. Various status flags get set, depending on what's returned in A: If it's a colon ($3A) or zero, both of which function as BASIC statement separators, the zero flag is set to 1. If it's a digit 0..9 ($30..$39), the carry flag comes back cleared to 0. If it's in the range $00..$2F or $3A..$FF, the carry flag comes back set to 1.

In Grafix 80, many of the calls to ChrGet occur when a routine is handling one of the new commands and needs to move along to the next element in a BASIC line. This is often in preparation for calling one of the higher-level BASIC parsing routines, like GetByt or GetWdByt (see Sections 2.26 and 2.27). In these cases, I don't even look at the character returned in the A register. Other times, I'm looking for specific tokens or punctuation marks, as in some of the parameter-fetching functions and the CommaCruz routine. In those cases, the call to ChrGet is followed by one or more CMP instructions and branches based on those comparisons.

2.11 LOW-MEMORY ROUTINE INDTXT ($03C9)

This is a useful routine that lets you look at parts of the BASIC line currently being parsed. It gives you an indexed look at the BASIC text, thus the name. As mentioned above, the BASIC parser maintains a pointer into the text, TxtPtr ($003D-$003E). Before calling IndTxt, you load the Y register with an offset value. IndTxt then uses that offset and TxtPtr to grab a byte from the BASIC line. In Grafix 80, IndTxt is used in the CommaCruz routine.

2.12 MEMORY LOCATION 13 ($000D) AKA COUNT

This memory location is used by many of the ROM routines. In Grafix 80, it's used in the IEscLkDetor token crunching routine. After a successful call to the undocumented ROM routine FndComTxt, which looks for a given command in a pointed-to table of legal commands, Count holds the found command's offset in that table. In Grafix 80, OurComsText is the table that's used. The table offset returned in Count is used by ROM routines to supply a selector token for the to-be-crunched command.

Got all that? Let me fill in a few holes. When you write a program, you enter it line by line. The BASIC interpreter takes each line and does some processing. One of the main processing tasks is changing valid BASIC commands from C-ASCII text to a condensed form. These condensed forms are called tokens, and may be one or two bytes long. Many commands transform into a one-byte token. But there are so many commands, some must be stored as two-byte tokens. The first byte of a token (the only byte in a one-byte token) has bit 7 set to 1. That makes it easy for the interpreter to find tokens, since bit 7 is easy to test (see all my previous discussions of 8502 bit-flagging), and no other parts of a BASIC line other than tokens have bit 7 set to 1. (Actually, characters in a string constant can have bit 7 set to 1, but since these occur in a string, demarcated with quotation marks, they're easily recognized and filtered out.) Since all the one-byte tokens are used up, new commands are implemented as two-byte tokens. In these two-byte tokens, the first byte (the one that has its high bit set to 1) is called the lead-in token or lead-in byte, the second byte the selector token or selector byte. Two lead-in tokens are used in the C-128, $FE and $CE. My 80-column graphics commands use $FE as the lead-in token, since a complete set of vectors exists that makes it easy to add commands that tokenize to $FE doubles. A given command's selector token is determined by taking the command's position in its table (OurComsText is the table in Grafix 80), and adding in some constants. Carefully check out the heavily-commented IEscLkDetor code to see how this all works.

2.13 THE IERROR VECTOR

This vector, located at memory locations $0300-$0301, allows the system, and any other semi-intelligent lifeform, to send BASIC error messages to the current default output device (usually the screen). To call it, load the X register with an error number, then do an indirect jump thru this vector. For example:

```
LDX #SyntaxError
JMP (IError)
```

Pages 644-647 of the C-128 Prg provide a list of the error numbers, their messages, and what they mean. In Grafix 80, I use this error facility when there's been a mistake involving the 80-column graphics commands.

2.14 THE IESCLK VECTOR

This vector, located at \$030C-\$030D, is called by BASIC's parser near the end of its token crunching activities. It's sometimes called the token crunching vector. The name comes from the fact it's an indirect jump providing an escape hatch during command lookup. As mentioned before (Section 2.12), BASIC commands are converted to tokens after a line is entered. This vector is there so you can provide a last-resort lookup routine, when all the built-in command lists have been searched without success. In Grafix 80, we point this vector to our **IEscLkDetor** routine, which checks for one of the new 80-column commands.

The system calls **IEscLk** as it does most vectors, with an indirect jump:

JMP (IEscLk)

At the time of this jump, the A register contains the character BASIC's parser is looking at; that is, the first byte of the current BASIC line element of interest. If you write a detour for this vector, your routine can check your own command lists. When you're done checking, you return to the parser loop by jumping to the regular **IEscLk** handler. If you've found a valid command, you go to this exit jump with the carry flag cleared to 0, the X register holding a code indicating whether the command crunches to an \$FE lead-in double-token (coded with a value of 0) or a \$CE lead-in double-token (coded with a value of 255), and the A register holding an adjusted version of the found command's selector token. Check out the **IEscLkDetor** code for more detail on this. If you haven't found a valid command, you go to this exit jump with the carry flag set to 1.

2.15 THE IESCPR VECTOR

This vector, located at \$030E-\$030F, is called by the BASIC interpreter when it's listing a tokenized BASIC line. The interpreter needs to convert tokens back into their full C-ASCII command form to list them. This process is called un-crunching, and this vector is sometimes called the token un-crunching vector. It's name comes from the fact it's an indirect jump capable of providing an escape hatch during command printing. It's normally set to jump right back into the BASIC interpreter code in the ROM, but you can redirect it and un-crunch double-byte tokens for commands you've added to BASIC. Upon entry to a routine indirectly jumped to thru this vector, the X register holds a code indicating whether the lead-in token is an \$FE (coded with a value of 0) or \$CE (coded with a value of 255). The A register holds the selector token 23.

If the jumped-to routine decides that the double-token coded by X and A is valid, and thus can be un-crunched, it ends by jumping to an undocumented un-crunching routine, **FndTknTxt** (see Section 2.21). To prepare for this exit jump, the X register gets stuffed with an adjusted version of the selector token, A gets the hi-byte of a pointer to a table of C-ASCII command names, and Y gets the lo-byte of that pointer. But if the jumped-to routine decides that the double-token coded by X and A is invalid, it sets the carry flag to 1 and jumps on to the regular **IEscPr** handler in the ROM.

Grafix 80 uses this vector for its new 80-column graphics routines. The vector is reset to point to the routine **IEscPrDetor**. Check the heavily-commented code for various implementation details.

2.16 THE IESCEX VECTOR

This vector rounds out the set that lets you add full-fledged commands to BASIC 7.0. It lives at $0310-$0311. The name comes from the fact it's an indirect jump providing an escape hatch during command execution. When the BASIC interpreter comes across an $FE double-token it doesn't recognize, it jumps to the routine pointed to by this vector. Normally, the vector points back into the ROM code. But it can be redirected to a routine of your own design. This is what I've done in Grafix 80.

Upon entry to a routine pointed to by IEscEx, the $FE double-token's selector token is in the A register. If this selector marks a valid command, you just go and carry out the command, then clear the carry flag to mark success and return via a simple RTS. If the selector is out of range, set the carry flag to mark failure and jump on to the regular IEscEx handler. Again, refer to the Grafix 80 code, in particular the IEscExDetor routine, for heavily-commented real world examples of using this vector.

2.17 TOKENS: CRUNCHING NEW BASIC COMMANDS

We've covered most of this but here's a mini-review and some further detail.

The key is redirecting the IEscLk vector to a special crunching routine, as covered in Section 2.14. The new commands will have double tokens, with an $FE lead-in. You also need to build a table containing the text of the new commands. One important feature of this table is the last character of each command must have its high bit (bit 7) set to 1. That's because of the way the ROM routines scan for commands. Most assemblers have a pseudo-op that takes care of this detail, though you can just figure out the proper code with all those extra neurons you're developing. The Merlin assembler supplies the DCI pseudo-op to cover this situation. In Grafix 80, the command table is called OurComsText. Finally, you need to associate selector tokens with each new command. These can be any integer in the range $26 to $7F. In Grafix 80, this is done with the constants listed in lines 255-263.

FndComText is an undocumented ROM routine you can call from the crunching detour to carry out the crunching gruntwork. Section 2.20 gives further details about using it.

2.18 TOKENS: DETECTING NEW BASIC COMMANDS

First, we have to come up with detour routines for IEscLk, IEscPr, and IEscEx. In crunching, we scan a table of command names, as mentioned in 2.17, and come up with tokens. Then, in un-crunching, and executing new BASIC commands, we just run some comparison tests on the token(s) to see if it's one of our new commands. New commands will have double tokens, so we look for a lead-in token match and a selector token match.

2.19 TOKENS: UN-CRUNCHING NEW BASIC COMMANDS

Again, we've covered most of this but here's a mini-review and some further detail.

The key is redirecting the IEscPr vector to a special un-crunching routine. Upon entry, the X register holds a code indicating an $FE or $CE lead-in token, A holds the selector token. If these two items indicate one of the new commands, we call on an undocumented ROM routine, FndTknTxt, which does the actual un-crunching. Section 2.21 details the calling protocol for FndTknTxt.

2.20 UNDOCUMENTED ROM ROUTINE FNDCOMTXT ($43E2)

In Grafix 80, this is the routine IEscLkDetor calls to see if the BASIC parser is looking at a command. To set up for the call, the processor's A register gets the high byte of a pointer to a table of command names, and Y gets the low byte of the pointer. This table holds a number of command names, and the last character in each command must have its high bit (bit 7) set to 1. Then FndComTxt is called with a JSR. It cruises through the table of command names, seeking a match for the character string that the BASIC interpreter is currently looking at via TxtPtr (see Sections 2.10 and 2.11). Upon return, the carry flag is set if the routine found that BASIC's sitting on a command from the pointed-to table, and cleared if not. If a valid command was found, memory location $000D holds the 0-based offset of the command within its table.

Be sure to refer to Chapter 3, Section 3.1 for more information on using undocumented ROM routines.

2.21 UNDOCUMENTED ROM ROUTINE FNDTKNTXT ($516A)

This routine, IEscPrDetor calls to un-crunch a token. To set up for the call, the processor's A register gets the high byte of a pointer to a table of command names, Y gets the low byte of the pointer, and X gets an adjusted offset into the 0-based table that shows the appropriate command name. This offset is slightly weird, in that it gets its high bit (bit 7) set. For example, if we want to print out the fifth command name in the table, the index passed to FndTknTxt is $84 (remember, the table is 0-based).

2.22 LOW-MEMORY ROUTINE CHRGOT

This routine is actually a subset of the ChrGet routine, engineered by entering that routine right after TxtPtr has been incremented. So, whereas ChrGet advances through a BASIC line, ChrGot grabs the currently pointed to byte. That is, it gets a character that's already been got. ChrGot comes in handy in parsing routines. You can call ChrGet to get the next character, play with it, even lose it in the play, then pick it up again with ChrGot. Also, some of the built-in parsing routines, like GetByt and GetWdByt, work through ChrGet. ChrGot lets you continue parsing after one of these calls. You'll find numerous real-world examples sprinkled throughout the Grafix 80 code, particularly in the various command parameter fetching routines.

2.23 8502 USAGE: DERIVING AN ABSOLUTE VALUE

This is a neat little trick, used in a couple of the Grafix 80 routines. Absolute value, in case you don't know, is the magnitude of a number, ignoring any negative signs. For example, the absolute value of both –5 and 5 is 5, the absolute value of both –126 and 126 is 126. In Grafix 80, absolute value is used when we're figuring the length of a line segment through subtraction, so we don't have to worry about the positions of the end-point coordinates in the subtraction.

As used, the procedure applies to one-byte signed values, which means they're in the range −128..127. The procedure involves a flip-flop followed by an incrementation. Here's a tidy little version:

```
        LDA RawValue         ; get the raw value
        BPL :Done            ; it's already positive
        EOR #$FF             ; for a negative value, flip-flop it
        CLC                  ; prepare to increment
        ADC #1               ; increment
:Done   STA AbsoluteValue    ; store the absolute value
```

To work the same procedure on multi-byte signed values, you just flip-flop each byte with an **EOR**, then add 1 to the result.

2.24 8502 USAGE: TWO-STAGE MASKING

Masking refers to the process of setting and clearing particular bits in a byte. It's synonym for the standard logical operations: **ANDing**, **ORing**, and **EXCLUSIVE ORing**. Grafix 80 does a lot of masking, much of it setting and clearing bits and bytes for the 80-column display and attribute memories. Sometimes, the masking is a two-stage process: an **AND** operation clears a range of bits, then an **OR** operation sets some of those bits. Or an **EXCLUSIVE OR** flip-flops a mask byte, followed by an **AND** and/or an **OR**.

Grafix 80 usually uses tables of masking bytes, picked up from the various routines through indexing. As you read the Grafix 80 code, you'll see I played some condensation tricks on these tables. I enjoyed discovering the close, symmetrical relationships between various **AND** and **OR** masks that let me squeeze these tables together.

2.25 8502 USAGE: NIBBLE TRANSFER

One of the holes in the 6502 family's assembly language is an easy way to shift nibbles within a byte. The difficulty stems from the fact that the **shift** and **rotate** instructions only move one bit position at a time. So **nibble** transfers—moving bits 0..3 into bits 4..7, or bits 4..7 into bits 0..3—have to be accomplished with multiple bit shifts. In Grafix 80, there's an example that shows how to move a byte's lo-nibble (bits 0..3) into the hi-nibble (bits 4..7). S/M ASM 2 has code that shows how to swap nibbles. If any of you readers come up with more efficient 6502 nibble transfer techniques, please write immediately to let me in on your discoveries.

2.26 UNDOCUMENTED ROM ROUTINE: GETBYT ($87F4)

This is the third undocumented ROM routine used in Grafix 80. Again, check out Chapter 3, Section 3.1 for more information on this dangerous practice, why I stooped to it, and other tidbits.

GetByt grabs a byte-sized integer parameter from a BASIC line. It doesn't matter whether the parameter's expressed as a constant or a variable. **GetByt** does all the necessary manipulations and conversions, then returns the recovered parameter's value in the processor's X register. The BASIC parser's pointer ends up at the next line element following the parameter. In Grafix 80, GetByt is called on to grab parameters for the various 80-column commands.

2.27 UNDOCUMENTED ROM ROUTINE: GETWDBYT ($8803)

This is the fourth and last undocumented ROM routine used in Grafix 80. It's a close relative toGetByt.

GetWdByt grabs a word-sized integer parameter and a byte-sized integer parameter from a BASIC line. It's a fairly common situation in BASIC 7.0 for a command to expect a pair of parameters to come in this order, separated by a comma. Thus, this function. Again, as in GetByt, it doesn't matter whether the parameters are expressed as constants or variables. GetWdByt does all the necessary manipulations and conversions, then returns the lo-byte of the word-sized parameter in memory location $0016, the hi-byte of the word-sized parameter in memory location $0017, and the byte-sized parameter in the processor's X register. The BASIC parser's pointer ends up at the next line element following the byte-sized parameter. In Grafix 80, GetWdByt is called on to grab parameters for the various 80-column commands. For example, a command may take a word-sized horizontal screen coordinate, and then a byte-sized vertical screen coordinate.

2.28 8502 USAGE: MULTI-BYTE DIVISION BY POWERS OF 2

Every time you shift a byte that represents an unsigned integer value one bit to the right, it has the same effect as dividing the byte's value by 2. This makes division by powers of 2 easy for such values, just shift right once for each power of 2 you want to divide by. But you may not know how to do the same kind of quick division when the number to be divided is stored in more than one byte. The trick is getting a bit 0 that shifts out of one of the bytes moved into bit 7 of the next-lowest byte in the number's multi-byte representation, so the chain of bit shifts is unbroken. This can be done with combinations of the LSR and ROR instructions. LSR shifts bits one position to the right, putting a 0 into bit 7 of the byte and moving bit 0 into the carry flag. ROR also shifts bits one position to the right, putting the contents of the carry flag into bit 7, then moving bit 0 into the carry flag. So, the way to accomplish division of a multi-byte unsigned integer value by 2 is to start with the hi-byte, do an LSR to get the ball rolling, then follow through with RORs on each of the lower bytes. Each time you carry this cycle through, each bit in the multi-byte representation moves 1 bit to the right, and the number gets divided by two. Repeat the cycle twice, you've divided by 4; three times, and you've divided by 8; and so on. Grafix 80 uses this technique to speed up a number of its calculations.

2.29 SCREEN CLEARING VIA BSOUT

This has been mentioned before, but Grafix 80 shows how sending a screen clearing character ($93) (147) to the kernel routine BSOut works on the 80-column VDC screen just as well as it does on the 40-column VIC screen.

2.30 KERNEL ROUTINE SWAPPER ($FF5F)

As I first scanned through my eagerly-awaited C-128 Prg, I remember coming across this built-in function and wondering how soon I'd be using it. Well, there it is in Grafix 80's ClrTx80 routine, written about three weeks after that first scan. Swapper switches you to the screen mode you're not in. That is, if you're in 40-column screen mode, it puts you into 80-column mode. If you're in 80-column screen mode, it puts you into 40-column mode. And it takes care of all the screen variables and tables that the system uses to fiddle with a screen. There's no preparation, just call it with a simple JSR.

2.31 CLEARING THE 80-COLUMN GRAPHICS SCREEN

Using BSOut to clear a screen only works when the screen's in text mode. Graphics mode is a little tougher. Grafix 80 has a little routine that clears the 80-column bit map. It uses the 80-column chip's block write capability. The algorithm is quite simple: for each 256 byte page of the 80-column bit map display memory, fill that page with zeros. 256 bytes is the most we can tell the VDC chip to work on in one block write command. Be sure to look at the code for the actual implementation of this algorithm.

2.32 SETTING THE 80-COLUMN CHIP FOR TEXT OR GRAPHICS

Bit 7 of VDC register 25 controls chip mode. Set the bit to 1 for bit map mode, 0 for text mode. In the normal, full-page (640 horizontal by 200 vertical) bit map situation, you'll have to disable attribute memory because there's no room for it. That's done by clearing bit 6 of register 25 to 0. When you go back to text mode, attribute memory is enabled by setting bit 6 of register 25 to 1.

In Grafix 80, I use the full 640 by 200 bit map. So, to enter bit map mode I clear bit 6 of VDC register 25 and set bit 7. Going back to text mode, I set bit 6 and clear bit 7 of VDC register 25.

2.33 80-COLUMN GRAPHICS COLOR CONTROL NIBBLES

In 80-column bit map mode, with a full 640 horizontal by 200 vertical display, almost all of the VDC chip's 16K of private RAM is taken up by bytes for the bit map. There's no room left over for attribute memory. So, without an attribute area, the bit map display is limited to two colors, one for the foreground and one for the background. Bits 0..3 (the lo-nibble) of VDC register 26 hold a color code in the 0..15 range for the background color, and bits 4..7 (hi-nibble) of the same register hold a code for the foreground color. If a byte in the bit map has a bit set to 1, it'll show up in the foreground color. If a bit is set to 0, it'll show up in the background color. See Section 2.35 for more information on the color codes used in 80-column operations.

2.34 80-COLUMN GRAPHICS PIXEL OPERATIONS

In order to draw or clear a dot on the 80-column graphics screen, you set or clear the appropriate bit in the appropriate byte of the VDC bit map memory. How to find the appropriate bit and byte? There are 200 vertical positions, or rows, in the bit map, and each row has 640 horizontal positions, or columns. Each byte in the bit map memory controls eight pixels, and eighty bytes map one row of the bit map (640 pixels). For once in my computer graphics work, the bit map is organized logically; that is, the first 80 bytes map the first row of pixels, the next 80 bytes map the second row, and so on.

Grafix 80 uses a routine called FigPoint to figure out bit and byte information for a given screen pixel. First, it figures out the address of the byte the pixel lives in. It takes the point's vertical coordinate, and uses it to index into some tables to build the address of the first byte in the point's row. The easy way is to have tables with an address for each of the 200 rows. To save some memory space, I've used shortened tables, and done some tricky indexing based on regularities in the addresses, but the idea's the same. After getting the address of the starting byte of the pixels's row, FigPoint takes the pixels's horizontal coordinate and divides it by 8, since there are eight pixels controlled by each bit map byte. The integer, part of the division's result tells us the row offset of our pixel's byte; that is, how many bytes into the row the pixel's byte is at. I add that row offset value to the row's starting address, and get the exact address of the pixel's byte. The remainder from the horizontal coordinate division gives me the bit position in that byte that controls the given pixel.

Okay. I've got a pixel's bit position in its byte, and the byte's location in the VDC RAM. To turn that pixel on, I just grab the byte, set the appropriate bit to 1, and store the byte back into position. Section 1.2.30 covers grabbing and storing VDC RAM bytes. Setting the appropriate bit to 1 is done with a mask and an ORA command. The pixel's bit position is used as an index into a table of ORing masks. Turning a pixel off is just about the same, except that this time we clear the appropriate bit to 0 with a mask and an AND command. This time, the pixel's bit position is used to index into a table of ANDing masks. In Grafix 80, all this stuff gets done in the routines PlotIt, GetTargByt, PutTargByt, and PixelPop.

2.35 BLOCK WRITE, BLOCK COPY, 80-COLUMN SCREEN REGISTERS 24 AND 30 (WORD COUNT) AND 32 (BLOCK START ADDRESS HI) AND 33 (BLOCK START ADDRESS LO)

Registers 24, 30, 32, and 33 let you carry out block writes and block copies. Block writes let you fill contiguous areas of VDC RAM with the same byte. Block copies let you copy the contents of one contiguous area of VDC RAM to another contiguous area. In both procedures, VDC register 30 is used to indicate how many bytes are in the memory block. Bit 7 of register 24 is used to indicate whether a block function is block copy or block write. Registers 32 and 33 are used in block copies to indicate the starting address of the transfer's source block. In Grafix 80, block writing is used to clear the graphics screen.

Here's the official C-128 Prg block write procedure:

1. Using the VDC register writing algorithm, set VDC register 18 to the hi-byte of the address of the initial byte in the block.
2. Similarly, set VDC register 19 to the lo-byte of the address of the initial byte in the block.
3. Using the VDC register writing algorithm, put the value you want to write to the block into VDC register 31 (the data register).

4. Using the VDC register writing algorithm, clear bit 7 of VDC register 24 to 0 to select the block write function.
5. Again using the register writing algorithm, put into VDC register 30 the number of bytes in the block less one (since step 3 already wrote 1 byte)

Here's the official C-128 Prg block copy procedure:

1. Using the VDC register writing algorithm, set VDC register 18 to the hi-byte of the address of the initial byte in the destination block.
2. Similarly, set VDC register 19 to the lo-byte of the address of the initial byte in the destination block.
3. Using the VDC register writing algorithm, set bit 7 of VDC register 24 to 1 to select the block copy function.
4. Using the VDC register writing algorithm, set VDC register 32 to the hi-byte of the address of the initial byte in the source block.
5. Similarly, set VDC register 33 to the lo-byte of the address of the initial byte in the source block.
6. Again using the register writing algorithm, put into VDC register 30 the exact number of bytes in the block.

2.36 80-COLUMN COLOR NIBBLES

When working with the 80-column screen, you get to set colors with nibble-sized codes. But, unlike VIC graphics, you can't just poke the standard C-128 BASIC color numbers into attribute memory or the color registers. That's because the VDC color nibble codes are used directly to control the four components of the chip's IBM-PC-like color scheme: red, green, blue, and intensity. There's a mistake in the C-128 Prg concerning this subject: on page 302, they tell you that the nibble codes are determined by taking the BASIC color codes and subtracting one. Nope. The codes are found by figuring out which components are necessary to produce each of the sixteen colors, then putting together a nibble by representing each present component with a one bit, each absent component with a zero bit. Four components, four bits, one nibble. Got it? Appendix I shows all the VDC colors, the BASIC codes, and the nibble codes. You can also find the same information at the end of Grafix 80, in the table HuNb80Tb.

Chapter 3: Program Notes

3.1 UNDOCUMENTED ROM ROUTINES

You should NEVER use undocumented ROM routines in a commercial software product. ROMs change quickly, and undocumented routines are usually misplaced in the change.

However, since (1) Grafix 80 is written for learning, not selling, and (2) the undocumented ROM routines I've used are long, twisted and handy, and (3) replacement routines would add many pages to the source code, and (4) the undocumented ROM routines are used only in the BASIC 7.0 interface to the 80-column routines. I've used four of the no-no's. Two are used in BASIC command text searches :FndComTxt and FndTknTxt. Two others are used while fetching graphics command parameters from BASIC: GetByt and GetWdByt. You'll find descriptions of these four routines in Chapter 2: Sections 2.20, 2.21, 2.26, and 2.27 respectively.

Of course, you may own a C-128 with different ROMs than mine. What to do if you want to use the 80-column graphics routines? There are two solutions. Here's one: you can call the routines from assembly language. That means you'll have to fiddle with the source code a bit, excising all the parts that have to do with BASIC interfacing, then call the remaining routines as needed. More precisely: remove Install, UnInstall, InsCrchDtr, InsUncrDtr, InsExecDtr, RmvCrchDtr, RmvUncrDtr, RmvExecDtr, IEscLkDetor, IEscPrDetor, IEscExDetor, CommaCruz, DoG80Box, G8BoxGtPs, DG80Color, G8ColGtPs, DoG80Draw, G8DrwGtPs, DoG80Graphic, and G8GrfGtPs. Remove the running mode check and call to uninstall from DoG80Scat. Remove the call to install from G80 Install.S. Recompile the code, then run G80 Install.

Now, to call one of the 80-column commands from assembly language, you set up the command's parameters, optionally call the command's parameter-checking routine (G8xxxChPs), then call the command's execution routine (G8xxxDoIt). Another tribute to heavily-modular code. Section 3.10 has a few more details on calling the routines from assembly language.

There's a second way to deal with a ROM change that moves the undocumented routines in Grafix 80. It's a little tougher to pull off, but it maintains Grafix 80's full power. What you do is find the four routines in your ROM, then replace the old addresses with the new ones. The replacement is done by changing lines 192, 194, 196, and 198 in the constants section at the beginning of the Grafix 80 source code. Finding the new routine addresses is done with the C-128 monitor. Each of the routines has a unique sequence of bytes, or signature, that lets you find it. These signatures have a good chance of surviving ROM changes. Figure 3-1 shows each routine's current unique signature.

Here's an example. If you want to find FndComTxt, enter the monitor and give this command:

```
H F4000 FFFFF 85 25 84 24 A0 00 84
```

This tells the monitor to hunt in the bank 15 memory range $4000 . . $FFFF for the sequence of seven bytes that make up FndComTxt's signature. On my C-128, the monitor comes back with the single address $43E2, which is where FndComTxt lives. Note that in some cases the address returned by the monitor after a signature search has to be adjusted; again, Fig. 3-1 tells all.

Sometimes these signatures do get changed. In that case, you have to use the monitor to search for assembly language similar to the start of the current routines. Figure 3-2 shows the first few commands for each of the four routines. This code is pretty posi-

Undoc'd ROM Routine's Name	Location In My C–128's ROM	Unique Signature In My C–128's ROM	Adjustments To Address Returned By Monitor Search
FndComTxt	$43E2	85 25 84 24 A0 00 84	none
FndTknTxt	$516A	85 25 84 24 A0 00 CA	none
GetByt	$87F4	A6 66 D0	subtract 6
GetWdByt	$8803	A6 67 4C	add 5
Start Monitor search command with: H F4000 FFFFF			

Fig. 3-1. If your C-128 has a different ROM than the author's, you may have some difficulty finding the four undocumented ROM routines used in the Grafix 80 project. This information should help you find them.

```
FndComTxt       STA     $25
                STY     $24
                LDY     #$00
                STY     $0D
                DEY
                INY
                LDA     ($3D),Y
                SEC
                SBC     ($24),Y

FndTknTxt       STA     $25
                STY     $24
                LDY     #$00
                DEX
                BPL     *+$11
                LDA     ($24),Y
                PHA
                INC     $24
                PLA

GetByt          JSR     $77D7
                JSR     $84AD
                LDX     $66
                BNE     *+$2F
                LDX     $67
                JMP     $386

GetWdByt        JSR     $77D7
                JSR     $8815
                JSR     $795C
                JMP     GetByt
```

Fig. 3-2. Here are the first few instructions for each of the four undocumented ROM routines used in the Grafix 80 project.

tion independent, so the odds are it won't change very much in any revised ROMs. And, if the routines do move, they usually don't move very far.

Phew. See why we never use undocumented ROM routines in commercial code?

3.2 MOVING BASIC UP TO FIT CODE BENEATH IT

The C-64 has a number of nice nooks and crannies for stuffing assembly language programs. The C-128 has even more. But, if your routines are going to interact heavily with BASIC's interpreter, it's particularly useful to put code in RAM bank 1, beneath BASIC's text area. That minimizes memory configuring when the assembly routines are running. The area \$1300-\$1C00 is usually available for this purpose. It provides 2304 bytes of memory space.

But the Grafix 80 code is large. I needed even more room. To get it, I moved the start of BASIC's text area up 512 bytes. G80 Install contains the code that does this, and Sections 2.2 and 2.3 describe the process.

3.3 HORIZONTAL LINE DRAWING ALGORITHMS

Horizontal lines can be drawn faster than any other lines on most bit-mapped computer displays, including the C-128's. That's because these displays use consecutive bytes of screen memory to represent adjacent horizontal screen pixels. This organization of data gives us what the computer science types call coherence. The fundamental idea behind all fast horizontal line drawing algorithms is finding the byte that controls the leftmost pixel in the line, adjusting that pixel's bit, then continuing to adjust subsequent bits up through the rightmost pixel's bit. Only the first pixel's byte's location has to be calculated or looked up; subsequent pixel's bytes follow consecutively. On the C-128 this coherence is especially helpful, due to the nature of 80-column screen RAM access.

The algorithm I use in the 80-column routines divides horizontal lines into three cases. The first case is what I call a one-part line, in which the line's pixels are controlled by one byte of screen memory. The second case is what I call a two-part line, in which the line's pixels are controlled by two bytes of screen memory. The third case is what I call a three-part line, in which the line's pixels are controlled by three or more bytes of screen memory. Figure 3-3 shows examples of these three line cases. And sheets 12 thru 15 of Fig. 7-2 give complete pseudo-code for the horizontal line drawing algorithms.

To draw a one-part line, I grab a mask for the left part of the line, grab one for the right part, take their intersection to get a mask that represents the whole line, then use that mask to adjust bits in the line's byte. Figure 3-4 gives a picture of the process.

To draw a two-part line, I grab a mask for the left part of the line, then use that mask to adjust bits in the left part's byte. Then I grab a mask for the right part of the line, and use it to adjust bits in the right part's byte. Figure 3-5 gives a picture of this process.

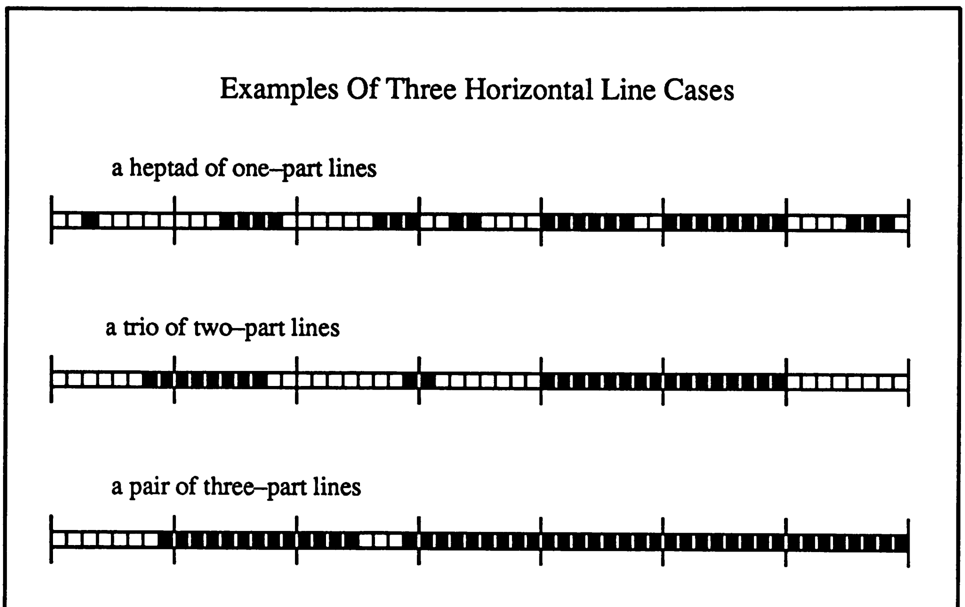


Fig. 3-3. Our algorithm for drawing horizontal lines divides such lines into three classes: one-part lines, two-part lines, and three-part lines. Here are some examples of each class.

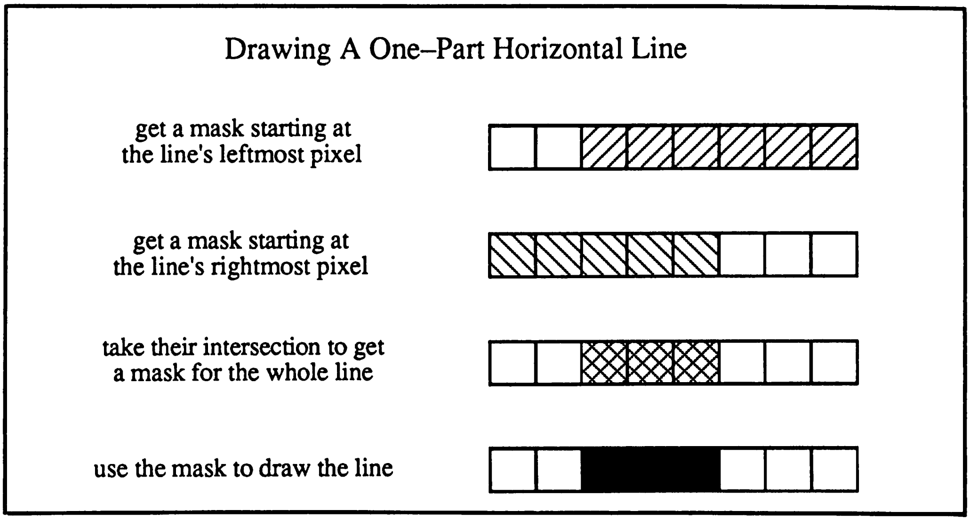


Fig. 3-4. Our algorithm for drawing a one-part horizontal line.

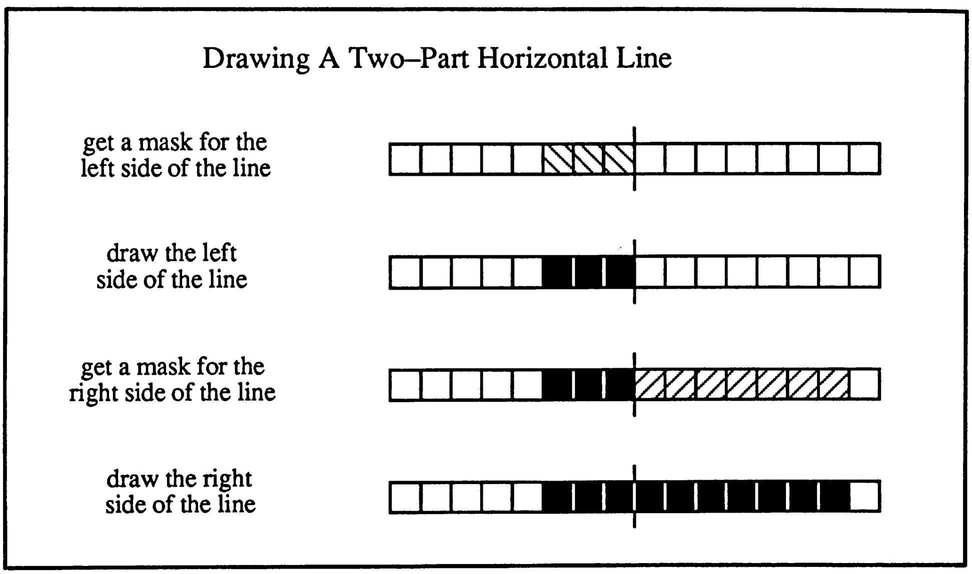


Fig. 3-5. Our algorithm for drawing a two-part horizontal line.

To draw a three-part line, I do the left and right parts the same as for a two-part line. The middle section of the line, which consists of one or more whole bytes, is done by preparing an adjusted byte, then storing it in each byte. Figure 3-6 gives a picture of this process.

3.4 VERTICAL LINE DRAWING ALGORITHM

Vertical lines are also easy to draw on most bit-mapped displays. Coherence in these cases comes from the fact that the pixels in a vertical line are controlled by bytes that

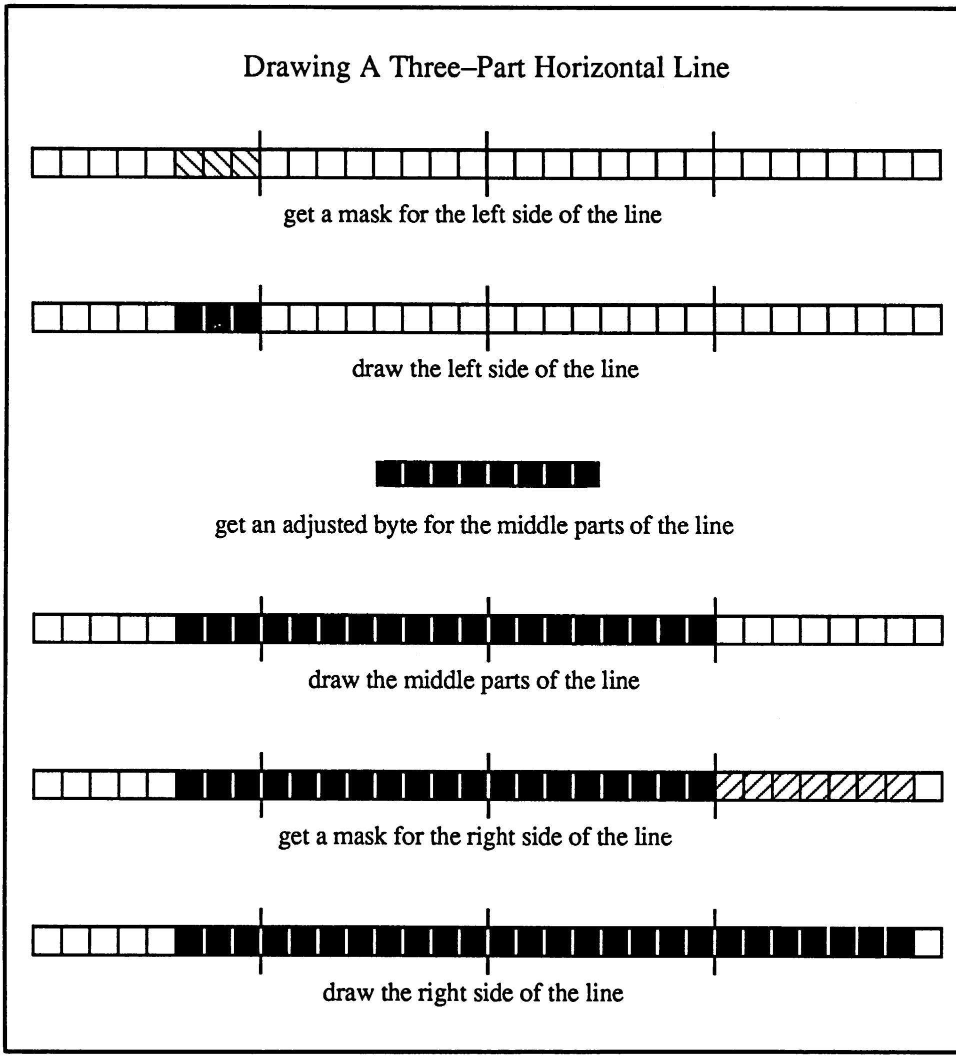


Fig. 3-6. Our algorithm for drawing a three-part horizontal line.

are separated by a constant number of memory locations. If you think about it, you'll realize that this constant is the number of bytes that controls a row of the bit-mapped display.

This leads to the simple algorithm used in Grafix 80. Start with the topmost point in the line. Plot it. Then move down to the next pixel's byte by adding the appropriate constant. See sheet 12 in Fig. 7-2.

3.5 BRESENHAM'S GENERALIZED LINE DRAWING ALGORITHM

Horizontal and vertical lines can be drawn quickly, as detailed before, since there's no need for heavy calculation once the first point of the line is located. Other lines—the slanted ones—aren't quite so simple. Early generalized line drawing algorithms required one or more multiplications to calculate each pixel's location. Multiplication is slow, as are the resulting algorithms.

Then along came a person named Bresenham, who did a little mental and algebraic manipulation, and produced a generalized line-drawing algorithm that only requires additions and subtractions. These operations can be done quickly. Bresenham's work is the basis for the generalized line-drawing algorithm used in Grafix 80. In the discussion that follows I'll refer to the algorithm as BGLDA (for Bresenham's Generalized Line Drawing Algorithm).

Think of drawing a line as moving a point from one of the line's endpoints to the other. When the point reaches its destination, it will have achieved a net change in both its vertical and horizontal positions. In an ideal world, where display screens have infinite resolution, each time the point moves one integer position horizontally, it moves some amount vertically. In most lines this amount won't be an integer value. Here in the real world, though, screens have finite resolution, and we can only move one pixel at a time; that is, we're limited to integer value motion. What the BGLDA does is keep track of the real vertical motion in a variable I call the erometer. Every now and then the erometer overflows. That is, the real motion reaches an integer value. At that point we move one position vertically, then reset the erometer.

The values used to adjust the erometer are based on the line's slope. Different values are used for steep (slope greater than 45°) and shallow (slope less than 45°) lines. Different values are used for lines that rise or fall as the line goes from left to right. And, just to complicate the previous paragraph's discussion, sometimes the erometer keeps track of horizontal motion, rather than vertical. All these implementation details can be seen in the BGLDA pseudo-code, located on sheets 15 thru 18 of Fig. 7-2. Even more detail can be found in the actual assembly language code, located on sheets 41 thru 46 of Fig. 8-2. But, remember, all of this is just detail. The big idea of the BGLDA is setting up a variable that keeps track of non-integer changes, then overflows once an integer value is reached.

3.6 MULTIPLE SOURCE CODE FILES

The C-128 doesn't have enough room to hold large assembly language source files. Most assemblers let you get around this by breaking a project up into a number of smaller source files. This is done in the Grafix 80 program. The code is broken up into six files. The files are connected by using the Merlin-128 pseudo-op **PUT**. See Appendix O for details on this pseudo-op's usage.

3.7 CHEAP BOX TRICKS

The Grafix 80 routines take it easy when it comes to drawing boxes. Outlined boxes are drawn by calling on the horizontal and vertical line drawing routines. Filled boxes are drawn by calling repeatedly on the horizontal line drawing routines. And, since these routines are so quick, Grafix 80 box drawing is much faster than the C-128's built-in 40-column box routines.

3.8 RANGE ADJUSTMENT TO OPTIMIZE TESTING

The Grafix 80 routine **G8ColChPs** checks to see if parameters to the G80Color command are in range. It pulls a little optimization trick when it checks the color number parameter. The color parameter can take on values in the range 1 . . 16. **G8ColChPs** decrements the parameter value so it can check for values in the range 0 . . 15. This is an easier range to check with assembly language code.

3.9 GENERALIZING A DRAW COMMAND WITH A POINT LIST

The G80Draw command operates like its 40-column counterpart, in that it can take a list of points as input and then draw a series of connected lines. In the Grafix 80 package this ability is implemented by building a list of points.

The point list in turn is implemented as an array of point entries, **G8DrwLst**. This array can hold up to 33 points. Each point is stored as three bytes: two bytes for a horizontal coordinate and one byte for a vertical coordinate. The variable **G8DLPts** keeps track of how many points are currently in the list. A zero-page variable, **DLPntr**, points to the beginning of the list. Finally, the variable **G8DLNdx** indexes off of **DLPntr** to point to the next open slot in the list.

G8DrwGtPs fetches parameters for the G80Draw command. It starts out by calling **InitPntLst** to create a blank point list. Then, as points are picked up from the BASIC line, it calls on the routine **StorPntLst** to add them to the point list.

G8DrwDoIt draws single points and line segments. It checks the point list to see what to do. No points, and it simply exits. One point, and it calls on **FigPoint** and **PlotIt** to draw a single point. More than one point, and it draws a series of line segments that connect the points.

Here's the simple algorithm **G8DrwDoIt** follows for point lists containing two or more points (stated slightly differently here than on sheet 10 of Section Fig. 7-2, but amounting to the same code):

```
grab two points from the point list
CALL on DoLine to draw a line connecting them
WHILE
  there's a next point to grab from the point list
DO the following
  grab that next point
  CALL on DoLine to draw a line connecting it to the last line drawn
```

3.10 CALLING THE 80-COLUMN GRAPHICS ROUTINES FROM ASSEMBLY LANGUAGE

This was mentioned briefly in Section 3.1. Each of the G80 Grafix commands sports a modular design, as follows: Each command has an execution routine, named **DoG80xxx**. This routine is followed by a set of command parameter variables. The execution routine calls on three subsidiary routines: **G8xxxGtPs**, which fetches any command parameters from the BASIC line; **G8xxxChPs**, which checks the legality of command parameters, and **G8xxxDoIt**, which carries out the command. To call one of the G80 Grafix commands from assembly language, start by plugging parameters directly into the command's parameter variables. If you don't trust your ability to pass clean parameters, call the command's **G8xxxChPs** routine to check their legality. Then just call the **G8xxxDoIt** routine to carry out the command.

You can see an example of this in the routine G8GrfDoIt, which sets up parameters and then calls on G8ColDoIt.

3.11 COMMAND VARIATION BASED ON WHETHER THE BASIC INTERPRETER'S IN DIRECT MODE OR RUNNING A PROGRAM

The Grafix 80 package shows you how to add commands to BASIC 7.0. When you add a command, it's easy to customize things so the command's behavior is dependent on whether the interpreter's running a program or working in direct mode. The key is memory location $007F. A zero value in this location indicates direct mode. A non-zero value indicates program mode. The DoG80Scat routine gives an example of using this location's value (the G80SCAT command works only in direct mode).

3.12 MODULARITY AND OPTIMIZATION

You'll notice that the routines used in the Grafix 80 package are highly modular. This has some effect on the speed of execution. If you wanted to optimize for speed, you might be tempted to combine some of the routines into more straight-line code. Be careful. Modularity is just too useful to be thrown out in any but the most time-critical situations. If you do need some extra speed, the first thing to do is examine your algorithms for possible speedups. Then look for ways to speed up inner loops. Unrolling a loop may be useful. Unrolling a loop just means that you do more things in the body of the loop. The tradeoff is space for time. For example, a loop like this:

```
FOR
    position = 1 TO 20
DO the following
    erase (position)
```

. . . can be unrolled into a loop like this:

```
FOR
    position = 1 to 20 STEP 4
DO the following
    erase (position)
    erase (position+1)
    erase (position+2)
    erase (position+3)
```

The second version's code will be longer, but will usually run faster.

3.13 TABLES, TABLES, TABLES

The Grafix 80 routines use a number of tables. There are tables for BASIC commands, masking pixels, addressing screen memory, and selecting colors. Tables let your programs run quickly and cleanly. And changes are easy to make. Actually, heavy table usage is just a subset of the following good idea: separate your code and data. That way you can make certain types of modifications to one without worrying about the other. Apple's Macintosh systems make powerful use of this idea with their implementation of the resource concept. But you can do similar things on any system, including the C-128.

3.14 ADDING COMMANDS TO THE PACKAGE

There are five steps to follow to add a command to the Grafix 80 package. The first four hook your command into BASIC 7.0, and are pretty mechanical. The fifth step demands a bit more of the mind: writing the code to execute the new command.

Let's go through the steps. First: go to the token stuff area of the Constants section of the Grafix 80 source code (lines 246-263 of Grafix 80 O.S.). Notice how there's a selector token equate for each of the commands already added to BASIC: **TknBox**, **TknColor**, **TknDraw**, **TknGraphic**, and **TknScat**. The values of these selector tokens begin at $27, with each command taking the next value. What you need to do is add an equate for your new command, using the next available value. For example, the first command you add could have an equate like this:

```
TknAnim = $2C     ;selector token for G80ANIMATE
```

Next step is to adjust the constant **OurLast**, in the same section of the source code, so it's equal to the selector token with the greatest value. For example, if the example above were the only command you were adding, the new equate would look like this:

```
OurLast = TknAnim     ;last selector token for our commands
```

The third hookup step is to add the command to the **OurComsText** table. This table is located at line 350 of the source file Grafix 80 5.S. It holds ASCII code for each command, with this twist: the last character of the command gets its high bit set. Commands are listed in the order of their selector tokens. The Merlin-128 assembler lets you put ASCII strings in the proper format for this list with the **DCI** pseudo-op. Here's how you'd add our example command:

```
DCI     'g80animate'
```

The fourth step is to add an execution branch for the new command to the **IEscExDetor** routine, located around line 240 of the source file Grafix 80 1.S. The pattern for these branches should be evident from the current **IEscExDetor** code. Here's how you'd add a branch for the example command we've been using:

```
:Tst6 CMP  #TknAnim     ; is it G80ANIMATE command?
      BNE  :Tst7        ; if not, next test
  (or BNE  :NotOurs       if this is the last test)
      JSR  DoG80Anim    ; it is, so execute the command
      BCC  :GoneZo1     ; if we make it back, always branches
```

So much for the mechanical steps. The fifth and final hookup step is writing the routine that'll actually carry out the command. This execution routine must satisfy the following pseudo-code conditional:

```
IF
    the routine executes successfully
```

THEN
    the A-, X-, and Y- registers are preserved
    return from the routine is via an RTS
    the carry flag returns cleared
ELSE {the routine ran into an error condition}
    the A- and Y- registers are preserved
    the X register gets loaded with an error code
    return from the routine is via a JMP thru the IError vector
    the carry flag returns set

The execution routine will usually pick up some parameters from the BASIC line, although that's not always the case (see the Grafix 80 routine **DoG80Scat**). The

draw 100 random dots				
	ten trials			
	all times are in seconds			
graphics chip	40	40	40	40
width of drawing area	160	160	320	320
processor speed	slow	fast	slow	fast
average time per trial	1.15	0.54	1.15	0.54
standard deviation	0.01	0.01	0.01	0.01
relative speed (1 = fastest)	2.13	1.00	2.13	1.00
graphics chip	80	80	80	80
width of drawing area	160	160	320	320
processor speed	slow	fast	slow	fast
average time per trial	1.17	0.56	1.18	0.56
standard deviation	0.01	0.01	0.01	0.01
relative speed (1 = fastest)	2.17	1.04	2.19	1.04
graphics chip	80	80		
width of drawing area	640	640		
processor speed	slow	fast		
average time per trial	1.18	0.56		
standard deviation	0.01	0.01		
relative speed (1 = fastest)	2.19	1.04		

Fig. 3-7. Results from performance tests on drawing random dots.

draw 100 random vertical lines				
	ten trials			
	all times are in seconds			
graphics chip	40	40	40	40
width of drawing area	160	160	320	320
processor speed	slow	fast	slow	fast
average time per trial	4.89	2.31	4.89	2.31
standard deviation	0.27	0.13	0.27	0.13
relative speed (1 = fastest)	2.19	1.04	2.19	1.04
graphics chip	80	80	80	80
width of drawing area	160	160	320	320
processor speed	slow	fast	slow	fast
average time per trial	4.73	2.24	4.74	2.23
standard deviation	0.26	0.13	0.26	0.12
relative speed (1 = fastest)	2.12	1.00	2.13	1.00
graphics chip	80	80		
width of drawing area	640	640		
processor speed	slow	fast		
average time per trial	4.76	2.24		
standard deviation	0.26	0.12		
relative speed (1 = fastest)	2.13	1.00		

Fig. 3-8. Results from performance tests on drawing random vertical lines.

DOG80xxxx routines in Grafix 80 provide a number of examples of setting up execution routines.

3.15 PERFORMANCE TESTING

I used the programs G80 Test Suite and G40 Test Suite to test the performance of the 80-column graphic commands. The results are summarized in Figs. 3-7 thru 3-12. The source code for the two programs is in Figs. 8-3 and 8-4.

I tested performance by drawing pseudo-random samples of six types of graphics objects: dots, vertical lines, horizontal lines, lines, outlined boxes, and filled boxes. For each type of graphic object, there were four test sets run on the 40-column screen using the BASIC 7.0 drawing commands, and six test sets on the 80-column screen using the

draw 100 random horizontal lines				
	ten trials			
	all times are in seconds			
graphics chip	40	40	40	40
width of drawing area	160	160	320	320
processor speed	slow	fast	slow	fast
average time per trial	4.37	2.07	6.64	3.14
standard deviation	0.14	0.06	0.26	0.13
relative speed (1 = fastest)	4.33	2.05	6.57	3.11
graphics chip	80	80	80	80
width of drawing area	160	160	320	320
processor speed	slow	fast	slow	fast
average time per trial	2.13	1.01	2.16	1.04
standard deviation	0.01	0.01	0.01	0.01
relative speed (1 = fastest)	2.11	1.00	2.14	1.03
graphics chip	80	80		
width of drawing area	640	640		
processor speed	slow	fast		
average time per trial	2.23	1.08		
standard deviation	0.01	0.01		
relative speed (1 = fastest)	2.21	1.07		

Fig. 3-9. Results from performance tests on drawing random horizontal lines.

Grafix 80 extensions to BASIC 7.0. Test sets varied by the width of the active drawing area and whether the processor speed was 1 or 2 megahertz. Ten trials, each based on a different random seed, were run for each test set. The ten random seeds were (1, 2, 45, 1291, 5987, 8711, 9261, 22222, 28835, 33287). Each trial drew 100 pseudo-randomly located and sized instances of the graphic object.

The figures give three performance values for each test. The first indicates the average time (in seconds) per trial of 100 instances. The second value—standard deviation, indicates the time variation between trials. The lower the standard deviation, the smaller the time variation, signalling an algorithm that provides more consistent performance over different data worlds. Finally, there's a number that compares the average trail times of the ten tests. The bigger the value here, the slower the test. 1.00 indicates the fastest test.

Among the primary design goals for the Grafix 80 routines were consistency, reliability, and lucidity. The reasonably low 80-column standard deviation values help indicate this. Speed concerns were met by algorithmic refinement rather than code trickery. And the bottleneck interface to 80-column display memory slows down any 80-column graphics work. Interestingly, the Grafix 80 routines run as fast or faster than their 40-column counterparts. And, unlike the 40-column routines, the Grafix 80 routines can be called directly from assembly language, which provides a marked speedup.

draw 100 random lines				
	ten trials			
	all times are in seconds			
graphics chip	40	40	40	40
width of drawing area	160	160	320	320
processor speed	slow	fast	slow	fast
average time per trial	5.86	2.77	7.67	3.62
standard deviation	0.22	0.10	0.28	0.13
relative speed (1 = fastest)	2.12	1.00	2.77	1.31
graphics chip	80	80	80	80
width of drawing area	160	160	320	320
processor speed	slow	fast	slow	fast
average time per trial	6.27	3.07	8.27	4.08
standard deviation	0.25	0.12	0.28	0.14
relative speed (1 = fastest)	2.26	1.11	2.99	1.47
graphics chip	80	80		
width of drawing area	640	640		
processor speed	slow	fast		
average time per trial	13.05	6.51		
standard deviation	0.54	0.27		
relative speed (1 = fastest)	4.71	2.35		

Fig. 3-10. Results from performance tests on drawing random lines.

draw 100 random outlined boxes				
	ten trials			
	all times are in seconds			
graphics chip	40	40	40	40
width of drawing area	160	160	320	320
processor speed	slow	fast	slow	fast
average time per trial	12.96	6.13	17.47	8.26
standard deviation	0.64	0.30	0.82	0.38
relative speed (1 = fastest)	3.52	1.67	4.75	2.24
graphics chip	80	80	80	80
width of drawing area	160	160	320	320
processor speed	slow	fast	slow	fast
average time per trial	7.76	3.68	7.82	3.72
standard deviation	0.52	0.24	0.52	0.24
relative speed (1 = fastest)	2.11	1.00	2.13	1.01
graphics chip	80	80		
width of drawing area	640	640		
processor speed	slow	fast		
average time per trial	7.93	3.80		
standard deviation	0.52	0.25		
relative speed (1 = fastest)	2.15	1.03		

Fig. 3-11. Results from performance tests on drawing random outlined boxes.

draw 100 random filled-in boxes				
	ten trials			
	all times are in seconds			
graphics chip	40	40	40	40
width of drawing area	160	160	320	320
processor speed	slow	fast	slow	fast
average time per trial	267.30	126.36	532.22	251.60
standard deviation	30.66	14.50	61.11	28.89
relative speed (1 = fastest)	44.11	20.85	87.83	41.52
graphics chip	80	80	80	80
width of drawing area	160	160	320	320
processor speed	slow	fast	slow	fast
average time per trial	11.83	6.06	13.55	7.32
standard deviation	0.93	0.49	1.13	0.63
relative speed (1 = fastest)	1.95	1.00	2.24	1.21
graphics chip	80	80		
width of drawing area	640	640		
processor speed	slow	fast		
average time per trial	16.83	9.74		
standard deviation	1.48	0.90		
relative speed (1 = fastest)	2.78	1.61		

Fig. 3-12. Results from performance tests on drawing random filled-in boxes.

Chapter 4: Stretching

4.1 SPECIAL CASING FOR 45° LINES

Right now the Grafix 80 routines have special code for horizontal and vertical lines. Another special case you may want to code for are 45° lines. They're easy to recognize: the vertical and horizontal displacements are equal. And they're pretty easy to draw: to get to the next pixel in a 45° line, just move one position vertically, then move one position horizontally. For simplicity, you'll probably want to start at the leftmost end-point of the line. The code for vertical lines in Grafix 80 shows you the details of moving vertically. And the Bresenham code shows you the details of moving horizontally.

4.2 OTHER GEOMETRIC FIGURES

Another way you can expand the Grafix 80 package is with commands to draw other geometric figures: circles, ovals, regular polygons.

The circle and oval algorithms are too complex to describe here, but I can point you at two good books: Artwick's *Applied Concepts in Microcomputer Graphics* and Foley & Van Dam's *Fundamentals Of Interactive Computer Graphics*. Some of the hottest algorithms for these figures are coded into the Apple Macintosh ROM, but you'll need to disassemble the code, not a trivial task.

Regular polygons (triangles, hexagons, pentagons, et al.) are simpler. A polygon is drawn as a series of connected lines. Given a starting point and an orientation, simple applications of trigonometry let you figure the endpoints of these connected lines. Refer to any good high school trigonometry textbook for the details.

4.3 CODE UNFOLDING

The Grafix 80 code is highly modular. This nurtures reliability and keeps the code size down. It also slows performance, due to the overhead of register-preserving function calls. If you like, you can speed things up by unfolding the code.

For example, the Grafix 80 line drawing routines (DoHorz, DoVert, and DoBres) go through a chain of calls to plot a point on the screen: they call on PlotIt, which calls on GetTargByt, PixelPop, and PutTargByt, which make calls to VDCRegPoke, VDCMemPeek, and VDCMemPoke. You could replace all of these subroutine calls with the actual subroutine code. That's code unfolding. Speed would be increased, at the expense of memory.

4.4 BIT-MAPPED TEXT

A feature missing from the Grafix 80 package is the ability to draw text characters on the bit-mapped screen. Actually, bit-mapped text drawing can be done at many levels of sophistication. I'll outline a few of those levels here, along with some implementation hints. Note: as with many programming tasks, text drawing sophistication and algorithmic complexity increase together.

The simplest form of text drawing on a bit-mapped screen simulates a text display. For the C-128's 80-column screen that means each character is drawn within a box that's eight pixels wide and eight pixels high, and these boxes are aligned on a grid that's 80 characters wide and 25 characters high. Eight bytes of data code a character, each byte representing a row of the character's image. To draw a character, the eight character image bytes are transferred to screen memory. Given the 80h by 25v alignment constraints, each image byte falls evenly within a byte of screen memory. This simplifies the transfer of image bytes. The next programming project in this book, the sound/music lab, draws aligned text on the 40-column bit-mapped screen. The routine DrawBMChar does the work; you can find its source code in Fig. 16-1. Drawing text on the 80-column screen differs only in the details of screen memory arrangement and access.

A more sophisticated level of bit-mapped text drawing lets you draw characters at any screen position, not just aligned to an 80h by 25v matrix. This means that the bytes of the character image won't necessarily line up with the bytes of screen memory. If the bytes do line up, transfer is as described in the previous paragraph. If they don't line up, each image byte has to be shifted across into two bytes. Then masks are made, and the transfer of image data can be carried out.

An even higher level of bit-mapped text drawing lets you draw characters of varying widths. This means that you need to have width information for each character, and that character images may fall anywhere in relation to screen memory byte boundaries.

Finally, how about being able to draw characters in any orientation on the screen? That is, not just placing them horizontally, left to right, but at various angles. This is most easily done by performing various planar transformations on the character image data. Details on these sorts of transformations can be found in Foley & Van Dam's *Fundamentals Of Interactive Computer Graphics.*

Chapter 5: Calling Structure Diagrams

This chapter consists of four figures, as follows:

Fig. 5-1—calling structure diagram for G80 Install (1 sheet).
Fig. 5-2—calling structure diagram for Grafix 80 (8 sheets).
Fig. 5-3—calling structure diagram for G80 Test Suite (1 sheet).
Fig. 5-4—calling structure diagram for G40 Test Suite (1 sheet).

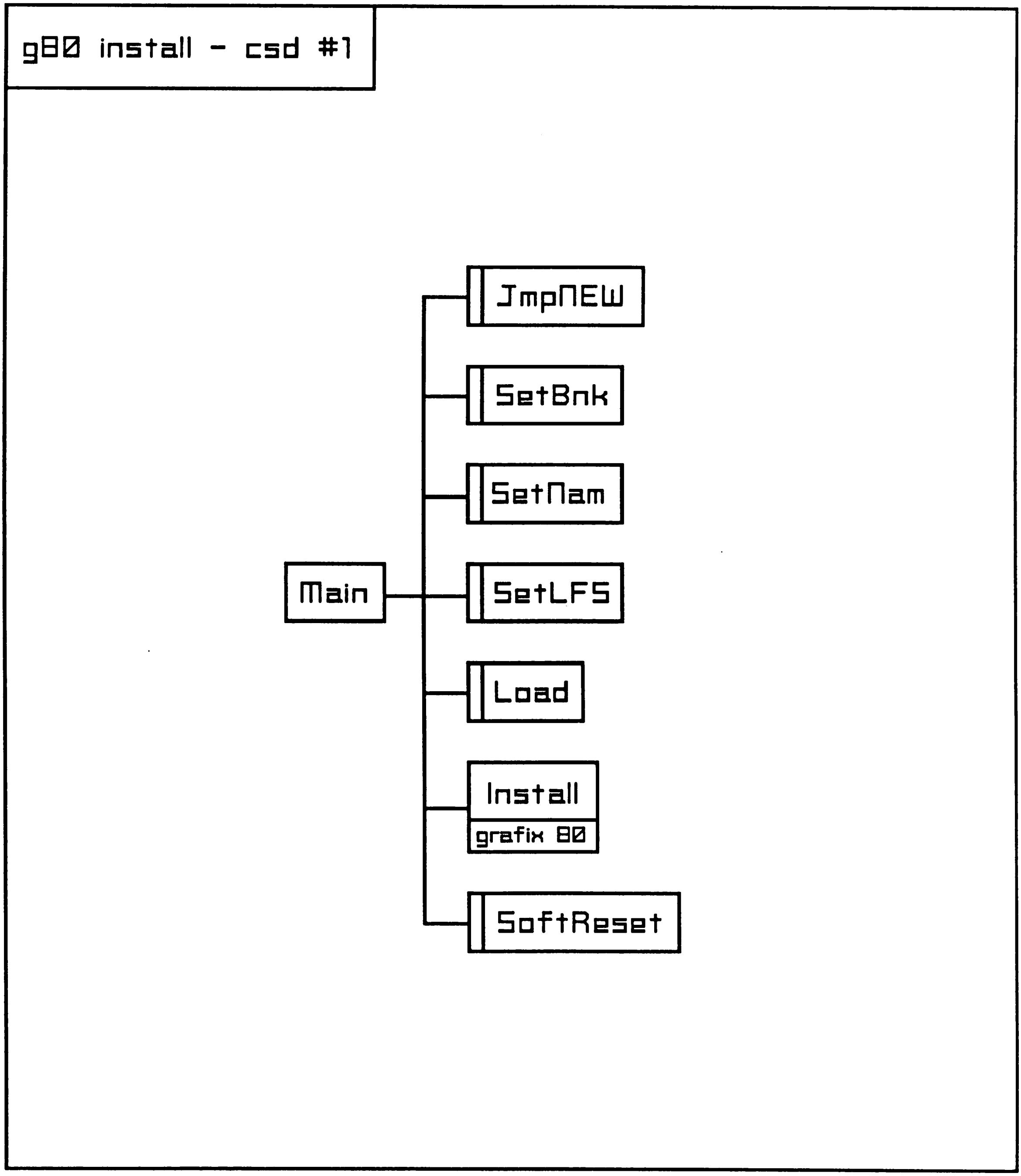


Fig. 5-1. Calling structure diagram for G80 Install.

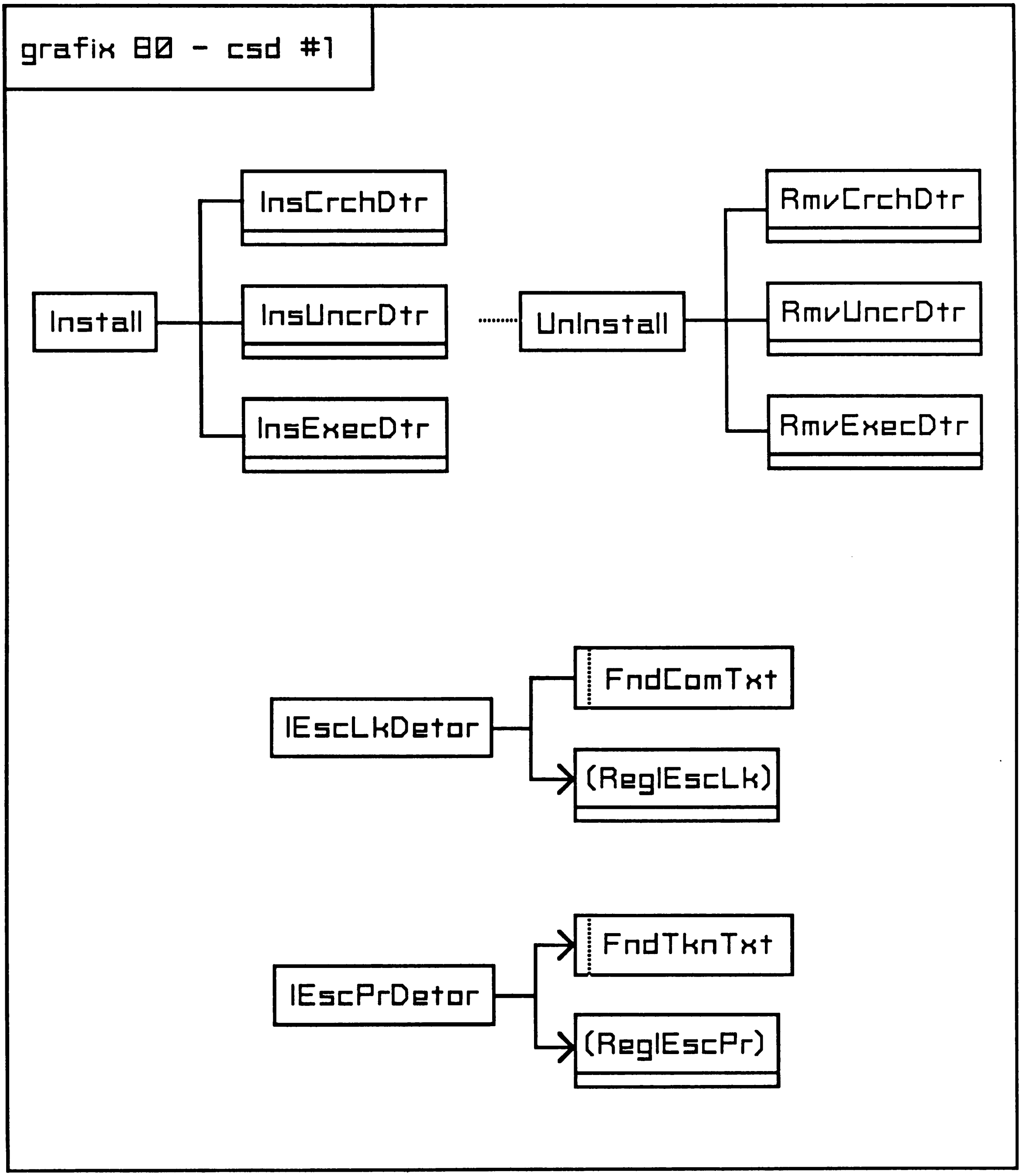
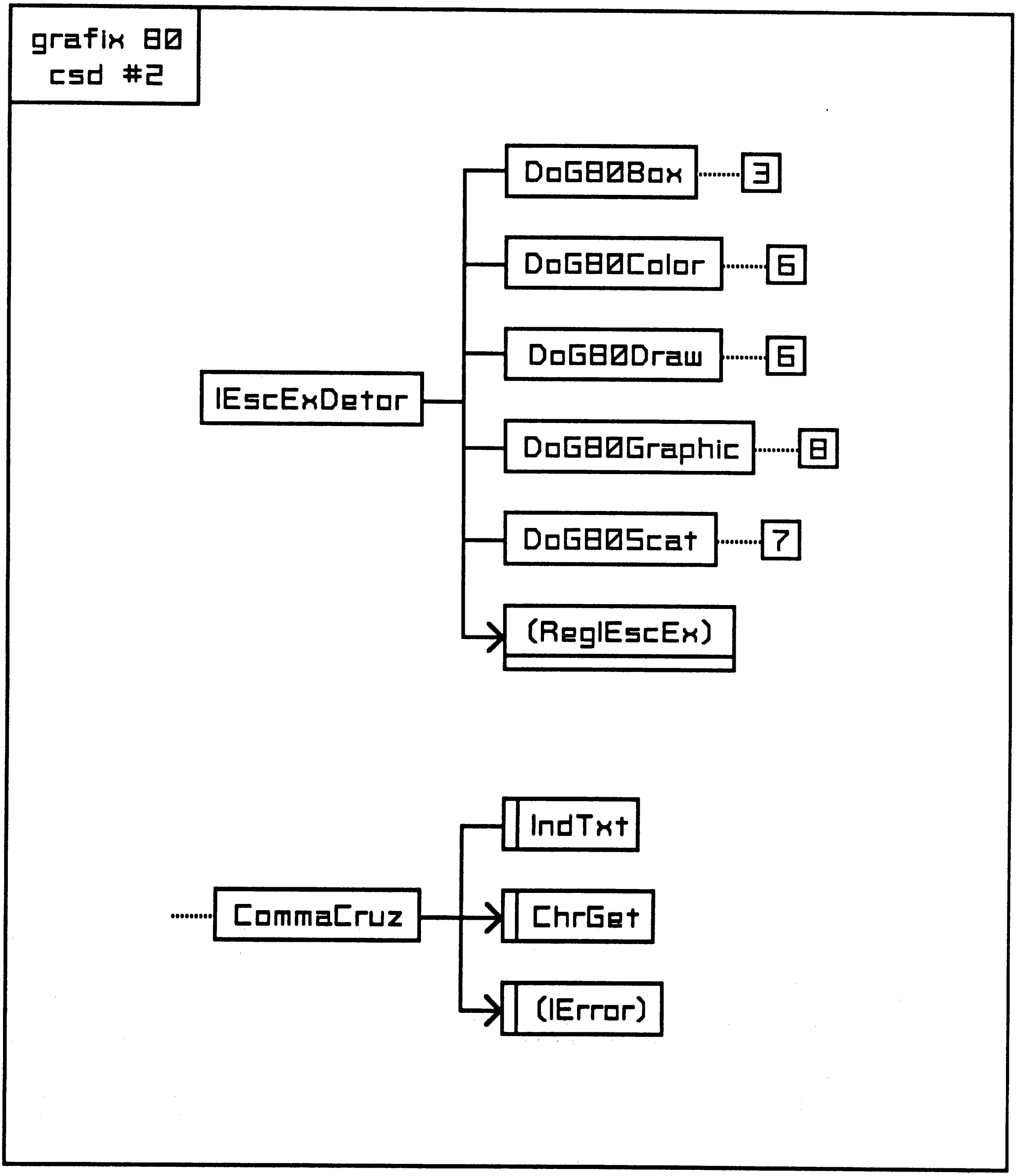
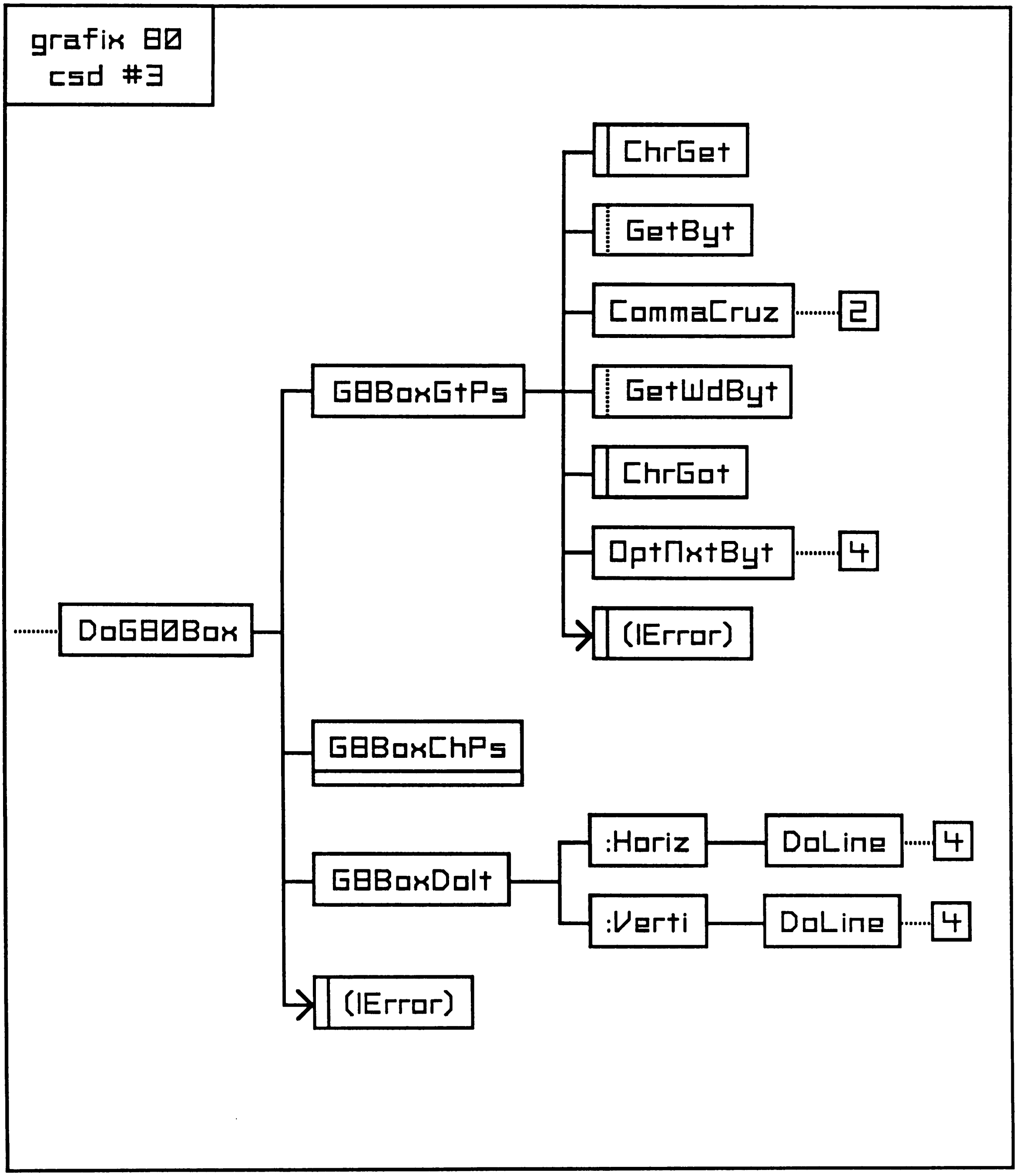
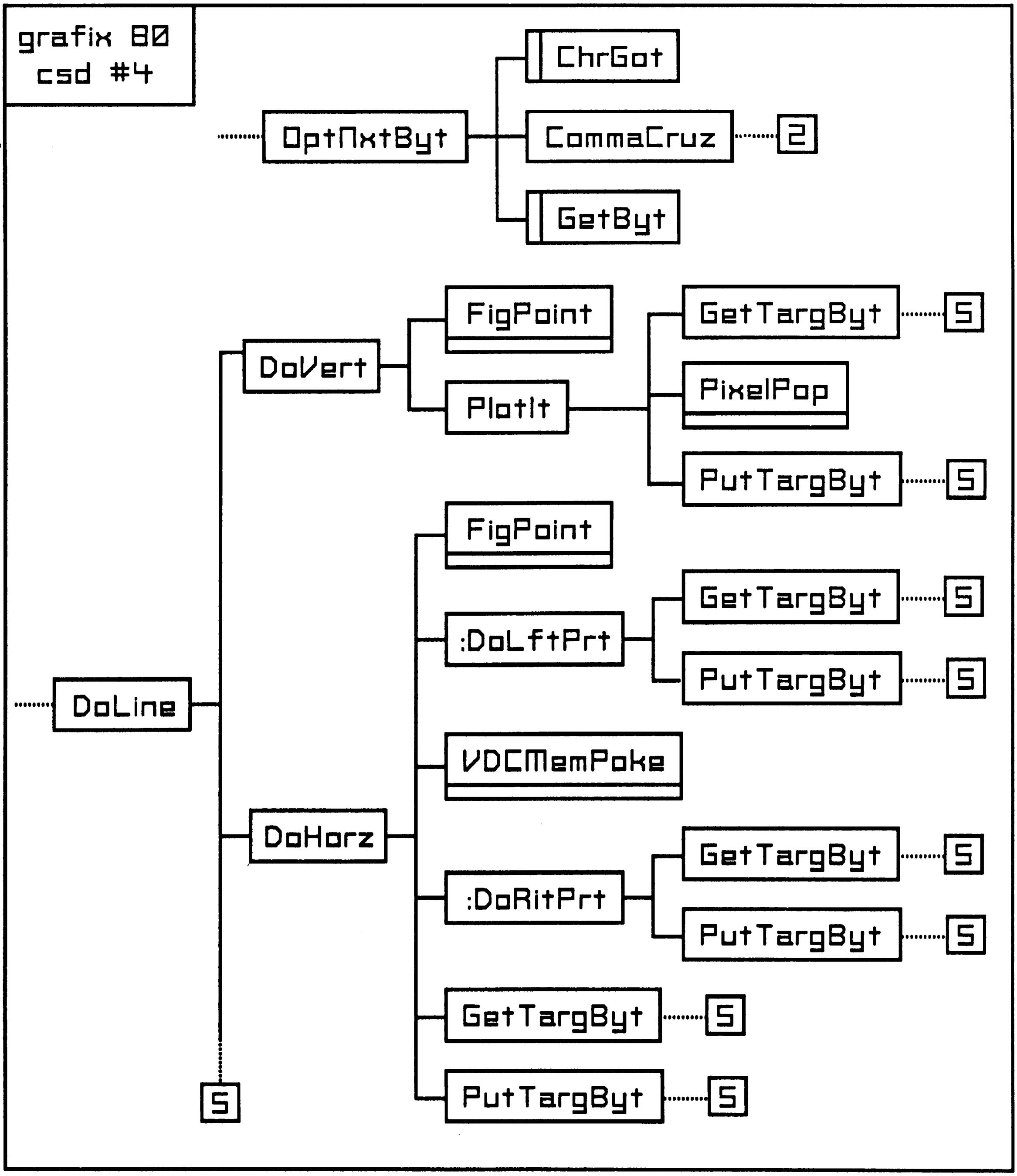
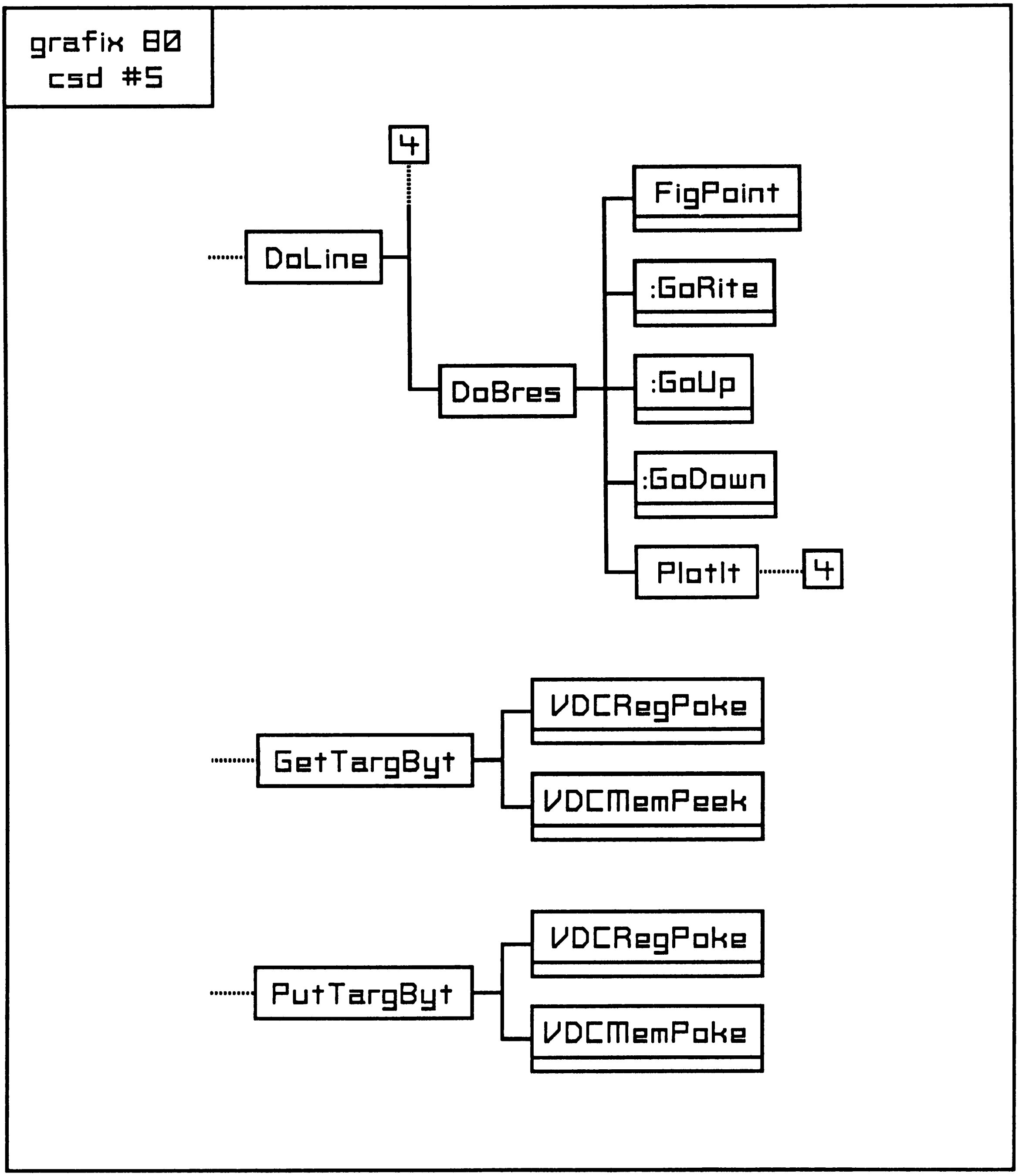


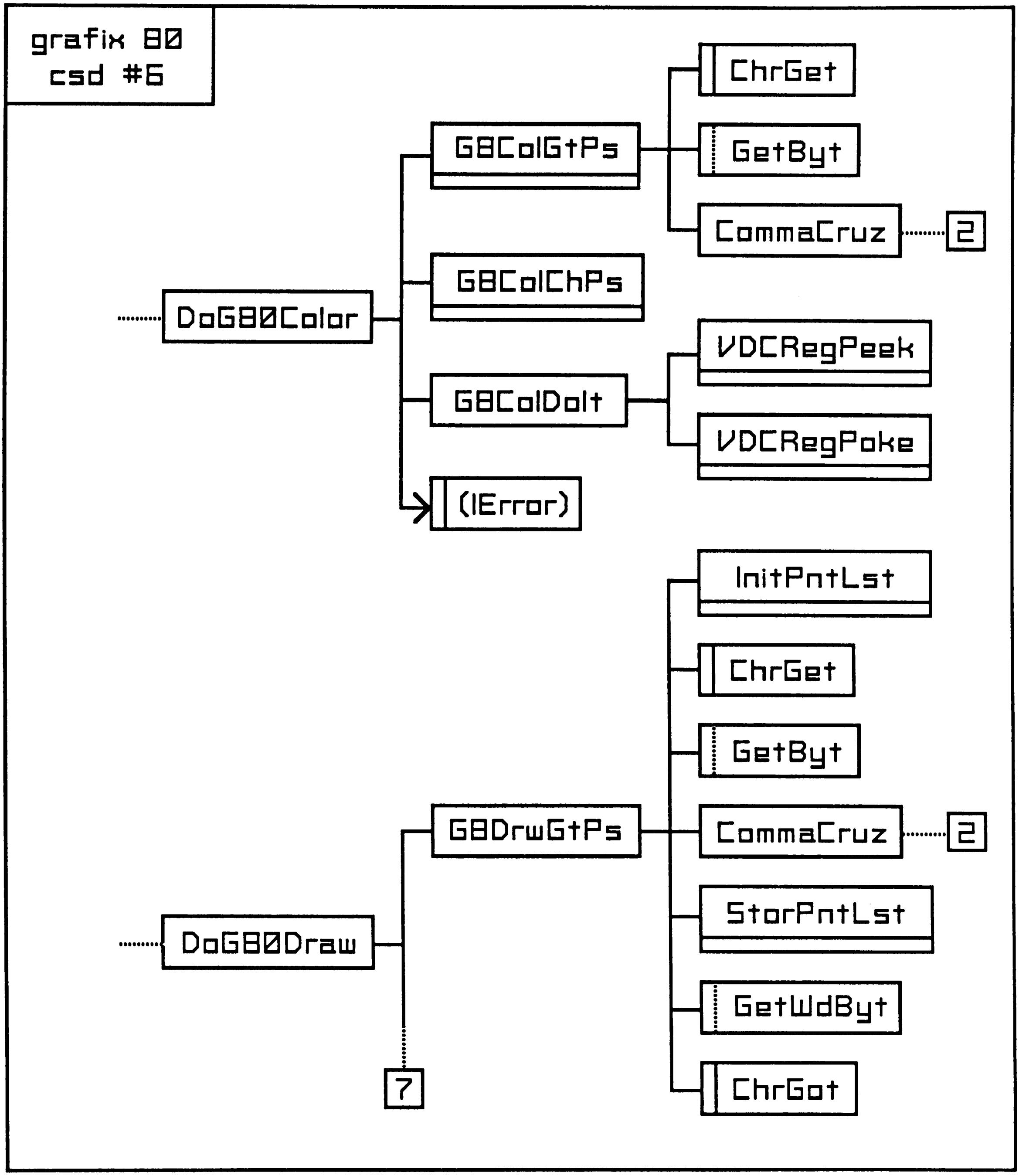
Fig. 5-2. Calling structure diagrams for Grafix 80.

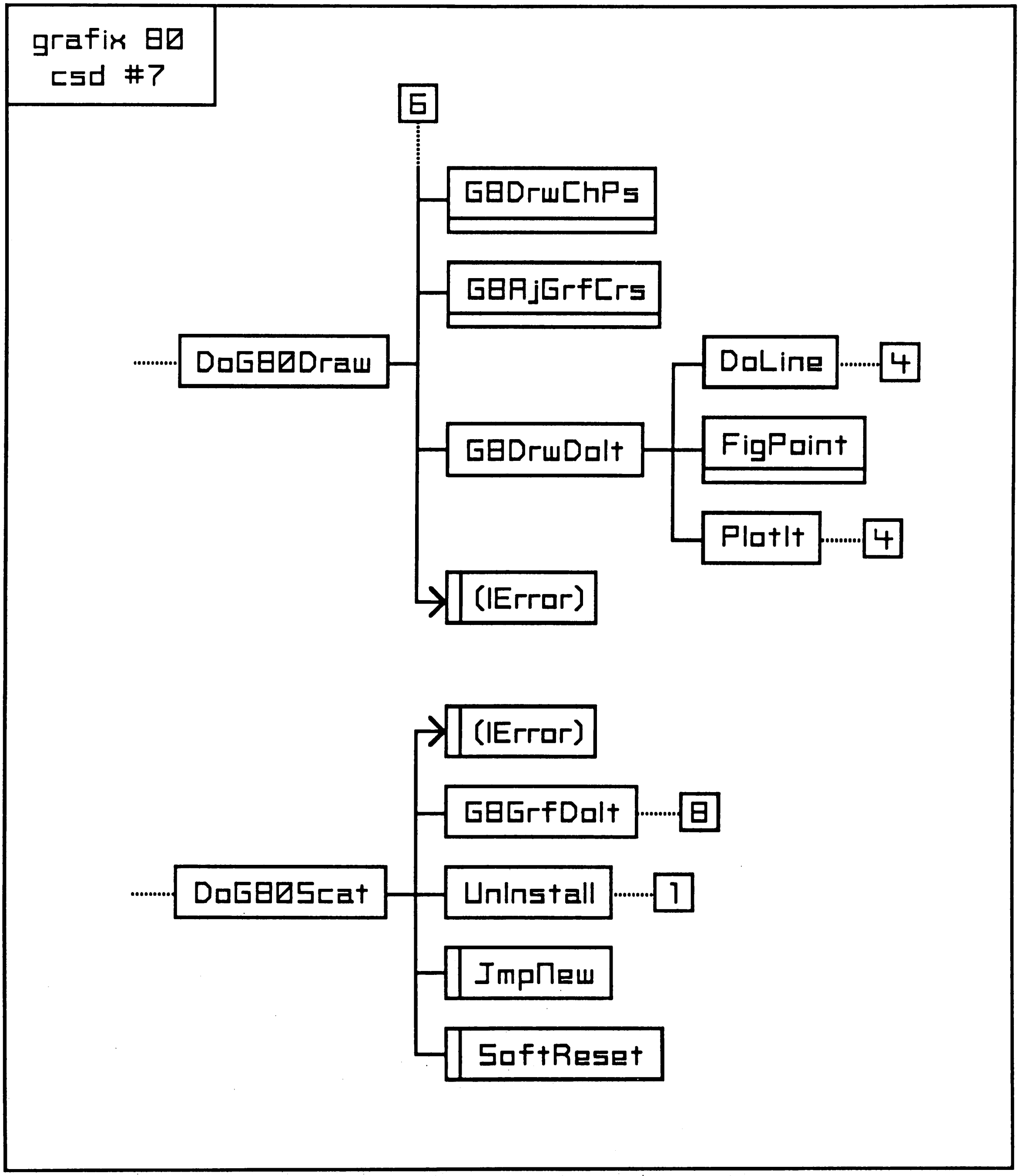

grafix 80
csd #2
IEscExDetor
DoG80Box
3
DoG80Color
6
DoG80Draw
6
DoG80Graphic
8
DoG80Scat
7
(RegIEscEx)
CommaCruz
IndTxt
ChrGet
(IError)

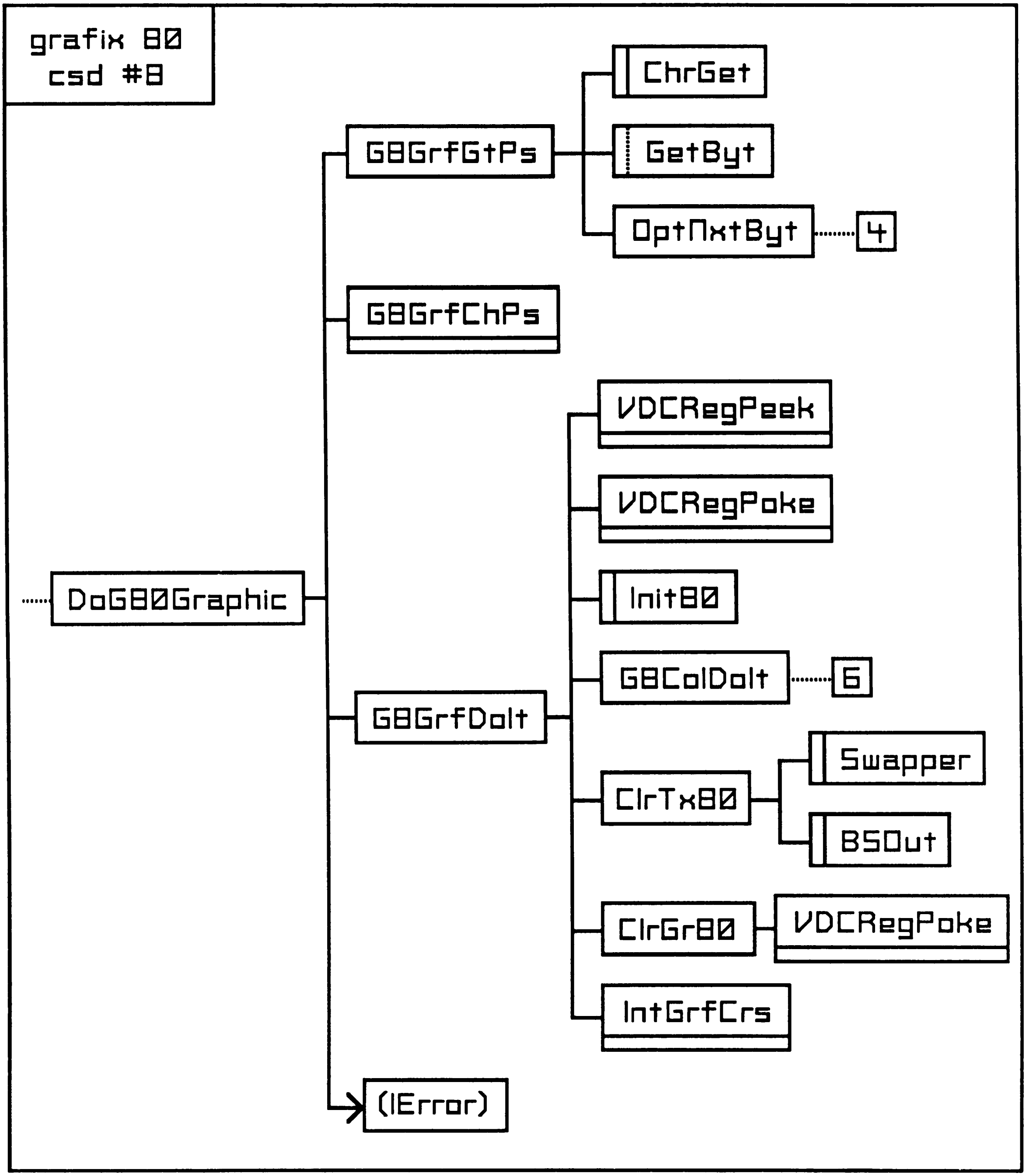

grafix 80
csd #3
DoG80Box
G8BoxGtPs
ChrGet
GetByt
CommaCruz
2
GetWdByt
ChrGot
OptNxtByt
4
(IError)
G8BoxChPs
G8BoxDoIt
:Horiz
DoLine
4
:Verti
DoLine
4
(IError)


grafix 80
csd #4
OptNxtByt
ChrGot
CommaCruz
2
GetByt
DoLine
DoVert
FigPoint
PlotIt
GetTargByt
5
PixelPop
PutTargByt
5
DoHorz
FigPoint
:DoLftPrt
GetTargByt
5
PutTargByt
5
VDCMemPoke
:DoRitPrt
GetTargByt
5
PutTargByt
5
GetTargByt
5
PutTargByt
5
5


grafix 80
csd #5
4
DoLine
DoBres
FigPoint
:GoRite
:GoUp
:GoDown
PlotIt
4
GetTargByt
VDCRegPoke
VDCMemPeek
PutTargByt
VDCRegPoke
VDCMemPoke


grafix 80
csd #6
DoG80Color
G8ColGtPs
ChrGet
GetByt
CommaCruz
2
G8ColChPs
G8ColDoIt
VDCRegPeek
VDCRegPoke
(IError)
DoG80Draw
G8DrwGtPs
InitPntLst
ChrGet
GetByt
CommaCruz
2
StorPntLst
GetWdByt
ChrGot
7


grafix 80
csd #7
6
G8DrwChPs
G8AjGrfCrs
DoG80Draw
DoLine
4
G8DrwDoIt
FigPoint
PlotIt
4
(IError)
(IError)
G8GrfDoIt
8
DoG80Scat
UnInstall
1
JmpNew
SoftReset


grafix 80
csd #8
DoG80Graphic
G8GrfGtPs
ChrGet
GetByt
OptNxtByt
4
G8GrfChPs
G8GrfDoIt
VDCRegPeek
VDCRegPoke
Init80
G8ColDoIt
6
ClrTx80
Swapper
BSOut
ClrGr80
VDCRegPoke
IntGrfCrs
(IError)

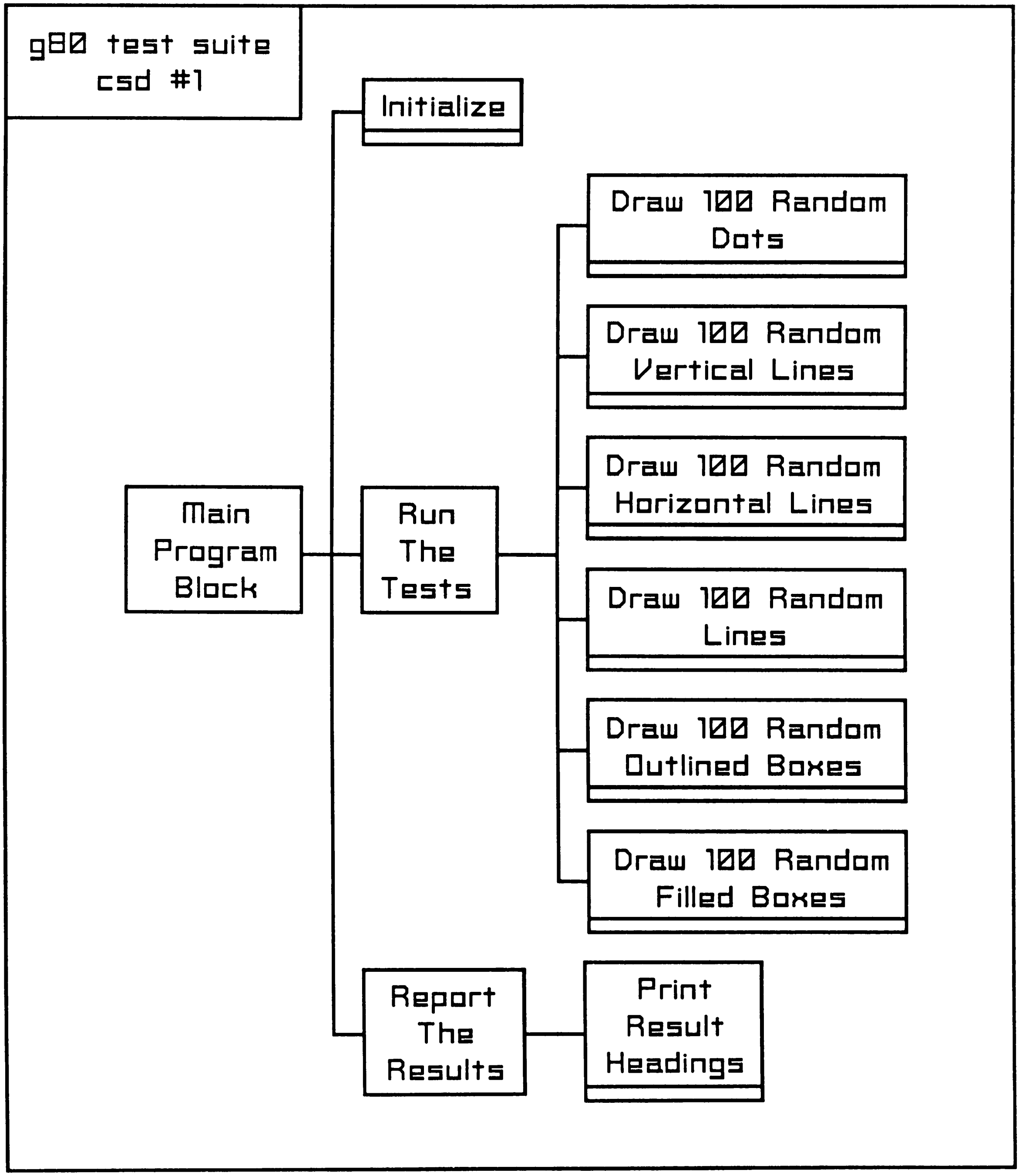


Fig. 5-3. Calling structure diagram for G80 Test Suite.

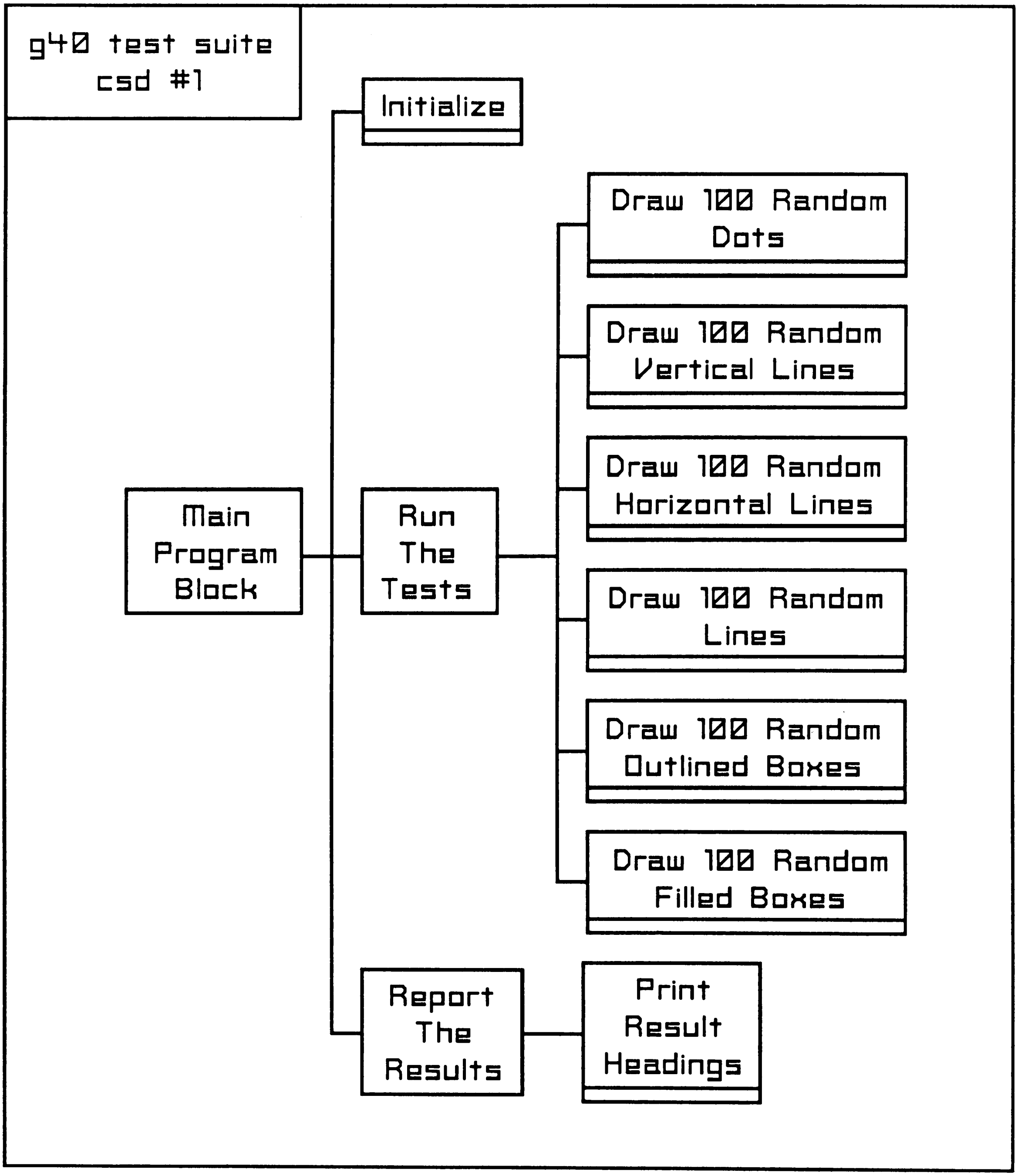


Fig. 5-4. Calling structure diagram for G40 Test Suite.

Chapter 6: Subroutine Line Starts

This chapter consists of three figures, as follows:

Fig. 6-1—list of subroutine line starts for Grafix 80 (3 sheets).
Fig. 6-2—list of subroutine line starts for G80 Test Suite (1 sheet).
Fig. 6-3—list of subroutine line starts for G40 Test Suite (1 sheet).

GRAFIX 80 - Subroutine Line Starts — Sheet 1 of 3

Install	0-317
UnInstall	0-331
InsCrchDtr	1-3
InsUncrDtr	1-30
InsExecDtr	1-57
RmvCrchDtr	1-84
RmvUncrDtr	1-105
RmvExecDtr	1-126
IEscLkDetor	1-147
IEscPrDetor	1-203
IEscExDetor	1-237
CommaCruz	1-280
DoG80Box	2-3
G8BoxGtPs	2-38
G8BoxChPs	2-127
G8BoxDoIt	2-190
:Horiz	2-285
:Verti	2-304
DoG80Color	2-321
G8ColGtPs	2-347
G8ColChPs	2-367
G8ColDoIt	2-405
DoG80Draw	3-3
G8DrwGtPs	3-40
InitPntLst	3-101
StorPntLst	3-126
G8DrwChPs	3-172
G8AjGrfCrs	3-243
G8DrwDoIt	3-283
DoG80Graphic	3-370
G8GrfGtPs	3-394
G8GrfChPs	3-413
G8GrfDoIt	3-454
DoG80Scat	3-549
IntGrfCrs	4-3
DoLine	4-23
DoVert	4-97
DoHorz	4-151

Fig. 6-1. List of subroutine line starts for Grafix 80.

GRAFIX 80 - Subroutine Line Starts

:DoLftPrt	4-320
:DoRitPrt	4-342
DoBres	4-393
:GoRite	4-647
:GoUp	4-661
:GoDown	4-674
ClrTx80	5-3
ClrGr80	5-42
FigPoint	5-89
PlotIt	5-166
GetTargByt	5-181
PutTargByt	5-186
PixelPop	5-219
VDCMemPoke	5-268
VDCRegPoke	5-270
VDCMemPeek	5-277
VDCRegPeek	5-279
OptNxtByt	5-287

G80 TEST SUITE - Subroutine Line Starts — Sheet 1 of 1

Main Program Block	1270
Initialize	1350
Run The Tests	1820
Report The Results	1930
Draw 100 Random Dots	2130
Draw 100 Random Vertical Lines	2320
Draw 100 Random Horizontal Lines	2510
Draw 100 Random Lines	2700
Draw 100 Random Outlined Boxes	2890
Draw 100 Random Filled Boxes	3080
Print Result Headings	3270

Fig. 6-2. List of subroutine line starts for G80 Test Suite.

G40 TEST SUITE - Subroutine Line Starts	Sheet 1 of 1
Main Program Block	1250
Initialize	1330
Run The Tests	1800
Report The Results	1910
Draw 100 Random Dots	2110
Draw 100 Random Vertical Lines	2300
Draw 100 Random Horizontal Lines	2490
Draw 100 Random Lines	2680
Draw 100 Random Outlined Boxes	2870
Draw 100 Random Filled Boxes	3060
Print Result Headings	3250

Fig. 6-3. List of subroutine line starts for G40 Test Suite.

Chapter 7: Selected Algorithms

This chapter consists of three figures, as follows:

Fig. 7-1—selected algorithms from G80 Install (1 sheet).
Fig. 7-2—selected algorithms from Grafix 80 (22 sheets).
Fig. 7-3—selected algorithms from G80 Test Suite (4 sheets).

Selected Algorithms From G80 Install — Sheet 1 of 1

main

```
    save some registers
    save current memory configuration
    set memory configuration to Bank 15 ( system bank )
    set BASIC text start to a new position
    zero out the byte just before BASIC text start
    do a BASIC NEW command by calling the ROM routine JmpNEW
    load in the GRAFIX 80 object code
    call the GRAFIX 80 routine Install to install the 80-column routines
    call the ROM routine SoftReset to do a BASIC warm start
    restore the entry memory configuration
    restore some registers
    RETURN
```

Fig. 7-1. Selected algorithms from G80 Install.

Selected Algorithms From GRAFIX 80 — Sheet 1 Of 22

Install

```
    call on InsCrchDtr to install a command crunching detour
    call on InsUncrDtr to install a command un-crunching detour
    call on InsExecDtr to install a command execution detour
    RETURN
```

UnInstall

```
    call on RmvCrchDtr to remove a command crunching detour
    call on RmvUncrDtr to remove a command un-crunching
        detour
    call on RmvExecDtr to remove a command execution detour
    RETURN
```

InsCrchDtr

```
    save some registers
    save the current command crunching vector
    point the command crunching vector at our detour routine
    restore some registers
    RETURN
```

Fig. 7-2. Selected algorithms from Grafix 80.

InsUncrDtr

save some registers
save the current command un–crunching vector
point the command un–crunching vector at our detour routine
restore some registers
RETURN

InsExecDtr

save some registers
save the current command execution vector
point the command execution vector at our detour routine
restore some registers
RETURN

RmvCrchDtr

save some registers
point the command crunching vector back at its original routine
restore some registers
RETURN

RmvUncrDtr

save some registers
point the command un–crunching vector back at its original routine
restore some registers
RETURN

RmvExecDtr

save some registers
point the command execution vector back at its original routine
restore some registers
RETURN

IEscLkDetor

save entry byte of program text
IF
    the entry byte is one of the following :
        end of input buffer
        colon
        question mark
        a token
        quotation mark
THEN do this
    restore entry byte

JUMP to the regular IEscLk routine with flag set for
no–command–found

call on the undocumented System routine FndComTxt to see
if the entry byte is the start of one of our 80-column
graphics commands

IF
FndComTxt says it didn't find one of our new commands

THEN
restore entry byte
JUMP to the regular IEscLk routine with flag set for
no–command–found

ELSE { one of our new commands was found)
set up registers for command tokenizing
JUMP to the regular IEscLk routine with flag set for
command–found

IEscPrDetor

IF
the lead-in token is not $FE
OR
the selector token is less than our first selector token
value
OR
the selector token is greater than our last selector token
value

THEN
JUMP to the regular IEscPr routine with flag set for
not-our-token

ELSE { we've found one of our token pairs }
set up for token un-crunching
JUMP to the undocumented System routine FndTknTxt to
un-crunch the token pair

IEscExDetor

IF
the selector token indicates the G80BOX command

THEN
call on DoG80Gox to carry out the command
RETURN

ELSE IF
the selector token indicates the G80COLOR command

THEN

call on DoG80Color to carry out the command
RETURN
ELSE IF
the selector token indicates the G80DRAW command
THEN
call on DoG80Draw to carry out the command
RETURN
ELSE IF
the selector token indicates the G80GRAPHIC command
THEN
call on DoG80Graphic to carry out the command
RETURN
ELSE IF
the selector token indicates the G80SCAT command
THEN
call on DoG80Scat to carry out the command
RETURN
ELSE
JUMP to the regular IEscEx routine with a flag set to
signal not-our-token

CommaCruz
call on the System routine IndTxt to grab the
currently–pointed–at byte of BASIC text
IF
the byte represents a comma
THEN
JUMP to the System routine ChrGet to grab the next
meaningful byte in the BASIC statement and RETURN
ELSE { the byte doesn't represent a comma }
JUMP to the System routine IError to signal 'syntax error'

DoG80Box
call on G8BoxGtPs to fetch any command parameters
call on G8BoxChPs to check the legality of any parameters
IF
there's a problem with one of the parameters
THEN
JUMP to the IError routine to signal an
'illegal quantity' error
ELSE { the parameters checked out okay }
call on G8BoxDoIt to carry out the command
RETURN signalling that all went well

G8BoxGtPs

    set the default paint parameter to 'no–paint'
    call on the System routine ChrGet to get the BASIC statement
        element that follows G80BOX
    IF
        the next element is a comma
    THEN
        set color source to foreground
    ELSE
        call on undocumented System routine GetByt to get a color
            source from the BASIC statement
    call on CommaCruz to cruise through a comma
    call on undocumented System routine GetWdByt to get a first
        point's horizontal and vertical coordinates
    store those coordinates
    IF
        there are no more elements to the BASIC statement
    THEN
        RETURN
    { there are more elements to the BASIC statement }
    call on CommaCruz to cruise through a comma
    IF
        the next element isn't a comma
    THEN
        call on undocumented System routine GetWdByt to get a
            second point's horizontal and vertical coordinates
        move the graphics cursor to this second point
    call OptNxtByt and store the result as the paint parameter
    IF
        a call to ChrGot shows there are more elements to the
            BASIC statement
    THEN
        JUMP to IError to signal a "syntax error"
    ELSE
        RETURN

G8BoxChPs

    save some registers
    IF
        color source is not set for foreground
            AND
        color source is not set for background

```
    THEN
        restore some registers
        RETURN, signalling an error
    IF
        first point's vertical coordinate is too large
            OR
        second point's vertical coordinate is too large
    THEN
        restore some registers
        RETURN, signalling an error

    IF
        first point's horizontal coordinate is too large
            OR
        second point's horizontal coordinate is too large
    THEN
        restore some registers
        RETURN, signalling an error
    IF
        paint parameter's not 0
            AND
        paint parameter's not 1
    THEN
        restore some registers
        RETURN, signalling an error
    restore some registers
    RETURN, signalling all is okay with the parameters

G8BoxDoIt
    save some registers
    save current memory configuration
    set memory configuration to Bank 15 ( system bank )
    use color source to set up for drawing or erasing
    IF
        the paint flag says "no paint"
    THEN  { we're working on an outlined box }
        call on :Horiz to draw the first horizontal line
        call on :Horiz to draw the second horizontal line
        call on :Verti to draw the first vertical line
        call on :Verti to draw the second vertical line
    ELSE  { we're working on a filled-in box }
        figure the height of the box -- that is, how many
            rows it contains
```

```
        FOR
            each of the box's rows
        DO the following :
            call on :Horiz to draw the row
    restore the entry memory configuration
    restore some registers
    RETURN

:Horiz
    set vertical coordinates
    set horizontal coordinates
    call DoLine to draw the line
    RETURN

:Verti
    set horizontal coordinates
    set vertical coordinates
    call DoLine to draw the line
    RETURN

:InitPntLst
    save some registers
    set the draw–list point counter to 0
    set the draw–list indexer to the 0th byte of the list
    set the draw–list pointer to the beginning of the list
    restore some registers
    RETURN

StorPntLst
    save some registers
    add the point's horizontal coordinate to the draw list using the
        draw–list indexer and the draw–list pointer
    add the point's vertical coordinate to the point list using the
        draw–list indexer and the draw–list pointer
    store the incremented  ( by three -- that's how many bytes
        were just stored ) draw–list indexer
    increment the draw–list point counter
    restore some registers
    RETURN

G8DrwDoIt
    save some registers
```

```
    save entry memory configuration
    set memory configuration to Bank 15 ( system bank )
    IF
        there are points to draw
    THEN
        set up to draw or erase, based on foreground or
            background being the color source
        IF
            there's just one point to draw
        THEN
            grab the point's coordinates off the draw list
            call on FigPoint to set up the point's vital plotting info
            call on PlotIt to draw the point
        ELSE  { there's more than one point to draw }
            FOR
                each of the line segments in the draw list
            DO the following
                grab the line segment's endpoint coordinates from
                    the draw–list, using the draw list pointer
                call on DoLine to draw the line segment
                move the draw–list pointer along
    restore the entry memory configuration
    restore some registers
    RETURN
```

DoG80Scat

```
    IF
        we're not in direct mode
    THEN
        JUMP to IError signalling a "direct mode only" error
    ELSE  { we're in direct mode }
        call on G8GrfDoIt to get a cleared 80-column text screen
        call on UnInstall to un-install the 80-column graphics
            commands
        zero out the byte just before the standard BASIC text
            start
        set the BASIC text start back to its standard position
        call on the System routine JmpNEW to execute a BASIC NEW
            command
        call on the System routine SoftReset to do a warm start of
            BASIC
        RETURN, signalling that all went well
```

DoLine

save some registers
IF
    the line is vertical
THEN
    call on DoVert to draw a vertical line
ELSE
    adjust line endpoints so first point is leftmost
    IF
        the line is horizontal
    THEN
        call on DoHorz to draw a horizontal line
    ELSE { the line is sloped }
        call on DoBres to draw a sloped line
restore some registers
RETURN

DoVert

save some registers
figure the height of the line
adjust points so the first point is topmost
call on FigPoint to set up the first point's vital plotting info
STARTING WITH
    the first point
FOR
    as many points as the line has height
DO the following
    call on PlotIt to plot a point
    move vital point–plotting info down to the next point in
        the line
restore some registers
RETURN

DoHorz

save some registers
figure the length of the line
call on FigPoint to set up the first point's vital plotting info
figure out the bit–in–byte position for the line's rightmost point
figure out the bit–in–byte position for the line's leftmost point
IF
    the line length is greater than 256
THEN
    { we have a three–part line–drawing situation }

```
        call on :3Part to draw the line
    ELSE IF
        the line length plus the leftmost point's bit–in–byte position
            is less than 8
    THEN
        { we have a one–part line–drawing situation }
        call on :1Part to draw the line
    ELSE IF
        the line length plus the leftmost point's bit–in–byte position
            is less than 16
    THEN
        { we have a two–part line–drawing situation }
        call on :2Part to draw the line
    ELSE
        { we have a three–part line–drawing situation }
        call on :3Part to draw the line
    restore some registers
    RETURN

:3Part
    call on :DoLftPrt to draw the left part of the line
    figure the number of bytes in the middle part of the line
    prepare a byte that'll either draw or erase pixels
    FOR
        each byte in the middle part of the line
    DO the following
        call on VDCMemPoke to store the prepared byte
    adjust a pointer to point to the right part of the line
    call on :DoRitPrt to draw the right part of the line
    RETURN

:2Part
    call on :DoLftPrt to draw the left part of the line
    adjust a pointer to point to the right part of the line
    call on :DoRitPrt to draw the right part of the line
    RETURN

:1Part
    get an OR mask for the left part of the line
    get an OR mask for the right part of the line
    AND the OR masks together to get a custom mask
    call on GetTargByt to grab the screen target byte
```

```
IF
        we're drawing ( turning bits on )
THEN
        OR the target byte with the custom mask
ELSE  { we're erasing ( turning bits off ) }
        invert the custom mask
        AND the target byte with the inverted custom mask
call on PutTargByt to store the screen target byte
RETURN

:DoLftPrt
    call on GetTargByt to grab the screen target byte
    IF
            we're drawing ( turning bits on )
    THEN
            OR the target byte with the appropriate left part OR mask
    ELSE  { we're erasing ( turning bits off ) }
            AND the target byte with the appropriate left part AND
                    mask
    call on PutTargByt to store the screen target byte
    RETURN

:DoRitPrt
    call on GetTargByt to grab the screen target byte
    IF
            we're drawing ( turning bits on )
    THEN
            OR the target byte with the appropriate right part OR
                    mask
    ELSE  { we're erasing ( turning bits off ) }
            AND the target byte with the appropriate right part AND
                    mask
    call on PutTargByt to store the screen target byte
    RETURN

DoBres
    save some registers
    figure out the line's horizontal position change ( Raw Delta X,
            which will always be positive )
    figure out the line's vertical position change ( Raw Delta Y ,
            which can be positive or negative, and a positive version,
            Absolute Delta Y )
    figure out whether the line rises or falls as it goes from left
```

```
        to right
figure out whether the line's slope is steep ( greater than 45° )
        or shallow ( less than 45° )
set increments, erometer, and counter as follows:
        IF
                the line is steep
        THEN
                set Increment One to twice Raw Delta X
                initialize the Erometer to Increment One minus
                        Absolute Delta Y
                set Increment Two to Erometer minus Absolute
                        Delta Y
                initialize the Counter to Absolute Delta Y plus one
        ELSE   { the line is shallow }
                set Increment One to twice Absolute Delta Y
                initialize the Erometer to Increment One minus
                        Raw Delta X
                set Increment Two to Erometer minus Raw Delta X
                initialize the Counter to Raw Delta X plus one
call on FigPoint to set up the first point's vital plotting info
figure out the starting point's bit-in-byte position
call on PlotIt to draw/erase the starting point
decrement the Counter
FOR
        the number of points in the Counter
DO the following :
        IF
                it's a shallow line
        THEN
                call on :GoRite to move right one position
        ELSE   { it's a steep line }
                IF
                        it's a rising steep line
                THEN
                        call on :GoUp to move up one position
                ELSE   { it's a falling steep line }
                        call on :GoDown to move down one position
        IF
                the Erometer value is negative
        THEN
                add Increment One to the Erometer
        ELSE   { the Erometer value is positive }
                add Increment Two to the Erometer
```

```
IF
     it's a shallow line
THEN
     IF
          it's a rising shallow line
     THEN
          call on :GoUp to move up one position
               ELSE  { it's a falling shallow line }
                    call on :GoDown to move down one position
          ELSE  { it's a steep line }
               call on :GoRight to move right one position
     store the point's bit-in-byte position
     call on PlotIt to plot the point
restore some registers
RETURN

:GoRite
     increment the target bit-in-byte position
     IF
          we've moved on into the next byte
     THEN
          reset the target bit-in-byte position to 0
          increment the target byte pointer
     RETURN

:GoUp
     subtract a line's worth of bytes from the target byte pointer
     RETURN

:GoDown
     add a line's worth of bytes to the target byte pointer
     RETURN

ClrTx80
     save some registers
     IF
          we're in 40-column screen mode
     THEN
          call on the System routine Swapper to change to
               80-columns from 40
     clear the screen through a call to the System routine BSOut
     IF
          we were in 40-column screen mode upon entry
```

THEN

call on the System routine Swapper to change to 40–columns from 80

restore some registers

RETURN

ClrGr80

save some registers

FOR

each 256-byte page in the VDC RAM memory

DO the following

call on VDCRegPoke to set the VDC Update Address registers to this page

call on VDCRegPoke to tell the VDC chip to fill this page with 256 zeroes

restore some registers

RETURN

FigPoint

{ upon entry the routine is given a point's horizontal and vertical coordinates }

save some registers

use the point's vertical coordinate to get the address of the first byte in the point's row

add in the point's horizontal coordinate to get the address of the point's byte

set the target byte pointer to that address

AND the lo-byte of the horizontal coordinate with %00000111 (+7) to get the point's bit–in–byte position

set the target bit–in–byte position to that value

restore some registers

RETURN

PlotIt

call on GetTargByt to fetch the target point's byte

call on PixelPop to set the target point's bit in its byte on or off

call on PutTargByt to store the target point's modified byte

RETURN

GetTargByt

save some registers

call on VDCRegPoke to aim the VDC Update Address registers at

the target byte
call on VDCMemPeek to grab the target byte from VDC RAM memory
restore some registers
RETURN

PutTargByt
save some registers
call on VDCRegPoke to aim the VDC Update Address registers at the target byte
call on VDCMemPoke to store the target byte into VDC RAM
memory
restore some registers
RETURN

PixelPop
save some registers
IF
we're turning a pixel on
THEN
OR the target byte with an on-mask customized to the
target pixel's bit-in-byte position
ELSE { we're turning a pixel off }
AND the target byte with an off-mask customized to the
target pixel's bit-in-byte position
store the modified target byte
restore some registers
RETURN

OptNxtByt
IF
a call to the ROM routine ChrGot shows there's nothing
left to fetch from the current BASIC statement
THEN
RETURN, signalling and carrying a default value of 0
ELSE IF
a call to CommaCruz to make sure there's a comma comes
back empty handed
THEN
RETURN, signalling and carrying a default value of 0
ELSE
call on the undocumented ROM routine GetByt to fetch a
byte-sized value
RETURN, signalling and carrying a fetched value

Selected Algorithms From G80 TEST SUITE Sheet 1 Of 3

```
Main Program Block
        Initialize constants and variables
        Run The Tests
        Report The Results
        RETURN

Initialize
        fetch a random seed ( 1..32768 ) from the user
        fetch a screen width ( 0..639 ) from the user
        give some feedback
        speed up to 2 megahertz speed
        FOR
                each of 100 array elements
        DO the following
                FOR
                        each of 4 coordinate arrays { T(), B(), L(), & R () }
                DO the following
                        set the array's element to a value chosen randomly
                                from the element's permissible range of values
        FOR
                each of the six tests
        DO the following
                read in the test's name label
        slow down to 1 megahertz speed
        RETURN

Run The Tests
        Draw 100 Random Dots
        Draw 100 Random Vertical Lines
        Draw 100 Random Horizontal Lines
        Draw 100 Random Lines
        Draw 100 Random Outlined Boxes
        Draw 100 Random Filled Boxes
        RETURN

Report The Results
        go to a cleared 80–column text screen
        Print Result Headings
```

Fig. 7-3. Selected algorithms from G80 Test Suite.

```
FOR
    each of the six tests
DO the following
    print the test name
    print the adjective 'slow'
    print the result of the test run at 1 megahertz speed
    print the test name
    print the adjective 'fast'
    print the result of the test run at 2 megahertz speed
print a blank line
RETURN
```

Draw 100 Random Dots

```
go to a cleared 80–column graphics screen
slow down to run the test at 1 megahertz speed
reset the timer
FOR
    each of 100 points
DO the following
    draw the point by using elements from the coordinate
        arrays L() and T()
stop the timer and record the time
speed up to run the test at 2 megahertz speed
reset the timer
FOR
    each of 100 points
DO the following
    draw the point by using elements from the coordinate arrays L() and T()
stop the timer and record the time
RETURN
```

Draw 100 Random Vertical Lines

```
go to a cleared 80–column graphics screen
slow down to run the test at 1 megahertz speed
reset the timer
FOR
    each of 100 lines
DO the following
    draw the line by using elements from the coordinate arrays B(), L(), and T()
stop the timer and record the time
speed up to run the test at 2 megahertz speed
reset the timer
```

```
FOR
    each of 100 lines
DO the following
    draw the line by using elements from the coordinate arrays B(), L(), and T()
stop the timer and record the time
RETURN
```

Chapter 8: Program Listings

This chapter consists of four figures, each of which lists code for a program:

Fig. 8-1—code for G80 Install
Fig. 8-2—code for Grafix 80
Fig. 8-3—code for G80 Test Suite
Fig. 8-4—code for G40 Test Suite

```
*---------------- program identification ----------------*
*                                                         *
*                     G80 INSTALL.S                       *
*                                                         *
* Installs the 80-column graphics commands contained      *
* in the file GRAFIX 80                                   *
*                                                         *
* Here's how to run it :                                  *
*     BLOAD "G80 INSTALL"                                 *
*     SYS 7592                                            *
*                                                         *
* Version :     1.00                                      *
* Timestamp :   2:11 PM PST    September 20, 1986         *
*                                                         *
* Programmed by Stan Krute                                *
* Copyright (C) 1986 by Stan Krute's Hacker & Nerd        *
*                       18617 Camp Creek Road             *
*                       Hornbrook, California   96044     *
*                       [916] 475-3428                    *
* All rights reserved                                     *
* Call or write for help, bug reports, licensing, etc.    *
*                                                         *
*---------------------------------------------------------*


*------------------ constants ---------------------------*

* built-in routines -- documented

Load     =     $FFD5      ; load from file
SetBnk   =     $FF68      ; set load and filename banks
SetLFS   =     $FFBA      ; set up logical file number,
                          ; ... device number, and
                          ; ... secondary address command
SetNam   =     $FFBD      ; set filename parameters


* BASIC goodies

JmpNEW   =     $AF84      ; does the NEW command
NewBS    =     $1E01      ; where we move the start of
                          ; ... BASIC text to
SoftReset =    $4003      ; does a warm start
TxtTab   =     $2D        ; pointer to start of BASIC text


* GRAFIX 80 routines

Install  =     $1300      ; address of the GRAFIX 80
                          ; ... installation routine


* low-memory goodies

FA       =     $BA        ; current file primary address


* memory management

Bank15   =     %00000000  ; configuration byte for Bank 15
MmuSCR   =     $FF00      ; always-available configuration
```

Fig. 8-1. Source code for G80 Install.

```
                                                    ; ... register


            * our codes

            DoLoad    =     0              ; code for ROM's Load routine
            LoadBank =      0              ; memory bank we'll Load to
            NameBank =      0              ; memory bank where filename is
            Nil       =     0              ; nice name for nothing
            SavdPosn =      $FF            ; secondary address command for
                                           ; ... loading a file at its
                                           ; ... saved-from position


            *----------------- Set Program Origin ------------------*

                      ORG   $1DA8          ; a lovely little spot
                                           ; 7592 in decimal form


            *-------------------- Main ----------------------------*

            * Main block of the installation program

            Main
            * save some registers
1DA8: 48              PHA
1DA9: 8A              TXA
1DAA: 48              PHA
1DAB: 98              TYA
1DAC: 48              PHA

            * save current memory configuration
1DAD: AD 00 FF        LDA   MmuSCR         ; grab it
1DB0: 48              PHA                  ; park it on the stack

            * set memory configuration to Bank 15
1DB1: A9 00           LDA   #Bank15
1DB3: 8D 00 FF        STA   MmuSCR         ; do it to it

            * set BASIC text start to new position
1DB6: A9 01           LDA   #<NewBS        ; this preps us for NEW
1DB8: 85 2D           STA   TxtTab
1DBA: A9 1E           LDA   #>NewBS
1DBC: 85 2E           STA   TxtTab+1

            * zero out the byte just before BASIC text
1DBE: A9 00           LDA   #0             ; also primes 0 flag
1DC0: 8D 00 1E        STA   NewBS-1

            * do a BASIC NEW command
1DC3: 20 84 AF        JSR   JmpNEW         ; must enter with zero
                                           ; ... flag primed

            * load in our object code file
            * set input memory banks
1DC6: A9 00           LDA   #LoadBank      ; set bank to load file to
1DC8: A2 00           LDX   #NameBank      ; set bank filename is in
1DCA: 20 68 FF        JSR   SetBnk         ; set input memory banks

            * set filename parameters
1DCD: A9 09           LDA   #:End-:TheFile ; get length of file name
1DCF: A2 F4           LDX   #<:TheFile     ; point to the file name
1DD1: A0 1D           LDY   #>:TheFile
1DD3: 20 BD FF        JSR   SetNam         ; set filename parameters
```

```
                  * set logical file number, device number, and secondary
                  * ... address command
1DD6: A9 00               LDA   #Nil       ; logical file number not used
                                           ; ... for Load command
1DD8: A6 BA               LDX   FA         ; use current file device
1DDA: A0 FF               LDY   #SavdPosn  ; this secondary address command
                                           ; ... sez to load the file at
                                           ; ... its saved-from position
1DDC: 20 BA FF            JSR   SetLFS     ; set up LA, FA, and SA

                  * load that file
1DDF: A9 00               LDA   #DoLoad    ; set code for a load
1DE1: 20 D5 FF            JSR   Load       ; load that file

                  * go do the GRAFIX 80 installation code
1DE4: 20 00 13            JSR   Install    ; hop into it

                  * do a warm start of BASIC
1DE7: 20 03 40            JSR   SoftReset

                  * restore the entry memory configuration
1DEA: 68                  PLA              ; remember, we parked it here
1DEB: 8D 00 FF            STA   MmuSCR

                  * restore some registers
1DEE: 68                  PLA
1DEF: A8                  TAY
1DF0: 68                  PLA
1DF1: AA                  TAX
1DF2: 68                  PLA

                  * return from Main
1DF3: 60                  RTS


                  *-------- local constants --------*

1DF4: 47 52 41    :TheFile TXT   'grafix 80'
1DF7: 46 49 58 20 38 30
                  :End

--End assembly, 85 bytes, Errors: 0
```

```

*---------------- program identification ---------------*
*                                                       *
*                      GRAFIX 80                        *
*                                                       *
* Provides BASIC 7.0 commands for 80-column graphics.   *
*                                                       *
* Commands are fully tokenized extensions to the BASIC  *
* 7.0 command set.  They may be used in either direct   *
* or programmed mode.  I've tried to keep the syntax    *
* and parameters close to BASIC's VIC graphics commands. *
*                                                       *
* Sits in Bank 0 RAM at $1300-$1D84.  This simplifies   *
* the program.  In real life, you might stick it in     *
* RAM 1, or on a cartridge.  Since it overlaps the      *
```

Fig. 8-2. Source code for Grafix 80.

```
* normal starting point for BASIC program text, we do      *
* some re-arranging before loading/unloading our goodies.  *
*                                                          *
* Also: don't use any VIC bitmap commands while these      *
* 80-column graphics commands are installed, since the     *
* code is sitting in the area BASIC uses for VIC bitmaps.  *
*                                                          *
* Also #2: I've used four undocumented ROM routines for    *
* tokenizing chores and parsing BASIC command parameters.  *
* Why ?  So this code wouldn't be even larger.  In a       *
* commercial application, you'd include/rewrite these      *
* undoc'd routines in/for your code.  But what if you      *
* just want to see this stuff run, and the friendly        *
* Commodore folks do some ROM changes?  How to cope ?      *
* Well, the four routines are marked in the Constants      *
* section of the program.  They're major routines, and     *
* experience with the C64 has shown that ROM changes will  *
* most likely only affect where they live.  And so they    *
* can be found.  See the text for ways to do that.         *
*                                                          *
* The program is broken up into six source files :         *
*      GRAFIX 80 0.S   (this one)                          *
*      GRAFIX 80 1.S                                       *
*      GRAFIX 80 2.S                                       *
*      GRAFIX 80 3.S                                       *
*      GRAFIX 80 4.S                                       *
*      GRAFIX 80 5.S                                       *
*                                                          *
* To install the new commands :                            *
*      BLOAD "G80 INSTALL"                                 *
*      SYS 7592                                            *
*                                                          *
* To remove the new commands :                             *
*      G80SCAT                                             *
*                                                          *
*                                                          *
* Here are the new commands :                              *
* ( syntax is in the same style as in the BASIC 7.0        *
*   Encyclopedia section of Commodore's 128                *
*   System Guide )                                         *
*                                                          *
*   G80BOX [color source], X1, Y1 [, [X2, Y2][,paint]]     *
*                                                          *
*      source number can take on a value of 5 or 6         *
*      X1,Y1 give one corner of the box                    *
*      X2,Y2 (if present) give a second corner             *
*      X1 and X2 must be in the range 0..639               *
*      Y1 and Y2 must be in the range 0.199                *
*      paint parameter tells whether box should be         *
*         painted or not, must have value of 0 or 1        *
*                                                          *
*      Draws a box on the 80-column screen.  Color source  *
*      defaults to foreground.  The second corner          *
*      defaults to the current position of the graphics    *
*      cursor.  Paint defaults to don't paint.  If set     *
*      to paint, fills box with color source.  Upon        *
*      completion, the graphics cursor stays/goes at/to    *
*      the second corner.                                  *
*                                                          *
*      color source 5 ....... bitmap foreground (draws)    *
*      color source 6 ....... bitmap background (erases)   *
*      paint 0 .............. do not paint                 *
*      paint 1 .............. paint                        *
*                                                          *
*                                                          *
```

```
*   G80COLOR color source, color number                    *
*                                                          *
*       color source can take on a value of 5 or 6         *
*       color number can take on a value of 1..16          *
*                                                          *
*       Sets colors for the 80-column bitmap's             *
*       foreground and background.  If this command is     *
*       not given, default is white foreground on black    *
*       background.                                        *
*                                                          *
*       color source 5 ....... bitmap foreground           *
*       color source 6 ....... bitmap background           *
*                                                          *
*       color numbers ......... the standard 80-column     *
*          colors, as specified in the COLOR section of    *
*          the BASIC 7.0 Encyclopedia mentioned above      *
*          (page 248)                                      *
*                                                          *
*                                                          *
*   G80DRAW [color source], X1, Y1 [TO X2,Y2] ...          *
*   G80DRAW TO X1,Y1 [TO X2,Y2] ...                        *
*                                                          *
*       color source can take on a value of 5 or 6         *
*       All X# and Y# are absolute screen coordinates      *
*       X# coordinates can be in the range 0..639          *
*       Y# coordinates can be in the range 0..199          *
*                                                          *
*       Draws individual points, lines, or connected lines *
*       on the 80-column bitmap screen.  Works mostly like *
*       BASIC 7.0's DRAW command.  Exceptions : the lack   *
*       of a relative coordinate option, and the need to   *
*       specify at least one coordinate point.  The color  *
*       source defaults to 5 (drawing) upon entry to 80-   *
*       column graphics.  After that, it defaults to the   *
*       color source used by the last G80DRAW command.     *
*       The graphics cursor ends up at the last point      *
*       drawn.  When the second form is used (G80DRAW TO), *
*       the line starts at the current location of the     *
*       graphics cursor.                                   *
*                                                          *
*       color source 5 ....... bitmap foreground (draws)   *
*       color source 6 ....... bitmap background (erases)  *
*                                                          *
*                                                          *
*   G80GRAPHIC mode [,clear]                               *
*                                                          *
*       mode can take on a value of 5 or 6                 *
*       clear can take on a value of 0 or 1                *
*                                                          *
*       Puts the 80 column chip into one of two modes :    *
*       mode 5 .......... 80-column text                   *
*       mode 6 .......... 640h x 200v bit-mapped graphics  *
*                                                          *
*       A clear parameter of 0 doesn't clear the screen    *
*       when entering a mode.                              *
*       A clear parameter of 1 clears the screen when      *
*       entering a mode.                                   *
*       If clear is unspecified, a value of 0 is assumed.  *
*       A clear parameter of 1 when entering graphics mode *
*          resets the graphics cursor to the upper-left    *
*          corner of the screen (the point 0,0).           *
*                                                          *
*                                                          *
*   G80SCAT                                                *
*                                                          *
```

```
*   Removes the new commands.  Reclaims memory by setting *
*   BASIC text area back down.  Erases any BASIC program  *
*   in memory.                                            *
*                                                         *
*                                                         *
* Version :    1.00                                       *
* Timestamp :   8:05 PM PST    July 28, 1986              *
*                                                         *
* Programmed by Stan Krute                                *
* Copyright (C) 1986 by Stan Krute's Hacker & Nerd        *
*                      18617 Camp Creek Road              *
*                      Hornbrook, California   96044      *
* All rights reserved                                     *
* Call or write for help, bug reports, licensing, etc.    *
*                                                         *
*---------------------------------------------------------*


*------------------ Constants ---------------------------*


* BASIC

TxtTab    =     $2D         ; pointer to start of BASIC text
StdBS     =     $1C01       ; standard start of BASIC text


* built-in routines -- documented

BSOut     =     $FFD2       ; output a character
ChrGet    =     $0380       ; get next character from text
ChrGot    =     $0386       ; get current character from text
IndTxt    =     $03C9       ; get current character from text
                            ; ... no matter what the current
                            ; ... memory configuration is
Init80    =     $FF62       ; initialize 80 column screen
JmpNEW    =     $AF84       ; does the NEW command
SoftReset =     $4003       ; does a BASIC warm start
Swapper   =     $FF5F       ; switch between 40 and 80 column
                            ; ... video displays


* built-in routines -- not documented
* these are the ones that might move
* see the text for dealing with such events

FndComTxt =     $43E2       ; routine that searches for
                            ; BASIC command text
FndTknTxt =     $516A       ; routine that searches for
                            ; BASIC command text
GetByt    =     $87F4       ; gets a 1-byte integer from
                            ; ... BASIC line
GetWdByt =      $8803       ; gets a 2-byte integer and
                            ; ... a comma-separated 1-byte
                            ; ... integer from BASIC line

* colors

Black     =     1           ; BASIC 7.0 code for black
White     =     2           ; BASIC 7.0 code for white


* Commodore ASCII codes

Colon     =     $3A         ; C-ASCII for a colon
```

```
QuestMrk =      $3F          ; C-ASCII for a question mark
Quotz    =      $22          ; C-ASCII for a quotation mark
ClearScrn =     $93          ; C-ASCII for clearing the screen
Comma    =      $2C          ; C-ASCII for a comma


* memory management

MmuSCR   =      $FF00        ; secondary MMU configuration
                             ; ... register
Bank15   =      %00000000    ; memory configuration byte
                             ; ... to get a Bank 15 setup


* page zero goodies

SynTmp   =      $79          ; used for temporary goodies
RunMod   =      $7F          ; flags direct or
                             ; ... program-running mode
                             ; ... ( 0 for direct )
Mode     =      $D7          ; flags 40 (bit 7 low) or
                             ; ... 80 (bit 7 high) columns

* sentinels

BufferEnd =     0            ; marks end of input buffer


* system error codes

SynErr   =      11           ; code for Syntax Error
BadNum   =      14           ; code for Illegal Quantity
DirOnly  =      34           ; code for Direct Mode Only


* token stuff

CmndOfst =      $0D          ; where the ROM crunching
                             ; ... routine leaves a command's
                             ; ... offset in its command
                             ; ... names table
FETokFlg =      0            ; signals an FE token group to
                             ; ... the ROM's crunch routine
TknTo    =      $A4          ; total token for TO
TknBox   =      $27          ; selector token for G80BOX
TknColor =      $28          ; selector token for G80COLOR
TknDraw  =      $29          ; selector token for G80DRAW
TknGraphic =    $2A          ; selector token for G80GRAPHIC
TknScat  =      $2B          ; selector token for G80SCAT
TokenStart =    $80          ; tokens start with this code
OurFirst =      TknBox       ; first selector token for our
                             ; ... commands
OurLast  =      TknScat      ; last selector token for our
                             ; ... commands


* VDC (8563) 80-column chip registers

VDCAdr   =      $D600        ; VDC port address register
VDCDat   =      $D601        ; VDC port data register
AdrHiReg =      18           ; VDC address ptr. hi byte reg.
ModeReg  =      25           ; VDC mode register
ColReg   =      26           ; VDC color register
BytCntReg =     30           ; VDC byte count register
DataReg  =      31           ; VDC data register
```

```
* VDC (8563) 80-column text screen miscellany

BitMapSiz =    16000      ; size (bytes) of VDC bit map
ScrWidth =     640        ; screen width in pixels
HrzMax   =     639        ; maximum horizontal coordinate
BytPerLin =    80         ; screen width in bytes
ScrHite  =     200        ; screen height in pixels
VrtMax   =     199        ; maximum vertical coordinate
Bakgrnd  =     6          ; parameter code for background
Forgrnd  =     5          ; parameter code for foreground
Text     =     5          ; parameter code for text


* vectors

IError   =     $0300      ; vector to the system's error
                          ; ... handler routine
IEscEx   =     $0310      ; vector to late part of token
                          ; ... execution routine
IEscLk   =     $030C      ; vector to early part of token
                          ; ... crunching routine
IEscPr   =     $030E      ; vector to late part of token
                          ; ... un-crunching routine


*------------------- Macros ----------------------------*

* a nice pseudo-unconditional branch
BRA      MAC
         CLV
         BVC   ]1
         <<<


*------------------ Set Program Origin -------------------*

         ORG   $1300      ; a lovely little spot
                          ; 4864 in decimal

*-------------------- Install --------------------------*

* Installs the 80 column graphics commands

Install
* do some installing
1300: 20 14 13         JSR   InsCrchDtr  ; install a crunching detour
1303: 20 2D 13         JSR   InsUncrDtr  ; install an un-crunching detour
1306: 20 46 13         JSR   InsExecDtr  ; install an execution detour

* return from Install
1309: 60               RTS


*------------------ UnInstall --------------------------*

* Uninstalls the 80 column graphics commands

UnInstall
* do some uninstalling
130A: 20 5F 13         JSR   RmvCrchDtr ; remove a crunching detour
130D: 20 6E 13         JSR   RmvUncrDtr ; remove an un-crunching detour
1310: 20 7D 13         JSR   RmvExecDtr ; remove an execution detour
```

```
                * return from UnInstall
1313: 60                RTS


                        TTL   "GRAFIX 80 1.S"
                *---------------- Here Comes Another Source File ---------*

                        PUT   "GRAFIX 80 1.S"


                *---------------------- InsCrchDtr ---------------------*

                * Insert a little command crunching detour

                InsCrchDtr
                * save some registers
1314: 48                PHA

                * save the regular command crunching vector
1315: AD 0C 03          LDA   IEscLk
1318: 8D 00 1D          STA   RegIEscLk
131B: AD 0D 03          LDA   IEscLk+1
131E: 8D 01 1D          STA   RegIEscLk+1

                * set command crunching vector to our little detour
1321: A9 8C             LDA   #<IEscLkDetor
1323: 8D 0C 03          STA   IEscLk
1326: A9 13             LDA   #>IEscLkDetor
1328: 8D 0D 03          STA   IEscLk+1

                * restore some registers
132B: 68                PLA

                * return from InsCrchDtr
132C: 60                RTS


                *--------------------- InsUncrDtr ----------------------*

                * Insert a little command un-crunching detour

                InsUncrDtr
                * save some registers
132D: 48                PHA

                * save the regular command un-crunching vector
132E: AD 0E 03          LDA   IEscPr
1331: 8D 02 1D          STA   RegIEscPr
1334: AD 0F 03          LDA   IEscPr+1
1337: 8D 03 1D          STA   RegIEscPr+1

                * set command un-crunching vector to our little detour
133A: A9 BC             LDA   #<IEscPrDetor
133C: 8D 0E 03          STA   IEscPr
133F: A9 13             LDA   #>IEscPrDetor
1341: 8D 0F 03          STA   IEscPr+1

                * restore some registers
1344: 68                PLA

                * return from InsUncrDtr
1345: 60                RTS


                *--------------------- InsExecDtr ----------------------

```

```
               * Insert a little command execution detour

               InsExecDtr
               * save some registers
1346: 48               PHA

               * save the regular command execution vector
1347: AD 10 03         LDA   IEscEx      ; save the regular vector
134A: 8D 04 1D         STA   RegIEscEx
134D: AD 11 03         LDA   IEscEx+1
1350: 8D 05 1D         STA   RegIEscEx+1

               * set command execution vector to our little detour
1353: A9 D6            LDA   #<IEscExDetor
1355: 8D 10 03         STA   IEscEx
1358: A9 13            LDA   #>IEscExDetor
135A: 8D 11 03         STA   IEscEx+1

               * restore some registers
135D: 68               PLA

               * return from InsExecDtr
135E: 60               RTS


               *---------------------- RmvCrchDtr ---------------------*

               * Remove a little command crunching detour

               RmvCrchDtr
               * save some registers
135F: 48               PHA

               * restore the regular command crunching vector
1360: AD 00 1D         LDA   RegIEscLk
1363: 8D 0C 03         STA   IEscLk
1366: AD 01 1D         LDA   RegIEscLk+1
1369: 8D 0D 03         STA   IEscLk+1

               * restore some registers
136C: 68               PLA

               * return from RmvCrchDtr
136D: 60               RTS


               *--------------------- RmvUncrDtr ----------------------*

               * Remove a little command un-crunching detour

               RmvUncrDtr
               * save some registers
136E: 48               PHA

               * restore the regular command un-crunching vector
136F: AD 02 1D         LDA   RegIEscPr
1372: 8D 0E 03         STA   IEscPr
1375: AD 03 1D         LDA   RegIEscPr+1
1378: 8D 0F 03         STA   IEscPr+1

               * restore some registers
137B: 68               PLA

               * return from InsUncrDtr
137C: 60               RTS
```

```
                *--------------------- RmvExecDtr ----------------------*
                * Remove a little command execution detour
                RmvExecDtr
                * save some registers
137D: 48                 PHA
                * restore the regular command execution vector
137E: AD 04 1D           LDA   RegIEscEx
1381: 8D 10 03           STA   IEscEx
1384: AD 05 1D           LDA   RegIEscEx+1
1387: 8D 11 03           STA   IEscEx+1
                * restore some registers
138A: 68                 PLA
                * return from InsExecDtr
138B: 60                 RTS
                *------------- IEscLkDetor ------------------------------*
                * A detour for token crunching
                * Detects and crunches our new commands
                * Upon entry, Accumulator holds a byte from program text
                * Our new commands get reduced to double tokens :
                *   The lead-in token is $FE
                *   The selector tokens start at OurFirst ($27)
                * If one of our commands is found, A & X are set up
                * ... for the ROM's continuation of the crunching process
                IEscLkDetor
                * save entry byte
138C: 8D 06 1D           STA   TheCode
                * run a number of screening tests
138F: C9 00     :Test0   CMP   #BufferEnd   ; check for end of input buffer
1391: F0 22              BEQ   :NotOneOfOurs
1393: C9 3A     :Test1   CMP   #Colon       ; check for a colon
1395: F0 1E              BEQ   :NotOneOfOurs
1397: C9 3F     :Test2   CMP   #QuestMrk    ; check for a question mark
1399: F0 1A              BEQ   :NotOneOfOurs
139B: C9 80     :Test3   CMP   #TokenStart ; check for >= $80
139D: B0 16              BCS   :NotOneOfOurs
139F: C9 22     :Test4   CMP   #Quotz       ; check for quotation mark
13A1: F0 12              BEQ   :NotOneOfOurs
                * if we get here, we made it thru the screening tests
                :LetsSearch
13A3: A0 07              LDY   #<OurComsText ; set A & Y to point to
13A5: A9 1D              LDA   #>OurComsText ; ... our table of command
13A7: 20 E2 43           JSR   FndComTxt     ; ... words and go search it
13AA: 90 09              BCC   :NotOneOfOurs ; not one of our commands
                * FndComTxt sets Carry if it found one of our commands
                * so we set up for Crunch to do its crunching
                :ItsOneOfOurs
13AC: A2 00              LDX   #FETokFlg   ; X flags an $FE token
13AE: A9 A6              LDA   #TokenStart+OurFirst-1
                                           ; the -1 is there 'cuz we got
                                           ; ... here with the Carry set
13B0: 65 0D              ADC   CmndOfst    ; add the offset of the com'nd
```

```
                                                 ; ... as found in our 0-based
                                                 ; ... table of command names
13B2: 18                  CLC                    ; set flag for IEscLk branch
13B3: 90 04               BCC   :GoReglar        ; and jump back in
                  * if we get here, it's not one of our commands
                  :NotOneOfOurs
13B5: AD 06 1D            LDA   TheCode            ; restore character
13B8: 38                  SEC                    ; set flag for IEscLk branch
                  * everybody jumps to the regular vector
13B9: 6C 00 1D    :GoReglar JMP  (RegIEscLk)
                  *--------------- IEscPrDetor ----------------------------*
                  * A detour for token un-crunching
                  * Detects and un-crunches our new commands
                  * Upon entry, X holds code for FE or CE lead-in token
                  *    and A holds selector token
                  IEscPrDetor
                  * run some tests
13BC: A2 00       :Tst1   LDX   #FETokFlg   ; our commands start with FE
13BE: D0 12               BNE   :NotOurTkn  ; not ours, so bag it
13C0: C9 27       :Tst2   CMP   #OurFirst   ; is it in our selector range ?
13C2: 90 0E               BCC   :NotOurTkn  ; if not, bag it
13C4: C9 2C       :Tst3   CMP   #OurLast+1
13C6: B0 0A               BCS   :NotOurTkn
                  * deal with one of our tokens
                  * set parameters and jump to a ROM un-crunching routine
                  :ItsOurTkn                     ; we come in with Carry clear
13C8: 69 59               ADC   #TokenStart-OurFirst ; set up X to index
13CA: AA                  TAX                    ; ... our selector tokens
                                                 ; ... with base at TokenStart
13CB: A0 07               LDY   #<OurComsText; set A & Y to point to table
13CD: A9 1D               LDA   #>OurComsText; ... of our commands' text
13CF: 4C 6A 51            JMP   FndTknTxt      ; ... and jump to ROM
                                                 ; ... un-crunching routine
                  * deal with a token that's not ours
                  * set signal and jump to regular vector
                  :NotOurTkn
13D2: 38                  SEC                    ; signal not ours
13D3: 6C 02 1D            JMP   (RegIEscPr)  ; slide back in to ROM
                  *-------------------- IEscExDetor -----------------------*
                  * Detour for tokenized command execution routine
                  * Main token is $FE, selector token is in A- register
                  IEscExDetor
                  * run some tests
13D6: C9 27       :Tst1   CMP   #TknBox      ; is it G80BOX command ?
13D8: D0 05               BNE   :Tst2        ; if not, next test
13DA: 20 17 14            JSR   DoG80Box     ; it is, so hop to it
13DD: 90 22               BCC   :GoneZo1     ; if we make it back, always
13DF: C9 28       :Tst2   CMP   #TknColor    ; is it G80COLOR command ?
13E1: D0 05               BNE   :Tst3        ; if not, next test
13E3: 20 72 15            JSR   DoG80Color ; it is, so bop to it
```

```
13E6: 90 19              BCC   :GoneZo1     ; if we make it back, always

13E8: C9 29     :Tst3    CMP   #TknDraw     ; is it G80DRAW command ?
13EA: D0 05              BNE   :Tst4        ; if not, next test
13EC: 20 F9 15           JSR   DoG80Draw    ; it is, so flop to it
13EF: 90 10              BCC   :GoneZo1     ; if we make it back, always

13F1: C9 2A     :Tst4    CMP   #TknGraphic ; is it G80GRAPHIC c'mnd ?
13F3: D0 05              BNE   :Tst5          ; if not, next test
13F5: 20 93 17           JSR   DoG80Graphic; it is, so cop to it
13F8: 90 07              BCC   :GoneZo1       ; if we make it back, always

13FA: C9 2B     :Tst5    CMP   #TknScat       ; is it G80SCAT command ?
13FC: D0 04              BNE   :NotOurs       ; if not, not ours to do
13FE: 20 45 18           JSR   DoG80Scat      ; it is, so mop to it

                * return from carrying out one of our commands
1401: 60        :GoneZo1 RTS                ; the carry is clear


                * return if it wasn't one of our commands
1402: 38        :NotOurs SEC                ; signal error if not our token
1403: 6C 04 1D  :GoneZo2 JMP   (RegIEscEx); dive back into execution
                                            ; ... routine


                *-------------------- CommaCruz --------------------------*

                * Parses through required commas in a BASIC input line

                * Makes sure a comma is there, then gets next
                * ... line element

                * If no comma's there, "Syntax Error"

                * Upon successful exit :   A- reg holds result of a ChrGet
                *                          Y- reg set to 0

                * Inspired by an undocumented C-128 ROM routine

                CommaCruz
                * go grab current BASIC text character of interest
1406: A0 00              LDY   #0           ; set Y- reg for IndTxt call
1408: 20 C9 03           JSR   IndTxt       ; grab byte of BASIC line

                * is it a comma ?
140B: C9 2C              CMP   #Comma

                * if not, signal a syntax error
140D: D0 03              BNE   :NotOkay

                * 'twas a comma, so get next BASIC line element and
                * ... return
140F: 4C 80 03  :Okay    JMP   ChrGet


                * no comma, so signal a syntax error and return
1412: A2 0B     :NotOkay LDX   #SynErr      ; send out "Syntax Error"
1414: 6C 00 03           JMP   (IError)     ; ... and return
                         TTL   "GRAFIX 80 2.S"
                *---------------- Here Comes Another Source File ---------*

                         PUT   "GRAFIX 80 2.S"


                *--------------------- DoG80Box -------------------------*
```

```
                  * Deals with the G80Box command

                  DoG80Box

1417: 20 2E 14             JSR   G8BoxGtPs  ; fetch any command parameters
141A: 20 84 14             JSR   G8BoxChPs  ; check legality of parameters
141D: B0 05                BCS   :BadParams ; branch if there's a problem
141F: 20 CE 14             JSR   G8BoxDoIt  ; carry out the command

1422: 18          :OkeDoke CLC              ; signal that all went well
1423: 60                   RTS              ; return from DoG80Box

                  :BadParams
1424: A2 0E                LDX   #BadNum    ; signal  'illegal quantity'
1426: 6C 00 03             JMP   (IError)


                  *-------------- Variables for G80BOX command -------------*

1429: 00          G8BxSrc  DS    1          ; color source parameter for
                                            ; ... G80BOX command
142A: 00          G8BxP1Vt DS    1          ; vertical coordinate for first
                                            ; ... point in G80BOX command
142B: 00          G8BxP1Lo DS    1          ; horizontal lo-byte coordinate
                                            ; ... for first point in
                                            ; ... G80BOX command
142C: 00          G8BxP1Hi DS    1          ; horizontal hi-byte coordinate
                                            ; ... for first point in
                                            ; ... G80BOX command
142D: 00          G8BxPtFg DS    1          ; paint/no paint flag for the
                                            ; ... G80BOX command



                  *-------------------- G8BoxGtPs -----------------------*

                  * Fetch any command parameters for the G80Box command

                  * Since we use ROM parsing, assume all registers scrambled

                  G8BoxGtPs
                  * default paint parameter to "no paint"
142E: A9 00                LDA   #0
1430: 8D 2D 14             STA   G8BxPtFg

                  * get next element in BASIC statement after G80BOX
1433: 20 80 03             JSR   ChrGet

                  * is it a comma, signifying default color source ?
1436: C9 2C                CMP   #Comma     ; a little comparison test
1438: D0 04                BNE   :FechSrc   ; no, so fetch the source param.

                  * it is a comma, so default color source to foreground
143A: A2 05                LDX   #Forgrnd   ; get the non-zero code
143C: D0 03                BNE   :StorSrc   ; always branch to store

                  * get the color source from the command line
143E: 20 F4 87    :FechSrc JSR   GetByt     ; it's a byte-sized integer

                  * store the color source
1441: 8E 29 14    :StorSrc STX   G8BxSrc    ; store it

                  * cruise through a comma
1444: 20 06 14             JSR   CommaCruz  ; if it's not there, syntax error

```

```
                  * get first point data from the BASIC line
1447: 20 03 88             JSR   GetWdByt   ; get word-length horizontal
                                            ; ... coord. and byte-length
                                            ; ... vertical

                  * store the first point data
144A: 8E 2A 14             STX   G8BxP1Vt   ; store the vertical coord.
                                            ; ... don't worry about top/botto
m
                                            ; ... correctness right now
144D: A6 16                LDX   $16        ; holds horiz. coord. lo-byte
144F: 8E 2B 14             STX   G8BxP1Lo   ; store it -- for now, we don't
                                            ; ... worry about left/right
1452: A6 17                LDX   $17        ; holds horiz. coord. hi-byte
1454: 8E 2C 14             STX   G8BxP1Hi

                  * now, see if there's more parameters to the command
1457: 20 86 03             JSR   ChrGot     ; get current BASIC line object
145A: F0 27                BEQ   :Bye       ; if command end, git out

                  * there's more on the line, so continue parsing
145C: 20 06 14             JSR   CommaCruz  ; make sure there's a comma
                                            ; ... after first point

                  * see if next BASIC line object is a comma indicating
                  * ... no second point data
145F: C9 2C                CMP   #Comma     ; is it a comma ?
1461: F0 10                BEQ   :LukPnt    ; yup, so go look for paint
                                            ; ... parameter

                  * it's not a comma, so fetch 2nd point data from BASIC line
1463: 20 03 88    :SecPnt  JSR   GetWdByt   ; get word-length horizontal
                                            ; ... coord. and byte-length
                                            ; ... vertical

                  * store the 2nd point data as new grafix cursor coords.
1466: 8E 76 16             STX   GCVrt      ; store the vertical coord.
1469: A6 16                LDX   $16        ; holds horiz. coord. lo-byte
146B: 8E 74 16             STX   GCHrzLo    ; store it
146E: A6 17                LDX   $17        ; holds horiz. coord. hi-byte
1470: 8E 75 16             STX   GCHrzHi    ; store it

                  * see if there's a paint parameter
1473: 20 D7 1C    :LukPnt  JSR   OptNxtByt  ; go look for a small integer
1476: 8E 2D 14             STX   G8BxPtFg   ; and store it as paint flag

                  * make sure there's nothing else left on the line
1479: 20 86 03             JSR   ChrGot     ; check current line item
147C: F0 05                BEQ   :Bye       ; at end, so all's well

                  * too much stuff in command, so do a "Syntax Error"
147E: A2 0B                LDX   #SynErr    ; get the error code
1480: 6C 00 03             JMP   (IError)   ; go to the error handler

                  * so long
                  :Bye
1483: 60                   RTS              ; return from G8BoxGtPs


                  *-------------------- G8BoxChPs -----------------------*

                  * Check legality of parameters for the G80Box command

                  G8BoxChPs
                  * save some registers
```

```
1484: 48                     PHA

                      * check color source
1485: AD 29 14               LDA   G8BxSrc     ; get color source parameter
1488: C9 05                  CMP   #Forgrnd    ; is it set for foreground ?
148A: F0 04                  BEQ   :VerTest    ; if so, on to next test
148C: C9 06                  CMP   #Bakgrnd    ; is it set for background ?
148E: D0 3B                  BNE   :NoGood     ; if neither fore- nor back-
                                               ; ... we have a problem

                      * check out vertical coordinates
1490: AD 2A 14        :VerTest LDA G8BxP1Vt    ; check out a vertical param.
1493: C9 C8                  CMP   #VrtMax+1   ; is it <= the max ?
1495: B0 34                  BCS   :NoGood     ; no, so we have a problem

1497: AD 76 16               LDA   GCVrt       ; check out a vertical param.
149A: C9 C8                  CMP   #VrtMax+1   ; is it <= the max ?
149C: B0 2D                  BCS   :NoGood     ; no, so we have a problem

                      * check out horizontal coordinates
149E: AD 2C 14               LDA   G8BxP1Hi    ; check hi byte first
14A1: C9 02                  CMP   #>HrzMax    ; is it < the hi max ?
14A3: 90 09                  BCC   :2ndHorz    ; yep, so go on
14A5: D0 24                  BNE   :NoGood     ; if it's > the hi max
14A7: AD 2B 14               LDA   G8BxP1Lo    ; if hi byte = max, check lo
14AA: C9 80                  CMP   #<HrzMax+1  ; is it <= the lo max ?
14AC: B0 1D                  BCS   :NoGood     ; no, so no good parm.

14AE: AD 75 16        :2ndHorz LDA GCHrzHi     ; check hi byte first
14B1: C9 02                  CMP   #>HrzMax    ; is it < the hi max ?
14B3: 90 09                  BCC   :PtFgTst    ; yep, so on to the next test
14B5: D0 14                  BNE   :NoGood     ; if it's > the hi max
14B7: AD 74 16               LDA   GCHrzLo     ; if hi byte = max, check lo
14BA: C9 80                  CMP   #<HrzMax+1  ; is it <= the lo max ?
14BC: B0 0D                  BCS   :NoGood     ; no, so no good parm.

                      * check out paint flag
14BE: AD 2D 14        :PtFgTst LDA G8BxPtFg    ; grab the flag
14C1: F0 05                  BEQ   :OkeDoke    ; 0 is okay
14C3: 38                     SEC               ; so is 1
14C4: E9 01                  SBC   #1
14C6: D0 03                  BNE   :NoGood     ; other values are no good

                      * if we get here, all was well
14C8: 18              :OkeDoke CLC             ; signal that params. are okay
14C9: 90 01                  BCC   :Bye        ; ... and return

                      * if we get here, there was a problem
14CB: 38              :NoGood  SEC             ; signal that there's a problem
                                               ; ... with the params. and return

                      * restore some registers and leave
14CC: 68              :Bye     PLA

14CD: 60                     RTS               ; return from G8BoxChPs


                      *-------------------- G8BoxDoIt ------------------------*

                      * Carry out the G80Box command

                      G8BoxDoIt
                      * save some registers
14CE: 48                     PHA
14CF: 8A                     TXA
```

```
14D0: 48               PHA

                * make sure we're in Bank 15 memory configuration
14D1: AD 00 FF         LDA   MmuSCR     ; save current configuration
14D4: 48               PHA              ; ... on the stack
14D5: A9 00            LDA   #Bank15    ; then set to Bank 15
14D7: 8D 00 FF         STA   MmuSCR

                * set up for draw or erase
14DA: 38               SEC              ; prepare to subtract
14DB: AD 29 14         LDA   G8BxSrc    ; foreground or background
14DE: E9 06            SBC   #Bakgrnd   ; adjust to $FF or $00
14E0: 8D F9 1C         STA   OnOrOff

                * branch on paint flag
14E3: AD 2D 14         LDA   G8BxPtFg   ; 0 means not to paint
14E6: D0 21            BNE   :Paint     ; 1 means to paint

                ** draw an outlined box
                :NoPaint
                * draw first horizontal line
14E8: AD 2A 14         LDA   G8BxP1Vt
14EB: 20 34 15         JSR   :Horiz

                * draw second horizontal line
14EE: AD 76 16         LDA   GCVrt
14F1: 20 34 15         JSR   :Horiz

                * draw first vertical line
14F4: AD 2B 14         LDA   G8BxP1Lo
14F7: AE 2C 14         LDX   G8BxP1Hi
14FA: 20 56 15         JSR   :Verti

                * draw second vertical line
14FD: AD 74 16         LDA   GCHrzLo
1500: AE 75 16         LDX   GCHrzHi
1503: 20 56 15         JSR   :Verti

                * and leave
                       BRA   :Bye
1506: B8               CLV
1507: 50 23            BVC   :Bye
                       <<<


                ** draw a filled-in box
                :Paint

                * figure out which vertical coordinate is highest on
                * ... the screen, and what the height of the box is
1509: 38               SEC              ; prepare to subtract
150A: AD 2A 14         LDA   G8BxP1Vt   ; one vertical coordinate
150D: ED 76 16         SBC   GCVrt      ; the other vert. coord.
1510: 90 08            BCC   :P1VtHi    ;if G8BxP1Vt is higher on screen

                * GCVrt is highest on the screen (and Carry's set)
1512: AE 76 16  :GCVrtHi LDX GCVrt      ; it'll be the starting point
1515: 8E 2A 14         STX   G8BxP1Vt   ; so store it
1518: B0 04            BCS   :MovHit    ; always branches

                * G8BxP1Vt is highest on the screen (and Carry's clear)
151A: 49 FF     :P1VtHi EOR  #$FF       ; get absolute value of
151C: 69 01            ADC   #1         ; ... vertical coord. difference

                * A-reg now holds box height less 1
```

```
                 * we'll use that value as a loop counter
151E: AA         :MovHit  TAX                  ; move height-1 into X
151F: E8                  INX                  ; now it's height straight

                 * the line drawing loop that'll paint a filled box
1520: AD 2A 14   :PtLup   LDA    G8BxP1Vt      ; set the vertical coord.
1523: 20 34 15            JSR    :Horiz        ; ah, the usefulness of a
                                               ; ... well-designed subrout.
1526: EE 2A 14            INC    G8BxP1Vt      ; move down the screen
1529: CA                  DEX                  ; down the counter
152A: D0 F4               BNE    :PtLup        ; go 'til we ground out

                 * say good night
152C: 68         :Bye     PLA                  ; restore entry memory config.
152D: 8D 00 FF            STA    MmuSCR        ; we put it on the stack

1530: 68                  PLA                  ; restore some registers
1531: AA                  TAX
1532: 68                  PLA

1533: 60                  RTS                  ; return from G8BoxDoIt


                 *--------------- local subroutines for G8BoxDoIt

                 *--------------- :Horiz

                 * draws one of a box's horizontal lines
                 * enter with vertical coordinate in A- register

1534: 8D FC 1C   :Horiz   STA    Vrt0          ; set vertical coordinates
1537: 8D FF 1C            STA    Vrt1
153A: AD 2B 14            LDA    G8BxP1Lo      ; set one horizontal coord.
153D: 8D FA 1C            STA    Hrz0Lo
1540: AD 2C 14            LDA    G8BxP1Hi
1543: 8D FB 1C            STA    Hrz0Hi
1546: AD 74 16            LDA    GCHrzLo       ; set other horizontal coord.
1549: 8D FD 1C            STA    Hrz1Lo
154C: AD 75 16            LDA    GCHrzHi
154F: 8D FE 1C            STA    Hrz1Hi
1552: 20 83 18            JSR    DoLine        ; draw the horizontal line
1555: 60                  RTS                  ; return from :Horiz


                 *--------------- :Verti

                 * draws one of a box's vertical lines
                 * enter with horizontal coordinate in A- and X- regs.

1556: 8D FA 1C   :Verti   STA    Hrz0Lo        ; set horizontal coordinates
1559: 8D FD 1C            STA    Hrz1Lo
155C: 8E FB 1C            STX    Hrz0Hi
155F: 8E FE 1C            STX    Hrz1Hi
1562: AD 76 16            LDA    GCVrt         ; set one vertical coord.
1565: 8D FC 1C            STA    Vrt0
1568: AD 2A 14            LDA    G8BxP1Vt      ; set other vertical coord.
156B: 8D FF 1C            STA    Vrt1
156E: 20 83 18            JSR    DoLine        ; draw the vertical line
1571: 60                  RTS                  ; return from :Verti


                 *-------------------- DoG80Color -----------------------*

                 * Deals with the G80COLOR command

```

```
                DoG80Color
1572: 20 86 15           JSR   G8ColGtPs  ; fetch any command parameters
1575: 20 99 15           JSR   G8ColChPs  ; check legality of parameters
1578: B0 05              BCS   :BadParams ; branch if there's a problem
157A: 20 B6 15           JSR   G8ColDoIt  ; carry out the command
157D: 18        :OkeDoke CLC              ; signal that all went well
157E: 60                 RTS              ; return from DoG80Color
                :BadParams
157F: A2 0E              LDX   #BadNum    ; signal  'illegal quantity'
1581: 6C 00 03           JMP   (IError)
                *-------------- Variables for G80COLOR command -----------*
1584: 00        G8ColSrc DS    1          ; color source parameter for
                                          ; ... G80COLOR command
1585: 00        G8ColNum DS    1          ; color number parameter for
                                          ; ... G80COLOR command
                *---------------------- G8ColGtPs ------------------------*
                * Fetch any command parameters for G80COLOR command
                * Since we use ROM parsing, assume all registers scrambled
                G8ColGtPs
                * get the color source parameter
1586: 20 80 03           JSR   ChrGet     ; get next BASIC line element
1589: 20 F4 87           JSR   GetByt     ; fetch a small integer
158C: 8E 84 15           STX   G8ColSrc   ; save as color source parameter
                * get the color number parameter
158F: 20 06 14           JSR   CommaCruz  ; cruise thru required comma
1592: 20 F4 87           JSR   GetByt     ; fetch a small integer
1595: 8E 85 15           STX   G8ColNum   ; save as color number parameter
1598: 60                 RTS              ; return from G8ColGtPs
                *---------------------- G8ColChPs ------------------------*
                * Check legality of parameters for G80COLOR command
                G8ColChPs
                * save some registers
1599: 48                 PHA
159A: 8A                 TXA
159B: 48                 PHA
                * test color source parameter
159C: A2 05     :Test1   LDX   #Forgrnd   ; code for 80-column foreground
159E: EC 84 15           CPX   G8ColSrc   ; was it that ?
15A1: F0 06              BEQ   :Test3     ; if so, test next parameter
15A3: E8        :Test2   INX              ; code for 80-column background
15A4: EC 84 15           CPX   G8ColSrc   ; was it that ?
15A7: D0 08              BNE   :BadParms  ; if not, error condition
                * test color number parameter
15A9: AE 85 15  :Test3   LDX   G8ColNum   ; now, test color number
15AC: CA                 DEX              ; adjust into easier test state
15AD: E0 10              CPX   #16        ; was number in the range 1..16 ?
15AF: 90 01              BCC   :ByeBye    ; yes, so return with signal set
```

```
                                             ; ... for okayed params.

                  * if we get here, something was amiss
                  :BadParms
15B1: 38                   SEC               ; set signal for 'illegal
                                             ; ... quantity' error

                  * restore some registers and leave
15B2: 68          :ByeBye  PLA
15B3: AA                   TAX
15B4: 68                   PLA

15B5: 60                   RTS               ; return from G8ColChPs


                  *--------------------- G8ColDoIt -----------------------*

                  * carry out the G80COLOR command
                  * upon entry, there's a legal source number in G8ColSrc
                  *            there's a legal color number in G8ColNum

                  G8ColDoIt
                  * save some registers
15B6: 48                   PHA
15B7: 8A                   TXA
15B8: 48                   PHA
15B9: 98                   TYA
15BA: 48                   PHA

                  * make sure we're in Bank 15 memory configuration
15BB: AD 00 FF             LDA   MmuSCR      ; save current configuration
15BE: 48                   PHA               ; ... on the stack
15BF: A9 00                LDA   #Bank15     ; then set to Bank 15
15C1: 8D 00 FF             STA   MmuSCR

                  * fetch the appropriate 80-column color nibble
15C4: AE 85 15             LDX   G8ColNum    ; grab the BASIC color number
15C7: CA                   DEX               ; put into indexing range 0..15
15C8: BD 71 1D             LDA   HuNb80Tb,X  ; grab the proper nibble
15CB: 8D 85 15             STA   G8ColNum    ; park the nibble value

                  * adjust color source value for indexing
15CE: AD 84 15             LDA   G8ColSrc    ; get the source
15D1: 38                   SEC               ; prepare for subtraction
15D2: E9 05                SBC   #Forgrnd    ; put into range 0..1 for
                                             ; ... indexing chores
15D4: A8                   TAY               ; move adjusted source to Y reg.
15D5: D0 0A                BNE   :Adjust     ; branch if doing background

                  * setting foreground, so move color nibble into
                  * ... hi bits of byte
                  * remember, it's parked on the stack
15D7: AD 85 15             LDA   G8ColNum     ; for foreground, move the
15DA: 0A                   ASL               ; ... color number into the
15DB: 0A                   ASL               ; ... high nibble
15DC: 0A                   ASL
15DD: 0A                   ASL
15DE: 8D 85 15             STA   G8ColNum

                  :Adjust
                  * adjust the VDC color byte
15E1: A2 1A                LDX   #ColReg     ; get current color values
15E3: 20 CB 1C             JSR   VDCRegPeek : ... from the VDC register
15E6: 39 32 1D             AND   HueNbMsk,Y  ; mask out the desired nibble
15E9: 0D 85 15             ORA   G8ColNum    ; OR in the desired color nibble
```

```
15EC: 20 BD 1C          JSR   VDCRegPoke ; set the VDC color byte

                  * restore entry memory configuration
15EF: 68                PLA              ; remember, it's on the stack
15F0: 8D 00 FF          STA   MmuSCR

                  * restore some registers and leave
15F3: 68                PLA
15F4: A8                TAY
15F5: 68                PLA
15F6: AA                TAX
15F7: 68                PLA

15F8: 60                RTS              ; return from G8ColDoIt command
                        TTL   "GRAFIX 80 3.S"
                  *---------------- Here Comes Another Source File ---------*

                        PUT   "GRAFIX 80 3.S"


                  *--------------------- DoG80Draw -------------------------*

                  * Implements the G80DRAW command

                  DoG80Draw
15F9: 20 77 16          JSR   G8DrwGtPs  ; fetch any command parameters
                                         ; (they're put into a list)
15FC: 20 E7 16          JSR   G8DrwChPs  ; check legality of parameters
15FF: B0 08             BCS   :BadParams ; branch if there's a problem
1601: 20 22 17          JSR   G8AjGrfCrs ; adjust graphics cursor
1604: 20 3C 17          JSR   G8DrwDoIt  ; carry out the command

1607: 18                CLC              ; signal that all went well
1608: 60          :Bye  RTS              ; return from DoG80Draw

                  :BadParams
1609: A2 0E             LDX   #BadNum    ; signal 'illegal quantity'
160B: 6C 00 03          JMP   (IError)


                  *-------------- Variables for G80DRAW command ---------*

160E: 00          G8DrwSrc DS  1         ; color source for G80DRAW
160F: 00          G8DLPts  DS  1         ; number of points in draw-list
1610: 00          G8DLNdx  DS  1         ; indexes open slot in draw-list
1611: 00 00 00    G8DrwLst DS  99        ; draw-list :  holds up to 33
1614: 00 00 00 00 00 00 00 00
161C: 00 00 00 00 00 00 00 00
1624: 00 00 00 00 00 00 00 00
162C: 00 00 00 00 00 00 00 00
1634: 00 00 00 00 00 00 00 00
163C: 00 00 00 00 00 00 00 00
1644: 00 00 00 00 00 00 00 00
164C: 00 00 00 00 00 00 00 00
1654: 00 00 00 00 00 00 00 00
165C: 00 00 00 00 00 00 00 00
1664: 00 00 00 00 00 00 00 00
166C: 00 00 00 00 00 00 00 00
                                         ; ... points
1674: 00          GCHrzLo  DS  1         ; graphics cursor horizontal
                                         ; ... lo byte
1675: 00          GCHrzHi  DS  1         ; graphics cursor horizontal
                                         ; ... hi byte
1676: 00          GCVrt    DS  1         ; graphics cursor vertical

```

```
                DLPntr   =      $FA         ; a zero-page pointer to the
                                            ; ... draw-list


                *--------------------- G8DrwGtPs -----------------------*

                * Fetch any command parameters for G80DRAW command
                * Put points in a point list
                * Since it uses ROM parsing, assume all registers scrambled

                G8DrwGtPs
                * initialize a nice, clean point list
1677: 20 B6 16           JSR    InitPntLst

                * get next element in BASIC statement
167A: 20 80 03           JSR    ChrGet

                * determine whether command is G80DRAW or G80DRAW TO
167D: C9 A4              CMP    #TknTo      ; is it G80DRAW TO ...  ?
167F: F0 14              BEQ    :DrawTo     ; if so, branch

                :JustDraw                   ; if not, it's just G80DRAW
                * do they want the default color source ?
1681: C9 2C              CMP    #Comma      ; is first element a comma ?
1683: D0 04              BNE    :FechSrc    ; no, so fetch a source
1685: A2 05              LDX    #Forgrnd    ; yes, so default to foreground
1687: D0 03              BNE    :StorSrc    ; always branch to store

                * get the color source from command line
1689: 20 F4 87  :FechSrc JSR    GetByt      ; it's a byte-sized integer
168C: 8E 0E 16  :StorSrc STX    G8DrwSrc    ; store it

                * cruise through a comma
168F: 20 06 14           JSR    CommaCruz   ; if not there, syntax error
                         BRA    :GetPntLn   ; then git on down to point
1692: B8                 CLV
1693: 50 13              BVC    :GetPntLn
                         <<<
                                            ; ... nabbing

                :DrawTo                     ; it's a G80DRAW TO command
                * store graphics cursor position as first point
1695: AD 74 16           LDA    GCHrzLo     ; just feed graphics cursor
1698: 85 16              STA    $16         ; ... position to the
169A: AD 75 16           LDA    GCHrzHi     ; ... StorPntLst routine
169D: 85 17              STA    $17
169F: AE 76 16           LDX    GCVrt
16A2: 20 C9 16           JSR    StorPntLst  ; that stores it

16A5: 20 80 03  :GetLnElm JSR   ChrGet      ; get next element from BASIC
                                            ; ... line
                :GetPntLn
                * get a point from BASIC line
16A8: 20 03 88           JSR    GetWdByt    ; get word-length horizontal
                                            ; ... coord. and byte-length
                                            ; ... vertical
16AB: 20 C9 16           JSR    StorPntLst  ; store point in list

                * look for a TO
16AE: 20 86 03           JSR    ChrGot      ; get current element from
                                            ; ... BASIC input line
16B1: C9 A4              CMP    #TknTo      ; is it the TO token ?
16B3: F0 F0              BEQ    :GetLnElm   ; if so, look for another pair
                                            ; ... of point coordinates

```

```
16B5: 60                 RTS               ; if not, return from G8DrwGtPs


                *------------------- InitPntLst ------------------------*

                * Initialize a fresh, clean, blank point list

                InitPntLst
                * save registers
16B6: 48                 PHA

                * reset length and index
16B7: A9 00              LDA   #0
16B9: 8D 0F 16           STA   G8DLPts     ; no points in the list yet
16BC: 8D 10 16           STA   G8DLNdx     ; 0th slot is next to fill

                * reset the draw list pointer to beginning of list
16BF: A9 11              LDA   #<G8DrwLst ; put the lo and hi
16C1: 85 FA              STA   DLPntr     ; ... bytes into the
16C3: A9 16              LDA   #>G8DrwLst ; ... zero-page pointer
16C5: 85 FB              STA   DLPntr+1

                * restore registers and leave
16C7: 68                 PLA

16C8: 60                 RTS               ; return from InitPntLst


                *------------------- StorPntLst ------------------------*

                * Store a point's coordinates in our point list

                * Upon entry :  $16 holds horizontal coordinate lo-byte
                *               $17 holds horizontal coordinate hi-byte
                *               X reg. holds vertical coordinate
                *               DLPntr points to the list of points
                *               G8DLNdx indexes next open slot in the list
                *               G8DLPts counts points in the list

                StorPntLst
                * save some registers
16C9: 48                 PHA
16CA: 98                 TYA
16CB: 48                 PHA

                * add horizontal coordinate to list
16CC: A5 16              LDA   $16         ; get horizontal lo-byte
16CE: AC 10 16           LDY   G8DLNdx     ; get our list indexer
16D1: 91 FA              STA   (DLPntr),Y  ; add to list
16D3: C8                 INY               ; up the indexer

16D4: A5 17              LDA   $17         ; get horizontal hi-byte
16D6: 91 FA              STA   (DLPntr),Y  ; add to list
16D8: C8                 INY               ; up the indexer

                * add vertical coordinate to list
16D9: 8A                 TXA               ; vertical coord. next
16DA: 91 FA              STA   (DLPntr),Y ; add to list
16DC: C8                 INY               ; up the indexer

                * store the changed list indexer
16DD: 8C 10 16           STY   G8DLNdx

                * increment the point counter
16E0: EE 0F 16           INC   G8DLPts
```

```
                * restore some registers and leave
16E3: 68                 PLA
16E4: A8                 TAY
16E5: 68                 PLA
16E6: 60                 RTS                ; return from StorPntLst
                *-------------------- G8DrwChPs ----------------------*
                * Check legality of parameters for G80DRAW command
                G8DrwChPs
                * save some registers
16E7: 48                 PHA
16E8: 8A                 TXA
16E9: 48                 PHA
16EA: 98                 TYA
16EB: 48                 PHA
                * check out value of source parameter
16EC: AD 0E 16           LDA   G8DrwSrc     ; grab it
16EF: C9 05              CMP   #Forgrnd     ; it can be foreground or
16F1: F0 04              BEQ   :ChkLst      ; ... background
16F3: C9 06              CMP   #Bakgrnd
16F5: D0 24              BNE   :BadPs       ; if neither, bad parms.
                * loop to check all points in the point list
                * prepare for looping
16F7: AE 0F 16  :ChkLst  LDX   G8DLPts      ; get number of points
16FA: A0 01              LDY   #1           ; initialize our indexer
                * top of loop
                * check a horizontal coordinate
                :HChkOne
16FC: B1 FA              LDA   (DLPntr),Y   ; compare hi-byte of horiz.
16FE: C9 02              CMP   #>ScrWidth   ; ... coord. with screen width
1700: 90 0A              BCC   :VChkOne     ; if less than, horz. point's
                                            ; ... okay, so check vertical
1702: D0 17              BNE   :BadPs       ; if greater than, itza nogudski
                :HChkTwo                    ; hi bytes match, check lo bytes
1704: 88                 DEY                ; slide back to horz. lo byte
1705: B1 FA              LDA   (DLPntr),Y
1707: C9 80              CMP   #<ScrWidth   ; check it out
1709: B0 10              BCS   :BadPs       ; has to be less than for kosher
170B: C8                 INY                ; slide indexer back
                * check a vertical coordinate
170C: C8        :VChkOne INY                ; move up to vertical value
170D: B1 FA              LDA   (DLPntr),Y
170F: C9 C8              CMP   #ScrHite     ; check against screen height
1711: B0 08              BCS   :BadPs       ; again, has to be less than
                * the point's coordinates were okay, so test for
                * ... loop doneness
                :OkPnt                      ; the point passed muster
1713: C8                 INY                ; up the indexer
1714: C8                 INY
1715: CA                 DEX                ; count another point done
1716: D0 E4              BNE   :HChkOne     ; if all not done, branch
                * we get here if all parameters were okay
1718: 18        :OkayPs  CLC                ; signal okay parameters
```

```
1719: 90 01            BCC   :Bye          ; skip next instruction

               * we get here if there was a problem
171B: 38       :BadPs  SEC                 ; signal bad parameters

               * restore some registers and leave
171C: 68       :Bye    PLA
171D: A8               TAY
171E: 68               PLA
171F: AA               TAX
1720: 68               PLA

1721: 60               RTS                 ; return from G8DrwChPs


               *------------------ G8AjGrfCrs -------------------------*

               * Adjust the graphics cursor for the draw command

               * Set it to the last point of the about-to-drawn point list

               * Done before the actual drawing to make life easier for
               * all of us.

               G8AjGrfCrs
               * save some registers
1722: 48               PHA
1723: 8A               TXA
1724: 48               PHA
1725: 98               TYA
1726: 48               PHA

               * locate last entry on point list
1727: AC 10 16         LDY   G8DLNdx       ; points to 1 past last entry
172A: 88               DEY                 ; now points to last entry's
                                           ; ... vertical coordinate
172B: A2 02            LDX   #2            ; it'll index into cursor

               * transfer that point's three bytes to graphics cursor
172D: B1 FA    :LupTop LDA   (DLPntr),Y    ; get a byte of last point
172F: 9D 74 16         STA   GCHrzLo,X     ; store it as a byte of GC
1732: 88               DEY                 ; down the indexes
1733: CA               DEX
1734: 10 F7            BPL   :LupTop       ; move three bytes

               * restore some registers and leave
1736: 68               PLA
1737: A8               TAY
1738: 68               PLA
1739: AA               TAX
173A: 68               PLA

173B: 60               RTS                 ; return from G8AjGrfCrs


               *------------------ G8DrwDoIt --------------------------*

               * Carry out the G80DRAW command

               G8DrwDoIt
               * save some registers
173C: 48               PHA
173D: 8A               TXA
173E: 48               PHA
173F: 98               TYA
```

```
1740: 48                PHA

                  * play with memory configuration
1741: AD 00 FF          LDA   MmuSCR      ; save current memory config.
1744: 48                PHA               ; ... on the stack
1745: A9 00             LDA   #0          ; make sure we're in Bank 15
1747: 8D 00 FF          STA   MmuSCR

                  * see if there are any points to draw
174A: AE 0F 16          LDX   G8DLPts     ; X holds # of points in list
174D: F0 3A             BEQ   :Bye        ; no points to draw

                  * there ARE points to draw
174F: 38                SEC               ; set up to draw or erase
1750: AD 0E 16          LDA   G8DrwSrc    ; foreground source draws,
1753: E9 06             SBC   #Bakgrnd    ; ... background source erases
1755: 8D F9 1C          STA   OnOrOff     ; bit 7 of this variable
                                          ; ... indicates source

                  * branch if just one point to draw
1758: CA                DEX               ; just 1 point to draw ?
1759: F0 1E             BEQ   :SolePoint  ; if so, do it

                  * there's 1 or more line segments to draw
                  * X holds number of line segments

                  * transfer the line's coordinates for function call
                  :CordStor
175B: A0 05             LDY   #5          ; 6 bytes of coords. to store
175D: B1 FA       :CordLup LDA (DLPntr),Y ; get a byte
175F: 99 FA 1C          STA   Hrz0Lo,Y    ; store it
1762: 88                DEY
1763: 10 F8             BPL   :CordLup

                  * call the line drawer
1765: 20 83 18          JSR   DoLine      ; draw that line

                  * check for more line segments
1768: CA                DEX               ; decrement the point counter
1769: F0 1E             BEQ   :Bye        ; when it hits zero, we done

                  * move our list pointer along
176B: 18                CLC
176C: A9 03             LDA   #3
176E: 65 FA             ADC   DLPntr
1770: 85 FA             STA   DLPntr
1772: 90 02             BCC   :more
1774: E6 FB             INC   DLPntr+1
                  :more BRA   :CordStor   ; always branch back up
1776: B8                CLV
1777: 50 E2             BVC   :CordStor
                        <<<

                  * draw a single point
                  :SolePoint
1779: A0 02             LDY   #2          ; index into point data
177B: B1 FA       :SetPnt LDA (DLPntr),Y  ; grab some point data
177D: 99 FA 1C          STA   Hrz0Lo,Y    ; transfer it
1780: 88                DEY
1781: 10 F8             BPL   :SetPnt     ; go 'til all transferred

1783: 20 17 1C          JSR   FigPoint    ; prepare to plot
1786: 20 67 1C          JSR   PlotIt      ; plot

                  * all done drawing
```

```
                :Bye
                * restore entry memory configuration
1789: 68                 PLA
178A: 8D 00 FF           STA   MmuSCR

                * restore some registers and leave
178D: 68                 PLA
178E: A8                 TAY
178F: 68                 PLA
1790: AA                 TAX
1791: 68                 PLA

1792: 60                 RTS                ; return from G8DrwDoIt


                *-------------------- DoG80Graphic ----------------------*

                * Implements the G80GRAPHIC command

                DoG80Graphic
1793: 20 A7 17           JSR   G8GrfGtPs    ; fetch any command parameters
1796: 20 B7 17           JSR   G8GrfChPs    ; check legality of parameters
1799: B0 05              BCS   :BadParams   ; branch if there's a problem
179B: 20 D7 17           JSR   G8GrfDoIt    ; carry out the command

179E: 18                 CLC                ; signal that all went well
179F: 60                 RTS                ; return from DoG80Color

                :BadParams
17A0: A2 0E              LDX   #BadNum      ; signal 'illegal quantity'
17A2: 6C 00 03           JMP   (IError)


                *------------- Variables for G80GRAPHIC command ---------*

17A5: 00        G8GrMod  DS    1            ; mode parameter for G80GRAPHIC
17A6: 00        G8GrClr  DS    1            ; clear parameter for G80GRAPHIC


                *--------------------- G8GrfGtPs -----------------------*

                * Fetch any command parameters for G80GRAPHIC command
                * Since we use ROM parsing, assume all registers scrambled

                G8GrfGtPs

                * get mode parameter
17A7: 20 80 03           JSR   ChrGet       ; get next BASIC line element
17AA: 20 F4 87           JSR   GetByt       ; get a byte-sized parameter
17AD: 8E A5 17           STX   G8GrMod      ; and store it

                * get optional clear parameter
17B0: 20 D7 1C           JSR   OptNxtByt    ; get optional byte-sized param.
17B3: 8E A6 17           STX   G8GrClr      ; and store it

17B6: 60                 RTS                ; return from G8GrfGtPs


                *--------------------- G8GrfChPs -----------------------*

                * Check legality of parameters for G80GRAPHIC command

                G8GrfChPs
                * save some registers
17B7: 48                 PHA
```

```
17B8: 8A                 TXA
17B9: 48                 PHA

                  * test mode parameter
17BA: A2 05       :Test1 LDX  #Text      ; code for 80-column text
17BC: EC A5 17           CPX  G8GrMod    ; was it that ?
17BF: F0 06              BEQ  :Test3     ; if so, test next parameter
17C1: E8          :Test2 INX             ; code for 80-column graphics
17C2: EC A5 17           CPX  G8GrMod    ; was it that ?
17C5: D0 08              BNE  :BadParms  ; if not, error condition

                  * test clear parameter
17C7: AE A6 17    :Test3 LDX  G8GrClr    ; now, test clear parameter
17CA: F0 06              BEQ  :OKParms   ; 0 is an okay parameter
17CC: CA                 DEX
17CD: F0 03              BEQ  :OKParms   ; so is 1

                  * if we get here, something's wackoso
                  :BadParms
17CF: 38                 SEC             ; set signal for 'illegal
                                         ; quantity' error
17D0: B0 01              BCS  :Bye       ; always branches

                  * if we get here, all's well
17D2: 18          :OKParms CLC           ; set signal for okay

                  * restore some registers and leave
17D3: 68          :Bye   PLA             ; restore registers
17D4: AA                 TAX
17D5: 68                 PLA

17D6: 60                 RTS             ; return from G8ColChPs


                  *------------------ G8GrfDoIt --------------------------*

                  * Carry out the G80GRAPHIC command

                  G8GrfDoIt
                  * save some registers
17D7: 48                 PHA
17D8: 8A                 TXA
17D9: 48                 PHA
17DA: 98                 TYA
17DB: 48                 PHA

                  * adjust memory configuration
17DC: AD 00 FF           LDA  MmuSCR     ; save current configuration
17DF: 48                 PHA             ; ... on the stack
17E0: A9 00              LDA  #Bank15    ; move into memory bank 15
17E2: 8D 00 FF           STA  MmuSCR

                  * turn graphics mode value into an index
17E5: AD A5 17           LDA  G8GrMod    ; adjust the mode into an index
17E8: 38                 SEC             ; subtract 5, so it goes to 0..1
17E9: E9 05              SBC  #Text      ; 0-text   1-grafix
17EB: A8                 TAY             ; we'll use the mode to index
                                         ; ... into some tables

                  * adjust text and attribute bits
17EC: A2 19              LDX  #ModeReg   ; get current setting
17EE: 20 CB 1C           JSR  VDCRegPeek

17F1: 39 2E 1D           AND  VAndMsks,Y; turn on textor disable
                                         ; ... attributes
```

```
17F4: 19 30 1D >485          ORA   VOrMsks,Y  ; enable attributes or turn on
               >486                           ; ... grafix
               >487
17F7: 20 BD 1C >488          JSR   VDCRegPoke ; put the adjusted setting
               >489                           ; ... back in its reg.
               >490
               >491 *    initialize things according to mode
17FA: 98       >492          TYA              ; sets flags for testing
17FB: 8D A5 17 >493          STA   G8GrMod    ; ... and saves a copy of mode
17FE: D0 06    >494          BNE   :GrfOnly   ; grafix mode branches
               >495
               >496 * do some initialization for text mode only
1800: 20 62 FF >497 :TxtOnly JSR   Init80     ; copy character set into
               >498                           ; ... VDC's RAM memory
               >499          BRA   :ClrChk    ; and branch
1803: B8       >499          CLV
1804: 50 1F    >499          BVC   :ClrChk
               >499          <<<
               >500
               >501 * do some command initialization for graphics mode only
1806: A9 05    >502 :GrfOnly LDA   #Forgrnd   ; default G80DRAW to draw,
1808: 8D 0E 16 >503          STA   G8DrwSrc   ; ... not erase
180B: A9 02    >504          LDA   #White     ; default to white foreground
180D: 8D 85 15 >505          STA   G8ColNum
1810: A9 05    >506          LDA   #Forgrnd
1812: 8D 84 15 >507          STA   G8ColSrc
1815: 20 B6 15 >508          JSR   G8ColDoIt
1818: A9 01    >509          LDA   #Black     ; default to black background
181A: 8D 85 15 >510          STA   G8ColNum
181D: A9 06    >511          LDA   #Bakgrnd
181F: 8D 84 15 >512          STA   G8ColSrc
1822: 20 B6 15 >513          JSR   G8ColDoIt
               >514
               >515 * see if we have to do some clearing
1825: AD A6 17 >516 :ClrChk  LDA   G8GrClr    ; check out the clear parameter
1828: F0 11    >517          BEQ   :Bye       ; zero sez 'no need to clear'
               >518
               >519 * see if we're to clear text or graphics
182A: AD A5 17 >520 :DoClr   LDA   G8GrMod    ; get mode index back
182D: D0 06    >521          BNE   :ClrGraf   ; branch for graphics
               >522
               >523 * clear 80-column text
               >524 :ClrTxt
182F: 20 D0 1B >525          JSR   ClrTx80    ; clear 80-column text
               >526          BRA   :Bye       ; skip next instructions
1832: B8       >526          CLV
1833: 50 06    >526          BVC   :Bye
               >526          <<<
               >527
               >528 * clear 80-column graphics
               >529 :ClrGraf
1835: 20 EE 1B >530          JSR   ClrGr80    ; clear 80-column graphics
1838: 20 75 18 >531          JSR   IntGrfCrs  ; initialize the 80-column
               >532                           ; ... graphics cursor
               >533
               >534 * clean up and leave
               >535 * restore entry memory configuration
183B: 68       >536 :Bye     PLA
183C: 8D 00 FF >537          STA   MmuSCR
               >538
               >539 * restore some registers and leave
183F: 68       >540          PLA
1840: A8       >541          TAY
1841: 68       >542          PLA
1842: AA       >543          TAX
```

```
1843: 68                  PLA

1844: 60                  RTS               ; return from G8GrfDoIt


         *----------------- DoG80Scat----------------------------*

         * Implements the G80SCAT command

         * Assume all registers trashed

         DoG80Scat

         * make sure we're in direct mode
1845: A5 7F               LDA   RunMod      ; check for direct or program
1847: F0 05               BEQ   :Okay       ; zero flags direct mode

         * not in direct mode, so signal an error
1849: A2 22               LDX   #DirOnly    ; code for "Direct Mode Only"
184B: 6C 00 03            JMP   (IError)    ; jump to the error handler

         * make sure 80 column screen is set back to text
         * ... by doing a G80GRAPHIC 5,1 command
184E: A9 05      :Okay    LDA   #5
1850: 8D A5 17            STA   G8GrMod
1853: A9 01               LDA   #1
1855: 8D A6 17            STA   G8GrClr
1858: 20 D7 17            JSR   G8GrfDoIt

         * then uninstall the new commands
185B: 20 0A 13            JSR   UnInstall

         * set BASIC back in space
         * zero out the byte just before BASIC text
185E: A9 00               LDA   #0
1860: 8D 00 1C            STA   StdBS-1

         * set BASIC text start to standard position
1863: A9 01               LDA   #<StdBS
1865: 85 2D               STA   TxtTab
1867: A9 1C               LDA   #>StdBS
1869: 85 2E               STA   TxtTab+1

         * do a BASIC NEW command
186B: A9 00               LDA   #0          ; prep zero flag for NEW
186D: 20 84 AF            JSR   JmpNEW      ; go do it

         * do a warm start of BASIC
1870: 20 03 40            JSR   SoftReset

         * git on out
1873: 18                  CLC               ; signal that all went well
1874: 60                  RTS               ; return from DoG80Scat
                          TTL   "GRAFIX 80 4.S"
         *--------------- Here Comes Another Source File ---------*

                          PUT   "GRAFIX 80 4.S"


         *------------------ IntGrfCrs --------------------------*

         * Initialize the 80-column graphics cursor

         IntGrfCrs
         * save some registers
```

```
1875: 48                PHA

                * do it to it
1876: A9 00             LDA   #0          ; the initializing value
1878: 8D 74 16          STA   GCHrzLo     ; set 'em up
187B: 8D 75 16          STA   GCHrzHi
187E: 8D 76 16          STA   GCVrt

                * restore some registers and leave
1881: 68                PLA

1882: 60                RTS               ; return from IntGrfCrs



                *--------------------- DoLine --------------------------*

                * Draws lines

                * Upon entry, endpoints of line are stored in Hrz0Lo,
                *             Hrz0Hi, Vrt0, Hrz1Lo, Hrz1Hi, Vrt1

                DoLine
                * save some registers
1883: 48                PHA
1884: 8A                TXA
1885: 48                PHA

                * if vertical line, special case it
1886: AD FA 1C  :TstVrt LDA   Hrz0Lo      ; if horizontal coordinates
1889: CD FD 1C          CMP   Hrz1Lo      ; ... match, it's vertical
188C: D0 0E             BNE   :AdjEPs     ; not vertical, so branch
188E: AD FB 1C          LDA   Hrz0Hi
1891: CD FE 1C          CMP   Hrz1Hi
1894: D0 06             BNE   :AdjEPs

1896: 20 E4 18  :Vertcal JSR  DoVert      ; we had a match, so verticalize
                        BRA   :Bye        ; and get on out
1899: B8                CLV
189A: 50 44             BVC   :Bye
                        <<<

                * adjust endpoints (if necessary)

                * this way, calls to the horizontal and sloped line
                * drawers get the first point leftmost, second point
                * rightmost

189C: AD FE 1C  :AdjEPs LDA   Hrz1Hi      ; start by comparing horz.
189F: CD FB 1C          CMP   Hrz0Hi      ; ... hi bytes
18A2: 90 0A             BCC   :EndAdj     ; do adjustment
18A4: D0 29             BNE   :TstHrz     ; no adjustment needed, so go on

18A6: AE FD 1C  :AdTs2  LDX   Hrz1Lo      ; hi bytes equal, so test
18A9: EC FA 1C          CPX   Hrz0Lo      ; ... lo bytes
18AC: B0 21             BCS   :TstHrz     ; no adjustment needed if
                                          ; ... greater than or equal
18AE: AE FB 1C  :EndAdj LDX   Hrz0Hi      ; swap endpoints
18B1: 8D FB 1C          STA   Hrz0Hi      ; first, horizontal hi bytes
18B4: 8E FE 1C          STX   Hrz1Hi

18B7: AD FA 1C          LDA   Hrz0Lo      ; now horizontal lo bytes
18BA: AE FD 1C          LDX   Hrz1Lo
18BD: 8E FA 1C          STX   Hrz0Lo
18C0: 8D FD 1C          STA   Hrz1Lo
```

```
18C3: AD FF 1C          LDA   Vrt1        ; now vertical bytes
18C6: AE FC 1C          LDX   Vrt0
18C9: 8D FC 1C          STA   Vrt0
18CC: 8E FF 1C          STX   Vrt1

                 * not vertical -- is it horizontal ?
18CF: AD FF 1C   :TstHrz LDA   Vrt1        ; if vertical coordinates
18D2: CD FC 1C          CMP   Vrt0        ; ... match, it's horizontal
18D5: D0 06             BNE   :Sloped     ; not horizontal, so branch

18D7: 20 1B 19   :Hrzntal JSR  DoHorz      ; we had a match, so go do it
                        BRA   :Bye        ; and get on out
18DA: B8                CLV
18DB: 50 03             BVC   :Bye
                        <<<

                 * neither vertical nor horizontal -- it's sloped
                 * we use a version of Bresenham's algorithm on these

18DD: 20 46 1A   :Sloped JSR   DoBres      ; draw all other lines

                 * restore some registers and leave
18E0: 68         :Bye    PLA
18E1: AA                TAX
18E2: 68                PLA

18E3: 60                RTS               ; return from DoLine


                 *------------------- DoVert ----------------------------*

                 * Draws vertical lines on the 80-column screen

                 DoVert
                 * save some registers
18E4: 48                PHA
18E5: 8A                TXA
18E6: 48                PHA

                 * figure topmost point and change in verticality (delta Y)
18E7: 38                SEC               ; prepare for subtraction
18E8: AD FC 1C          LDA   Vrt0        ; subtract one from the other
18EB: ED FF 1C          SBC   Vrt1
18EE: B0 04             BCS   :Start1     ; if result positive, branch

18F0: 49 FF      :Start0 EOR   #$FF        ; get absolute value of delta Y
18F2: 69 01             ADC   #1          ; remember, Carry's clear

18F4: AA         :Start1 TAX               ; absolute value into X
18F5: 90 06             BCC   :PlotFirst ; want lowest value in Vrt0
18F7: AD FF 1C          LDA   Vrt1
18FA: 8D FC 1C          STA   Vrt0

                 * plot the first point
                 :PlotFirst
18FD: 20 17 1C          JSR   FigPoint    ; figure out stuff for first poin
t
                                          ; ... Hrz0 and Vrt0
1900: E8                INX               ; up so it can come down
1901: D0 0E             BNE   :VPlot      ; and branch always to plot it

                 * adjust the address of the target byte -- down 1 line
1903: 18         :NxtVPt CLC               ; prepare to add
1904: AD F4 1C          LDA   BtByLo      ; get current byte address
1907: 69 50             ADC   #BytPerLin ; add a line's worth of bytes
```

```
1909: 8D F4 1C              STA   BtByLo
190C: 90 03                 BCC   :VPlot
190E: EE F5 1C              INC   BtByHi
                     * plot a pixel
1911: 20 67 1C       :VPlot JSR   PlotIt       ; plot pixel
                     * see if we're done yet
1914: CA             :VTest DEX                ; one more pixel done
1915: D0 EC                 BNE   :NxtVPt      ; go until down to zero
                     * restore some registers and leave
1917: 68             :Bye   PLA
1918: AA                    TAX
1919: 68                    PLA
191A: 60                    RTS                ; return from DoVert
                     *------------------- DoHorz ----------------------------*
                     * Draws horizontal lines on the 80-column screen
                     * Upon entry, first point is leftmost, second point is
                     * rightmost
                     DoHorz
                     * save some registers
191B: 48                    PHA
191C: 8A                    TXA
191D: 48                    PHA
191E: 98                    TYA
191F: 48                    PHA
                     * figure delta X  -- line length is delta X plus one
1920: 38                    SEC                ; subtract leftmost point from
1921: AD FD 1C              LDA   Hrz1Lo       ; ... rightmost point
1924: ED FA 1C              SBC   Hrz0Lo
1927: 8D 2F 1A              STA   :DXLo        ; store result at :DXLo-:DXHi
192A: AD FE 1C              LDA   Hrz1Hi
192D: ED FB 1C              SBC   Hrz0Hi
1930: 8D 30 1A              STA   :DXHi
                     * figure out initial drawing information for leftmost point
1933: 20 17 1C              JSR   FigPoint     ; set up to draw
                     * figure out right pixel index
1936: AD FD 1C              LDA   Hrz1Lo       ; take right horizontal
1939: 29 07                 AND   #7           ; ... position mod 8
193B: 8D 32 1A              STA   :RitNdx      ; store for later use
                     * figure out left pixel index
193E: AD F7 1C              LDA   BitPos       ; FigPoint already found it
1941: 8D 31 1A              STA   :LftNdx      ; store for later use
                     * branch for 1, 2, or 3 part lines
1944: AE 30 1A              LDX   :DXHi        ; check out hi byte of delta X
1947: D0 0E                 BNE   :3Part       ; it's 3 parts if non-zero
1949: 18                    CLC                ; add left index to delta X
194A: 6D 2F 1A              ADC   :DXLo        ; remember, hi DX is 0 if we
                                               ; ... got here
194D: B0 08                 BCS   :3Part       ; if > 255, it's 3 parts
194F: C9 08                 CMP   #8           ; if sum is < 8, it's one part
1951: 90 6C                 BCC   :1Part       ; branch if so
1953: C9 10                 CMP   #16          ; if sum is < 15, it's two part
```

```
1955: 90 57              BCC   :2Part      ; branch if so
                                           ; otherwise, we've got 3 parts


                   **  deal with a three-part line
                   :3Part

                   * do the left part of the line
1957: 20 EF 19           JSR   :DoLftPrt

                   * do the middle part -- figure # of bytes, then do 'em
                   * # of bytes = (length - (left length + right length) ) / 8
                   * = (delta X + 1 - (8 - left index + right index + 1) ) / 8
                   * = (delta X + 1 - 8 + left index - right index - 1 ) / 8
                   * = ( (delta X + left index) - (right index + 8) ) / 8
195A: 18                 CLC               ; delta X + left index
195B: AD 2F 1A           LDA   :DXLo       ; result in X and Y registers
195E: 6D 31 1A           ADC   :LftNdx
1961: AA                 TAX
1962: AD 30 1A           LDA   :DXHi
1965: 69 00              ADC   #0
1967: A8                 TAY

1968: AD 32 1A           LDA   :RitNdx     ; right index + 8
196B: 69 08              ADC   #8          ; result put in :MidCont
196D: 8D 33 1A           STA   :MidCont

1970: 38                 SEC               ; first group minus second
1971: 8A                 TXA               ; result left in :MidCont
1972: ED 33 1A           SBC   :MidCont    ; ... and A register
1975: 8D 33 1A           STA   :MidCont
1978: 98                 TYA
1979: E9 00              SBC   #0

197B: 4A                 LSR               ; divide it all by 8
197C: 6E 33 1A           ROR   :MidCont
197F: 4A                 LSR
1980: 6E 33 1A           ROR   :MidCont
1983: 4A                 LSR
1984: 6E 33 1A           ROR   :MidCont    ; it now holds # of middle bytes

                   * the middle drawing loop
1987: AC 33 1A     :MidSet LDY :MidCont    ; it'll count the bytes
198A: A9 FF              LDA   #$FF        ; start with a byte full of 1's
198C: 2C F9 1C           BIT   OnOrOff     ; are we turning on or off
198F: 30 02              BMI   :MidStor    ; branch for on

1991: 49 FF        :MidOff EOR #$FF        ; turn byte into 0's for off

1993: 20 BB 1C     :MidStor JSR VDCMemPoke ; store byte
                                           ; storage address updated by VDC

1996: 88           :MidTst DEY             ; down the counter
1997: D0 FA              BNE   :MidStor    ; loop until done

                   * draw the right part of the line
                   * adjust byte pointer for right part of line
1999: 38                 SEC               ; add in the middle part's
199A: AD F4 1C           LDA   BtByLo      ; ... byte count plus 1
199D: 6D 33 1A           ADC   :MidCont
19A0: 8D F4 1C           STA   BtByLo
19A3: 90 03              BCC   :3Rit
19A5: EE F5 1C           INC   BtByHi

                   * go draw that right part and get out
```

```
19A8: 20 0F 1A   :3Rit    JSR   :DoRitPrt   ; draw the right part of line
                          BRA   :Bye        ; and get out
19AB: B8                  CLV
19AC: 50 3B               BVC   :Bye
                          <<<


                 ** deal with a two-part line
                 :2Part

                 * do the left part
19AE: 20 EF 19            JSR   :DoLftPrt

                 * adjust byte pointer for right part of line
19B1: EE F4 1C            INC   BtByLo
19B4: 90 03               BCC   :2Rit
19B6: EE F5 1C            INC   BtByHi

                 * do the right part and get out
19B9: 20 0F 1A   :2Rit    JSR   :DoRitPrt
                          BRA   :Bye
19BC: B8                  CLV
19BD: 50 2A               BVC   :Bye
                          <<<


                 ** deal with a one-part line
                 :1Part

                 * do some fancy masking
19BF: AE 31 1A            LDX   :LftNdx     ; get the left OR mask
19C2: BD 35 1A            LDA   :LfOrMsk,X
19C5: 8D 34 1A            STA   :OneMask
19C8: AE 32 1A            LDX   :RitNdx     ; get the right OR mask
19CB: BD 3E 1A            LDA   :RtOrMsk,X
19CE: 2D 34 1A            AND   :OneMask    ; get their 1's intersection

19D1: 20 71 1C            JSR   GetTargByt  ; get the target byte
19D4: 2C F9 1C            BIT   OnOrOff     ; do we turn bits on or off ?
19D7: 10 05               BPL   :OneOff     ; branch for off

19D9: 0D F8 1C   :OneOn   ORA   TargByt     ; turn appropriate bits on
19DC: D0 05               BNE   :WrtOne     ; branch always

19DE: 49 FF      :OneOff  EOR   #$FF        ; invert the mask
19E0: 2D F8 1C            AND   TargByt     ; turn appropriate bits off

                 * draw the sucker
19E3: 8D F8 1C   :WrtOne  STA   TargByt     ; store the result
19E6: 20 75 1C            JSR   PutTargByt  ; store the left byte


                 * restore some registers and leave
19E9: 68         :Bye     PLA               ; restore registers
19EA: A8                  TAY
19EB: 68                  PLA
19EC: AA                  TAX
19ED: 68                  PLA

19EE: 60                  RTS               ; return from DoHorz


                 *------------------- local subroutines for DoHorz

                 *------------------- :DoLftPrt
```

```
                * do the left part of a horizontal line

                :DoLftPrt
19EF: AE 31 1A            LDX   :LftNdx     ; get the already-calc'd index
19F2: 20 71 1C            JSR   GetTargByt  ; get the target byte
19F5: 2C F9 1C            BIT   OnOrOff     ; do we turn bits on or off ?
19F8: 10 08               BPL   :LeftOff    ; branch for off

19FA: BD 35 1A  :LeftOn   LDA   :LfOrMsk,X  ; get the mask
19FD: 0D F8 1C            ORA   TargByt     ; turn appropriate bits on
1A00: D0 06               BNE   :WrtLft     ; branch always

1A02: BD 3D 1A  :LeftOff  LDA   :LfNdMsk,X   ; get the mask
1A05: 2D F8 1C            AND   TargByt       ; turn appropriate bits off

1A08: 8D F8 1C  :WrtLft   STA   TargByt     ; store the result
1A0B: 20 75 1C            JSR   PutTargByt  ; store the left byte
1A0E: 60                  RTS               ; return from DoLftPrt


                *------------------- :DoRitPrt

                * do the right part of a horizontal line

                :DoRitPrt
1A0F: AE 32 1A            LDX   :RitNdx     ; get the already-calc'd index
1A12: 20 71 1C            JSR   GetTargByt  ; get the target byte
1A15: 2C F9 1C            BIT   OnOrOff     ; do we turn bits on or off ?
1A18: 10 08               BPL   :RiteOff    ; branch for off

1A1A: BD 3E 1A  :RiteOn   LDA   :RtOrMsk,X  ; get the mask
1A1D: 0D F8 1C            ORA   TargByt     ; turn appropriate bits on
1A20: D0 06               BNE   :WrtRit     ; branch always

1A22: BD 36 1A  :RiteOff  LDA   :RtNdMsk,X   ; get the mask
1A25: 2D F8 1C            AND   TargByt       ; turn appropriate bits off

1A28: 8D F8 1C  :WrtRit   STA   TargByt     ; store the result
1A2B: 20 75 1C            JSR   PutTargByt  ; store the right byte
1A2E: 60                  RTS               ; return from DoRitPrt


                *------------ local variables & constants for DoHorz

1A2F: 00        :DXLo     DS    1           ; horizontal length less 1
1A30: 00        :DXHi     DS    1
1A31: 00        :LftNdx   DS    1           ; index into byte from left
1A32: 00        :RitNdx   DS    1           ; index into byte from right
1A33: 00        :MidCont  DS    1           ; bytes worth of middle for
                                            ; ... a 3-part line
1A34: 00        :OneMask  DS    1           ; mask for a 1-part line

1A35: FF        :LfOrMsk  DFB   %11111111   ; ORing masks, left-indexed bytes
1A36: 7F        :RtNdMsk  DFB   %01111111   ; ANDing masks, rite-indexed byts
1A37: 3F                  DFB   %00111111
1A38: 1F                  DFB   %00011111
1A39: 0F                  DFB   %00001111
1A3A: 07                  DFB   %00000111
1A3B: 03                  DFB   %00000011
1A3C: 01                  DFB   %00000001
1A3D: 00        :LfNdMsk  DFB   %00000000   ; ANDing masks, left-indexed byts
1A3E: 80        :RtOrMsk  DFB   %10000000   ; ORing masks, rite-indexed bytes
1A3F: C0                  DFB   %11000000
1A40: E0                  DFB   %11100000
```

```
1A41: F0                DFB   %11110000
1A42: F8                DFB   %11111000
1A43: FC                DFB   %11111100
1A44: FE                DFB   %11111110
1A45: FF                DFB   %11111111


      *------------------- DoBres ----------------------------*

      * Draws all kinds of lines on the 80-column screen

      * Bresenham's line algorithm idea
      * With slight touches by S. Krute
      * See text for insane pseudo-code

      * Upon entry, first point is leftmost, second point is
      * rightmost

      DoBres
      * save some registers
1A46: 48                PHA
1A47: 8A                TXA
1A48: 48                PHA
1A49: 98                TYA
1A4A: 48                PHA

      * figure deltas (horizontal and vertical changes)
      :FigDelts
1A4B: 38                SEC                  ; figure raw delta X
1A4C: AD FD 1C          LDA   Hrz1Lo
1A4F: ED FA 1C          SBC   Hrz0Lo
1A52: 8D C3 1B          STA   :RawDX         ; store raw delta X lo byte
1A55: AD FE 1C          LDA   Hrz1Hi
1A58: ED FB 1C          SBC   Hrz0Hi
1A5B: 8D C4 1B          STA   :RawDX+1       ; store raw delta X hi byte

1A5E: 38                SEC                  ; figure raw delta Y
1A5F: AD FF 1C          LDA   Vrt1
1A62: ED FC 1C          SBC   Vrt0
1A65: 8D C5 1B          STA   :RawDY         ; store raw delta Y

                                             ; figure absolute delta Y
1A68: B0 04             BCS   :StorAbs       ; positive already, so branch
1A6A: 49 FF             EOR   #$FF           ; negative, so positivize it
1A6C: 69 01             ADC   #1             ; remember, carry's clear
1A6E: 8D C6 1B :StorAbs STA   :AbsDY         ; store absolute delta Y

      * set line type :
      * steep or shallow slope
      * rising or falling on screen
1A71: 6E C7 1B          ROR   :LinTyp        ; rising or falling flagged
                                             ; ... by bit 6 :  0-rise
                                             ; ... 1-fall

1A74: AE C4 1B          LDX   :RawDX+1       ; check hi byte of delta X
1A77: F0 03             BEQ   :NxtTst        ; check lo byte if hi clear
1A79: 18                CLC                  ; if non-zero, bigger than
1A7A: 90 03             BCC   :SetSlop       ; ... absolute delta Y,
                                             ; so set flag for shallow
1A7C: CD C3 1B :NxtTst  CMP   :RawDX         ; check lo byte vs. delta Y

1A7F: 6E C7 1B :SetSlop ROR   :LinTyp        ; steep or shallow flagged
                                             ; ... by bit 7 : 0-shallow
                                             ; ... 1-steep
```

```
* initialize increments, errometer, counter
1A82: A9 00          LDA   #0          ; initialize some high bytes
1A84: 8D C9 1B       STA   :IncOne+1
1A87: 8D CF 1B       STA   :Counter+1

1A8A: 2C C7 1B       BIT   :LinTyp      ; case out on line type
1A8D: 10 37          BPL   :InitShallow; shallow types, so branch

* initialize for a steep line
:InitSteep
* set :IncOne
1A8F: AD C3 1B       LDA   :RawDX      ; :IncOne = 2 * :RawDX
1A92: 0A             ASL
1A93: 8D C8 1B       STA   :IncOne
1A96: AA             TAX               ; store a copy
1A97: AD C4 1B       LDA   :RawDX+1
1A9A: 2A             ROL
1A9B: 8D C9 1B       STA   :IncOne+1
1A9E: A8             TAY               ; store a copy

* set :Erometr
1A9F: 38             SEC               ; :Erometr = (2 * :RawDX) - :AbsD
Y
1AA0: 8A             TXA               ;          = :IncOne - :AbsDY
1AA1: ED C6 1B       SBC   :AbsDY      ; that's why we stored the copies
1AA4: 8D CC 1B       STA   :Erometr
1AA7: AA             TAX               ; copy storage again
1AA8: 98             TYA
1AA9: E9 00          SBC   #0
1AAB: 8D CD 1B       STA   :Erometr+1
1AAE: A8             TAY               ; copy storage

* set :IncTwo
1AAF: 38             SEC               ; :IncTwo = 2 * (:RawDX - :AbsDY)
1AB0: 8A             TXA               ;         = :Erometr - :AbsDY
1AB1: ED C6 1B       SBC   :AbsDY
1AB4: 8D CA 1B       STA   :IncTwo
1AB7: 98             TYA
1AB8: E9 00          SBC   #0
1ABA: 8D CB 1B       STA   :IncTwo+1

* set :Counter
1ABD: AE C6 1B       LDX   :AbsDY      ; :Counter = :AbsDY + 1
1AC0: E8             INX
1AC1: 8E CE 1B       STX   :Counter
1AC4: D0 3F          BNE   :DrawPrep   ; prepare for drawing (always)

* initialize for a shallow line
:InitShallow
* set :IncOne
1AC6: AD C6 1B       LDA   :AbsDY      ; :IncOne = 2 * :AbsDY
1AC9: 0A             ASL
1ACA: 8D C8 1B       STA   :IncOne
1ACD: 2E C9 1B       ROL   :IncOne+1

* set :Erometr
1AD0: 38             SEC               ; :Erometr = (2 * :AbsDY) - :RawD
X
1AD1: ED C3 1B       SBC   :RawDX      ;          = :IncOne - :RawDX
1AD4: 8D CC 1B       STA   :Erometr
1AD7: AA             TAX
1AD8: AD C9 1B       LDA   :IncOne+1
```

```
1ADB: ED C4 1B            SBC   :RawDX+1
1ADE: 8D CD 1B            STA   :Erometr+1
1AE1: A8                  TAY

                 * set :IncTwo
1AE2: 38                  SEC                 ; :IncTwo = 2 * (:AbsDY - :RawDX)
1AE3: 8A                  TXA                 ;         = :Erometr - :RawDX
1AE4: ED C3 1B            SBC   :RawDX
1AE7: 8D CA 1B            STA   :IncTwo
1AEA: 98                  TYA
1AEB: ED C4 1B            SBC   :RawDX+1
1AEE: 8D CB 1B            STA   :IncTwo+1

                 * set :Counter
1AF1: AD C3 1B            LDA   :RawDX        ; :Counter = :RawDX + 1
1AF4: 8D CE 1B            STA   :Counter
1AF7: AD C4 1B            LDA   :RawDX+1
1AFA: 8D CF 1B            STA   :Counter+1
1AFD: EE CE 1B            INC   :Counter
1B00: D0 03               BNE   :DrawPrep
1B02: EE CF 1B            INC   :Counter+1

                 * prepare for drawing
                 :DrawPrep
                 * figure vital point plotting information
1B05: 20 17 1C            JSR   FigPoint

                 * see where starting pixel is in its byte
1B08: AD FA 1C            LDA   Hrz0Lo        ; take horizontal position
1B0B: 29 07               AND   #7            ; ... mod 8
1B0D: AA                  TAX                 ; store it in X to figure out
                                              ; ... where pixel is in byte

                 * enter at the bottom of the drawing loop
                          BRA   :DrawBtm
1B0E: B8                  CLV
1B0F: 50 5F               BVC   :DrawBtm
                          <<<

                 * top of drawing loop
                 :LupTop
                 * branch for steep or shallow line
1B11: 2C C7 1B            BIT   :LinTyp       ; steep or shallow ?
1B14: 30 05               BMI   :Steep

                 * for shallow line, move to the right
1B16: 20 95 1B   :Shallow JSR   :GoRite       ; move to the right
1B19: D0 0E               BNE   :ErrTest      ; branch always

                 * for steep line : branch for rising or falling
1B1B: 2C C7 1B   :Steep   BIT   :LinTyp       ; differences for steepitude
1B1E: 70 06               BVS   :StpFal

                 * for steep rising line, move up
1B20: 20 A5 1B   :StpRis  JSR   :GoUp         ; adjust byte pointer up
                          BRA   :ErrTest      ; branch always
1B23: B8                  CLV
1B24: 50 03               BVC   :ErrTest
                          <<<

                 * for steep falling line, move down
1B26: 20 B4 1B   :StpFal  JSR   :GoDown       ; adjust byte pointer down

                 * adjust the :Erometr according to its current value
1B29: 2C CD 1B   :ErrTest BIT   :Erometr+1    ; is :Erometr positive ?
```

```
1B2C: 10 16     >570          BPL   :ItsPos     ; yes, so branch
                >571
                >572  * :Erometr is negative
1B2E: 18        >573  :ItsNeg CLC               ; add :IncOne to a negative
1B2F: AD CC 1B  >574          LDA   :Erometr    ; ... :Erometr
1B32: 6D C8 1B  >575          ADC   :IncOne
1B35: 8D CC 1B  >576          STA   :Erometr
1B38: AD CD 1B  >577          LDA   :Erometr+1
1B3B: 6D C9 1B  >578          ADC   :IncOne+1
1B3E: 8D CD 1B  >579          STA   :Erometr+1
                >580          BRA   :DrawBtm    ; go to bottom of drawing loop
1B41: B8        >580          CLV
1B42: 50 2C     >580          BVC   :DrawBtm
                >580          <<<
                >581
                >582  * :Erometr is positive
1B44: 18        >583  :ItsPos CLC               ; add :IncTwo to a positive
1B45: AD CC 1B  >584          LDA   :Erometr    ; ... :Erometr
1B48: 6D CA 1B  >585          ADC   :IncTwo
1B4B: 8D CC 1B  >586          STA   :Erometr
1B4E: AD CD 1B  >587          LDA   :Erometr+1
1B51: 6D CB 1B  >588          ADC   :IncTwo+1
1B54: 8D CD 1B  >589          STA   :Erometr+1
                >590
                >591  * for was-positive :Erometr, branch on steep or shallow
1B57: 2C C7 1B  >592          BIT   :LinTyp     ; are we steep or shallow ?
1B5A: 30 11     >593          BMI   :Steep2     ; branch appropriately
                >594
                >595  * for was-positive :Erometr and shallow line, branch on
                >596  * ... rising or falling
1B5C: 2C C7 1B  >597  :Shalo2 BIT   :LinTyp     ; are we rising or falling ?
1B5F: 70 06     >598          BVS   :ShlFal     ; branch appropriately
                >599
                >600  * for was-positive :Erometr and shallow rising line,
                >601  * ... move up a pixel
1B61: 20 A5 1B  >602  :ShlRis JSR   :GoUp       ; move up a pixel
                >603          BRA   :DrawBtm    ; branch always
1B64: B8        >603          CLV
1B65: 50 09     >603          BVC   :DrawBtm
                >603          <<<
                >604
                >605  * for was-positive :Erometr and shallow falling line,
                >606  * ... move down a pixel
1B67: 20 B4 1B  >607  :ShlFal JSR   :GoDown     ; move down a pixel
                >608          BRA   :DrawBtm    ; branch always
1B6A: B8        >608          CLV
1B6B: 50 03     >608          BVC   :DrawBtm
                >608          <<<
                >609
                >610  * for was-positive :Erometr and steep line,
                >611  * ... move right a pixel
1B6D: 20 95 1B  >612  :Steep2 JSR   :GoRite     ; move right a pixel
                >613
                >614
                >615  * bottom of drawing loop
                >616  :DrawBtm
                >617  * draw a point
1B70: 8E F7 1C  >618          STX   BitPos      ; set pixel bit position
1B73: 20 67 1C  >619          JSR   PlotIt      ; plot the point
                >620
                >621  * see if we're done looping
                >622  :LupTst
1B76: CE CE 1B  >623          DEC   :Counter    ; counter = counter - 1
1B79: AD CE 1B  >624          LDA   :Counter    ; down the lo byte, then check
1B7C: C9 FF     >625          CMP   #$FF        ; do we have to do hi byte ?
```

```
1B7E: D0 06              BNE   :ZerTst     ; no, so check zeroness
1B80: CE CF 1B           DEC   :Counter+1  ;
                         BRA   :LupTop
1B83: B8                 CLV
1B84: 50 8B              BVC   :LupTop
                         <<<
1B86: C9 00     :ZerTst  CMP   #0          ; is lo byte zero ?
1B88: D0 87              BNE   :LupTop     ; no, so branch to loop top
1B8A: AD CF 1B           LDA   :Counter+1  ; is hi byte zero ?
1B8D: D0 82              BNE   :LupTop     ; no, so branch to loop top
                                           ; if both bytes zero, we gone

                * restore some registers and leave
1B8F: 68                 PLA               ; restore registers
1B90: A8                 TAY
1B91: 68                 PLA
1B92: AA                 TAX
1B93: 68                 PLA

1B94: 60                 RTS               ; return from DoBres


                *------------------- local subroutines for DoBres

                *------------------- :GoRite

                * move right a pixel

1B95: E8        :GoRite  INX               ; up the pixel index
1B96: E0 08              CPX   #8          ; into next byte ?
1B98: 90 0A              BCC   :GRDun      ; if not, git on back
1B9A: A2 00              LDX   #0          ; yes, so reset bit counter
1B9C: EE F4 1C           INC   BtByLo      ; ... and move byte pointer
1B9F: D0 03              BNE   :GRDun      ; ... one to the right
1BA1: EE F5 1C           INC   BtByHi
1BA4: 60        :GRDun   RTS               ; return from :GoRite


                *------------------- :GoUp

                * move up a pixel

1BA5: 38        :GoUp    SEC               ; just subtract a line's
1BA6: AD F4 1C           LDA   BtByLo      ; ... worth of bytes
1BA9: E9 50              SBC   #BytPerLin
1BAB: 8D F4 1C           STA   BtByLo
1BAE: B0 03              BCS   :GUDun
1BB0: CE F5 1C           DEC   BtByHi
1BB3: 60        :GUDun   RTS               ; return from :GoUp


                *------------------- :GoDown

                * move down a pixel
1BB4: 18        :GoDown  CLC               ; just add a line's
1BB5: AD F4 1C           LDA   BtByLo      ; ... worth of bytes
1BB8: 69 50              ADC   #BytPerLin
1BBA: 8D F4 1C           STA   BtByLo
1BBD: 90 03              BCC   :GDDun
1BBF: EE F5 1C           INC   BtByHi
1BC2: 60        :GDDun   RTS               ; return from :GoDown


                *------------ local variables for DoBres

```

```
1BC3: 00 00    :RawDX   DS    2           ; horizontal length less one
1BC5: 00       :RawDY   DS    1
1BC6: 00       :AbsDY   DS    1           ; absolute value of vertical
                                          ; ... length less one
1BC7: 00       :LinTyp  DS    1           ; type of line
1BC8: 00 00    :IncOne  DS    2           ; increment for :Erometr
1BCA: 00 00    :IncTwo  DS    2           ; increment for :Erometr
1BCC: 00 00    :Erometr DS    2           ; tracks relation between
                                          ; ... real and ideal line
1BCE: 00 00    :Counter DS    2           ; tracks horizontal or vertical
                                          ; ... pixels needed to complete
                                          ; ... the line
                        TTL   "GRAFIX 80 5.S"
               *---------------- Here Comes Another Source File ---------*

                        PUT   "GRAFIX 80 5.S"


               *--------------------- ClrTx80 -------------------------*

               * Clear the 80-column text screen

               ClrTx80
               * save some registers
1BD0: 48                PHA
1BD1: 8A                TXA
1BD2: 48                PHA
1BD3: 98                TYA
1BD4: 48                PHA

               * make sure we're in 80 column screen mode
1BD5: A5 D7             LDA   Mode        ; see if we're in 40 or 80 mode
1BD7: 48                PHA               ; park mode on stack
1BD8: 30 03             BMI   :Clear      ; branches if already in 80 mode

1BDA: 20 5F FF          JSR   Swapper     ; change to 80 mode from 40

               * clear the screen
1BDD: A9 93    :Clear   LDA   #ClearScrn  ; get clear screen character
1BDF: 20 D2 FF          JSR   BSOut       ; send it out

               * return to entry screen mode
1BE2: 68                PLA               ; get mode back from stack
1BE3: 30 03             BMI   :Bye        ; branches if we were in 80

1BE5: 20 5F FF          JSR   Swapper     ; change back to 40 mode from 80

               * restore some registers and leave
1BE8: 68       :Bye     PLA
1BE9: A8                TAY
1BEA: 68                PLA
1BEB: AA                TAX
1BEC: 68                PLA

1BED: 60                RTS               ; return from ClrTx80


               *--------------------- ClrGr80 -------------------------*

               * Clear the 80-column graphics screen

               ClrGr80
               * save some registers
1BEE: 48                PHA
1BEF: 8A                TXA
```

```
1BF0: 48                PHA
1BF1: 98                TYA
1BF2: 48                PHA

                 * prepare for page clearing loop
1BF3: A0 3E             LDY   #>BitMapSiz; we'll clear that many pages
                                         ; ... plus 1 of memory
1BF5: 98                TYA              ; starting with that page

                 * body of the page clearing loop
1BF6: A2 12      :LupOne LDX  #AdrHiReg  ; set VDC update location
1BF8: 20 BD 1C          JSR   VDCRegPoke ; ... to page whose # is
1BFB: E8                INX              ; ... in the A register
1BFC: A9 00             LDA   #0
1BFE: 20 BD 1C          JSR   VDCRegPoke

1C01: A2 1F             LDX   #DataReg   ; store 0 as the data to
1C03: 20 BD 1C          JSR   VDCRegPoke ; ... be written

1C06: A2 1E             LDX   #BytCntReg ; tell the chip to store
1C08: 20 BD 1C          JSR   VDCRegPoke ; 256 copies of it

                 * loop continuation test
1C0B: 88                DEY              ; count another memory page
                                         ; ... done
1C0C: 30 03             BMI   :Bye       ; if we're all done, branch
1C0E: 98                TYA              ; not all done, so move page
1C0F: 10 E5             BPL   :LupOne    ; ... down and loop-de-loop

                 * restore some registers and leave
1C11: 68         :Bye   PLA
1C12: A8                TAY
1C13: 68                PLA
1C14: AA                TAX
1C15: 68                PLA

1C16: 60                RTS              ; return from ClrGr80


                 *------------------------ FigPoint ----------------------
*

                 * Figures out vital information for point plotting
                 * Sets up a number of variables

                 * On entry : horizontal coordinate is in Hrz0Lo and Hrz0Hi
                 *            vertical coordinate is in Vrt0
                 * On exit  : pixel's position in byte (7..0) is
                 *            ... stored in BitPos
                 *            pixel's byte's address is in BtByLo - BtByHi
                 *            pixel's byte's row offset (0..79) is in RowOff

                 FigPoint
                 * save some registers
1C17: 48                PHA
1C18: 8A                TXA
1C19: 48                PHA
1C1A: 98                TYA
1C1B: 48                PHA

                 * prepare to use vertical coordinate
1C1C: AD FC 1C          LDA   Vrt0       ; get vertical coordinate
1C1F: 48                PHA              ; save a copy

                 * figure address of first byte in point's row
```

```
                  *    get the low byte
1C20: 29 0F              AND    #%00001111 ; take vert. coord mod 16
1C22: AA                 TAX               ; we'll use it to index
1C23: BD 34 1D           LDA    RwLoBs,X   ; get low byte of address
1C26: 8D F4 1C           STA    BtByLo     ; store low byte of address

                  *   get the high byte
1C29: 68                 PLA               ; A holds vertical coord again
1C2A: 4A                 LSR               ; divide it by 16
1C2B: 4A                 LSR               ; ends up in range 0..12
1C2C: 4A                 LSR
1C2D: 4A                 LSR
1C2E: 18                 CLC               ; prepare to add
1C2F: A8                 TAY               ; we want to index
1C30: B9 44 1D           LDA    RHiBMst,Y  ; get most of hi byte
1C33: 7D 51 1D           ADC    RHiBAdj,X  ; add in adjustment
1C36: 8D F5 1C           STA    BtByHi     ; store hi byte of address

                  * adjust address for horizontal coordinate
1C39: AD FB 1C           LDA    Hrz0Hi     ; get the horizontal hi byte
1C3C: 8D EC 1C           STA    ScratPad
1C3F: AD FA 1C           LDA    Hrz0Lo     ; then the lo byte
1C42: 48                 PHA               ; save copy for later
1C43: 4E EC 1C           LSR    ScratPad   ; now, divide horizontal
1C46: 6A                 ROR               ; ... coordinate by eight
1C47: 4E EC 1C           LSR    ScratPad   ; this gives us the
1C4A: 6A                 ROR               ; ... bit's byte's row
1C4B: 4A                 LSR               ; ... offset (0..79)
1C4C: 8D F6 1C           STA    RowOff     ; store that row offset

1C4F: 18                 CLC               ; add it to address
1C50: 6D F4 1C           ADC    BtByLo
1C53: 8D F4 1C           STA    BtByLo
1C56: 90 03              BCC    :LocateBit
1C58: EE F5 1C           INC    BtByHi

                  :LocateBit
1C5B: 68                 PLA               ; fetch low byte of horizontal
                                           ; ... cooordinate
1C5C: 29 07              AND    #7         ; horz. coord. mod 8
1C5E: 8D F7 1C           STA    BitPos     ; store it

                  * restore some registers and leave
1C61: 68                 PLA
1C62: A8                 TAY
1C63: 68                 PLA
1C64: AA                 TAX
1C65: 68                 PLA

1C66: 60                 RTS               ; return from FigPoint


                  *------------------ PlotIt -----------------------------*

                  * plot a pixel on the 80-column screen

                  PlotIt
1C67: 20 71 1C           JSR    GetTargByt ; get the target pixel's byte
1C6A: 20 9E 1C           JSR    PixelPop   ; set the target pixel on or off
1C6D: 20 75 1C           JSR    PutTargByt ; put target pixel's byte back
1C70: 60                 RTS               ; return from PlotIt


                  *---------------- GetTargByt & PutTargByt ---------------*

                  * two routines: get the target pixel's byte or put it back
```

```
GetTargByt
1C71: 08                    PHP                  ; save Status
1C72: 18                    CLC                  ; clear Carry for Get
1C73: 90 02                 BCC    MorSav        ; skip next entry

            PutTargByt
1C75: 08                    PHP                  ; save Status
1C76: 38                    SEC                  ; set Carry for Put

1C77: 48        MorSav      PHA                  ; save registers
1C78: 8A                    TXA
1C79: 48                    PHA

1C7A: A2 12                 LDX    #AdrHiReg     ; set VDC update location to
1C7C: AD F5 1C              LDA    BtByHi        ; ... target byte
1C7F: 20 BD 1C              JSR    VDCRegPoke
1C82: E8                    INX
1C83: AD F4 1C              LDA    BtByLo
1C86: 20 BD 1C              JSR    VDCRegPoke

1C89: B0 08                 BCS    :PutIt        ; branch for get or put

1C8B: 20 C9 1C  :GetIt      JSR    VDCMemPeek    ; get the target byte
1C8E: 8D F8 1C              STA    TargByt       ; and store it
1C91: 90 06                 BCC    :GPBye        ; always

1C93: AD F8 1C  :PutIt      LDA    TargByt       ; get the target byte
1C96: 20 BB 1C              JSR    VDCMemPoke    ; and store it

                * restore some registers and leave
1C99: 68        :GPBye      PLA
1C9A: AA                    TAX
1C9B: 68                    PLA
1C9C: 28                    PLP                  ; even the Status register

1C9D: 60                    RTS                  ; return from GetTargByt


                *------------------- PixelPop ---------------------------*

                * set the target pixel on or off

                PixelPop
                * save some registers
1C9E: 48                    PHA
1C9F: 8A                    TXA
1CA0: 48                    PHA

1CA1: AD F8 1C              LDA    TargByt       ; get target byte
1CA4: AE F7 1C              LDX    BitPos        ; get bit index
1CA7: 2C F9 1C              BIT    OnOrOff       ; check for pixel on or off
1CAA: 10 05                 BPL    :PixOff       ; branch accordingly

1CAC: 1D 61 1D  :PixOn      ORA    OnPixel,X     ; force bit to 1 for pixel on
1CAF: D0 03                 BNE    :Stikit       ; always branches

1CB1: 3D 69 1D  :PixOff     AND    OffPixel,X    ; force bit to 0 for pixel off

1CB4: 8D F8 1C  :Stikit     STA    TargByt       ; save fixed target byte

                * restore some registers and leave
1CB7: 68                    PLA
1CB8: AA                    TAX
1CB9: 68                    PLA
```

```
1CBA: 60                     RTS              ; return from PixelPop


                     *------------------ 8563 Access Routines ---------------*

                     * Four routines to access the 8563 80-column VDC
                     *   ( Video Display Chip)

                     * VDCMemPoke puts a byte into 8563's video RAM
                     *    A holds the byte, VDC registers 18 & 19 point to
                     *    the RAM location

                     * VDCRegPoke puts a byte into an 8563 chip register
                     *    A holds the byte, X holds the register #

                     * VDCMemPeek gets a byte from 8563's video RAM
                     *    VDC registers 18 & 19 point to the RAM location,
                     *    A gets the byte

                     * VDCRegPeek gets a byte from an 8563 chip register
                     *    X holds the register #, A gets the byte

                     VDCMemPoke
1CBB: A2 1F                  LDX   #DataReg   ; the read/write data register
                     VDCRegPoke
1CBD: 8E 00 D6               STX   VDCAdr     ; store the register number
1CC0: 2C 00 D6       PokeLup BIT   VDCAdr     ; test VDC digestion
1CC3: 10 FB                  BPL   PokeLup    ; wait 'til it's been eaten
1CC5: 8D 01 D6               STA   VDCDat     ; store the data
1CC8: 60                     RTS              ; and return

                     VDCMemPeek
1CC9: A2 1F                  LDX   #DataReg   ; the read/write data register
                     VDCRegPeek
1CCB: 8E 00 D6               STX   VDCAdr     ; store the register number
1CCE: 2C 00 D6       PeekLup BIT   VDCAdr     ; test VDC digestion
1CD1: 10 FB                  BPL   PeekLup    ; wait 'til it's been eaten
1CD3: AD 01 D6               LDA   VDCDat     ; fetch the data
1CD6: 60                     RTS              ; and return


                     *--------------------- OptNxtByt ----------------------*

                     * fetch an optional byte-sized value from a BASIC input
                     * ... line -- if no value there, defaults to 0

                     * exits with X holding the value and Carry indicating
                     * ... default value (clear) or fetched (set)

                     * as with ROM parsing routines, consider registers trashed

                     OptNxtByt
                     * anything around to fetch ?  test 1
1CD7: 20 86 03               JSR   ChrGot     ; anything left to get ?
1CDA: F0 0C                  BEQ   :DfltBye   ; no, so leave with default 0

                     * anything around to fetch ?  test 2
1CDC: 20 06 14               JSR   CommaCruz  ; make sure there's a comma
1CDF: C9 2C                  CMP   #Comma     ; is value left out (indicated
                                              ; ... by a comma as next BASIC
                                              ; ... line element) ?
1CE1: F0 05                  BEQ   :DfltBye   ; yup, so leave with default 0

                     * something around to fetch, so do so and return
1CE3: 20 F4 87               JSR   GetByt     ; no, so fetch the value
```

```
1CE6: 38                      SEC                  ; indicate fetched value
1CE7: 60                      RTS                  ; return from OptNxtByt

            * nothing to fetch, so set default and return
1CE8: A2 00       :DfltBye LDX    #0              ; load up with default value
1CEA: 18                   CLC                    ; indicate default value
1CEB: 60                   RTS                    ; return from OptNxtByt


            *---------------------- Variables -----------------------*

1CEC: 00 00 00    ScratPad DS     8               ; a scratchpad area
1CEF: 00 00 00 00 00

1CF4: 00          BtByLo   DS     1               ; holds address of a bit's
1CF5: 00          BtByHi   DS     1               ; ... byte

1CF6: 00          RowOff   DS     1               ; offset in the row (0..79) of
                                                  ; ... a bit's byte
1CF7: 00          BitPos   DS     1               ; bit's position (0..7) in its
                                                  ; ... byte
1CF8: 00          TargByt  DS     1               ; target byte for graphics work
1CF9: 00          OnOrOff  DS     1               ; whether pixel goes on or off

1CFA: 00          Hrz0Lo   DS     1               ; point 0 coords. for drawing
1CFB: 00          Hrz0Hi   DS     1
1CFC: 00          Vrt0     DS     1

1CFD: 00          Hrz1Lo   DS     1               ; point 1 coords. for drawing
1CFE: 00          Hrz1Hi   DS     1
1CFF: 00          Vrt1     DS     1

1D00: 00 00       RegIEscLk DS    2               ; stores regular IEscLk vector
1D02: 00 00       RegIEscPr DS    2               ; stores regular IEscPr vector
1D04: 00 00       RegIEscEx DS    2               ; stores regular IEscEx vector
1D06: 00          TheCode  DS     1               ; stores entry code


            *--------------------- Tables ----------------------------*

            * our additions to the BASIC 7.0 command set

            OurComsText
1D07: 47 38 30             DCI    'g80box'
1D0A: 42 4F D8
1D0D: 47 38 30             DCI    'g80color'
1D10: 43 4F 4C 4F D2
1D15: 47 38 30             DCI    'g80draw'
1D18: 44 52 41 D7
1D1C: 47 38 30             DCI    'g80graphic'
1D1F: 47 52 41 50 48 49 C3
1D26: 47 38 30             DCI    'g80scat'
1D29: 53 43 41 D4
1D2D: 00                   HEX    00


            * VDC masks for setting text and disabling attributes
            VAndMsks
1D2E: 7F                   DFB    %01111111   ; clears bit 7, setting text
1D2F: BF                   DFB    %10111111   ; clears bit 6, disabling
                                              ; ... attributes

            * VDC masks for setting grafix and enabling attributes
            VOrMsks
1D30: 40                   DFB    %01000000   ; sets bit 6, enabling
                                              ; ... attributes
```

```
1D31: 80                  DFB   %10000000  ; sets bit 7, setting grafix


                 * masks for clearing VDC color byte

                 HueNbMsk
1D32: 0F                  DFB   %00001111
1D33: F0                  DFB   %11110000


                 * table of low bytes of 80-column screen addresses for
                 * leftmost byte in each row ... cycles every 16 rows

                 RwLoBs
1D34: 00 50 A0            HEX   00,50,A0
1D37: F0 40 90            HEX   F0,40,90
1D3A: E0 30 80            HEX   E0,30,80
1D3D: D0 20 70            HEX   D0,20,70
1D40: C0 10 60            HEX   C0,10,60
1D43: B0                  HEX   B0


                 * table holding high bytes of 80-column screen addresses
                 * for leftmost byte in each row -- done for [m6[m6v[m6[m6ry 1
                 * row, starting with row 0

                 RHiBMst
1D44: 00 05 0A            HEX   00,05,0A
1D47: 0F 14 19            HEX   0F,14,19
1D4A: 1E 23 28            HEX   1E,23,28
1D4D: 2D 32 37            HEX   2D,32,37
1D50: 3C                  HEX   3C


                 * table holding adjustments for high bytes of 80-column
                 * screen addresses for leftmost byte in each row --
                 * cycles every 16 rows

                 RHiBAdj
1D51: 00 00 00            HEX   00,00,00
1D54: 00 01 01            HEX   00,01,01
1D57: 01 02 02            HEX   01,02,02
1D5A: 02 03 03            HEX   02,03,03
1D5D: 03 04 04            HEX   03,04,04
1D60: 04                  HEX   04

                 OnPixel
1D61: 80                  DFB   %10000000
1D62: 40                  DFB   %01000000
1D63: 20                  DFB   %00100000
1D64: 10                  DFB   %00010000
1D65: 08                  DFB   %00001000
1D66: 04                  DFB   %00000100
1D67: 02                  DFB   %00000010
1D68: 01                  DFB   %00000001

                 OffPixel
1D69: 7F                  DFB   %01111111
1D6A: BF                  DFB   %10111111
1D6B: DF                  DFB   %11011111
1D6C: EF                  DFB   %11101111
1D6D: F7                  DFB   %11110111
1D6E: FB                  DFB   %11111011
1D6F: FD                  DFB   %11111101
1D70: FE                  DFB   %11111110
```

```
             
             
             * table holding 80-column color nibbles in low bytes
             * each comment includes a decimal equivalent, the color,
             * and the BASIC color number
             
             HuNb80Tb
             * the signals:  ----rgbi  ; red, green, blue, and intensity
1D71: 00              DFB  %00000000  ;  0  black         (1)
1D72: 0F              DFB  %00001111  ; 15  white         (2)
1D73: 08              DFB  %00001000  ;
1D74: 07              DFB  %00000111  ;  7  light cyan    (4)
1D75: 0B              DFB  %00001011  ; 11  light purple  (5)
1D76: 04              DFB  %00000100  ;  4  dark green    (6)
1D77: 02              DFB  %00000010  ;  2  dark blue     (7)
1D78: 0D              DFB  %00001101  ; 13  light yellow  (8)
1D79: 0A              DFB  %00001010  ; 10  dark purple   (9)
1D7A: 0C              DFB  %00001100  ; 12  dark yellow  (10)
1D7B: 09              DFB  %00001001  ;  9  light red    (11)
1D7C: 06              DFB  %00000110  ;  6  dark cyan    (12)
1D7D: 01              DFB  %00000001  ;  1  medium gray  (13)
1D7E: 05              DFB  %00000101  ;  5  light green  (14)
1D7F: 03              DFB  %00000011  ;  3  light blue   (15)
1D80: 0E              DFB  %00001110  ; 14  light gray   (16)

--End assembly, 2689 bytes, Errors: 0
```

```
1000 REM ----- PROGRAM IDENTIFICATION -----
1010 :
1020 REM         G80 TEST SUITE
1030 :
1040 REM   RUNS A SERIES OF PERFORMANCE TESTS
1050 REM   ... ON THE 80-COLUMN GRAPHICS ROUTINES
1060 :
1070 REM   BE SURE TO INSTALL THE 80-COLUMN
1080 REM   ... GRAPHICS ROUTINES BEFORE LOADING
1090 REM   ... AND RUNNING THESE TESTS
1100 :
1110 REM   YOU MIGHT ALSO WANT TO INSTALL THE TEXT
1120 REM   ... SCREEN DUMP ROUTINES TO GET HARDCOPY
1130 REM   ... OF THE RESULTS
1140 :
1150 REM   VERSION :      1.00
1160 REM   TIMESTAMP :    3:35 PM PST    SEPTEMBER 20, 1986
1170 :
1180 REM   PROGRAMMED BY STAN KRUTE
1190 REM   COPYRIGHT (C) 1986 BY STAN KRUTE'S HACKER & NERD
1200 REM                      18617 CAMP CREEK ROAD
1210 REM                      HORNBROOK, CALIFORNIA   96044
1220 REM                      [916] 475-3428
1230 REM   ALL RIGHTS RESERVED
1240 REM   CALL OR WRITE FOR HELP, BUG REPORTS, LICENSING, ETC.
1250 :
1260 :
1270 REM ----- MAIN PROGRAM BLOCK -----
1280 :
1290 GOSUB 1370     :REM   INITIALIZE
1300 GOSUB 1840     :REM   RUN THE TESTS
1310 GOSUB 1950     :REM   REPORT THE RESULTS
1320 END            :REM   THAT'S THAT
1330 :
1340 :
```

Fig. 8-3. Source code for G80 Test Suite.

```
1350 REM ----- INITIALIZE -----
1360 :
1370 PRINT CHR$ (147)                                   :REM   CLEAR SCREEN
1380 PRINT "PLEASE GIVE ME A RANDOM SEED 1-32768 " ;    :REM   SEED PROMPT
1390 INPUT RS                                           :REM   GET RANDOM SEED
1400 :
1410 PRINT CHR$(147)                                    :REM   CLEAR SCREEN
1420 PRINT "PLEASE GIVE ME A SCREEN WIDTH " ;           :REM   WIDTH PROMPT
1430 INPUT SW                                           :REM   GET SCREEN WIDTH
1440 :
1450 PRINT CHR$(147)                               :REM   CLEAR SCREEN
1460 PRINT "INITIALIZING RANDOM ARRAYS " ;         :REM   GIVE SOME FEEDBACK
1470 PRINT "WITH 100 VALUES ..." ;
1480 :
1490 FAST                                          :REM   LET'S MOVE
1500 :
1510 DIM T(99), B(99), L(99), R(99)                :REM   DIMENSE COORDINATE ARAYZ
1520 DIM RS (11)                                   :REM   DIMENSE RESULT ARRAY
1530 N = RND (-RS)                                 :REM   SEED RANDOM NUMBERS
1540 :
1550 FOR N = 1 TO 100                              :REM   FILL ARRAYS
1560 :    T (N-1) = INT ( RND (1) * 200 )          :REM   T () GETS VALUES 0..199
1570 :    B (N-1) = INT ( RND (1) * 200 )          :REM   B () GETS VALUES 0..199
1580 :    L (N-1) = INT ( RND (1) * SW  )          :REM   L () GETS VALUES 0..SW-1
1590 :    R (N-1) = INT ( RND (1) * SW  )          :REM   R () GETS VALUES 0..SW-1
1600 :    IF N/10 <> INT (N/10) THEN 1660          :REM   FEEDBACK EVERY 10 VALUES
1610 :       N$ = STR$ (N)                         :REM      STRINGIZE THE COUNT
1620 :       PRINT N$ ;                            :REM      PRINT THE COUNT
1630 :       FOR P = 1 TO LEN (N$)                 :REM      MOVE CURSOR BACK
1640 :          PRINT CHR$ (157) ;                 :REM         MOVE CURSOR LEFT
1650 :          NEXT P
1660 :    NEXT N
1670 :
1680 RESTORE 1730                                  :REM   PREP FOR DATA READ
1690 FOR N = 0 TO 5                                :REM   READ 6 TEST NAME LABELS
1700 :    READ N$(N)                               :REM      READ A LABEL
1710 :    NEXT N
1720 :
1730 DATA "100 RANDOM DOTS",            "100 RANDOM VERTICAL LINES"
1740 DATA "100 RANDOM HORIZONTAL LINES", "100 RANDOM LINES"
1750 DATA "100 RANDOM OUTLINED BOXES",   "100 RANDOM FILLED BOXES"
1760 :
1770 SLOW                                          :REM   SLOW DOWN
1780 :
1790 RETURN
1800 :
1810 :
1820 REM ----- RUN THE TESTS -----
1830 :
1840 GOSUB 2150     :REM   DRAW 100 RANDOM DOTS
1850 GOSUB 2340     :REM   DRAW 100 RANDOM VERTICAL LINES
1860 GOSUB 2530     :REM   DRAW 100 RANDOM HORIZONTAL LINES
1870 GOSUB 2720     :REM   DRAW 100 RANDOM LINES
1880 GOSUB 2910     :REM   DRAW 100 RANDOM OUTLINED BOXES
1890 GOSUB 3100     :REM   DRAW 100 RANDOM FILLED BOXES
1900 RETURN
1910 :
1920 :
1930 REM ----- REPORT THE RESULTS -----
1940 :
1950 G80GRAPHIC 5,1                          :REM   80-COLUMN SCREEN, CLEARED
1960 :
1970 GOSUB 3290                              :REM   PRINT RESULT HEADINGS
1980 :
1990 FOR T = 0 TO 5                          :REM   FIVE TESTS ( 2 VARIANTS EACH )
```

```
2000 :    CHAR ,  0, T*3 + 4, N$(T)                   :REM   PRINT TEST NAME
2010 :    CHAR , 50, T*3 + 4, "SLOW"                  :REM   PRINT VARIANT MODE
2020 :    CHAR , 60, T*3 + 4, STR$ ( RS ( T*2 ))      :REM   PRINT RESULT
2030 :
2040 :    CHAR ,  0, T*3 + 5, N$(T)                   :REM   PRINT TEST NAME
2050 :    CHAR , 50, T*3 + 5, "FAST"                  :REM   PRINT VARIANT MODE
2060 :    CHAR , 60, T*3 + 5, STR$ ( RS ( T*2 + 1 )) :REM   PRINT RESULT
2070 :    NEXT T
2080 :
2090 PRINT                                  :REM   A NICE BLANK LINE
2100 RETURN
2110 :
2120 :
2130 REM ----- DRAW 100 RANDOM DOTS -----
2140 :
2150 G80GRAPHIC 6,1                   :REM   80-COLUMN GRAPHICS, CLEARED
2160 SLOW                             :REM   SET SPEED
2170 :                                :REM   HERE COMES THE TEST
2180 S = TI
2190 FOR N = 0 TO 99 : G80DRAW , L(N), T(N) : NEXT
2200 RS(0) = TI - S                   :REM   STORE RESULT
2210 :
2220 G80GRAPHIC 6,1                   :REM   80-COLUMN GRAPHICS, CLEARED
2230 FAST                             :REM   SET SPEED
2240 :                                :REM   HERE COMES THE TEST
2250 S = TI
2260 FOR N = 0 TO 99 : G80DRAW , L(N), T(N) : NEXT
2270 RS(1) = TI - S                   :REM   STORE RESULT
2280 :
2290 RETURN
2300 :
2310 :
2320 REM ----- DRAW 100 RANDOM VERTICAL LINES -----
2330 :
2340 G80GRAPHIC 6,1                            :REM   80-COLUMN GRAPHICS, CLEARED
2350 SLOW                                      :REM   SET SPEED
2360 :                                         :REM   HERE COMES THE TEST
2370 S = TI
2380 FOR N = 0 TO 99 : G80DRAW , L(N), T(N) TO L(N), B(N) : NEXT
2390 RS(2) = TI - S                            :REM   STORE RESULT
2400 :
2410 G80GRAPHIC 6,1                            :REM   80-COLUMN GRAPHICS, CLEARED
2420 FAST                                      :REM   SET SPEED
2430 :                                         :REM   HERE COMES THE TEST
2440 S = TI
2450 FOR N = 0 TO 99 : G80DRAW , L(N), T(N) TO L(N), B(N) : NEXT
2460 RS(3) = TI - S                            :REM   STORE RESULT
2470 :
2480 RETURN
2490 :
2500 :
2510 REM ----- DRAW 100 RANDOM HORIZONTAL LINES -----
2520 :
2530 G80GRAPHIC 6, 1                           :REM   80-COLUMN GRAPHICS, CLEARED
2540 SLOW                                      :REM   SET SPEED
2550 :                                         :REM   HERE COMES THE TEST
2560 S = TI
2570 FOR N = 0 TO 99 : G80DRAW , L(N), T(N) TO R(N), T(N) : NEXT
2580 RS(4) = TI - S                            :REM   STORE RESULT
2590 :
2600 G80GRAPHIC 6, 1                           :REM   80-COLUMN GRAPHICS, CLEARED
2610 FAST                                      :REM   SET SPEED
2620 :                                         :REM   HERE COMES THE TEST
2630 S = TI
2640 FOR N = 0 TO 99 : G80DRAW , L(N), T(N) TO R(N), T(N) : NEXT
```

```
2650 RS(5) = TI - S                                   :REM   STORE RESULT
2660 :
2670 RETURN
2680 :
2690 :
2700 REM ----- DRAW 100 RANDOM LINES -----
2710 :
2720 G80GRAPHIC 6, 1                                  :REM   80-COLUMN GRAPHICS, CLEARED
2730 SLOW                                             :REM   SET SPEED
2740 :                                                :REM   HERE COMES THE TEST
2750 S = TI
2760 FOR N = 0 TO 99 : G80DRAW , L(N), T(N) TO R(N), B(N) : NEXT
2770 RS(6) = TI - S                                   :REM   STORE RESULT
2780 :
2790 G80GRAPHIC 6, 1                                  :REM   80-COLUMN GRAPHICS, CLEARED
2800 FAST                                             :REM   SET SPEED
2810 :                                                :REM   HERE COMES THE TEST
2820 S = TI
2830 FOR N = 0 TO 99 : G80DRAW , L(N), T(N) TO R(N), B(N) : NEXT
2840 RS(7) = TI - S                                   :REM   STORE RESULT
2850 :
2860 RETURN
2870 :
2880 :
2890 REM ----- DRAW 100 RANDOM OUTLINED BOXES -----
2900 :
2910 G80GRAPHIC 6, 1                                  :REM   80-COLUMN GRAPHICS, CLEARED
2920 SLOW                                             :REM   SET SPEED
2930 :                                                :REM   HERE COMES THE TEST
2940 S = TI
2950 FOR N = 0 TO 99 : G80BOX , L(N), T(N), R(N), B(N) : NEXT
2960 RS(8) = TI - S                                   :REM   STORE RESULT
2970 :
2980 G80GRAPHIC 6, 1                                  :REM   80-COLUMN GRAPHICS, CLEARED
2990 FAST                                             :REM   SET SPEED
3000 :                                                :REM   HERE COMES THE TEST
3010 S = TI
3020 FOR N = 0 TO 99 : G80BOX , L(N), T(N), R(N), B(N) : NEXT
3030 RS(9) = TI - S                                   :REM   STORE RESULT
3040 :
3050 RETURN
3060 :
3070 :
3080 REM ----- DRAW 100 RANDOM FILLED BOXES -----
3090 :
3100 G80GRAPHIC 6, 1                                  :REM   80-COLUMN GRAPHICS, CLEARED
3110 SLOW                                             :REM   SET SPEED
3120 :                                                :REM   HERE COMES THE TEST
3130 S = TI
3140 FOR N = 0 TO 99 : G80BOX , L(N), T(N), R(N), B(N), 1 : NEXT
3150 RS(10) = TI - S                                  :REM   STORE RESULT
3160 :
3170 G80GRAPHIC 6, 1                                  :REM   80-COLUMN GRAPHICS, CLEARED
3180 FAST                                             :REM   SET SPEED
3190 :                                                :REM   HERE COMES THE TEST
3200 S = TI
3210 FOR N = 0 TO 99 : G80BOX , L(N), T(N), R(N), B(N), 1 : NEXT
3220 RS(11) = TI - S                                  :REM   STORE RESULT
3230 :
3240 RETURN
3250 :
3260 :
3270 REM ----- PRINT RESULT HEADINGS -----
3280 :
3290 CHAR ,  0, 0, "G80 TEST SUITE"
```

```
3300 CHAR , 19, 0, "RANDOM SEED ="
3310 PRINT RS ;
3320 CHAR , 44, 0, "SCREEN WIDTH ="
3330 PRINT SW
3340 CHAR ,  0, 2, "TEST DESCRIPTION"
3350 CHAR , 50, 2, "MODE"
3360 CHAR , 60, 2, "TIME (IN JIFFIES)"
3370 CHAR ,  0, 3, "============================================"
3380 CHAR , 50, 3, "===="
3390 CHAR , 60, 3, "================="
3400 RETURN
```

```
1000 REM ----- PROGRAM IDENTIFICATION -----
1010 :
1020 REM           G40 TEST SUITE
1030 :
1040 REM   RUNS A SERIES OF PERFORMANCE TESTS
1050 REM   ... ON THE 40-COLUMN GRAPHICS ROUTINES
1060 :
1070 REM   WRITTEN TO OUTPUT RESULTS TO 80-COLUMN TEXT SCREEN
1080 REM   ( 'CUZ THERE'S MORE ROOM THERE )
1090 REM   :
1100 REM   YOU MIGHT WANT TO INSTALL THE TEXT SCREEN DUMP
1110 REM   ... ROUTINES TO GET HARDCOPY OF THE RESULTS
1120 :
1130 REM   VERSION :      1.00
1140 REM   TIMESTAMP :    3:28 PM PST    SEPTEMBER 20, 1986
1150 :
1160 REM   PROGRAMMED BY STAN KRUTE
1170 REM   COPYRIGHT (C) 1986 BY STAN KRUTE'S HACKER & NERD
1180 REM                         18617 CAMP CREEK ROAD
1190 REM                         HORNBROOK, CALIFORNIA   96044
1200 REM                         [916] 475-3428
1210 REM   ALL RIGHTS RESERVED
1220 REM   CALL OR WRITE FOR HELP, BUG REPORTS, LICENSING, ETC.
1230 :
1240 :
1250 REM ----- MAIN PROGRAM BLOCK -----
1260 :
1270 GOSUB 1350     :REM   INITIALIZE
1280 GOSUB 1820     :REM   RUN THE TESTS
1290 GOSUB 1930     :REM   REPORT THE RESULTS
1300 END            :REM   THAT'S THAT
1310 :
1320 :
1330 REM ----- INITIALIZE -----
1340 :
1350 PRINT CHR$ (147)                                   :REM   CLEAR SCREEN
1360 PRINT "PLEASE GIVE ME A RANDOM SEED 1-32768 " ;    :REM   SEED PROMPT
1370 INPUT RS                                           :REM   GET RANDOM SEED
1380 :
1390 PRINT CHR$(147)                                    :REM   CLEAR SCREEN
1400 PRINT "PLEASE GIVE ME A SCREEN WIDTH " ;           :REM   WIDTH PROMPT
1410 INPUT SW                                           :REM   GET SCREEN WIDTH
1420 :
1430 PRINT CHR$(147)                               :REM   CLEAR SCREEN
1440 PRINT "INITIALIZING RANDOM ARRAYS " ;         :REM   GIVE SOME FEEDBACK
1450 PRINT "WITH 100 VALUES ..." ;
1460 :
1470 FAST                                          :REM   LET'S MOVE
1480 :
1490 DIM T(99), B(99), L(99), R(99)                :REM   DIMENSE COORDINATE ARAYZ
1500 DIM RS (11)                                   :REM   DIMENSE RESULT ARRAY
```

Fig. 8-4. Source code for G40 Test Suite.

```
1510 N = RND (-RS)                                 :REM   SEED RANDOM NUMBERS
1520 :
1530 FOR N = 1 TO 100                              :REM   FILL ARRAYS
1540 :    T (N-1) = INT ( RND (1) * 200 )          :REM   T () GETS VALUES 0..199
1550 :    B (N-1) = INT ( RND (1) * 200 )          :REM   B () GETS VALUES 0..199
1560 :    L (N-1) = INT ( RND (1) * SW  )          :REM   L () GETS VALUES 0..SW-1
1570 :    R (N-1) = INT ( RND (1) * SW  )          :REM   R () GETS VALUES 0..SW-1
1580 :    IF N/10 <> INT (N/10) THEN 1640          :REM   FEEDBACK EVERY 10 VALUES
1590 :       N$ = STR$ (N)                         :REM      STRINGIZE THE COUNT
1600 :       PRINT N$ ;                            :REM      PRINT THE COUNT
1610 :       FOR P = 1 TO LEN (N$)                 :REM      MOVE CURSOR BACK
1620 :          PRINT CHR$ (157) ;                 :REM         MOVE CURSOR LEFT
1630 :          NEXT P
1640 :    NEXT N
1650 :
1660 RESTORE 1710                                  :REM   PREP FOR DATA READ
1670 FOR N = 0 TO 5                                :REM   READ 6 TEST NAME LABELS
1680 :    READ N$(N)                               :REM      READ A LABEL
1690 :    NEXT N
1700 :
1710 DATA "100 RANDOM DOTS",            "100 RANDOM VERTICAL LINES"
1720 DATA "100 RANDOM HORIZONTAL LINES", "100 RANDOM LINES"
1730 DATA "100 RANDOM OUTLINED BOXES",   "100 RANDOM FILLED BOXES"
1740 :
1750 SLOW                                          :REM   SLOW DOWN
1760 :
1770 RETURN
1780 :
1790 :
1800 REM ----- RUN THE TESTS -----
1810 :
1820 GOSUB 2130     :REM   DRAW 100 RANDOM DOTS
1830 GOSUB 2320     :REM   DRAW 100 RANDOM VERTICAL LINES
1840 GOSUB 2510     :REM   DRAW 100 RANDOM HORIZONTAL LINES
1850 GOSUB 2700     :REM   DRAW 100 RANDOM LINES
1860 GOSUB 2890     :REM   DRAW 100 RANDOM OUTLINED BOXES
1870 GOSUB 3080     :REM   DRAW 100 RANDOM FILLED BOXES
1880 RETURN
1890 :
1900 :
1910 REM ----- REPORT THE RESULTS -----
1920 :
1930 GRAPHIC 0,1 : GRAPHIC 5,1               :REM   80-COLUMN SCREEN, CLEARED
1940 :
1950 GOSUB 3270                              :REM   PRINT RESULT HEADINGS
1960 :
1970 FOR T = 0 TO 5                          :REM   FIVE TESTS ( 2 VARIANTS EACH )
1980 :    CHAR ,  0, T*3 + 4, N$(T)                    :REM   PRINT TEST NAME
1990 :    CHAR , 50, T*3 + 4, "SLOW"                   :REM   PRINT VARIANT MODE
2000 :    CHAR , 60, T*3 + 4, STR$ ( RS ( T*2 ))       :REM   PRINT RESULT
2010 :
2020 :    CHAR ,  0, T*3 + 5, N$(T)                    :REM   PRINT TEST NAME
2030 :    CHAR , 50, T*3 + 5, "FAST"                   :REM   PRINT VARIANT MODE
2040 :    CHAR , 60, T*3 + 5, STR$ ( RS ( T*2 + 1 )) :REM   PRINT RESULT
2050 :    NEXT T
2060 :
2070 PRINT                                   :REM   A NICE BLANK LINE
2080 RETURN
2090 :
2100 :
2110 REM ----- DRAW 100 RANDOM DOTS -----
2120 :
2130 GRAPHIC 1,1                       :REM   40-COLUMN HI-RES, CLEARED
2140 SLOW                              :REM   SET SPEED
2150 :                                 :REM   HERE COMES THE TEST
```

```
2160 S = TI
2170 FOR N = 0 TO 99 : DRAW , L(N), T(N) : NEXT
2180 RS(0) = TI - S                    :REM   STORE RESULT
2190 :
2200 GRAPHIC 1,1                       :REM   40-COLUMN HI-RES, CLEARED
2210 FAST                              :REM   SET SPEED
2220 :                                 :REM   HERE COMES THE TEST
2230 S = TI
2240 FOR N = 0 TO 99 : DRAW , L(N), T(N) : NEXT
2250 RS(1) = TI - S                    :REM   STORE RESULT
2260 :
2270 RETURN
2280 :
2290 :
2300 REM ----- DRAW 100 RANDOM VERTICAL LINES -----
2310 :
2320 GRAPHIC 1,1                                 :REM   40-COLUMN HI-RES, CLEARED
2330 SLOW                                        :REM   SET SPEED
2340 :                                           :REM   HERE COMES THE TEST
2350 S = TI
2360 FOR N = 0 TO 99 : DRAW , L(N), T(N) TO L(N), B(N) : NEXT
2370 RS(2) = TI - S                              :REM   STORE RESULT
2380 :
2390 GRAPHIC 1,1                                 :REM   40-COLUMN HI-RES, CLEARED
2400 FAST                                        :REM   SET SPEED
2410 :                                           :REM   HERE COMES THE TEST
2420 S = TI
2430 FOR N = 0 TO 99 : DRAW , L(N), T(N) TO L(N), B(N) : NEXT
2440 RS(3) = TI - S                              :REM   STORE RESULT
2450 :
2460 RETURN
2470 :
2480 :
2490 REM ----- DRAW 100 RANDOM HORIZONTAL LINES -----
2500 :
2510 GRAPHIC 1,1                                 :REM   40-COLUMN HI-RES, CLEARED
2520 SLOW                                        :REM   SET SPEED
2530 :                                           :REM   HERE COMES THE TEST
2540 S = TI
2550 FOR N = 0 TO 99 : DRAW , L(N), T(N) TO R(N), T(N) : NEXT
2560 RS(4) = TI - S                              :REM   STORE RESULT
2570 :
2580 GRAPHIC 1,1                                 :REM   40-COLUMN HI-RES, CLEARED
2590 FAST                                        :REM   SET SPEED
2600 :                                           :REM   HERE COMES THE TEST
2610 S = TI
2620 FOR N = 0 TO 99 : DRAW , L(N), T(N) TO R(N), T(N) : NEXT
2630 RS(5) = TI - S                              :REM   STORE RESULT
2640 :
2650 RETURN
2660 :
2670 :
2680 REM ----- DRAW 100 RANDOM LINES -----
2690 :
2700 GRAPHIC 1,1                                 :REM   40-COLUMN HI-RES, CLEARED
2710 SLOW                                        :REM   SET SPEED
2720 :                                           :REM   HERE COMES THE TEST
2730 S = TI
2740 FOR N = 0 TO 99 : DRAW , L(N), T(N) TO R(N), B(N) : NEXT
2750 RS(6) = TI - S                              :REM   STORE RESULT
2760 :
2770 GRAPHIC 1,1                                 :REM   40-COLUMN HI-RES, CLEARED
2780 FAST                                        :REM   SET SPEED
2790 :                                           :REM   HERE COMES THE TEST
2800 S = TI
```

```
2810 FOR N = 0 TO 99 : DRAW , L(N), T(N) TO R(N), B(N) : NEXT
2820 RS(7) = TI - S                               :REM   STORE RESULT
2830 :
2840 RETURN
2850 :
2860 :
2870 REM ----- DRAW 100 RANDOM OUTLINED BOXES -----
2880 :
2890 GRAPHIC 1,1                                  :REM   40-COLUMN HI-RES, CLEARED
2900 SLOW                                         :REM   SET SPEED
2910 :                                            :REM   HERE COMES THE TEST
2920 S = TI
2930 FOR N = 0 TO 99 : BOX , L(N), T(N), R(N), B(N) : NEXT
2940 RS(8) = TI - S                               :REM   STORE RESULT
2950 :
2960 GRAPHIC 1,1                                  :REM   40-COLUMN HI-RES, CLEARED
2970 FAST                                         :REM   SET SPEED
2980 :                                            :REM   HERE COMES THE TEST
2990 S = TI
3000 FOR N = 0 TO 99 : BOX , L(N), T(N), R(N), B(N) : NEXT
3010 RS(9) = TI - S                               :REM   STORE RESULT
3020 :
3030 RETURN
3040 :
3050 :
3060 REM ----- DRAW 100 RANDOM FILLED BOXES -----
3070 :
3080 GRAPHIC 1,1                                  :REM   40-COLUMN HI-RES, CLEARED
3090 SLOW                                         :REM   SET SPEED
3100 :                                            :REM   HERE COMES THE TEST
3110 S = TI
3120 FOR N = 0 TO 99 : BOX , L(N), T(N), R(N), B(N), , 1 : NEXT
3130 RS(10) = TI - S                              :REM   STORE RESULT
3140 :
3150 GRAPHIC 1,1                                  :REM   40-COLUMN HI-RES, CLEARED
3160 FAST                                         :REM   SET SPEED
3170 :                                            :REM   HERE COMES THE TEST
3180 S = TI
3190 FOR N = 0 TO 99 : BOX , L(N), T(N), R(N), B(N), , 1 : NEXT
3200 RS(11) = TI - S                              :REM   STORE RESULT
3210 :
3220 RETURN
3230 :
3240 :
3250 REM ----- PRINT RESULT HEADINGS -----
3260 :
3270 CHAR ,  0, 0, "G40 TEST SUITE"
3280 CHAR , 19, 0, "RANDOM SEED ="
3290 PRINT RS ;
3300 CHAR , 44, 0, "SCREEN WIDTH ="
3310 PRINT SW
3320 CHAR ,  0, 2, "TEST DESCRIPTION"
3330 CHAR , 50, 2, "MODE"
3340 CHAR , 60, 2, "TIME (IN JIFFIES)"
3350 CHAR ,  0, 3, "============================================="
3360 CHAR , 50, 3, "===="
3370 CHAR , 60, 3, "================="
3380 RETURN
```

Chapter 9: Human Interface

This second programming project is a sound and music recording lab. You get to play with BASIC 7.0's built-in sound commands via a Macintosh-like graphic user interface. The lab lets you record, play, print, store, and load sound/musical compositions. This is a full-scale project, with all kinds of hot programming concepts for you to fiddle with. A number of programs and files are involved. I'll discuss each program's human interface. Let's start with the lab itself. There's a lot to cover.

9.1 GETTING THE LAB GOING

Prepare a disk that contains the following files: the BASIC 7.0 program Sound/Music Lab (see Fig. 16-8 for its listing), the compiled object code for S/M Asm 1 (see Fig 16-1 for its assembly language source code), the compiled object code for S/M Asm 2 (see Fig. 16-2 for its assembly language source code), the sequential data file S/M Help Pack (which is created by running S/M Help Packer—see Section 9.17 and Fig. 16-3 for its listing), the sequential data file S/M Vars (which is created by running Make S/M Vars—see Section 9.15, and Fig. 16-4 for its listing), and the binary data file Finger Cursor (which you get by sending in for the program disk, working with the C-128's sprite-editing tools, or using the C-128's built-in monitor—see Chapter 11 for the do-it-yourselfer instructions.)

Okay, you've got a disk packed with these six files. If you have a joystick, plug it into control port 2. To start up the lab, give this command:

```
RUN "SOUND/MUSIC LAB"
```

Go out and make yourself a snack while the program loads itself and its various tools. It takes a few minutes with stock Commodore drives. When you come back you'll see a screen that looks like Fig. 9-1. The lab's message window tells you it's ready to roll.

9.2 THE LAB SCREEN

Let me describe the screen. At the top is a SOUND command window. There are two parts to this window, a title area containing the label SND, and a parameter area containing labels and (when you're working with the command) values for the SOUND command's eight parameters: voice, frequency value, duration, step direction for sweep, minimum frequency for sweep, step value for sweep, waveform, and pulse width. Check out the command descriptions in the C-128 Prg and other references for more detail on all the BASIC 7.0 sound commands and their parameters.

Beneath the SOUND window is a PLAY command window. It also has two parts: a title area on the left, and a PLAY string editing area on the right. This is where you work with BASIC's most powerful sound/music command, PLAY.

Beneath the PLAY window and on the left is the ENVELOPE window. This is where you work with BASIC 7.0's ENVELOPE command. There is a title area, and a large

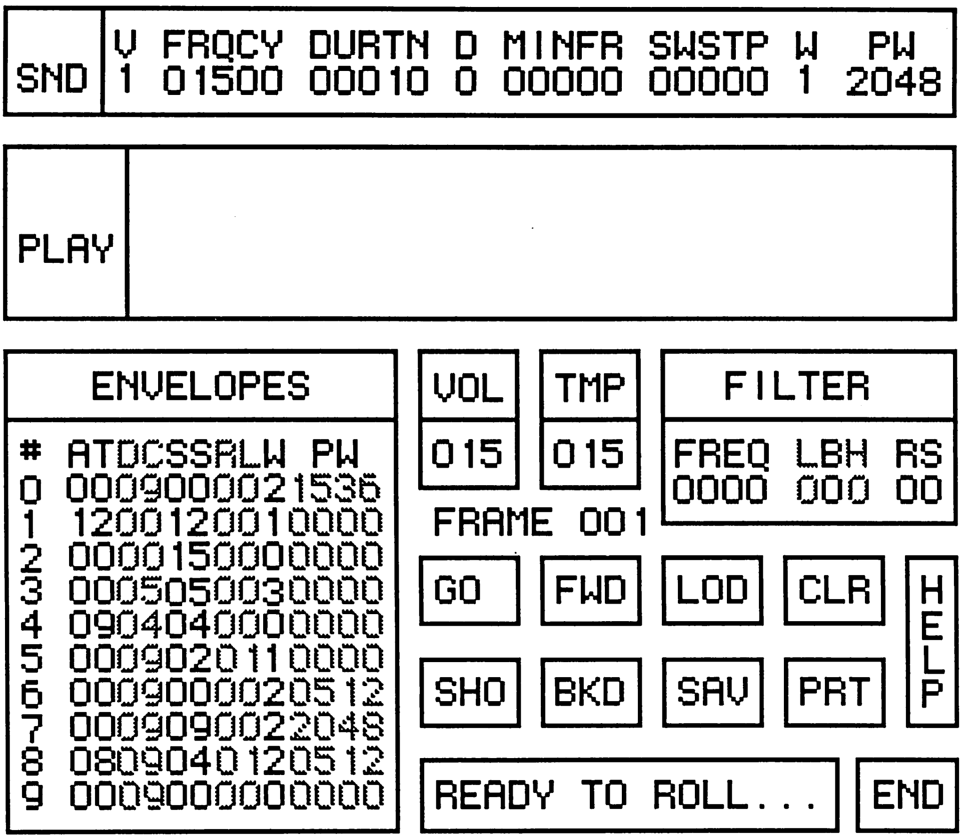


Fig. 9-1. The sound and music laboratory's main screen upon startup.

data area beneath. Each of the ten configurable envelopes has a line in the data area. Each line holds that envelope's six parameters: attack rate, decay rate, sustain length, release rate, waveform, and pulse width. There are labels for each parameter, and a number for each envelope.

Three smaller windows lie to the right of the **ENVELOPE** window: a **VOLume** window, a **TEMPO** window, and a **FILTER** window. Each has a title area and a data area. Each lets you use the corresponding BASIC 7.0 command. The **VOLume** window lets you set a volume level. The **TEMPO** window lets you set the speed of **PLAYed** sound/music commands. The **FILTER** window lets you set the five **FILTER** command parameters: filter cut-off frequency, low-pass filter on/off switch, band-pass filter on/off switch, high-pass filter on/off switch, and amount of resonance. Again, refer to the C-128 Prg for more detail on these BASIC 7.0 commands and their parameters.

Beneath the **VOLume** and **TEMPO** windows is a frame counter. Each time you tell the lab to record a particular command, it stores the command as an individual sound frame and moves on to the next frame. I call a collection of recorded lab commands a song. The frame counter tells you what frame of the current song the lab's working on. When the lab starts up, you're working on frame 1 of an empty song. There's a maximum of 1000 frames, although complex recordings might fill up memory before that frame limit is reached.

Beneath the frame counter and the **FILTER** window are ten buttons and a message window. The button labelled **GO** is used to try out sounds and start playbacks. The button labelled **FWD** is used to move the frame counter forward. The button labelled **LOD** is used to load in lab songs saved to disk. The button labelled **CLR** is used to clear the lab back to a fresh starting condition. The button labelled **HELP** lets you access the lab's help screens. The button labelled **SHO** lets you see the contents of any recorded frame. The button labelled **BKD** moves the frame counter backward. The button labelled **SAVE** is used to save the current lab song to disk. The button labelled **PRT** is used to print the current lab song on a printer. The button labelled **END** is how you leave the lab. To its left is the message window, which lets you and the lab trade messages drawn in green.

9.3 MOVING & CLICKING AROUND THE LAB

Moving the joystick moves the finger-cursor around the screen. You can also move it with the C-128's top row cursor keys. Pressing the joystick button or the C-128's Return key tells the program you want to select, or do, whatever the finger-cursor is touching. I call this joystick-keyboard-sprite point & select system a pseudo-mouse. When I say pseudo-mouse button, I mean either the joystick button or the C-128's return key. The tip of the finger-cursor's pointing finger is its hot spot, the part that the program tests when it's looking for the pseudo-mouse' location. If I tell you to move to a part of the lab, I mean that you should move the pseudo-mouse there. And when I tell you to click a part of the lab, I mean that you should move the finger-cursor's hot spot to that part of the screen and then press the pseudo-mouse button. You can click anywhere you want at any time. Finally, when you're not working in any of the lab's windows, just floating around freely, I call that the ready state. When the lab starts up, you're in the ready state. You can always get to the ready state by clicking outside any of the lab's windows or buttons.

9.4 USING THE HELP SCREENS

Move the pseudo-mouse over to the **HELP** button. Click the pseudo-mouse. The lab image disappears, and one of 22 help screens appears. See Fig. 9-2 for an example. Five buttons are at the bottom of each help screen. Clicking one of the four leftmost buttons takes you to different help screens (with wraparound): **FIRST**, **LAST**, **PREVIOUS**, or **NEXT**. The fifth button, labelled **LAB**, takes you back to the lab screen and the ready state. The lab's **HELP** button and the help screens' **LAB** button are hooked together so that mere clicking takes you back and forth, with no need for pseudo-mouse motion.

The help screens summarize the lab's operation. They're not activated in a context-sensitive way. (See Chapter 12 for hints on changing that). The first **HELP** click in a user's lab session activates help screen 1. The help screens do have a memory, so subsequent **HELP** clicks during a lab session take the user to the last-viewed help screen.

9.5 USING THE SOUND WINDOW

Click anywhere inside the **SOUND** window to wake it up. The current **SOUND** command parameter values appear, and the lab sets you up to enter/edit the **SOUND** command parameter nearest your click.

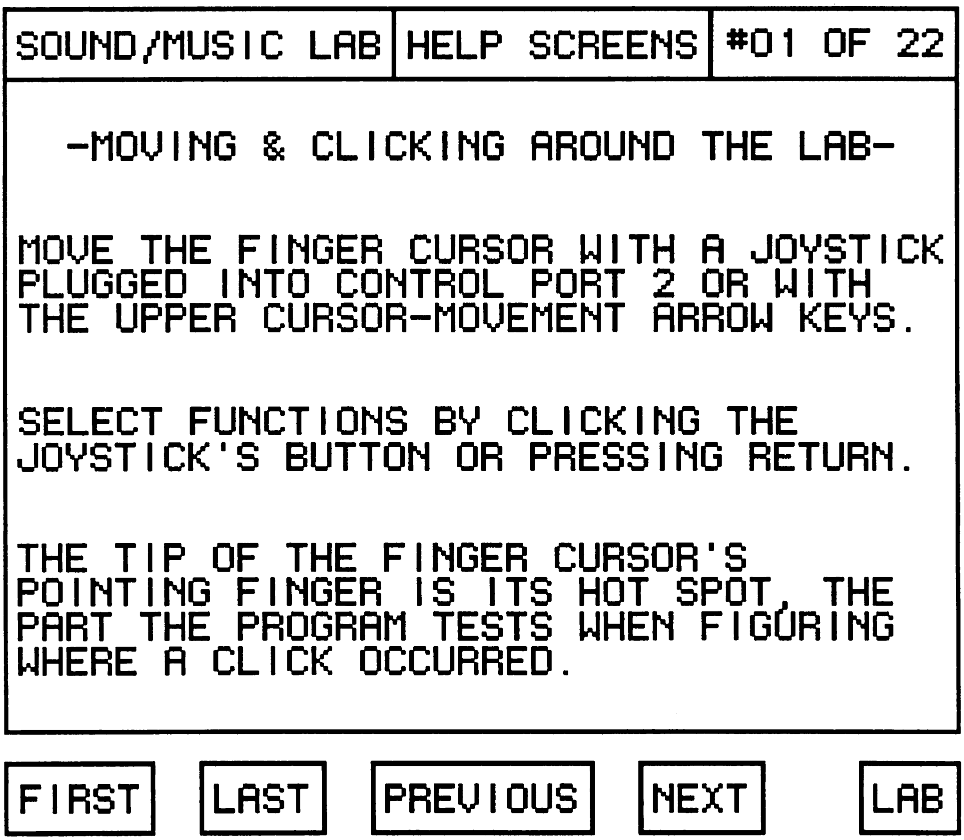


Fig. 9-2. One of the sound and music laboratory's help screens.

You get to enter/edit text and numerical data in a number of the lab's windows. A set of common editing routines, detailed elsewhere in this chapter, are used. They operate similarly to the C-128's standard screen editing routines, with a few improvements. I'll describe editing in the **SOUND** window, but be aware that the same techniques work throughout the lab.

The editing process centers around the editing cursor. It's an inverse block that indicates the current site of editing action. It moves along as you type. You can also move the editing cursor with the C-128's lower cursor keys (remember, the upper ones move the sprite finger-cursor). The insert key inserts spaces at the editing cursor, moving everything else to the right, just like standard C-128 screen editing. But unlike standard C-128 screen editing, one or more insert keypresses do not change the way the machine interprets that many subsequent keypresses (a feature/bug that drives me nuts). The delete key deletes the character to the left of the editing cursor, just as it does in standard C-128 screen editing.

So type in a value for the currently-selected parameter, and use the above-mentioned editing keys as necessary. When you finish, clicking the pseudo-mouse ends the data entry/editing session. What happens next depends on where the ending click occurred, whether the entered value is acceptable, and how any attempted recording may have gone. Here's a little chart that shows what happens:

if the value entered is...	*and the ending click is in the ...*	*this happens:*
acceptable	SOUND window data area	the parameter goes to the entered Value and the lab sets you up to enter/edit the SOUND parameter nearest the ending click.
acceptable	SOUND window title area	The parameter goes to the entered value & the lab plays and tries to record the displayed SOUND command—if recording works, the lab sets you up to enter/edit the same SOUND parameter again, if recording fails, the SOUND window deactivates, and you're back in the ready state.
acceptable	GO button	The parameter goes to the entered value, the lab plays the displayed SOUND command, and the lab sets you up to enter/edit the same SOUND parameter again.
acceptable	any other part of the lab	The parameter reverts to its prior value, the SOUND window deactivates, and the lab acts on the click as if it were entered from the ready state.
no good	SOUND window data area, SOUND window title area, or GO button	The lab burps and tells you the entered value is no good, the param-

		eter reverts to its prior value, and the lab sets you up to enter/edit the same SOUND parameter again.
no good	any other part of the lab	The lab burps and tells you the entered value is no good, the parameter reverts to its prior value, the SOUND window deactivates, and the lab acts on the click as if it were entered from the ready state.

This may look intricate here on paper, but in the lab it's pretty intuitive. Here's a general summary: If you want to try out a SOUND command, you click on the GO button. If you want to record a SOUND command, you click on the SOUND window's title area. If you want to work on another SOUND parameter, click in its vicinity. If you want to stop working with the SOUND command, click somewhere else in the lab. If your entered parameter value is no good, the lab has you try it again, unless you indicated you wanted to go somewhere else in the lab.

The lab's other windows follow this general pattern. Clicking in a window's title area means you want to record what you've edited. Clicking in a window's data area means you want the editing to take effect. Clicking outside the window means you want to forget what you've done. Of course, the nature of each window adds slight variations, but mostly there's consistency to the lab's responses.

9.6 SOME RECORDING CONCEPTS

When you choose to record a SOUND command, as detailed above, it's recorded at the current frame, then the frame counter advances. That's how all commands are recorded. The recording process always makes sure there's enough memory left to make the recording. If you want to re-record a frame, just move the frame counter back to that frame, as noted below in the frame counter discussion, and record a new frame over the old one.

You can record seven types of commands: a SOUND command, a PLAY command, an ENVELOPE command, a VOLume command, a TEMPO command, a FILTER command, and a Frame command. The first six correspond directly to BASIC 7.0's sound/music commands. The seventh command type, FRAME, lets a song jump to any of its frames. It gives the lab's recording process one of the rudimentary features of a programming language, the ability to branch.

At any time you can play back the current song's recorded frames. Normally when the lab plays back a song it goes from one frame to another in sequential order. The FRAME command lets you change that by telling the lab's playback mechanism to jump to any frame.

Recording frames does not mean they're saved to disk. For that, you have to explicitly use the SAV button.

9.7 USING THE PLAY WINDOW

Click anywhere inside the PLAY window to wake it up. The last-worked-on PLAY string will appear, and the lab sets you up to edit that string.

Editing the PLAY string works similarly to the editing process described for the SOUND window, with one useful addition: the pseudo-mouse can be used to move the editing cursor. So if you don't want to use the lower cursor keys to move the editing cursor, you can do it with the pseudo-mouse. Just click it somewhere in the PLAY window's data area while you're editing, and the editing cursor moves to that spot.

Clicking the pseudo-mouse outside the PLAY window's data area ends the editing session. The results are similar to what happens when you finish editing a SOUND window parameter, and depending on where the edit-ending click occurs, whether the entered string is acceptable, and how any attempted recording goes. Here's a summary: If you want to try out a PLAY string, click on the Go button. If you want to record a PLAY command, click on the PLAY window's title area. If you want to stop working with the PLAY command, click somewhere else in the lab. If you try to PLAY a lousy PLAY string via clicking the Go button or the PLAY window's title area, the lab will let you know with an error message, then return you to work on the string. By the way, sometimes a PLAY string may look legal, but still result in an error message when you try to play it. Try cleaning out any invisible weirdness by cruising over blank areas via the spacebar. That's always worked for me.

PLAY is BASIC 7.0's most powerful sound/music command. You can do remarkable things with it. Take a look at the C-128 Prg (as usual) for more details on what you can use in a PLAY string, then spend some time trying things out in the lab. Actually, easy exploration of PLAY string possibilities is what led me to design and program the sound/music lab.

9.8 USING THE ENVELOPE WINDOW

The ENVELOPE window lets you adjust BASIC 7.0's ten built-in envelopes. Once again, refer to the C-128 Prg for information on the ENVELOPE command. After changing an envelope, you can listen to the effect by using the 'T' option in a PLAY string.

After playing with the SOUND and PLAY windows, using this window should be easy. All ten envelopes and their current parameter settings are shown. The parameters are attack rate, decay rate, sustain level, release rate, waveform code, and pulse width. Just click on the parameter value you want to change and edit/enter a new value. The editing is done just as it was in the SOUND window.

Again, a click of the pseudo-mouse ends parameter value entry/editing. If you click on the ENVELOPE window title, it'll record a frame that gives the ENVELOPE command for the envelope whose parameter value you've worked with. If you click elsewhere in the ENVELOPE, the lab will just carry out the ENVELOPE command for the envelope whose parameter you've edited. If you click outside the ENVELOPE window, the parameter reverts to its previous value. If your entered/edited value is out of range, the parameter reverts to its previous value.

9.9 USING THE VOLUME WINDOW

The VOLume widow lets you set the lab's volume via BASIC 7.0's VOL command. Click anywhere inside the VOLume window to wake it up. Then edit the volume setting just as you edited other settings. It can take on values in the range 0..15. When you finish editing, click the pseudo-mouse. If you click it outside the window, the value reverts to its prior value.. If you click it in the window's data area, the lab's volume goes to the new value. If you click it in the window's title area, the lab's volume goes to the new value and the lab records a VOLume command frame.

9.10 USING THE TEMPO WINDOW

The TEMPO window lets you set the rate the PLAY command runs at. It does this via BASIC 7.0's TEMPO command, which takes values from 1 (slowest) to 255 (fastest). Click anywhere inside the TEMPO window to wake it up. Then edit the tempo setting. When you finish editing, click the pseudo-mouse. If you click it outside the window, the value reverts to its prior value.. If you click it in the window's data area, the lab's TEMPO goes to the new value. If you click it in the window's title area, the lab's TEMPO goes to the new value and the lab records a TEMPO command frame.

9.11 USING THE FILTER WINDOW

The FILTER window lets you play with BASIC 7.0's FILTER command. Five parameters can be worked with: filter cut-off frequency, low-pass filter on/off switch, band-pass filter on/off switch, high-pass filter on/off switch, and amount of resonance.

As with the other command windows, you get to edit/enter values for each parameter. Refer to the C-128 Prg for details on appropriate values. The edit-ending pseudo-mouse click works in the standard way: click in the window's title area to record a FILTER command frame, click in the data area to affirm the editing/entry, click outside the window to ignore any changes and revert to the prior parameter value.

The FILTER command's effects can be quite subtle. They really let you fine-tune the C-128's sounds. There's a lot to learn, so play away.

9.12 USING THE FRAME COUNTER

The frame counter's located just under the VOLume and TEMPO windows. It's there to show you what frame of its current song the lab's dealing with. It can also be used in a more active way.

Click in the frame counter title or data area. That sets you up to enter/edit the frame counter value. The frame counter can take on values from 1 to 1000. As usual, click on the pseudo-mouse to end the editing. If the pseudo-mouse click is outside the frame counter's title or data areas, the frame counter will just revert to its prior value. If the click is in the frame counter's data area, the lab will accept the change and jump to that frame.

Now, if the click is in the frame counter's title, the lab will try to record a Frame change command at the frame indicted by the counter's prior value, then advance the counter one frame. The recorded Frame command tells the lab to jump to the frame whose value you entered.

Confused? Here's an example. The lab starts up with the frame counter indicating frame 1. Suppose you record a PLAY string command at frame 1. The frame counter then increments to frame 2. Now you record a SOUND command as frame 2. The frame counter then increments to frame 3. Now you click on the frame counter and edit the value to 1. To end this editing/entry, you click on the frame counter's title. The lab will record frame 3 as a jump to frame 1, then increment to frame 4. Now click on the frame counter again and edit the frame counter value back to 1. This time, click on the frame counter data to end editing. That jumps the lab back to frame 1. Click the GO button to play this little three-frame song. The lab will do frame 1's PLAY command. Then it'll do frame 2's SOUND command. Then it'll do frame 3's Frame command and jump back to frame 1. Then it'll do frame 1's PLAY command again, then frame 2, and so on. 1,2,3,1,2,3,1,2,3,....The song will play endlessly until you click the pseudo-mouse to stop it.

9.13 USING THE TEN BUTTONS

Okay, now let's go over the lab's ten buttons. We've already discussed clicking the GO button while working in the SOUND or PLAY windows: it lets you try out the SOUND or PLAY command. Clicking the GO button from the ready state tells the lab to play back the current song's recorded frames. It'll start at the current frame, as indicated by the frame counter, and end with the last recorded frame. If you want to halt the playback, just click the pseudo-mouse button outside the GO button. When you're done, the frame counter reverts to the value it had before GO was clicked. That way, it's easy to listen to a song over and over.

The FWD button advances the frame counter.Click it once, and the lab advances to the next frame. Hold the pseudo-mouse button down, and the lab will keep advancing until you let up on it. When you screech to a stop, the lab lets you know whether the moved-to frame has been recorded or not.

The LOD button lets you load pre-recorded sound/music lab songs from the disk drive you used to start the lab. Click it once, and the message window turns into a data entry window. Enter the name of the file you want to load. You've got all the standard lab editing tools, including the pseudo-mouse-controlled editing cursor movement described before under the PLAY window's explanation. When you finish entering the file name, click the LOD button again to tell the lab to go ahead and try to load the file from the disk in the startup drive. Or click outside the LOD button if you have second thoughts and decide not to load the file. As it does throughout the lab's operations, the message window will keep you apprised of the loading process. If the file loads successfully, the recorded song becomes the lab's current song, and the frame counter sets to frame 1.

The CLR button resets the lab to its starting state. Any recorded frames are erased, so be sure you've saved any valuable work before clicking CLR.

The HELP button,described earlier, gives you access to several help screens that describe the lab's operations.

The SHO button lets you review any recorded frames. Click it once, and the current frame's command appears. For example, a SOUND command SHOws up by lighting up the SND window and displaying the command's parameter values. Keep clicking the SHO button, and subsequent frames appear. When you want to stop the review, just click outside the SHO button. The frame counter remains at the last frame SHOwn. That makes it easy to edit something that's caught your eye.

The BKD button moves the frame counter backwards. Click it once, and the lab goes back to the previous frame. Hold the pseudo-mouse button down, and the lab will keep retreating until you let up on it. When you stop, the lab lets you know whether the moved-to frame has been recorded.

The **SAV** button lets you save the lab's current song to a disk file. Click it once, and the message window turns into a data entry window. Enter the file name, just as described above for the **LOD** button. When you finish, click the **SAV** button to go ahead with the save attempt, or click outside the **SAV** button if you chicken out.

The **PRT** command prints the current song's recorded frames on a printer. Be sure the printer is on-line before clicking this one. Figure 9-3 shows a sample printout.

Finally, there's the **END** button. Clicking **END** takes you out of the lab, back to the C-128's **READY** prompt. As with **CLR**, be sure you've saved any senses-shattering symphonia before clicking this button.

9.14 LAB WRAPUP

Okay, we've covered the lab's operation. It's a fairly robust program, able to handle almost any user input without crashing. Response time isn't bad, considering the small ratio of assembly language to BASIC 7.0 used in the coding. If you've used a Macintosh, Amiga, Atari ST, or other mouse-controlled machine, the user interface should be particularly intuitive. The key is the level of consistency and logicality in the lab's response to user actions. I designed the program as an exploratory learning tool, but you can actually use it to come up with some sophisticated recordings. Have fun pseudo-mousing around with it.

9.15 USING MAKE S/M VARS

Because Sound/Music Lab is so large, I created a separate program to prepare a file full of pre-initialized variables. That program's called Make S/M Vars. It creates a file called S/M Vars. Sound/Music Lab reads in the variable values contained in S/M Vars.

If you bought this book's program disks, you don't need to run Make S/M Vars, since S/M Vars is supplied ready-made. Otherwise, you do. Here's how:

Prepare a disk that contains the BASIC 7.0 program Make S/M Vars (See Fig. 16-4 for its listing). Then give this command to load the file-maker:

```
DLOAD "MAKE S/M VARS"
```

Next, insert the disk you want the file of variable values on. This'll usually be the disk that contains the BASIC 7.0 program Sound/Music Lab.

Finally, run the file-maker with this command:

```
RUN
```

That's all there is to it. To check things out, do a catalog of the disk. You'll see a new file on it, S/M Vars.

9.16 USING MAKE 40C SCREENS

Make 40C Screens is a utility program I used to create the sound/music lab's help screens. It lets you edit and save complete 40-column text screens. Since it uses a fast assembly language editing routine, it's got a nice responsive feel. The screen files can be used in other programs.

As written, Make 40C Screens uses two monitors: a 40-column device hooked to the C-128's composite video output, and an 80-column device hooked to the 80-column video output. This eased the programming for me, but you may want to change it so it'll work without the 80-column monitor. See Chapter 12.

Once again, if you bought the program disks, you have no need (beyond curiosity) to run Make 40C Screens. But if you didn't, you'll want to run it to create the screens required by the sound/music lab. Here's how:

Prepare a disk that contains the BASIC 7.0 program Make 40C Screens (See Fig. 16-5 for its listing), and the compiled object code for 40C Edit (See Fig. 16-6 for its assembly language source code).

VERY IMPORTANT: If you want to print out the help screens, you'll also need the compiled object code for a program called Text Dumps. The Text Dumps object code comes on the disk of book programs, available from TAB. Source code is available from the author. If you don't have Text Dumps, you need to **delete the following lines** from Make 40C Screens: 1540, 1890, and 3080. Sorry we didn't include the source code for Text Dumps, but this book is already too big, and the functionality it adds to Make 40C Screens is a luxury.

Once you've got your disk ready, rev up the C-128, turn on both monitors, and give this command:

RUN "MAKE 40C SCREENS"

The 80-column screen now displays a menu of command choices:

Edit The Screen
Clear The Screen
Save The Screen
Load The Screen
Print The Screen {if you've got TEXT DUMPS}
Quit The Program

This is called command mode. From here, you can invoke any of the six displayed commands by typing its first letter.

Press **E** to edit the 40-column screen. An editing cursor appears on that screen. You're now in 40-column editing mode, ready to work. You can type any printable character key. You can move the editing cursor with the cursor keys. You can use insert and delete to move things around. You can use reverson and reverseoff to get reversed characters. All other keys are disabled, with one exception: the shift-return combination. Press that pair simultaneously to leave editing mode and return to command mode.

The second command lets you clear the 40-column screen. Just press **C**, and the deed is done.

The third command lets you save the 40-column screen.character information to a disk file (color information is not saved—see Chapter 12 for hints on changing this). Press **S** to save the current 40-column screen. The program will prompt you for a file name. If you're creating files for Sound/Music Lab, the file should be named S/M Help #, where the # is replaced by a number in the range 1..22. For example, the file for the lab's first help screen should be S/M Help 1, and the 22nd screen help file should be S/M Help 22. After getting a file name, the program will prompt you for a disk drive device number. Usually, that'll be 8 or 9, depending on how your drives are set up. If you've developed cold feet and don't want to save the screen after all, type in a spurious device number, like 32456. If you type in a valid device number, the program proceeds to save the screen.

The fourth command lets you load in saved 40-column screens. Press **L** to do so. The program will prompt you for a file name and a device number, just as it did for saving screens. If your entries are valid, the program goes ahead and loads the saved screen.

The fifth command uses Text Dumps to print out the 40-column screen. Remember, if you don't have Text Dumps, you should disable this command by deleting lines 1540, 1890, and 3080. If you do have Text Dumps, get your printer ready, then just press **P** to print out the screen.

Finally, the sixth command lets you quit the program. Just press **Q**.

Pretty simple program, eh? In fact, this is the minimal functionality needed for any text editor. And I stress minimal.

Okay, now you need to save 22 help screens, named as detailed above, onto a disk. Since this book is already too large, I haven't included the text for the 22 help screens. They're just a synopsis of Sections 9.1 through 9.14. So you get to make up your own screens. If you're not feeling creative, just save 22 screens worth of garbage. But you do need 22 appropriately-named help screens for the program S/M Help Packer (and, in turn, Sound/Music Lab) to run.

9.17 USING S/M HELP PACKER

When the sound/music lab is running, the help screens and their entourage sit in the top half of RAM bank 1's memory. To facilitate loading, I wrote the program S/M Help Packer, which gets everything saved as one big block of binary data, the file S/M Help Pack.

Again, if you bought the program disks there's no need to run S/M Help Packer. For everyone else, here's how to do it:

Prepare a disk that contains the BASIC 7.0, program S/M Help Packer (See Fig. 16-3 for its listing). Then give this command:

```
DLOAD "S/M HELP PACKER"
```

Now insert a disk that contains the following: the twenty-two help screen files (discussed in the previous section), and the binary data file Finger Cursor (discussed back in Section 9.1). Put it into the drive you loaded S/M Help Packer from, then give this command:

```
RUN
```

It'll take a few minutes for the program to load in all the information it'll be saving. When it finishes, it'll prompt you for another disk. Put the disk you want to run Sound/Music Lab from into the same drive you've been using, making sure with a **CATALOG** command that the disk has at least 126 free sectors. Then press the spacebar. It'll take another few minutes for the program to save S/M Help Pack to this disk. When it finishes, it'll print a disk catalog on the screen. You should see the new file, S/M Help Pack, with a size of 126 blocks.

Chapter 10: System Interface

Due to the size of this project, I've only got room to touch on a small selection of system interface issues. You can refer to Appendix D:System Interface Summary to locate instances of the following items and others in the project's programs' code.

10.1 USING THE STANDARD TEXT SCREEN RAM FOR ASSEMBLY LANGUAGE ROUTINES

Sound/Music Lab is a very large BASIC program. In fact, it eats up almost all of RAM bank 0. And it uses a number of assembly language routines. Due to what these routines do, and the C-128's system architecture, coding's a lot easier if they also live in RAM bank 0. One group of routines, S/M Asm 2, lives in the part of RAM bank 0 that's set aside for such stuff: $1300-$1BFF. But there's more, so I had to look around for another area. Since Sound/Music Lab uses the 40-column bit-mapped graphics screen, the 40-column text screen area, $0400-$07FF, is free. And that's where I stick the other group of routines, S/M Asm 1.

The only detail to attend to before loading the routines into that area is switching to the graphics screen. In Sound/Music Lab, this is done in the subroutine Draw A Fresh Screen (lines 2910-3040). Then the assembly language routines are loaded via the next subroutine, Update The Screen, in line 3220.

10.2 READING FROM ANY MEMORY BANK VIA INDFET

The C-128 kernel provides a useful routine for reading from any byte in any memory bank, IndFet. When this routine is called, the A register should contain the address of a zero-page location that contains (along with the next zero-page location) the address of the byte in memory you want IndFet to read, and the X register should contain the number of the memory bank. Upon return, the A register contains the desired byte. IndFet is used in the subroutine :SetStrgz, lines 40-139 of S/M Asm 1 B.S. IndFet is used there to examine bytes of a BASIC string that's stored in RAM bank 1.

10.3 WRITING TO ANY MEMORY BANK VIA INDSTA

IndSta is a C-128 kernel routine that lets you write a value to any byte in any memory bank. When this routine is called, the A register should contain the value to be stored, the X register should contain the number of the memory bank, and memory location StaVec ($02B9) should contain the address of the byte in memory you want IndSta to write to. IndSta is also used in the subroutine :SetStrgz, lines 40-139 of S/M Asm 1 B.S. IndSta is used there to store bytes into a BASIC string that's stored in RAM bank 1.

10.4 CHANGING A BASIC CHARACTER STRING FROM ASSEMBLY LANGUAGE

The :insert and :delete subroutines, lines 569-636 of S/M Asm 1 B.S and lines 3-79 of S/M Asm 1 C.S, show how a BASIC character string can be changed from assembly language. The keys to this process are the IndFet and IndSta routines mentioned in Sections 10.2 and 10.3. Those routines are used to move characters into and out of the string.

BASIC strings are stored in RAM bank 1. How to find a specific string? Just use BASIC 7.0's POINTER command. It returns a pointer to the first byte of a variable. Examples can be found in lines 6800 and 6810 of Sound/Music Lab.

Important note: The strings worked on by :insert and :delete have a constant length. The characters in a string are changed, but the length is not. It IS possible to change the lengths of BASIC strings from assembly language, but it's a pain.

10.5 DRAWING CHARACTERS ON THE 40-COLUMN BIT-MAPPED GRAPHICS SCREEN

Standard C-128 text characters are drawn in an 8-horizontal-dot by 8-vertical-dot matrix. A byte codes 8 horizontal dots. Eight bytes code one character's graphic pattern. When a character's on the text screen, the VIC chip grabs that character's group of 8 pattern bytes from the C-128's character ROM and pops it into memory.

To draw standard characters on the bit-mapped screen, we just mimic in software what the VIC chip does via hardware. The routine DrawBMChar, in S/M Asm 1 C.S, shows the details. Here's a summary: DrawBMChar is called with a C-ASCII character code and a location. The location is in a 40-column by 25-row matrix, same as the 40-column text screen. First, convert the character's C-ASCII code into a poke code. Next, locate the character ROM. Then multiply the character's poke code by 8 to get an offset into the character ROM. Add that offset to the ROM's starting address to get the starting address of the character's eight pattern bytes. Then the character's row and column location is used to figure out the corresponding set of eight bytes in the bit-mapped screen's RAM buffer. This is done by taking the starting location of the bit-map screen and adding an offset. Check out the source code or the algorithms for the details of figuring the offset. Now we've got a pointer into the character ROM and a pointer into the bit-map screen RAM. We move into a bank 14 memory configuration, which lets us read from character ROM and write to bit-map screen RAM. Then the eight pattern bytes are transferred.

10.6 CONVERTING C-ASCII CODES TO SCREEN POKE CODES

Like many simple translation problems, converting from a C-ASCII code to a screen poke code can be done in two ways: use a translation table or an algorithm. In this case, a table is very fast, but uses one byte for each character in the character set. An algorithm is a bit slower than a table, but takes up less room due to coherence—repetitive patterns—in the translations. In the subroutine **CAsc2Pok1**, located in S/M Asm 1 C.S, I use the algorithmic method. After all, since we're working in assembly language, a little more time is a very little more time. And the code only takes up 50 bytes, which is important in this case because the Sound/Music Lab is squeezed tight for space.

How does the translation algorithm work? Well, the relationship between C-ASCII and poke codes works in clumps. That is, a range of C-ASCII values, usually 32 or 64 at a crack, stay together when transformed into poke codes. The algorithm is just a set of tests that test a C-ASCII code against the boundaries of such a range; when the appropriate range, or clump, is found, the C-ASCII code can be easily adjusted into its corresponding poke code. A little study of **CAsc2Pok1** should show the elegant simplicity of this approach.

10.7 FINDING THE 40-COLUMN TEXT SCREEN

You can move the C-128's 40-column text screen buffer almost anywhere in RAM memory (See Sections 10.20 and 10.21). But how to find it? Well, there are two things to determine: which memory bank it's in, and what's its starting address.

The 40-column text screen buffer's memory bank is indicated by bit 6 of memory location $D506. If this bit is cleared to 0, the 40-column screen buffer's in RAM bank 0. If it's set to 1, the buffer's in RAM bank 1.

Finding the starting address is a bit more complex. Addresses are 16 bits long. The 40-column text screen buffer can only start on a 1K boundary. That means that, no matter where it's living, bits 0-9 of the starting address will be 0. Bits 10-13 of the starting address come from bits 4-7 of $D018, which is VIC register 24. You can also find those four bits in bits 4-7 of $0A2C (2604), which is a shadow register for $D018. Finally, bits 14-15 of the starting address are derived by flip-flopping bits 0-1 of memory location $DD00.

If all that seemed a bit obscure, you can look at the routine **BasBnk40**, in lines 579-645 of S/M Asm 2 B.S. This routine gets used by the Sound/Music Lab when it needs to find one of its help screens, which can live anywhere.

10.8 USING A KEYCHK DETOUR TO HIDE SELECTED KEYS FROM THE SYSTEM

The Sound/Music Lab lets you move a sprite cursor around on the screen by using a joystick or pressing one of the C-128's upper arrow keys. Pressing the return key is made equivalent to pressing the joystick's button.

Normally, these keys are handled by the system. What we need to do is intercept any presses of these joystick-equivalent keys and hide them from the system. Then we can do our own keyboard scan (See Section 10.9), and, if one of the keys is pressed, act on the press in our own way.

The trick is done by detouring the C-128's **KeyChk** routine. **KeyChk** is part of the system's regular keyboard scan. On entry to **KeyChk**, the A register holds the keycode of a pressed key. We reroute the call to **KeyChk** to a detour routine. The detour routine checks to see if the entry keycode is one we want to hide. If the keycode is such a beast, we hide it by clearing the A register to 0. Otherwise we leave A alone. In either case, the detour ends by jumping to the normal **KeyChk** routine.

You can see an example of this in S/M Asm 2 A.S. **OurKeyChk** is the detour routine. The routine **Install** installs it, and **UnInstall** removes it.

10.9 DETOURING IIRQ TO IMPLEMENT A PSEUDO-MOUSE AND CURSOR

Sound/Music Lab detours calls to the C-128's regular **IIRQ** routine in order to implement a pseudo-mouse and cursor sprite. **IIRQ** is a system heartbeat routine, called every sixtieth of a second to do things like reading the keyboard and updating the VIC chip. It's tied to the system's vertical retrace interrupt. The detour routine is called **OurIIRQ** in S/M Asm 2 A.S, is our detour. Like **OurKeyChk**, it gets installed by **Install**, and removed by **UnInstall**.

Details of **OurIIRQ** can be seen in Fig. 15-2, sheets 2-4, and the S/M Asm 2 A.S source code. Here's a summary: First, it checks to see if any of the upper arrow keys are being pressed. Then it looks to see if the joystick is being pressed in any direction. If the joystick and one of the arrow keys is giving directional information, the joystick info takes precedence.

Next: if the cursor sprite is currently in motion, and the user is indicating, via joystick or upper arrow keys, a change in that motion, the cursor sprite is stopped.

If the cursor sprite is stopped, and the user is indicating, via joystick or upper arrow keys, that he wants it to move, a call is made to the routine **SetMoshn** to get it moving in the appropriate direction.

Then **OurIIRQ** checks to see if the joystick button or the return key is pressed. It sets a pseudo-mouse click state flag to "pressed" or "not pressed" based on the result. Finally, **OurIIRQ** jumps to the regular **IIRQ** routine.

10.10 DIRECTLY READING THE KEYBOARD FROM ASSEMBLY LANGUAGE

The routine **OurIIRQ**, mentioned in Section 10.9, directly reads the C-128 keyboard from assembly language. How's it done? Well, the C-128 keyboard is wired as a matrix of control lines, which we'll call horizontal and vertical control lines. There's a nice illustration of this on page 642 of the C-128 Prg. The basic idea for reading the keyboard is that you send signals out on the vertical control lines, and look for results on the horizontal control lines. Now for some details:

The horizontal control lines are called R0-R7 (R stands for Row). They're accessed via bits 0-7 of memory location $DC01, also known as CIA port A, set up for input. The vertical control lines are C0-C7 and K0-K2 (C and K stand for Column). C0-C7 are accessed via bits 0-7 of $DC00, which is CIA port B, set up for output. K0-K2 are accessed via bits 0-2 of $D02F, which is the VIC chip's register 47.

To check a key, you clear its vertical control line's bit to 0, set all other vertical control line bits to 1, then read the key of interest's horizontal control line's bit. The key is pressed if the horizontal control line bit shows up cleared to 0.

OurIIRQ has two specific examples of this. The first: In lines 398-404, the code looks at the four upper arrow keys. The four keys share one vertical control line, K2, and four consecutive horizontal control lines, R3-R6. The K2 line is cleared to 0, all other vertical control lines are set to 1, and the result is read from $DC01. Since the keys have consecutive horizontal control lines, after appropriate masking and shifting we use the four bits of interest in the result as an index into a table of directions. This lets us handle cases where more than one key is pressed.

The second keyboard-reading example from OurIIRQ is in lines 480-489, where we look for a press of the return key. Its vertical control line is C0, so we clear that to 0, and set all others to 1. The return key's horizontal control line is R1, so if bit 1 of $DC01 comes back clear, we know that return is being pressed.

10.11 DIRECTLY READING THE JOYSTICK FROM ASSEMBLY LANGUAGE

The four directional switches for joystick 1, connected through control port 2, are linked to bits 0-3 of $DC00. Bit 4 reflects the state of the joystick's button. The four directional switches for joystick 2, connected through control port 1, are linked to bits 0-3 of $DC01. Bit 4 reflects the state of the joystick's button. A bit cleared to 0 indicates a joystick directional switch or button being pressed. You can use the pattern of set and cleared directional switch bits as an index into a table of direction codes. If you're just using one joystick, joystick 1 is preferred, since its control lines more easily avoid keyboard interference. That's what we do with Sound/Music Lab.

An example of joystick reading is seen in lines 412-419 and 473-478 of OurIIRQ. In 412-419, joystick 1's byte is read from $DC00, non-directional bits (4-7) masked out, the result used as index into a table of direction codes, and the appropriate direction code then stored away. In lines 473-478, joystick 1's byte is read from $DC00, non-button bits masked out, and the result used to determine the state of the joystick's button.

10.12 SPRITE MOTION FROM ASSEMBLY LANGUAGE

Sprites that you've activated in the normal ways can be set into motion from assembly language. All you need do is store an appropriate sprite motion data record into the sprite speed and direction tables. The system's standard vertical retrace interrupt routine will then take over, and adjust the sprite's position as needed every sixtieth of a second.

Appendix G gives a complete description of these speed and direction tables. And the code for the routine SetMoshn, in S/M Asm 2 B.S, shows how to use assembly-language to store a sprite motion data record into the tables.

10.13 SPRITE POSITIONING FROM ASSEMBLY LANGUAGE

Unless you disable some low-level flags, you can't position sprites from assembly language by just changing the values in the appropriate VIC registers. What you need to do is store position data in a set of sprite shadow registers. That's because routines triggered by the system's standard vertical retrace interrupt updates the VIC registers every sixtieth of a second with values taken from these shadow registers.

Appendix G gives a complete description of these sprite shadow registers.

10.14 INVERTING A CELL OF THE 40-COLUMN BIT-MAPPED SCREEN

In standard bit map mode, color for each 8-pixel by 8-pixel region, or cell, is determined by a color byte in a video matrix that usually lives at memory locations $1C00-$1FE7. This video matrix views the 320-pixel wide by 200-pixel high bit map screen as 40 cells wide and 25 cells high, set up like a 40-column text screen. The upper nibble of a cell's color byte gives the cell's foreground color, the lower nibble gives the background color. To invert a cell, just swap the nibbles in its color byte.

The routine **HRBandInvt**, located at lines 360-447 of S/M Asm 2 B.S, inverts bands of bit-mapped cells for the Sound/Music Lab. Lines 411-428, in particular, show the requisite nibble swap.

10.15 INVERTING A CELL OF THE 40-COLUMN TEXT SCREEN

There's an easy way to invert characters on the 40-column text screen. For each character in either of the two character sets, there's a reversed image of the character whose poke code is different only in that bit 7's set to 1. So you can invert a character by flip-flopping bit 7. In 6502 assembly language, this can be done by **EORing** the poke code with the mask value $10000000.

The routine **TX40BandInvt**, located at lines 448-530, inverts bands of 40-column text screen characters for the Sound/Music Lab. Line 505 shows the high bit flip-flopping. By the way, this routine works no matter where the 40-column text screen's living.

10.16 LOADING DATA INTO MEMORY BANK 1

The Sound/Music Lab loads twenty-two help screens into RAM bank 1, above BASIC's variables. There are two steps to this process.

First, the top of BASIC variables is moved down to make room for the help screen data. Two pointers have to be adjusted: **FreTop**, located at $0035-$0036, and **Max_Mem_1**, located at $0039-$003A. This is done in lines 1750-1770 of Sound/Music Lab.

Second, the help screens are loaded from a disk file with the bank parameter set to 1. This is done in line 3230 of Sound/Music Lab.

10.17 DEALING WITH SPRITES IN ALTERNATE TEXT SCREENS

When the user clicks the Sound/Music Lab's help button, the display changes from the bit-mapped image of the lab to a 40-column text screen showing one of the help pages. The sprite cursor needs to come along for the ride.

How is this done? Well, when VIC is showing a 40-column text screen, there's got to be a pointer to any sprite data located just above the 1000-byte text screen buffer. In addition, the sprite data itself must be living somewhere in the same 16K quadrant as the text screen.

Take a look at the program S/M Help Packer (Fig. 16-3). It takes the lab's help screens, each of which is just the saved image of an information packed 40-column screen buffer, and loads them into RAM bank 1 memory at the addresses they're to live at. For each screen, it also pokes a pointer to sprite data into the appropriate address, 1016 bytes above the start of the screen. Finally, copies of the lab's finger cursor screen data are stored into empty real estate in each of the help screen's two quadrants. Then the whole block, with its help screens, sprite data pointers, and sprite data, is saved as the binary file S/M Help Pack.

The Sound/Music Lab loads this file right back to where it was saved from. Then, when the lab tells VIC to switch to a help screen, the appropriate sprite data pointer and sprite data are right where they're supposed to be, and VIC can show the lab's sprite finger cursor with hardly a blink.

10.18 DIRECT TEXT DISPLAY FROM ASSEMBLY LANGUAGE

The basic idea is simple: Given a C-ASCII character, you figure out its poke code, then store that code at a particular location in the 40-column screen memory that corresponds to a desired row and column position.

Section 10.6 above describes how poke codes are derived from C-ASCII. Section 10.7 shows how the base of 40-column screen memory is derived. A particular location corresponding to a row and column position is figured by multiplying the 0-based row by 40, then adding in the 0-based column, then adding that offset to the screen memory's base address.

Example code can be found in a key routine from 40C Edit (Fig. 16-6), **PrintChar**. Located in lines 308-350, **PrintChar** prints a character to the 40-column screen, then advances the cursor one position to the right, with wraparound at the end of a line and at the end of the screen. The algorithm for **PrintChar** is shown in sheets 3-4 of Fig. 15-6.

10.19 BASIC-ASSEMBLY LANGUAGE PARAMETER PASSING

BASIC's **SYS** command lets you call an assembly routine, and (optionally) pass information via the A, X, Y, and Status registers. For example,

```
SYS 1123, 12, 13, 18, 138
```

. . . will jump to the assembly language subroutine at 1123 ($423), with A containing 12 ($0C), X containing 13 ($0D), Y containing 18 ($12), and the Status register containing 138 ($8A).

You can also pass information back to BASIC from an assembly language routine. The **RREG** command, not documented by Commodore but quite functional, lets you do this. The syntax is similar to that for **SYS**. For example,

```
RREG AV, XV, YV, SV
```

. . . will assign to the BASIC variable AV the value the A register had when the last assembly language subroutine called by **SYS** returned. XV will get the value the X register had, YV will get the value of Y, and SV will get the value of the Status register.

RREG is usually used right after a **SYS** command. For example, take a look at line 1500 of Sound/Music Lab:

```
1500 SYS HA(3), HA(4), HA(5): RREG MA
```

This **SYS** command calls on the **AreaSearch** routine from S/M Asm 2, passing via A and X a pointer to the **LabAreas** data table. **AreaSearch** returns an area result code in the A register, and the **RREG** command here assigns it to the BASIC variable MA.

Note that you can pass more than three values to an assembly language routine or BASIC by using two of the processor's registers to pass a pointer to a block of values. Take a look at lines 6770-6920 of Sound/Music Lab, for example: a pointer to an array of values is passed via A and X to the assembly language routine **StrngRectEdit**.

10.20 SETTING VIC'S RAM BANK AND QUADRANT (VIDEO BANK)

The C-128's VIC chip can look at one 16K video bank, or quadrant, of memory at a time. It defaults to looking at video bank 0 ($0000-$3FFF) in RAM bank 0.

In Sound/Music Lab, we move VIC's focus when the user chooses to look at the help screens. They live in quadrants 2 ($8000-$BFFF) and 3 ($C000-$FFFF) of RAM bank 1.

Bit 6 of $D506 (54534) selects VIC's RAM bank. Set the bit to 1 to get VIC looking at RAM bank 1, clear it to 0 for RAM bank 0. Bits 0 and 1 of $DD00 (56576) select the video bank: %11 selects video bank 0 ($0000-$3FFF), %10 selects video bank 1 ($4000-$7FFF), %01 selects video bank 2 ($8000-$BFFF), and %00 selects video bank 3 ($C000-$FFFF).

Lines 11680, 11690, 12020, 12090, and 12100 of Sound/Music Lab give examples of how these bits are set/cleared.

10-21 SETTING VIC'S 40-COLUMN TEXT SCREEN STARTING ADDRESS

As mentioned in Section 10.7, the C-128's 40-column text screen can live anywhere. How to move it to a particular spot?

Well, first you set the VIC RAM and video banks, as shown in Section 10.20. Then it's time to place the 40-column text screen on a 1K boundary within VIC's 16K video bank. There are (obviously) 16 possible locations. And four bits are needed to encode 16 values. The upper nibble—bits 4-7— of VIC register 24 ($D018) are used to indicate the 1K boundary the text screen sits at. This VIC register has a shadow register at $0A2C (2604). Plug a value into the upper nibble of 2604, and the system's raster interrupt routines automatically stick it into VIC register 24.

You can see an example of moving this text screen pointer in line 12010 of Sound/Music Lab. By just changing the pointer, we can move very quickly between help screens.

10.21 CHANGING PROCESSOR SPEED

The C-128's processor can run at two speeds. Unfortunately, the VIC chip goes to sleep and blanks the screen when the processor's running at its fastest. So we can't run the Sound/Music Lab at high speed if we want to see anything.

However, sometimes we don't need to see anything: during lab initialization and resetting. And those are both lengthy processes that appreciate the speedup. So I run the processor at its faster (2 megahertz) speed.

You can see examples of this in the Sound/Music Lab subroutines Set Up The Lab and Clear Click.

10.22 DETERMINING WHICH DRIVE WE (PROBABLY) CAME FROM

When Sound/Music Lab starts up, it needs to load a number of auxiliary files. It expects them to be on the same disk that it was loaded from. How can it tell which drive that disk is in? Memory location 186 ($00BA) holds the device number of the most recently accessed file system device. If we look there right after loading in Sound/Music Lab, before any new file system calls mess up the value, we'll find the device number of the drive Sound/Music Lab came from.

Take a look at lines 1830-1840 and 3200-3230 of Sound/Music Lab for examples showing how this address and its value are used to load in auxiliary disk files. Of particular note is the fact that you can use a variable to provide a parameter in a file system command, so long as the variable name is enclosed in parentheses.

10.23 USING THE CHAR COMMAND FOR PRECISE TEXT POSITIONING

One of the nicest additions to BASIC 7.0 is the CHAR command. It lets you easily print strings at precise locations on the text or bit-mapped screen. On the C-64 you had to go through some contortions to achieve the same effect. You'll notice I've used CHAR throughout Sound/Music Lab for text printing. For example, take a look at lines 3450, 3690, 3840, 3950, 4060, and 4210.

10.24 DEALING WITH DISK WACKINESS

Commodore disk drives are notorious for wacky behavior. There's not a lot a program can do about the drives. What it CAN do is note when a disk error's occurred, and recover gracefully.

The reserved variable DS (for Disk Status) takes on a non-zero value when there's a disk error. If you check it after disk operations, and act appropriately, you can avoid most disk-caused program crashes. For example, the Sound/Music Lab subroutine LOAD CLICK makes two checks of DS after critical disk operations; see lines 10620-10640, 10900, and 10960-10980. And DS has a companion, the reserved variable DS$. It'll contain a descriptive error string if there's been a problem. Note how we use DS$ in line 10970 to notify the user of what happened if LOAD CLICK's disk operations have hit a snag.

10.25 IS A SOUND STILL SOUNDING?

There are times in the operation of the Sound/Music Lab when we want a sound to finish sounding before the lab goes on to another task. In fact, there's a subroutine in Sound/Music Lab dedicated to just that purpose. It's called (surprise!) LET SOUND FINISH (lines 15810-15850). The key to this subroutine are two memory locations, $1282 (4738), called SoundTimeLo, and $1285 (4741), called SoundTimeHi. When both those locations take on the value 255, a sound triggered by one of the BASIC 7.0 sound commands has finished sounding. So LET SOUND FINISH just waits until that's the case.

10.26 SETTING UP AN ERROR HANDLER

I'm obsessed with creating programs that don't crash too often. Besides providing the DS/DS$ mechanisms mentioned in Section 10.24, BASIC 7.0 gives us the TRAP, ERR$, ER, EL, RESUME, and NEXT. TRAP lets you designate an error-handling subroutine that'll come into play when there's a problem executing a BASIC statement. ERR$ is a function that produces a descriptive string corresponding to an error-code number. After an error that triggers TRAP, the reserved variable ER will contain an error-code for the wacky event, and EL will contain the line that was being executed when it occurred. After such an error, RESUME tells the computer to start up again somewhere: if followed by a line number, at that line; if followed by NEXT, at the line following EL.

I use all this stuff in Sound/Music Lab. Line 1340 uses TRAP to set up an error-handling routine at line 16240. The routine at 16240, imaginatively named ERROR HANDLER, uses ERR$, ER, and EL to report the error to the user. Then it uses RESUME,NEXT to keep the program going. And it usually works.

Chapter 11: Program Notes

Again, due to the size of this project, I can only touch on some of the more interesting program features. Be sure to scan the selected algorithms and source code for more goodies.

11.1 USE OF OUTSIDE RESOURCES

The Sound/Music Lab is a large project. In such work, on the C-128 or other computers, it's often convenient to use one or more outside files to support the main program. On some machines, such as Apple's Macintosh, you can use separate files during development, then easily combine them into one big application when everything's finished. That's a bit tougher, though not impossible, on the C-128. The Sound/Music Lab program uses five files of outside resources: Finger Cursor holds data for a sprite finger cursor; S/M Asm 1 and S/M Asm 2 hold a number of useful assembly language routines called by the main program; S/M Vars contains a number of initialized values for Sound/Music Lab variables; and S/M Help Pack holds twenty two help screen images and associated data. Loading in all those files is the reason the lab takes so long to set itself up.

11.2 THE HA( )ARRAY

Sound/Music Lab uses a number of assembly language routines and tables, and has to remember their addresses. During development, these addresses change frequently. Rather than have them scattered throughout the BASIC code, hard to find when changes are needed, I created an array of integer variables, HA( ), to contain the addresses.

That way all I have to do when one of them changes is adjust its value in a single HA( ) assignment statement. The primary assignment statements for the HA( ) array are located in lines 1550-1650 of Make S/M Vars. And Fig. 11-1 shows the value and assembly language label associated with each of the fourteen HA( ) entries.

11.3 DATA STRUCTURES AND VARIABLES FOR RECORDING SOUND/MUSIC FRAMES

The Sound/Music Lab lets you record a sequence of BASIC 7.0 sound and music commands for later playback and editing. Each recorded command is called a frame. I had to come up with a set of data structures and variables to record the frames.

SOUND/MUSIC LAB HA() Array Information

HA(#)	Label	Qualifier	Hexadecimal	Decimal
HA(0)	Install		$1300	+4864
HA(1)	UnInstall		$1345	+4933
HA(2)	ButnStat		$1B14	+6932
HA(3)	AreaSearch		$1468	+5224
HA(4)	LabAreas	lo-byte	$2F	+47
HA(5)	LabAreas	hi-byte	$16	+22
HA(6)	HlpAreas	lo-byte	$0F	+15
HA(7)	HlpAreas	hi-byte	$19	+25
HA(8)	HRBandInvt		$1512	+5394
HA(9)	HRRectInvt		$14C6	+5318
HA(10)	LabInvRex	lo-byte	$33	+51
HA(11)	LabInvRex	hi-byte	$19	+25
HA(12)	LabAreas		$162F	+5679
HA(13)	Tx40BandInvt		$1553	+5459

Fig. 11-1. Information concerning the HA ( ) array from Sound/Music Lab.

The fundamental data structure I use for recording frames is a stack. The classic metaphor for describing a stack is a pile of plates stored in a spring-loaded plate dispenser. The stack starts out empty. The first plate you add becomes the topmost plate, the top of the stack. Add another plate, and it becomes the topmost plate. Remove a plate, and you get the topmost element in the stack, the last plate added.

Computers lack spring-loaded plate dispensers. One way around this is to use an array to represent the stack, and a separate variable to indicate which element of the array is currently the top of the stack. This is what I do in the Sound/Music Lab.

FD(MD) is an array used to implement a stack of frame data. It can hold up to MD (for Maximum amount of Data) values; as the program is written, MD is set equal to 3000, but that can be changed. TD indicates the top of the stack.

As mentioned in Section 9.6, the lab lets you record seven types of commands: **SOUND, PLAY, ENVELOPE, VOLume, TEMPO, FILTER,** and **jump-to-frame**. Each of these commands stores a different amount of data into FD ( ). Different size plates, as it were. To keep track of each frame's data on the stack, there's the array FF%(MF), where MF is the maximum number of frames (1000 as the program is written, but that can be changed). Every time a frame is recorded, its data is added to FD ( ), and a pointer to that data is stored in FF%. TF is a variable that holds the number of the highest recorded frame; it also functions as a pointer to the topmost valid entry of FF%( ).

Six commands always use the same amount of storage for a frame. The **PLAY** command is different, since it needs to store a play string. The array FS$(MS) holds those strings. MS is the maximum number of strings. When a **PLAY** command is recorded, its string is added to FS$( ), and a pointer to that string is what's stored into FD( ).

DS(7) is an array that holds the size of frame data for each of the seven command types. MF holds the maximum number of frames, which is 1000. FR is the current frame the lab's working on.

11.4 ALGORITHM FOR RECORDING A SOUND/MUSIC FRAME

Here's a summary of what happens when a user chooses to record a sound/music command: The routine handling that particular command prepares data, in the array D( ), that describes the command. D(O) gets a code for the type of command, and other elements of D( ) get the command's data. Then there's a call to the subroutine Record A Sound Frame. Record A Sound Frame works as follows: It first checks to see if there's room to record the command. If there's no room, an appropriate message goes to the user, and the subroutine returns, with a result variable set to failure. If there is room, the elements of D( ) that describe the command get stored on the FD( ) stack. Then a pointer to that data's position in FD( ) is stored in FF%( ). Feedback gets sent to the user. The frame counter gets bumped up a frame, and wraps around if past its maximum. The subroutine returns, with a result variable set to success.

11.5 CREATING THE SPRITE FINGER CURSOR

If you didn't buy the program disk, you need to create a file called "Finger Cursor" that contains the data for a sprite finger cursor. What you do is enter the data with the C-128's built-in sprite editor or built-in monitor, then save the area of memory the data's in as a binary file.

Figure 11-2 shows a sprite coding form for the finger cursor sprite. The finger cursor is a multi-color sprite. It uses pixels colored by transparent screen color, sprite color, and multicolor register 0. I use black for the sprite color and white for multicolor register 0. That way the sprite finger cursor shows up as a white outline of a black hand with a pointing finger.

Refer to the C-128 Prg for instructions on using the built-in sprite editor or built-in monitor to enter the sprite data shown in Fig. 11-2. Note that the figure gives the data in decimal form. Refer to Fig. 11-3 for the same data in hexadecimal. If you use the sprite editor, be sure you design the sprite as sprite 1. If you use the monitor, enter the data starting at $0E00 (3584), the pre-defined position for sprite 1's data.

Leave the editor or monitor after the data's entered. Then give this command to save the data as a binary file:

```
BSAVE "FINGER CURSOR", BO, P3584 TO P3648
```

11.6 EVENT-DRIVEN PROGRAMMING

The key paradigm for programs like Sound/Music Lab is that they're event-driven. That is: the program waits around for something to happen, then reacts to that event.

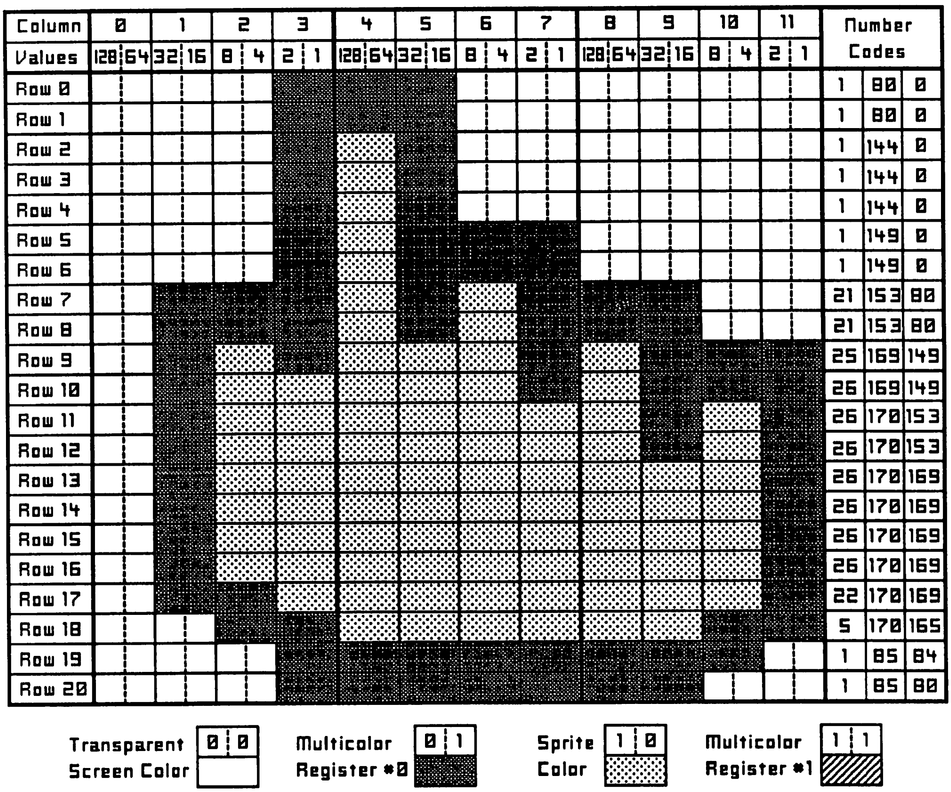


Fig. 11-2. Sprite coding form for the Sound/Music Lab's sprite finger cursor.

Finger Cursor Sprite Data In Hexadecimal

```
$01 $50 $00
$01 $50 $00
$01 $90 $00
$01 $90 $00

$01 $90 $00
$01 $95 $00
$01 $95 $00
$15 $99 $50

$15 $99 $50
$19 $A9 $95
$1A $A9 $95
$1A $AA $99

$1A $AA $99
$1A $AA $A9
$1A $AA $A9
$1A $AA $A9

$1A $AA $A9
$16 $AA $A9
$05 $AA $A5
$01 $55 $54

$01 $55 $50
```

Fig. 11-3. The Sound/Music Lab's sprite finger cursor's data in hexadecimal form.

The heart of such programs is a main event loop. In Sound/Music Lab, that loop is the subroutine Lab Event Loop. That subroutine looks for a pseudo-mouse click. If there is one, it figures out where the click occurred. If the click was in an active area of the lab screen, program control jumps to a subroutine handling clicks in that area. Those subroutines deal with subsequent events, using their own event loops, until the user takes an action ending their control. Control then returns to Lab Event Loop.

11.7 POSITIONING THE SPRITE ON LAB/HELP SCREEN JUMPS

When the user clicks the Sound/Music Lab's HELP button, the lab disappears and a help screen appears. The program takes the liberty of repositioning the sprite finger cursor so it's hot spot is centered in the help screen's **LAB** button. That way the user can simply click the pseudo-mouse to get back to the lab. When the user returns to the lab, the program shifts the sprite finger cursor so it's hot spot is back inside the **HELP** button.

11.8 MODE AVOIDANCE

The Sound/Music Lab tries to avoid modes. That is: when you're doing one thing, you can easily leave it and do something else. For example, you can be working in the **SOUND** window, then click in the **ENVELOPES** window; **SOUND** is deactivated, and you get to work in **ENVELOPES** without the program making a protest. This behavior is implemented in each lab window's subroutine's event loop. For example, take a look at line 6220 of Sound/Music Lab.

Why avoid modes? Modes confuse users. The user wants to do something they know can be done, because they've done it before, but the program won't let them do it because it's not in the correct mode. Avoid modes, keep your users happy, and lessen confusion in today's complex world.

11.9 HEAVY MODULARITY AND USE OF SUBROUTINES

As already noted many times, the Sound/Music Lab is a very large program. And yet it's essentially bug-free. How can you write such a large project, with so much error-prone assembly language, and get it to work so well? Heavy modularity, achieved through a liberal use of well-designed subroutines. Each subroutine should perform a well-defined task. It shouldn't be too large; if its task is complex, break it into sub-tasks, making each sub-task a subroutine. You can see this throughout the source code for Sound/Music Lab and its support programs.

11.10 VARIABLE INITIALIZATION VIA DATA STATEMENTS AND RESTORES

The Sound/Music Lab has to initialize a lot of variables. In another attempt to increase robustness via modularity, I prefer to use **DATA** statements for mass initialization. There can be a problem with **DATA** statements, however: making sure the right **DATA** statement gets read. The solution is the use of the **RESTORE** statement followed by the line number of the **DATA** statement you want read. You can see examples of this throughout Make S/M Vars and the initialization sections of Sound/Music Lab.

11.11 EDITORS

The Commodore computers have always had the best built-in text editors of any personal computer, and the C-128 is no exception. But the Sound/Music Lab needed more specialized editing capabilities. So I developed a package of assembly language editing routines, headed up by **StrngRecEdit**. They're located in S/M Asm 1. **StrngRecEdit** is called when the user's entering data in a Sound/Music Lab window. It lets the user type characters, use the insert and delete keys, use the cursor keys, and use the joystick-controlled mouse to move the cursor. All this happens within a specified rectangle, with wraparound on all sides. One little thing it does different than the Commodore editor: after pressing the insert key, you can do other things than just type in characters.

40C Edit is another package of assembly language editing routines. It's called on by Make 40C Screens. It lets you create and edit 40-column screens. Many of the routines are similar to those used in StrngRecEdit.

Both of these editing packages work through a main event loop. The loop waits for the user to type something, acts accordingly, then returns to the loop if the user's action didn't signify the end of the editing process.

11.12 MOVING THE CURSOR WITH A PSEUDO-MOUSE

As mentioned in Section 11.11, StrngRectEdit lets you use the Sound/Music Lab's pseudo-mouse to position an editing cursor. How's it done?

If StrngRectEdit's event loop picks up a press of the pseudo-mouse, it calls on the routine :DealMouse. :DealMouse starts by calling on AreaSearch (from S/M Asm 2) to see where the p-m click occurred. If the click occurred outside the rectangular StrngRectEdit's working on, :DealMouse returns with a result code indicating it's time to end the editing session. If the click occurred inside the editing rectangle, there's a call to :InvertCursor to un-invert the editing cursor. Then there's a calculation to convert the pseudo-mouse location to text screen (40H × 25V) coordinates. Refer to the source code for details of this conversion. The new coordinates are adjusted so that they remain within the editing rectangle. Then the cursor is moved to that spot. :DealMouse returns with a result code indicating editing should continue. The top of the event loop inverts the cursor, which is at its new position.

11.13 INPUT FILTERING

A program should never crash because the user enters unreasonable data. Like a patient friend, it should deal gracefully with the user's mistake.

I try to live up to this ideal in the Sound/Music Lab. When you enter a bad value into one of the Sound/Music Lab's windows, you'll get a beep and a hopefully helpful error message. The program won't crash.

Example: The Sound/Music Lab has a routine, Fetch A Parameter, that handles numeric data entry. Fetch A Parameter calls on StrngRectEdit to fetch a value. When StrngRectEdit returns, the fetched value gets checked to see if it's within range for that particular parameter. An array of parameter bounds, PF( ), is used for the checking; take a look at the Fetch A Parameter source code, line 15250 in particular.

11.14 ASSEMBLY LANGUAGE TABLES

Tables are used throughout the Sound/Music Lab project. Using tables in your code encourages flexibility, generality, and mutational speed. I'd like to point out some of the more interesting ones that occur in some of the project's assembly language code:

WherTab (lines 352-374 of S/M Asm 1 C.S)
: This is a table of pointers that tells the :StorStuf routine where to stick various items when it unpacks a parameter block sent to StrngRectEdit

:TstCodz (lines 352-374 of S/M Asm 1 C.S)
: This is a table of keycodes for keys whose presses OurKeyChk wants to hide from the C-128 system. The keys are the four upper cursor movement keys and the return key.

:**DirTab** (lines 531-536 of S/M Asm 2 A.S)
This table is used to translate raw joystick and upper cursor key data, both of which have values in the range 0..15, into a direction code.

:**MDLo** and :**MDHi** (lines 540-556of S/M Asm 2 A.S)
These tables are used to translate a direction code, in the range 0..7, into a pointer to a set of sprite motion data tied to that direction. See the description of the **North, NorthEast,** etc. tables.

HRRowsLo, Rows4025Lo, HRRowsHi, and **Rows4025Hi** (lines 18-50 of S/M Asm 2 C.S)
These tables are used to convert a text screen row number, in the range 0..24, to pointers to the first byte in that row.

LabAreas (lines 66-698 of S/M Asm 2 C.S)
This table gives the rectangular boundaries, in bit map screen 320h by 200v coordinates, and an identification code for each area of the Sound/Music Lab screen where a pseudo-mouse click is significant. When a click occurs in the lab, this is the table that's searched to see where it happened.

HlpAreas (lines 711-742 of S/M Asm 2 C.S)
This table gives the rectangular boundaries, in bit map screen 320h by 200v coordinates, and an identification code for each area of a Sound/Music Lab help screen where a pseudo-mouse click is significant. When a click occurs in the help screen, this is the table that's searched to see where it happened.

LabInvRex (lines 759-1073 of S/M Asm 2 C.S)
This table gives the boundaries, in text screen 40h by 25v coordinates, for inversion rectangles assigned to each area of the lab described in the **LabAreas** table.

LabInvRex (lines 759-1073 of S/M Asm 2 C.S)
This table gives the boundaries, in text screen 40h by 25v coordinates, for inversion rectangles assigned to each area of the lab described in the LabAreas table.

North, NorthEast, East ... West, NorthWest (lines 1078-1155 of S/M Asm 2 C.S)
These are tables of sprite motion data, one for each of 8 primary directions, that, when plugged into the sprite speed and direction tables described in Appendix G, produce sprite motion in a particular direction.

11.15 DISPLAYING VARIABLE-LENGTH STRINGS IN A FIXED-LENGTH AREA

Error messages come in various sizes. The Sound/Music Lab has an error message window that's got a fixed size: 16 characters wide. So how can we display error messages that may be longer than that? There are two possible solutions: The first, and the way we do it in Sound/Music Lab, is to chew off pieces of the error message that'll fit in the window, and show one piece after another until the entire message has been displayed. See lines 16090-16190 of Sound/Music Lab. The second, and more elegant way, is to write a routine that scrolls the message across the window, like the moving message on that building in New York's Times Square. Such a routine should be written in assembly language; I just didn't have the time or space to do it for this project.

Chapter 12: Stretching

There's not enough room to provide complete solutions here, but I can give some strong hints.

12.1 CHANGING MAKE 40C SCREENS SO IT WORKS WITHOUT AN 80-COLUMN MONITOR

As written, Make 40C Screens uses an 80-column monitor as a control screen. You can rewrite the program to use a 40-column control screen. Possibly the simplest way is to use the bit-mapped screen for control operations. Wherever a command in Make 40C Screens involves the 80-column screen, just replace it with an equivalent bit-map screen command. Remember, you print text on the bit-map screen via the **CHAR** command.

For example, line 1340 would be changed from Graphic 5,1 to Graphic 1,1. And line 1990 would be changed from PRINT "BAD CHOICE" to something like **CHAR** 1, 10, 16, "BAD CHOICE".

12.2 CHANGING MAKE 40C SCREENS SO IT SAVES 40-COLUMN SCREENS WITH COLOR INFORMATION

When Make 40C Screens saves a 40-column text screen, it only saves the 1000 memory locations holding poke codes. The simplest way to change it so it saves color information is to add a line that saves color RAM as a companion binary file. What to name this companion file? Well, possibly the simplest thing to do is add a suffix to the name the user chooses for the file of text information. For example, if the user chose to save the screen as ''HELP SCREEN 22'', the color information would be saved as ''HELP SCREEN 22C''.

This can be done by adding a line like this to Make 40C Screens:

```
2425 BSAVE ("@" + NM$ + "C"), U(DN), P55296 TO P56296
```

One more detail: Since an extra character is added to NM$, you have to make sure your name has 15 characters or less. You might want to add a length filter of some sort to the Fetch File Name And Device Number subroutine.

12.3 MORE VISUALS

Here are a number of ways you might make the lab more visual. No time here for many code hints, but I can make some design suggestions.

It would be nice if the Sound/Music Lab provided some visual representation of the individual frames. Then, as recording and playback occurred, a display of frame representations could scroll by. This scrolling visual frame display would provide another channel of information for the user, concretizing their work in the lab.

The C-128 has some hardware features that aid such scrolling. But they don't help if you're just scrolling one small area of the screen. So you'd have to write some scrolling routines. Things will be easier if you make the scrolling display go from one side of the screen to the other.

How to represent a frame's command? Well, you might develop a distinctive icon for each of the seven Sound/Music Lab commands. Different colors can go with each icon. You could have a text tag, giving the details of a frame's command, that travels with each icon.

Another idea is to replace the numeric parameter display of some of the windows with graphic controls. For example, rather than typing in a value for the **VOLume** command, you could move a slide switch. Another example: envelopes could be represented as a set of slideable points, connected with lines. The user would grab a point, and just move it up or down.

12.4 MORE SOPHISTICATED PLAYBACK CONTROLS

The **FRAME** command gives the Sound/Music Lab a primitive language capability during playback: it can jump from one frame to another. For example, you can **FRAME** a command at the end of a recording that jumps to the beginning of the recording. But these are unconditional jumps. You could add more sophistication. For example, a recorded **FRAME** command could have a count, so the jump would only occur a certain number of times. Or, a bit more complex, it could adjust one of the values in another recorded command, test it, then jump based on the results of the test.

Chapter 13: Calling Structure Diagrams

This chapter consists of six figures, as follows:

Fig. 13-1—calling structure diagrams for S/M Asm 1 (3 sheets).
Fig. 13-2—calling structure diagrams for S/M Asm 2 (1 sheet).
Fig. 13-3—calling structure diagrams for S/M Help Packer and Make S/M Vars (1 sheet).
Fig. 13-4—calling structure diagrams for Make 40C Screens (1 sheet).
Fig. 13-5—calling structure diagrams for 40C Edit (1 sheet).
Fig. 13-6—calling structure diagrams for Sound/Music Lab (18 sheets).

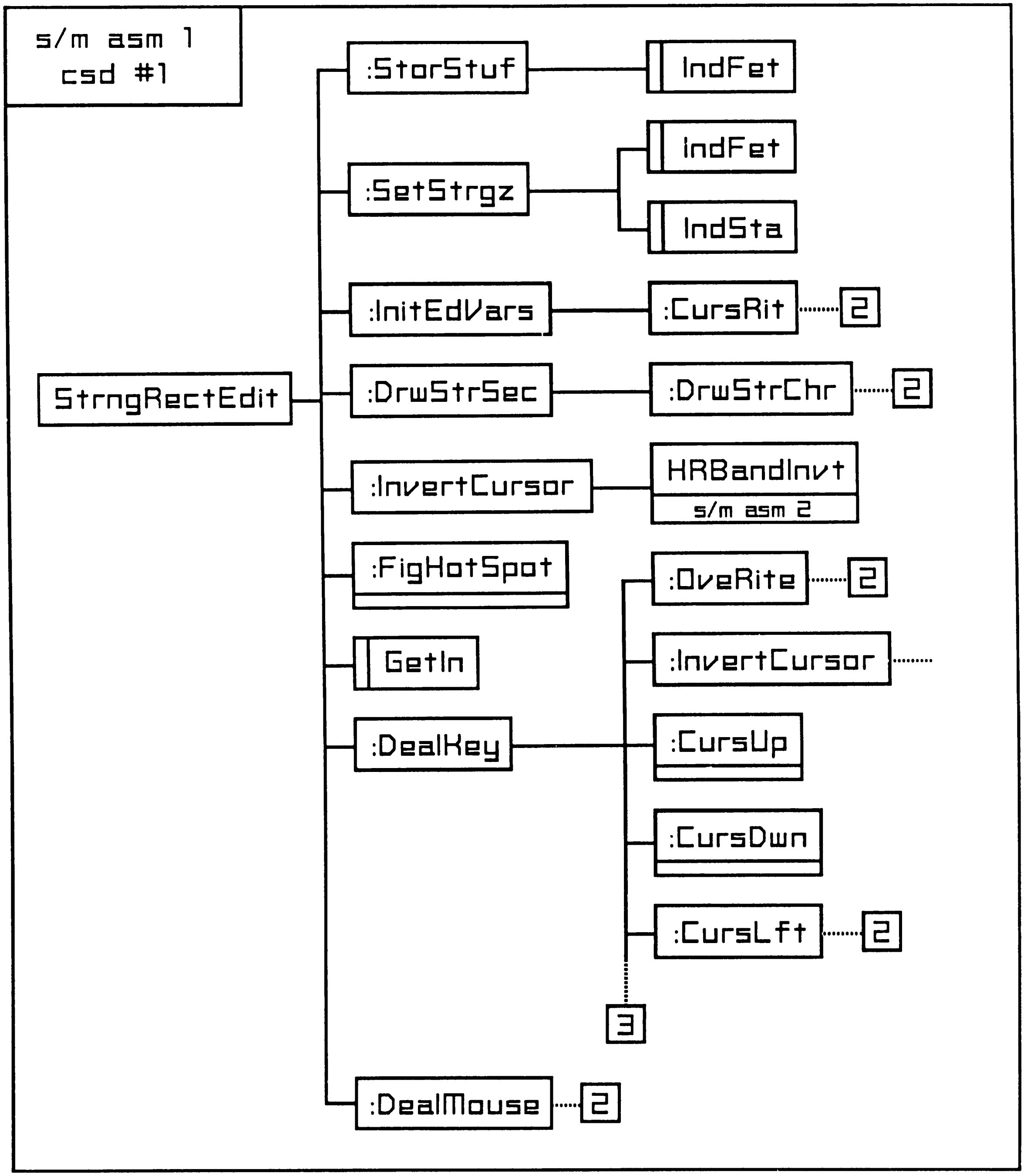
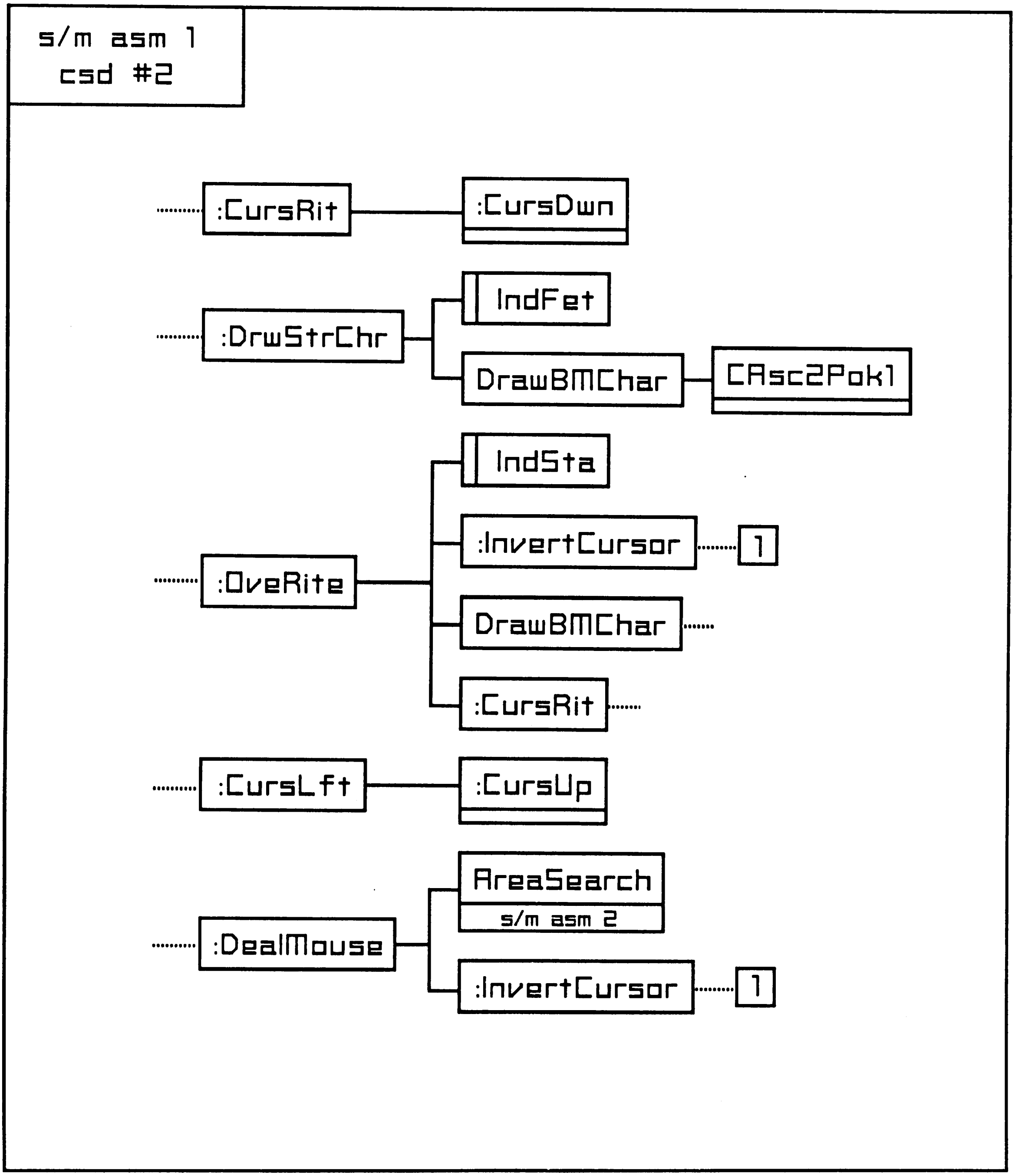
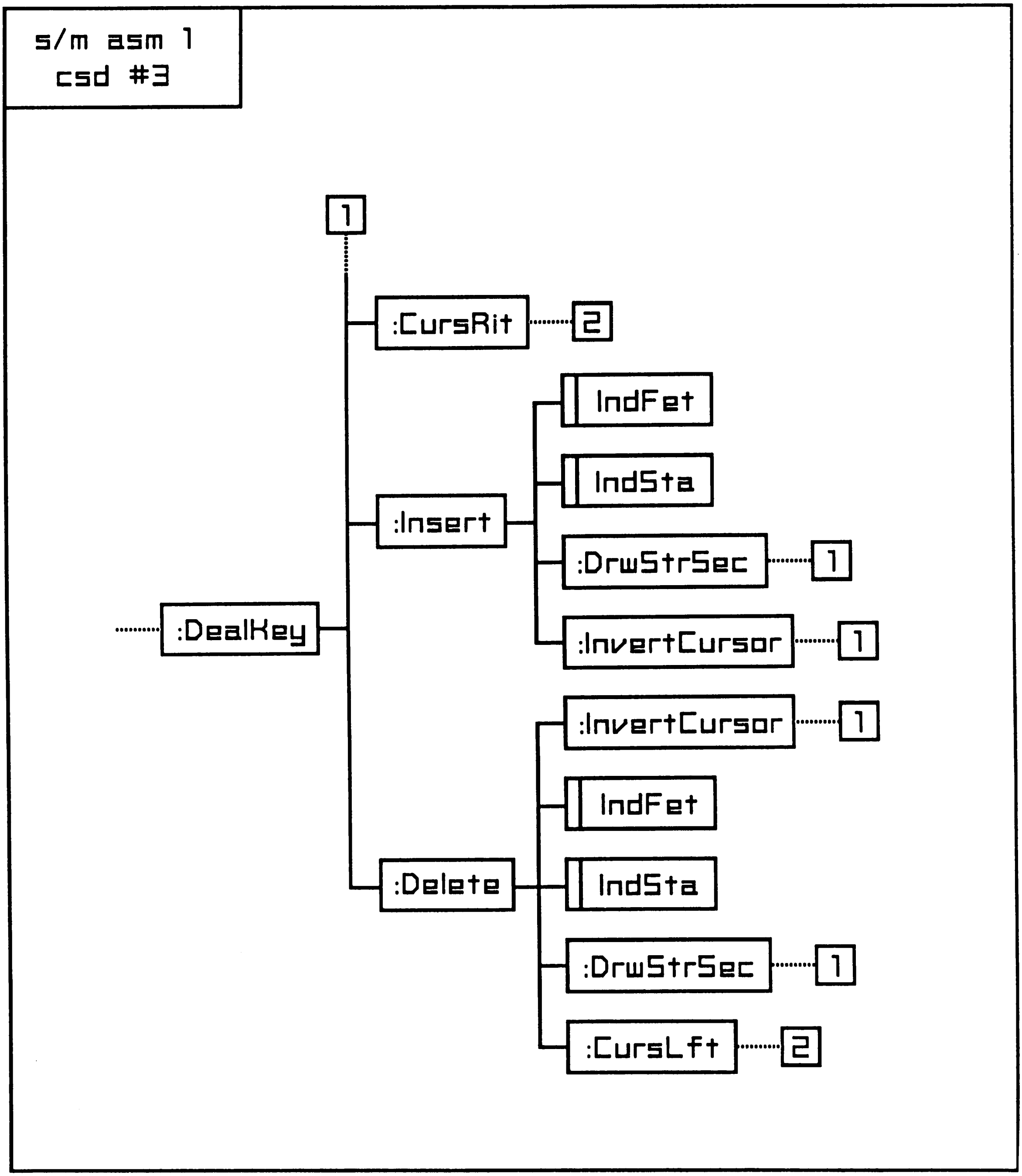


Fig. 13-1. Calling structure diagrams for S/M Asm 1.



s/m asm 1
csd #2
:CursRit
:CursDwn
:DrwStrChr
IndFet
DrawBMChar
CAsc2Pok1
IndSta
:InvertCursor
1
:OveRite
DrawBMChar
:CursRit
:CursLft
:CursUp
AreaSearch
s/m asm 2
:DealMouse
:InvertCursor
1


s/m asm 1
csd #3
1
:CursRit
2
IndFet
IndSta
:Insert
:DrwStrSec
1
:DealKey
:InvertCursor
1
:InvertCursor
1
IndFet
IndSta
:Delete
:DrwStrSec
1
:CursLft
2

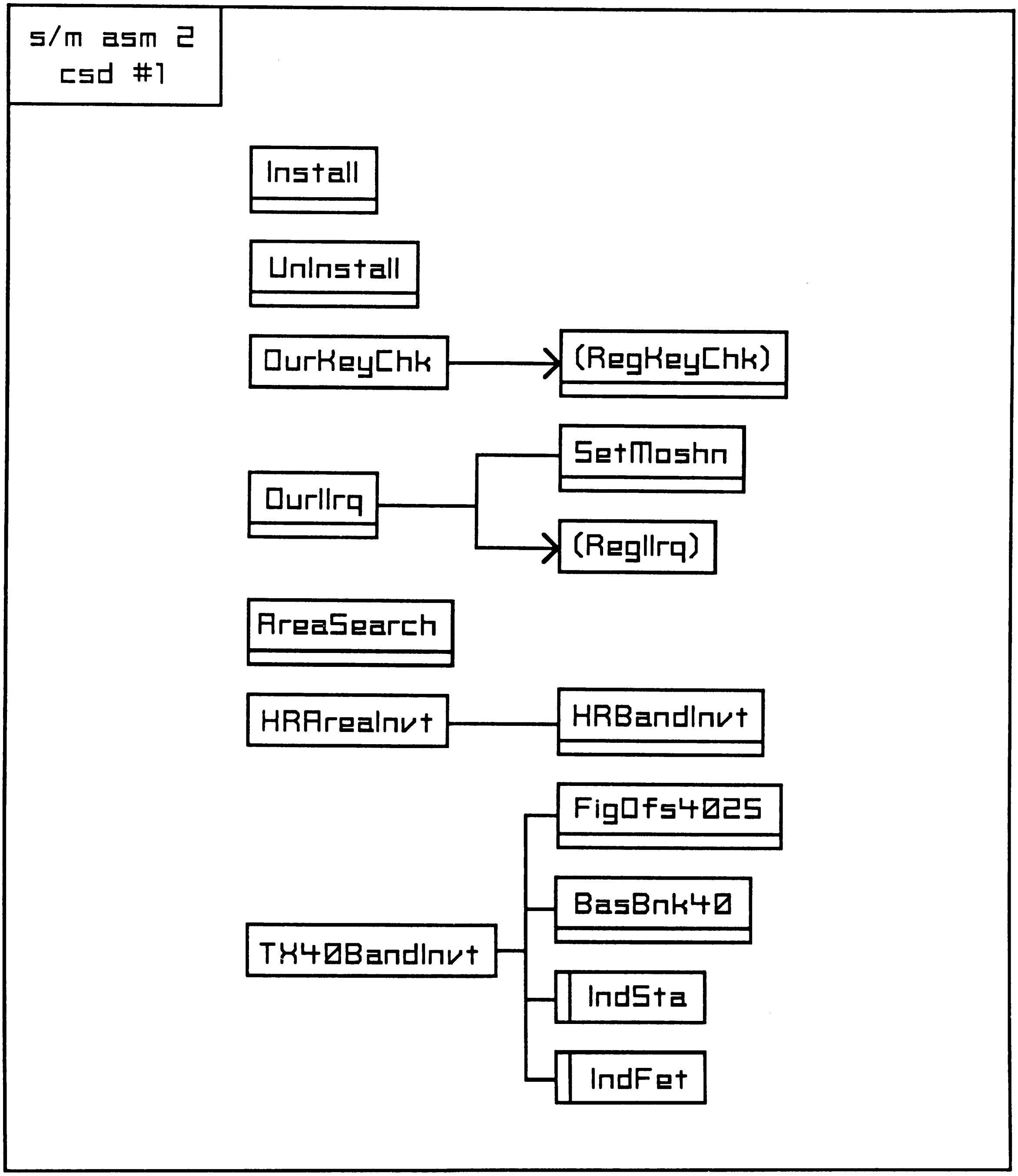
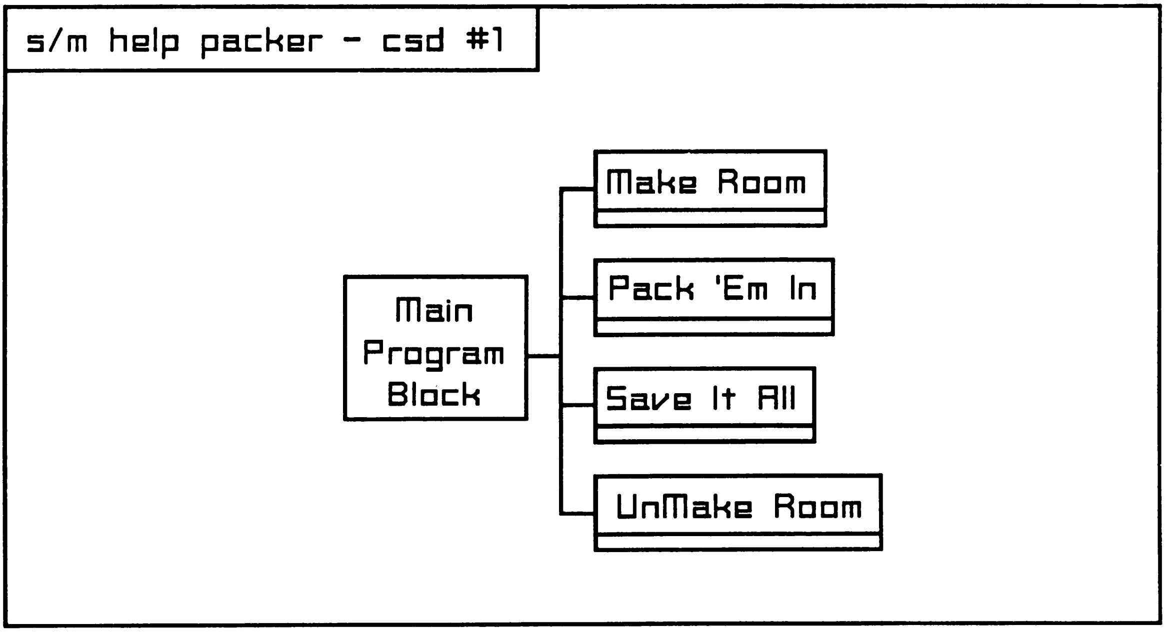


Fig. 13-2. Calling structure diagram for S/M Asm 2.



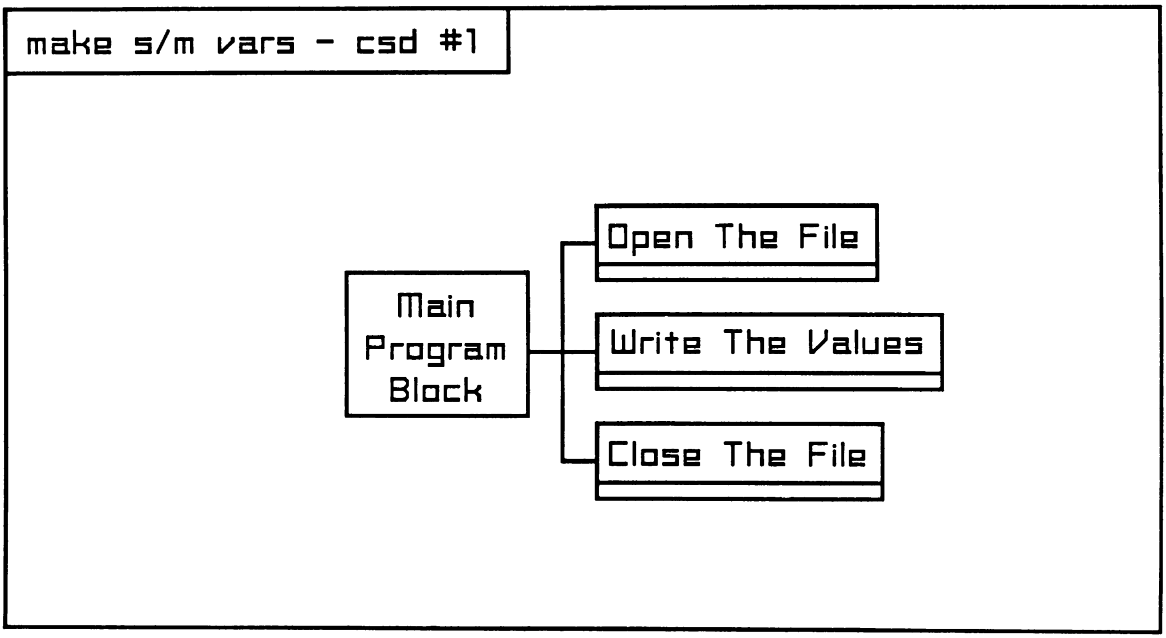


Fig. 13-3. Calling structure diagrams for S/M Help Packer and Make S/M Vars.

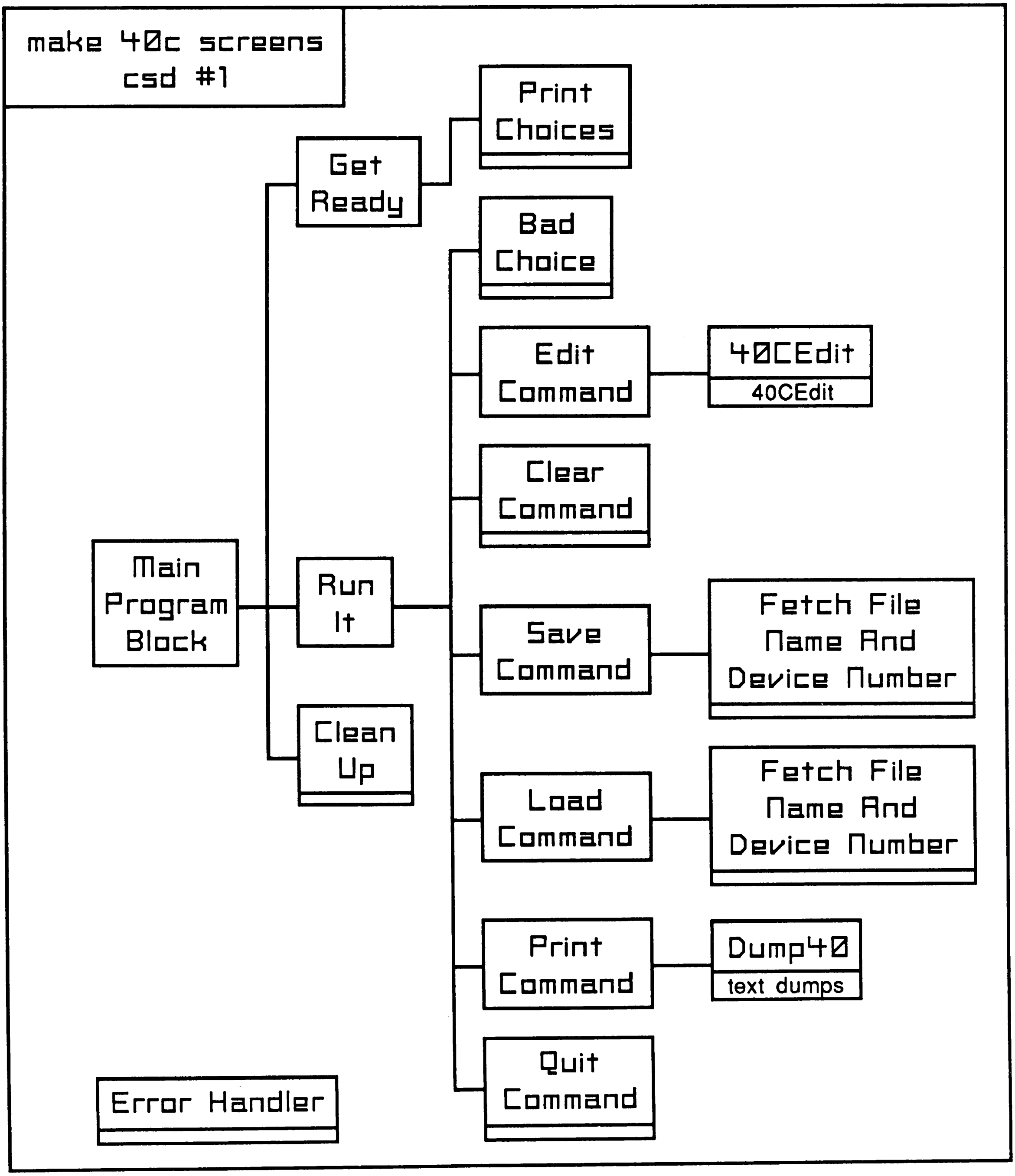


Fig. 13-4. Calling structure diagram for Make 40C Screens.

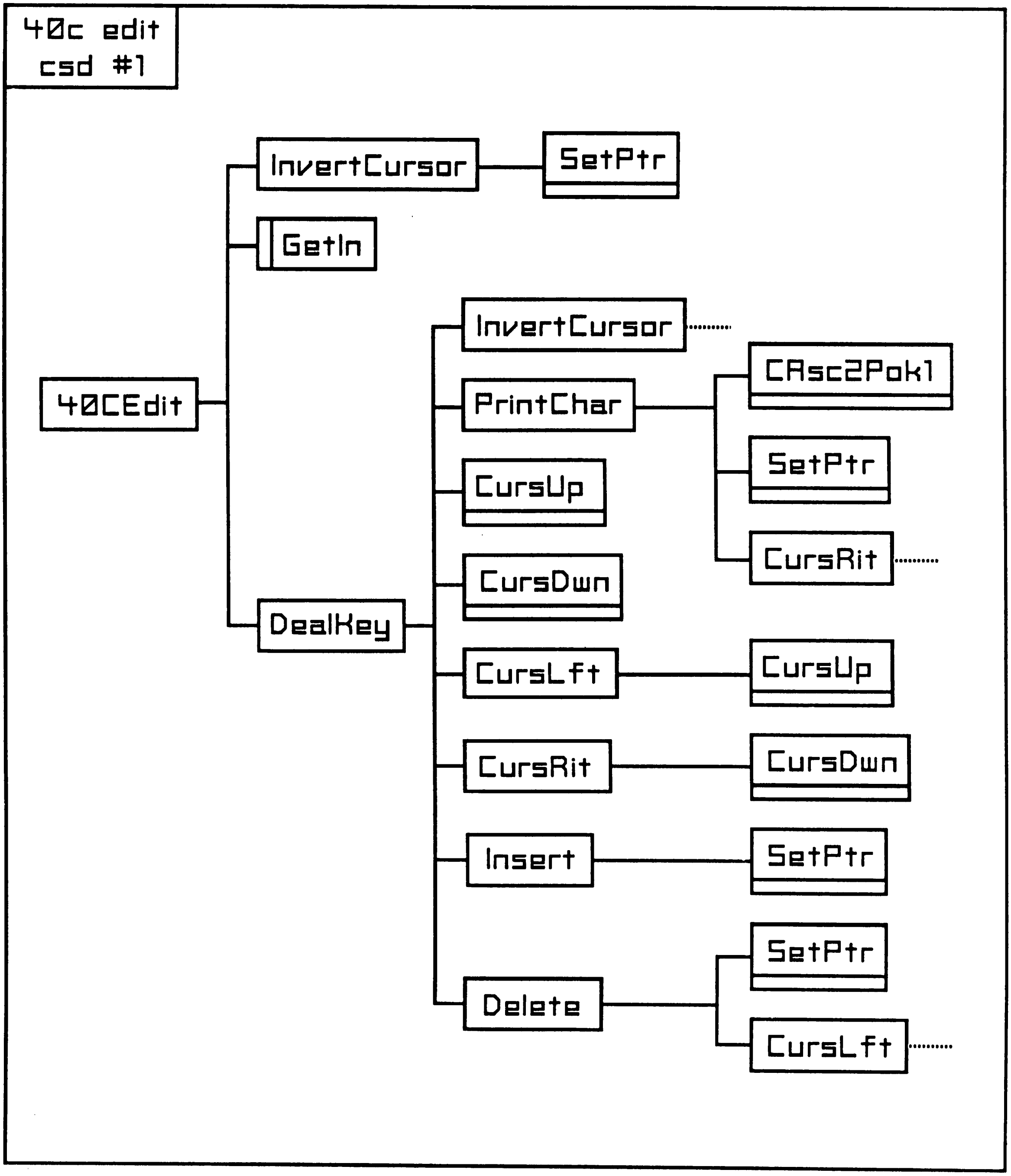


Fig. 13-5. Calling structure diagram for 40C Edit.

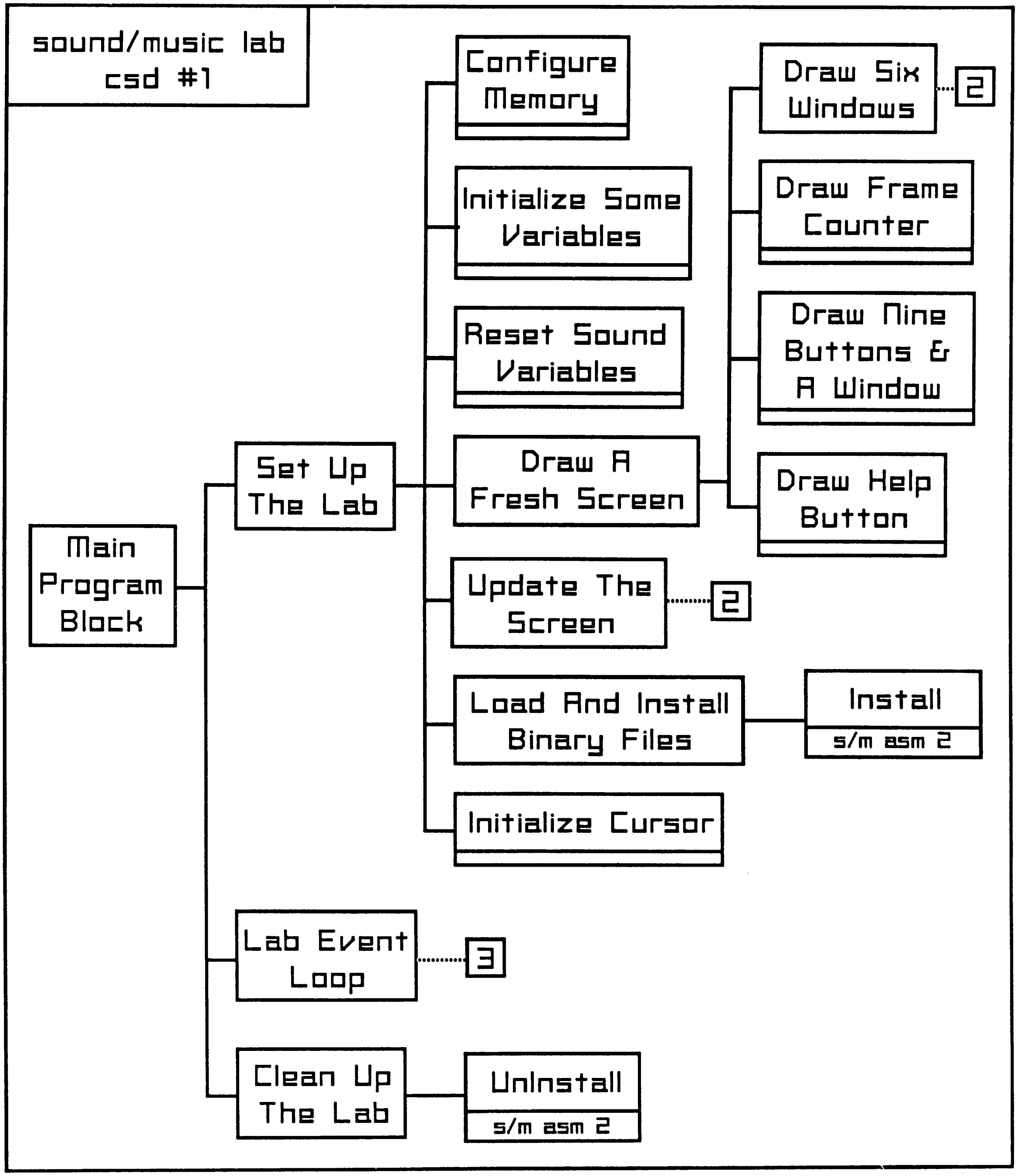
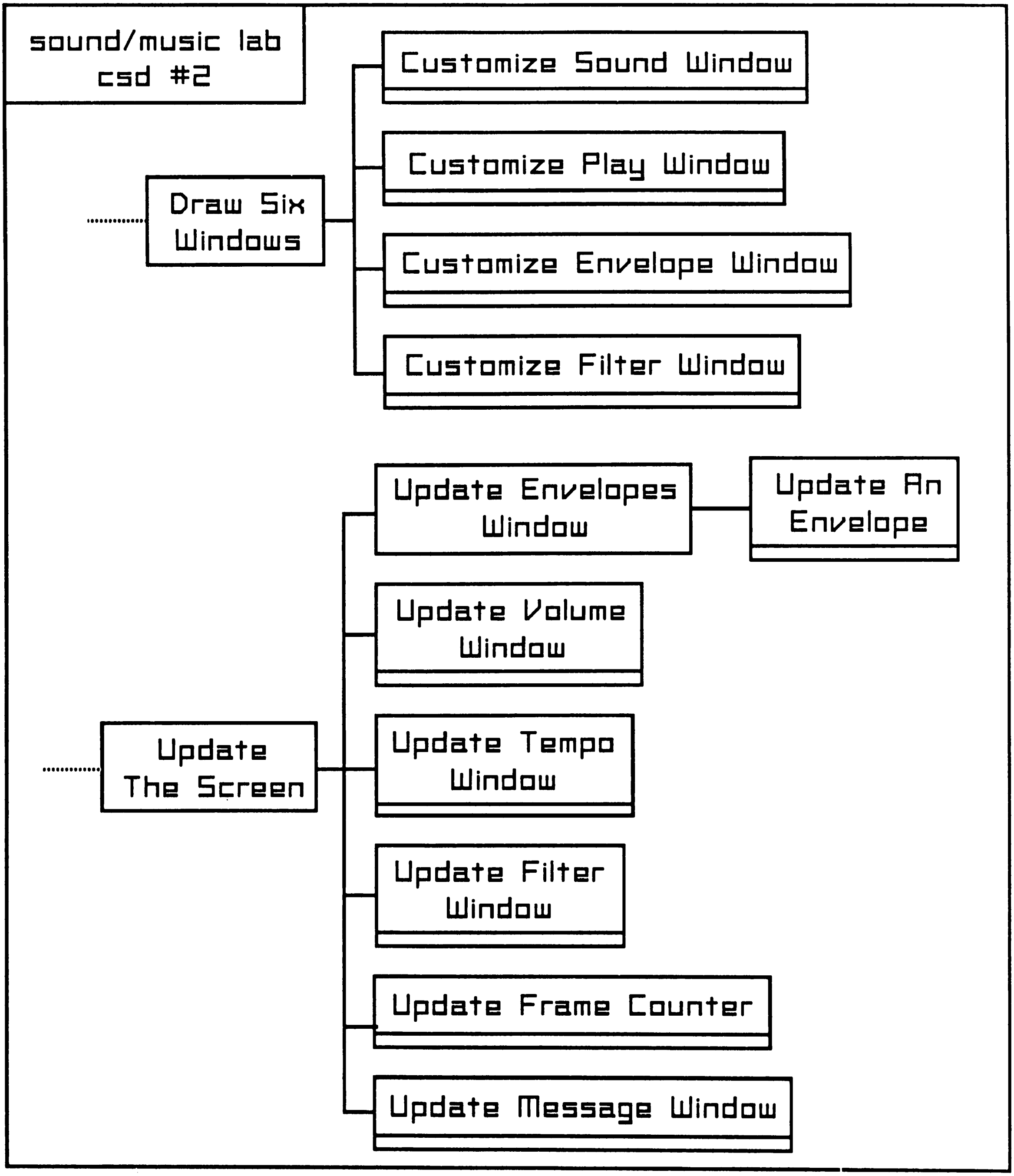
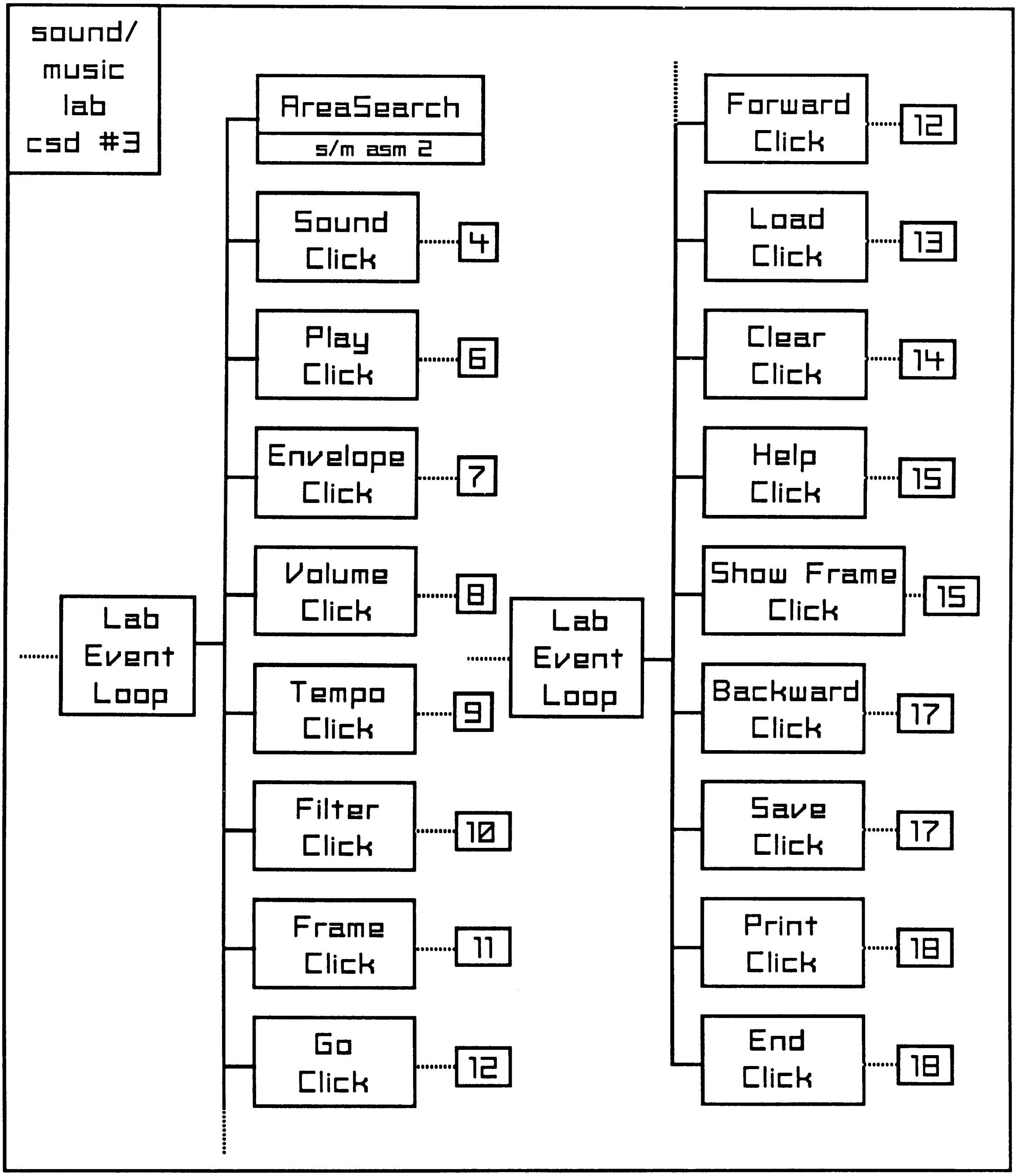
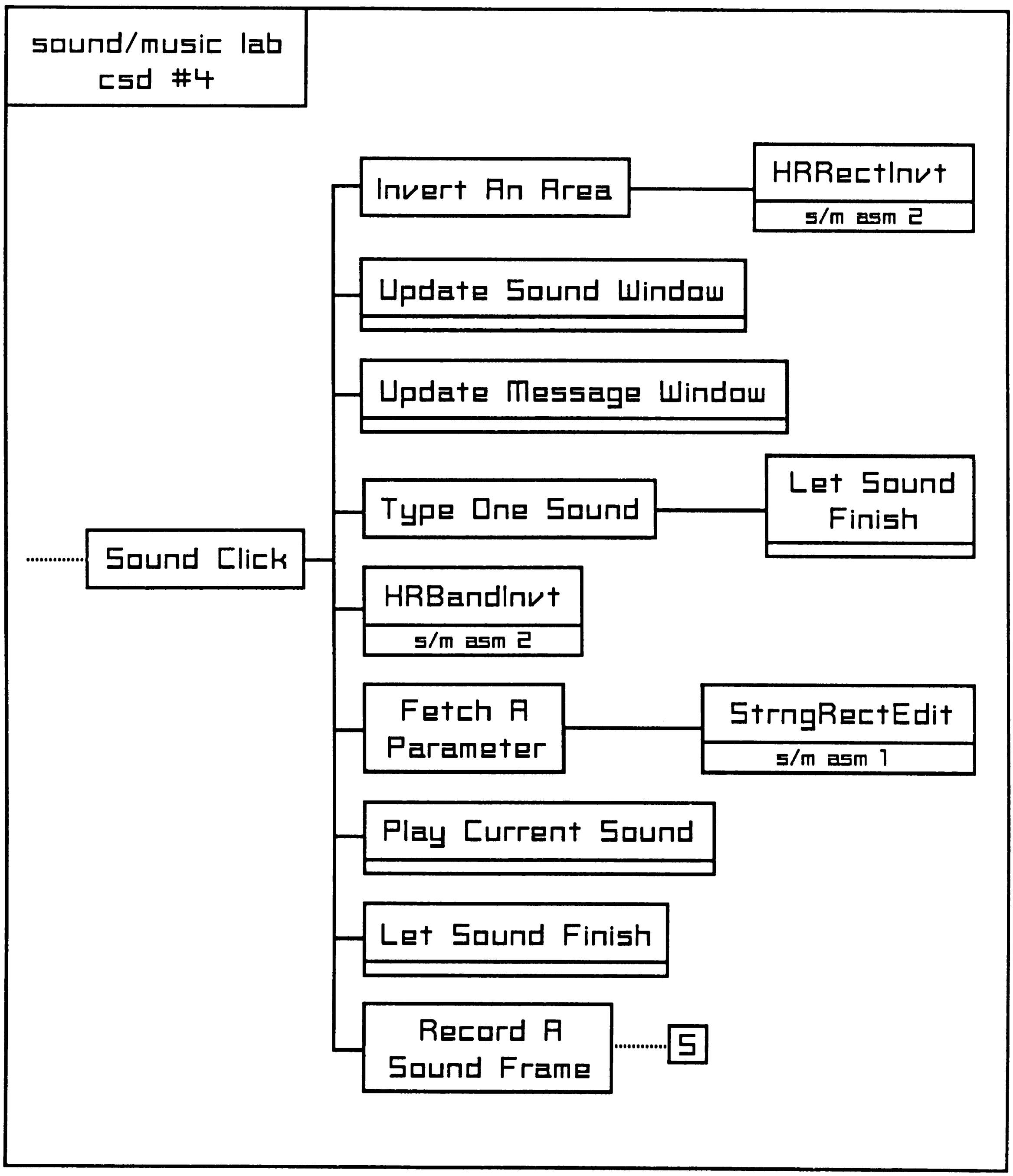
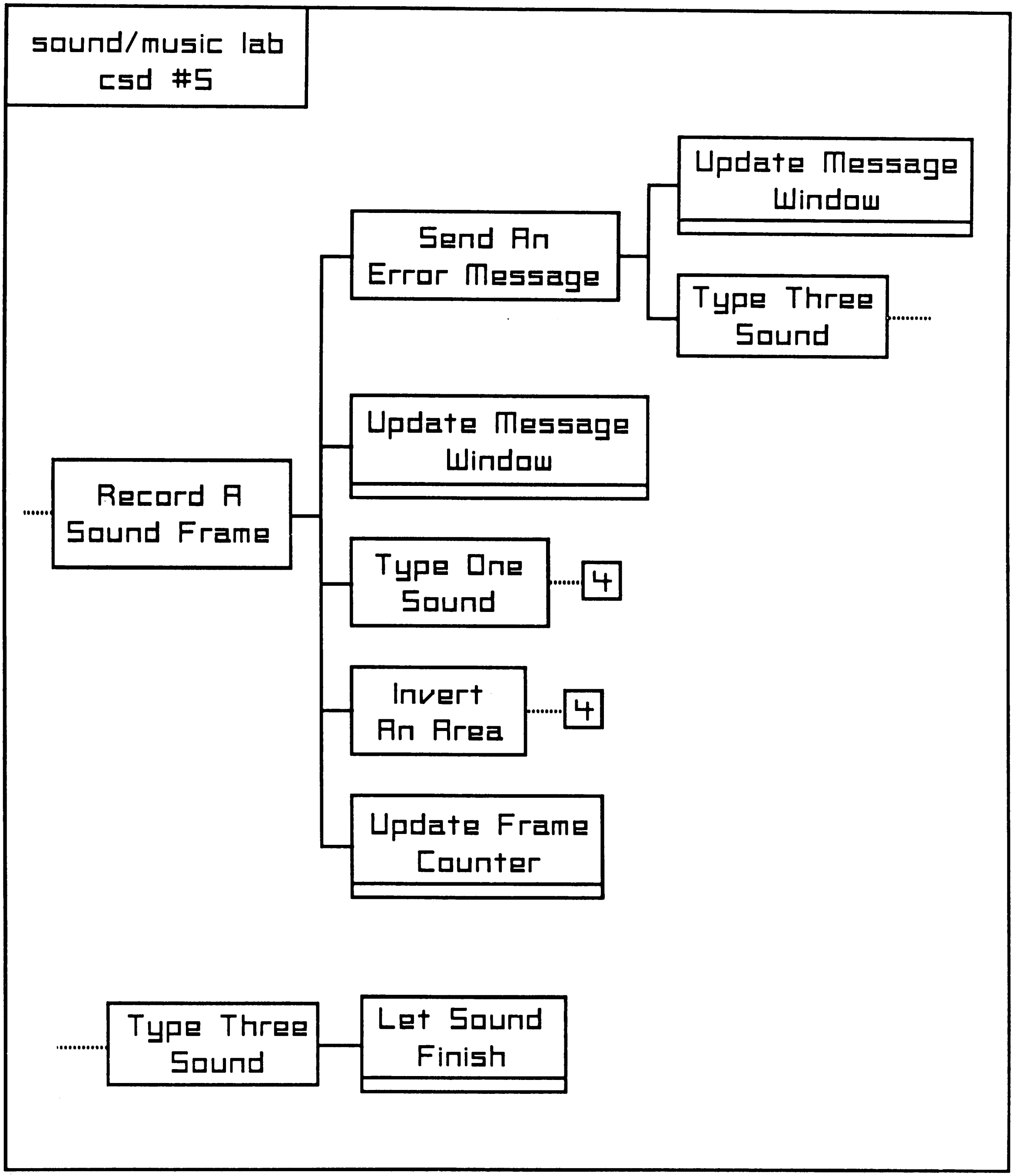


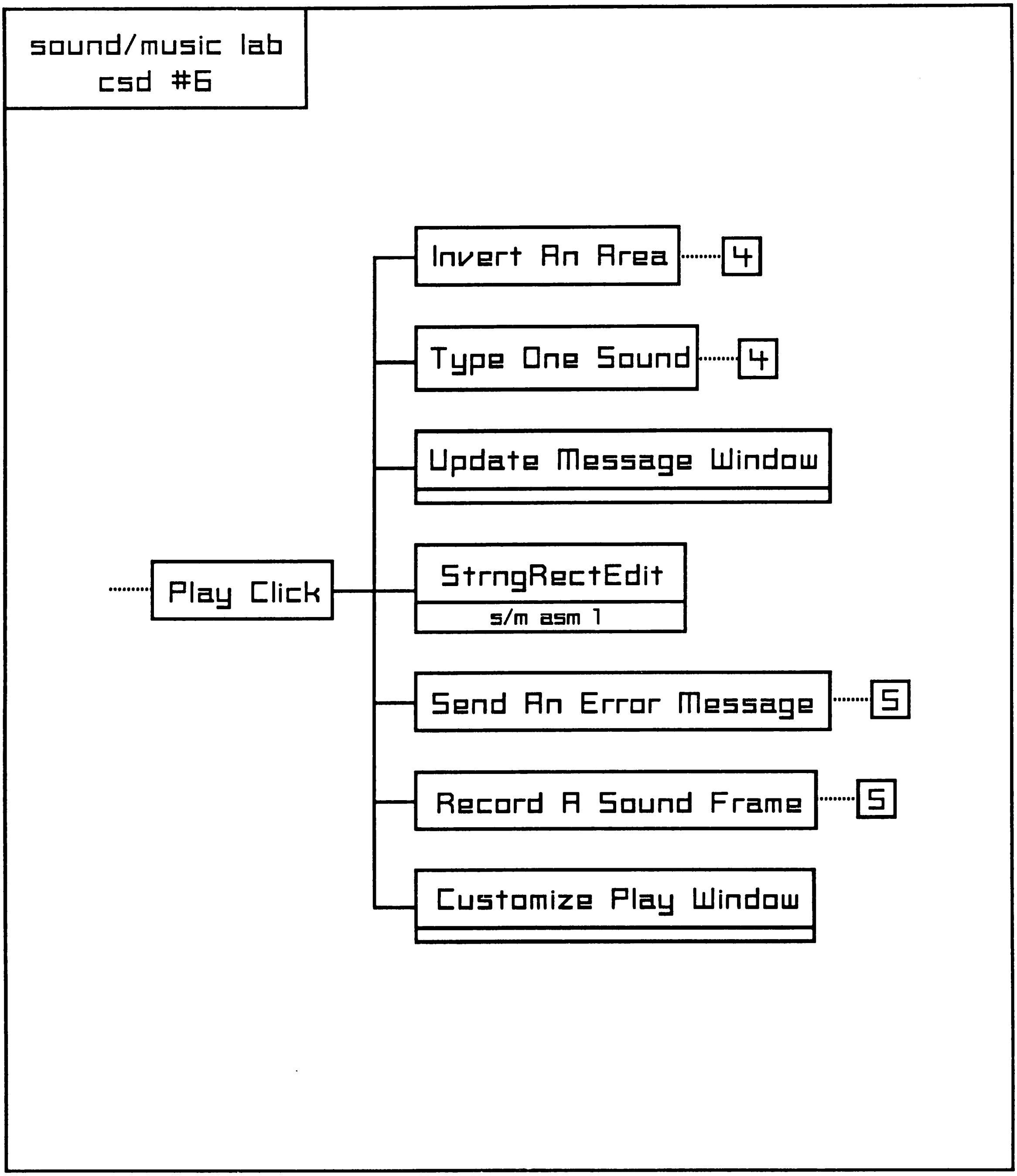
Fig. 13-6. Calling structure diagrams for Sound/Music Lab.

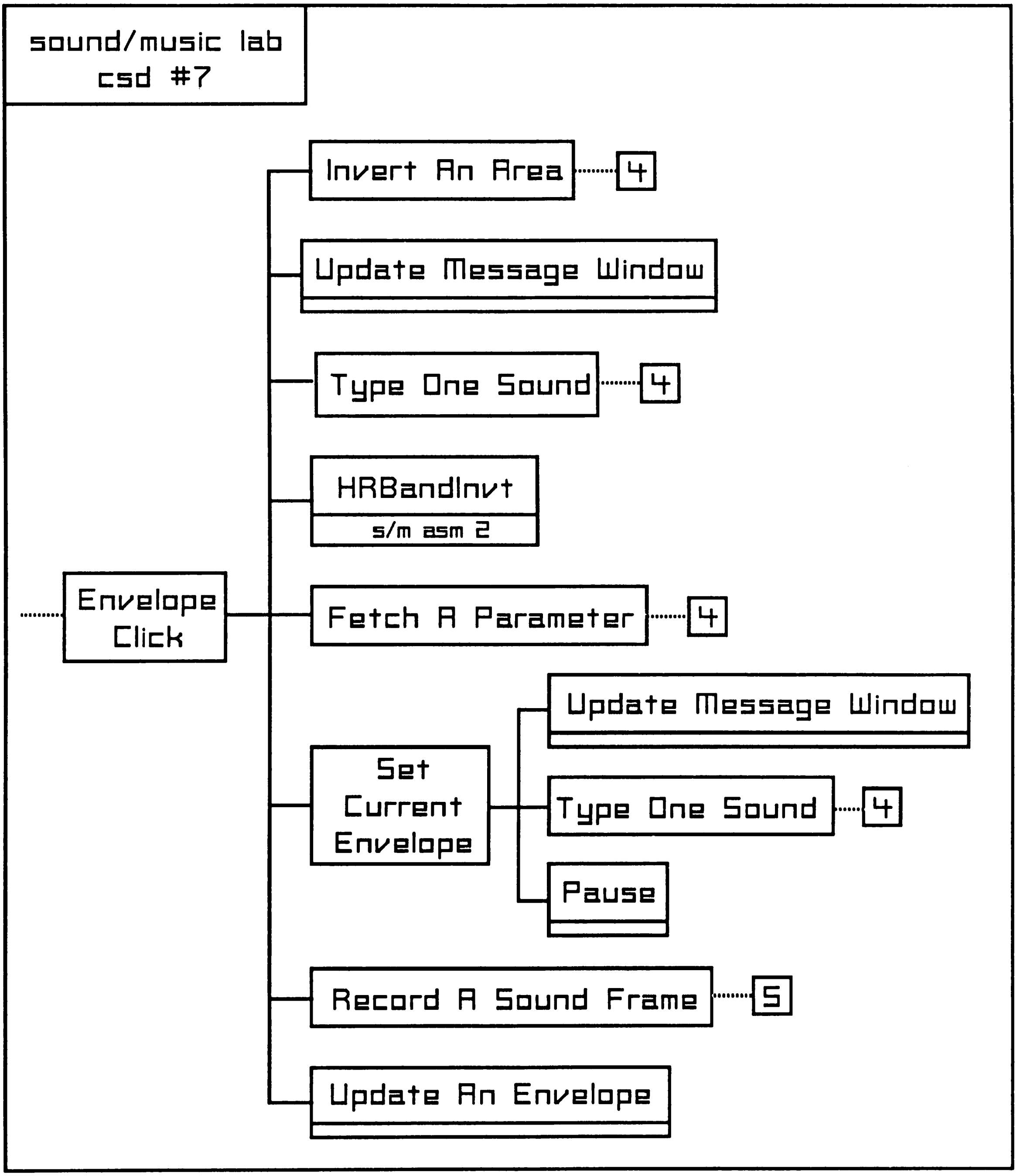

sound/music lab
csd #2
Draw Six Windows
Customize Sound Window
Customize Play Window
Customize Envelope Window
Customize Filter Window
Update The Screen
Update Envelopes Window
Update An Envelope
Update Volume Window
Update Tempo Window
Update Filter Window
Update Frame Counter
Update Message Window

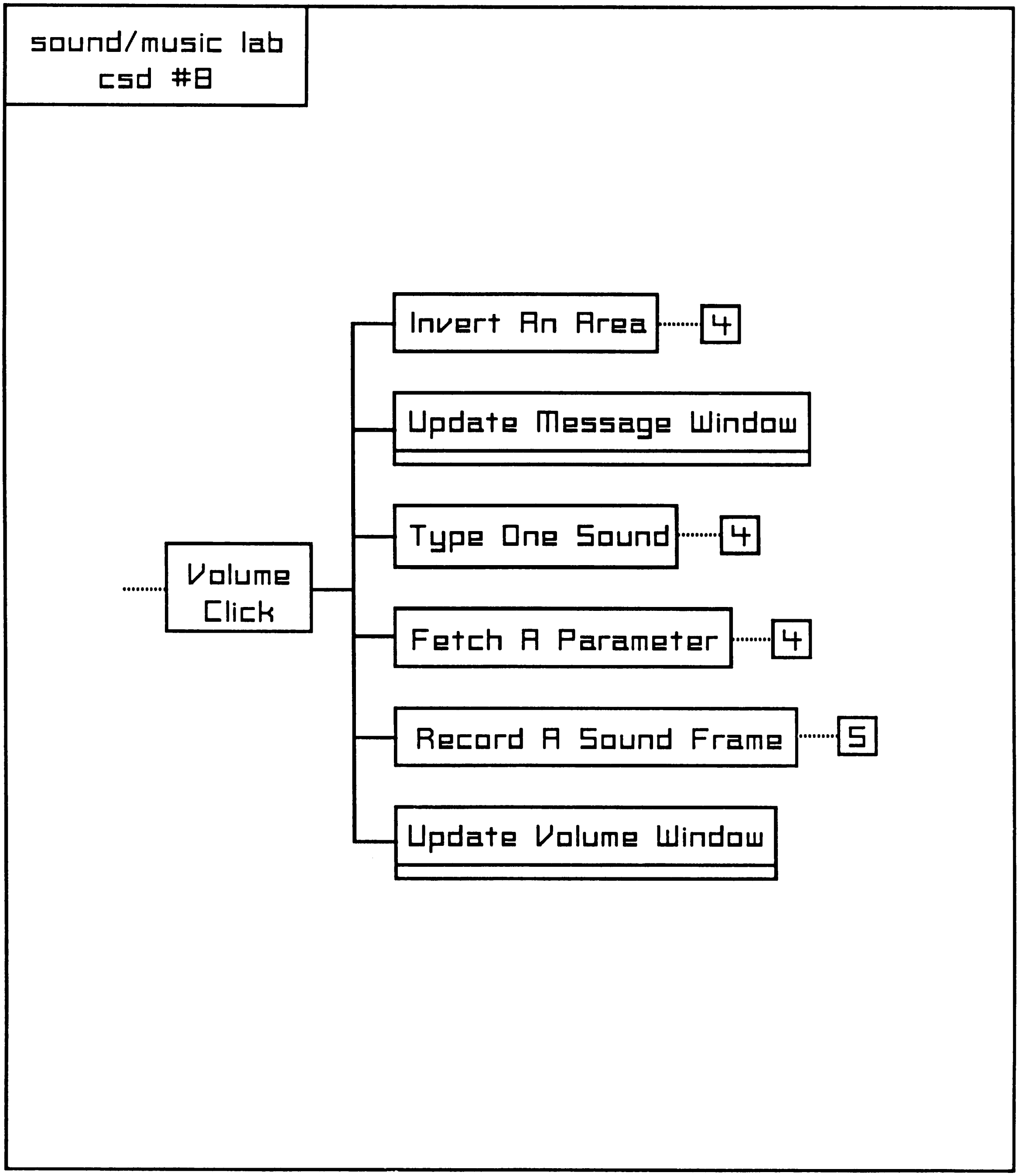

sound/ music lab csd #3
Lab Event Loop
AreaSearch
s/m asm 2
Sound Click
4
Play Click
6
Envelope Click
7
Volume Click
8
Tempo Click
9
Filter Click
10
Frame Click
11
Go Click
12
Lab Event Loop
Forward Click
12
Load Click
13
Clear Click
14
Help Click
15
Show Frame Click
15
Backward Click
17
Save Click
17
Print Click
18
End Click
18

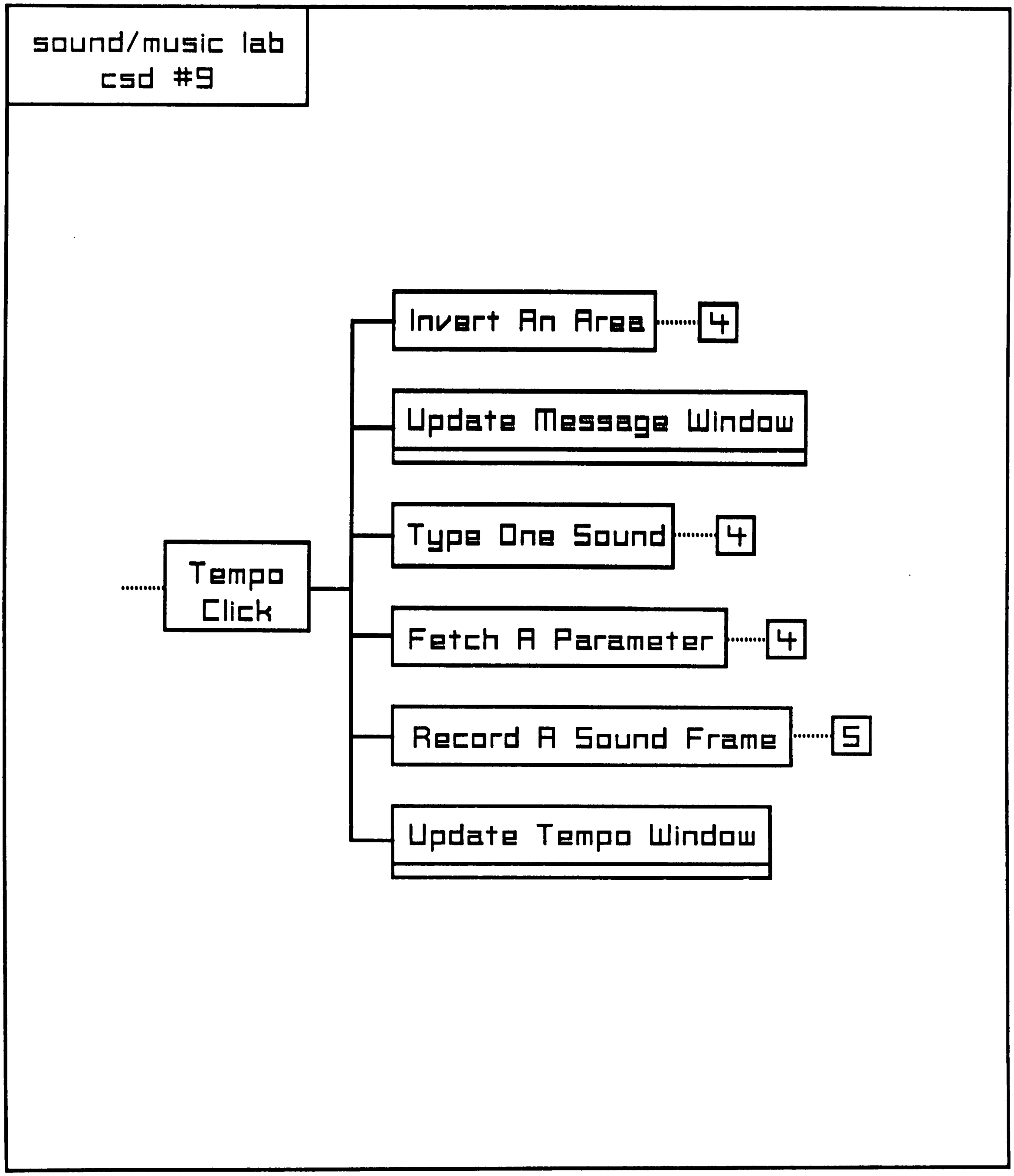

sound/music lab
csd #4
Sound Click
Invert An Area
HRRectInvt
s/m asm 2
Update Sound Window
Update Message Window
Type One Sound
Let Sound Finish
HRBandInvt
s/m asm 2
Fetch A Parameter
StrngRectEdit
s/m asm 1
Play Current Sound
Let Sound Finish
Record A Sound Frame
5

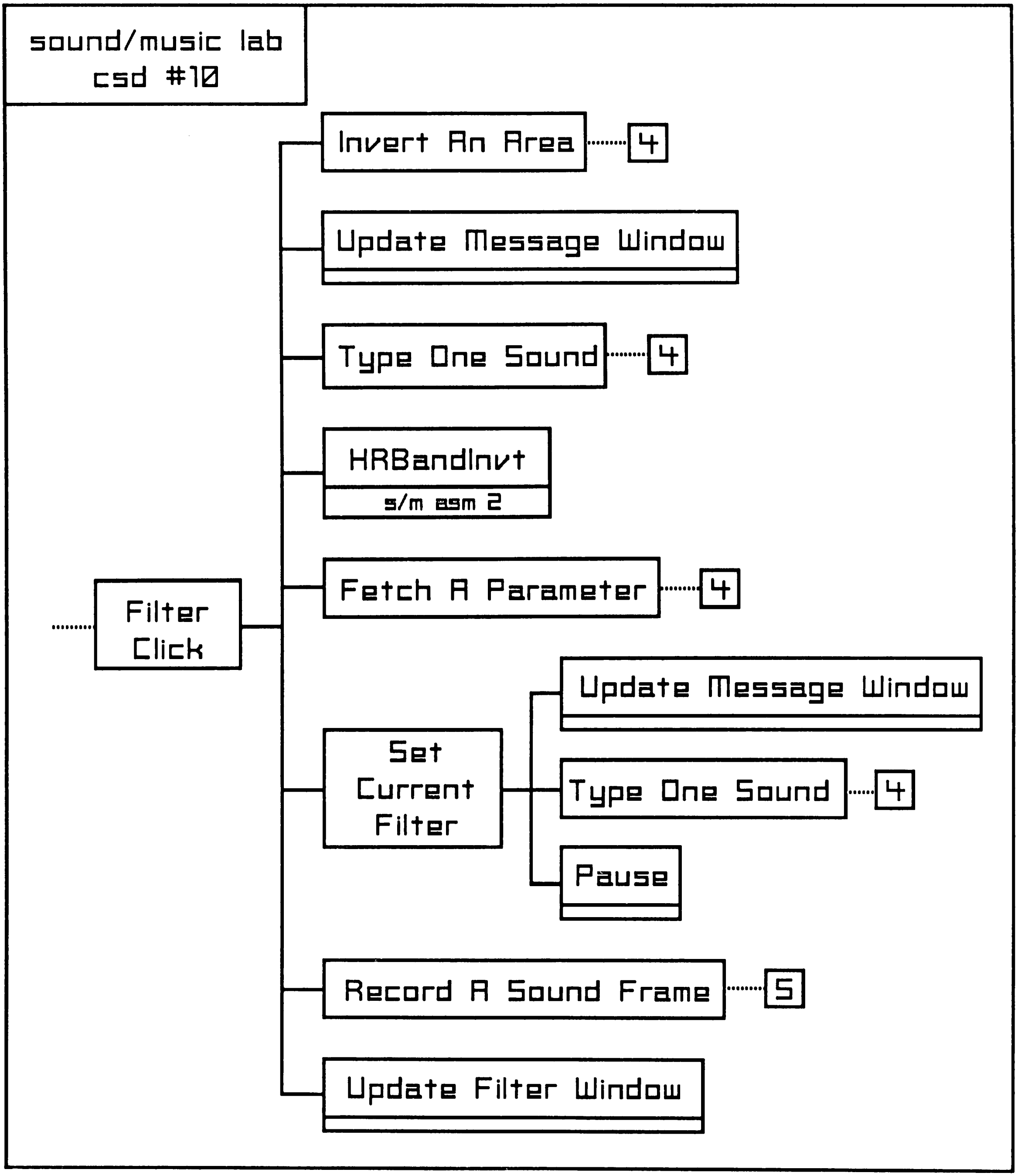

sound/music lab
csd #5
Update Message Window
Send An Error Message
Type Three Sound
Update Message Window
Record A Sound Frame
Type One Sound
4
Invert An Area
4
Update Frame Counter
Type Three Sound
Let Sound Finish

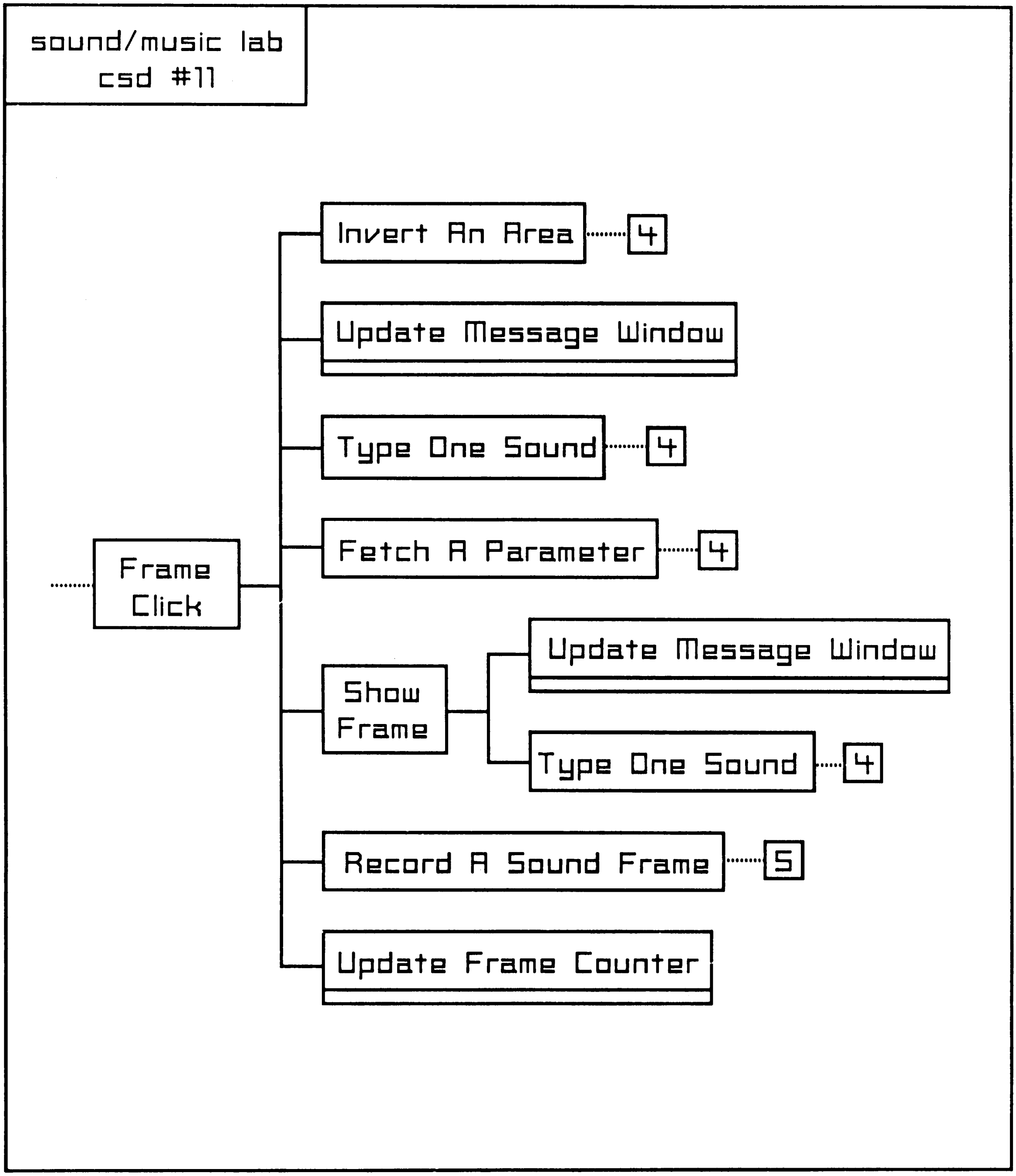

sound/music lab
csd #6
Play Click
Invert An Area
4
Type One Sound
4
Update Message Window
StrngRectEdit
s/m asm 1
Send An Error Message
5
Record A Sound Frame
5
Customize Play Window

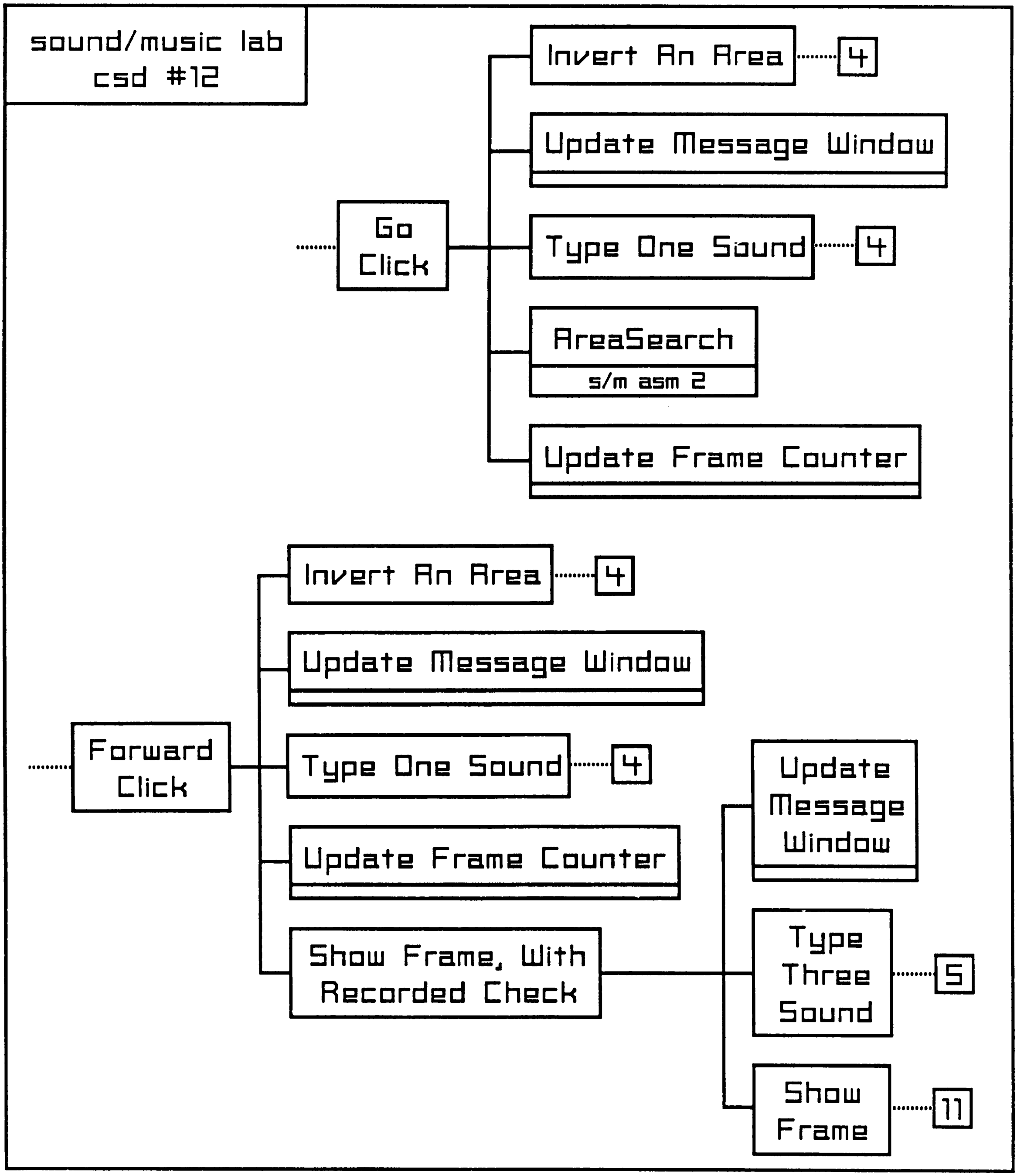

sound/music lab
csd #7
Envelope Click
Invert An Area
4
Update Message Window
Type One Sound
4
HRBandInvt
s/m asm 2
Fetch A Parameter
4
Set Current Envelope
Update Message Window
Type One Sound
4
Pause
Record A Sound Frame
5
Update An Envelope

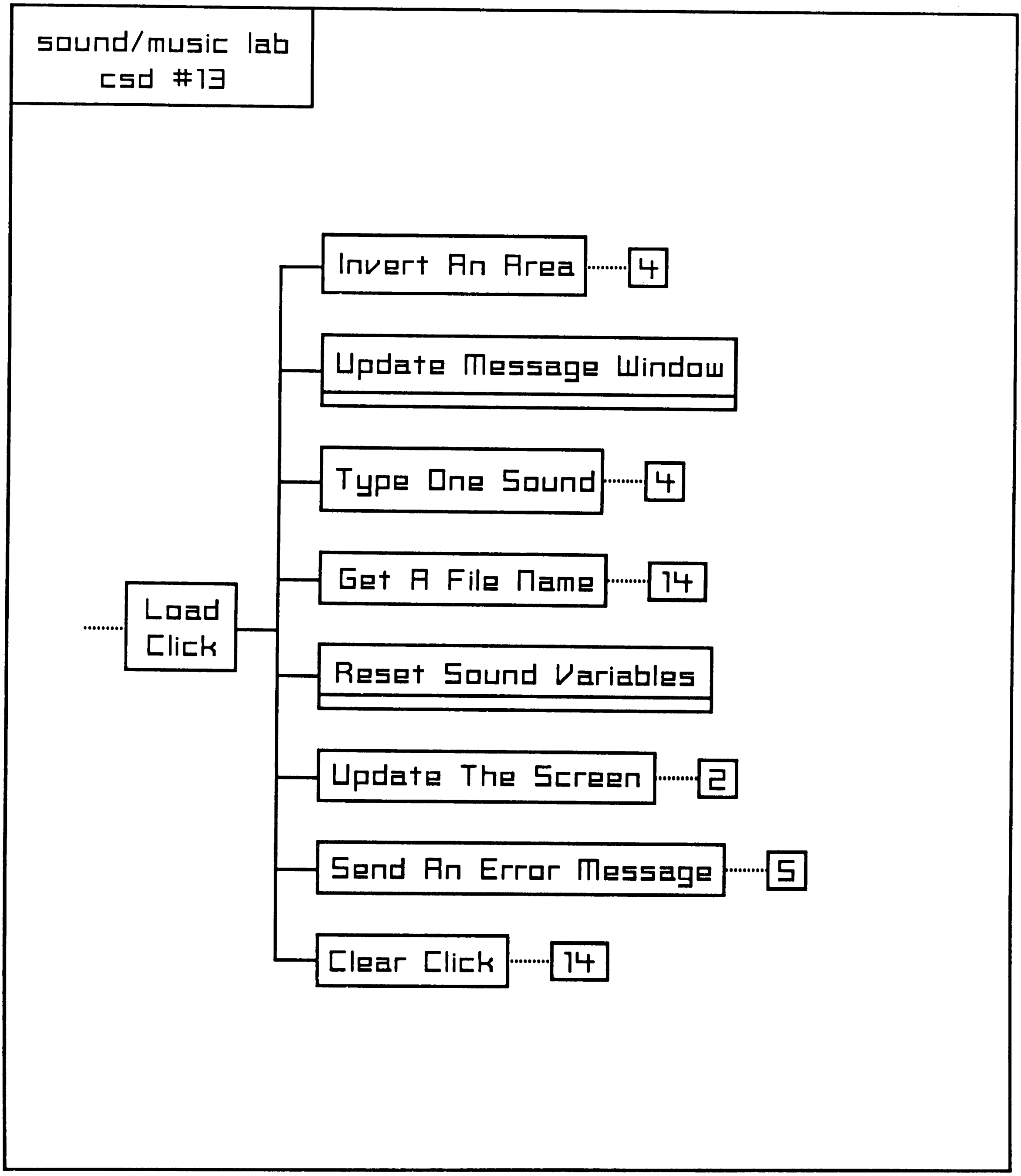

sound/music lab
csd #8
Volume
Click
Invert An Area
4
Update Message Window
Type One Sound
4
Fetch A Parameter
4
Record A Sound Frame
5
Update Volume Window

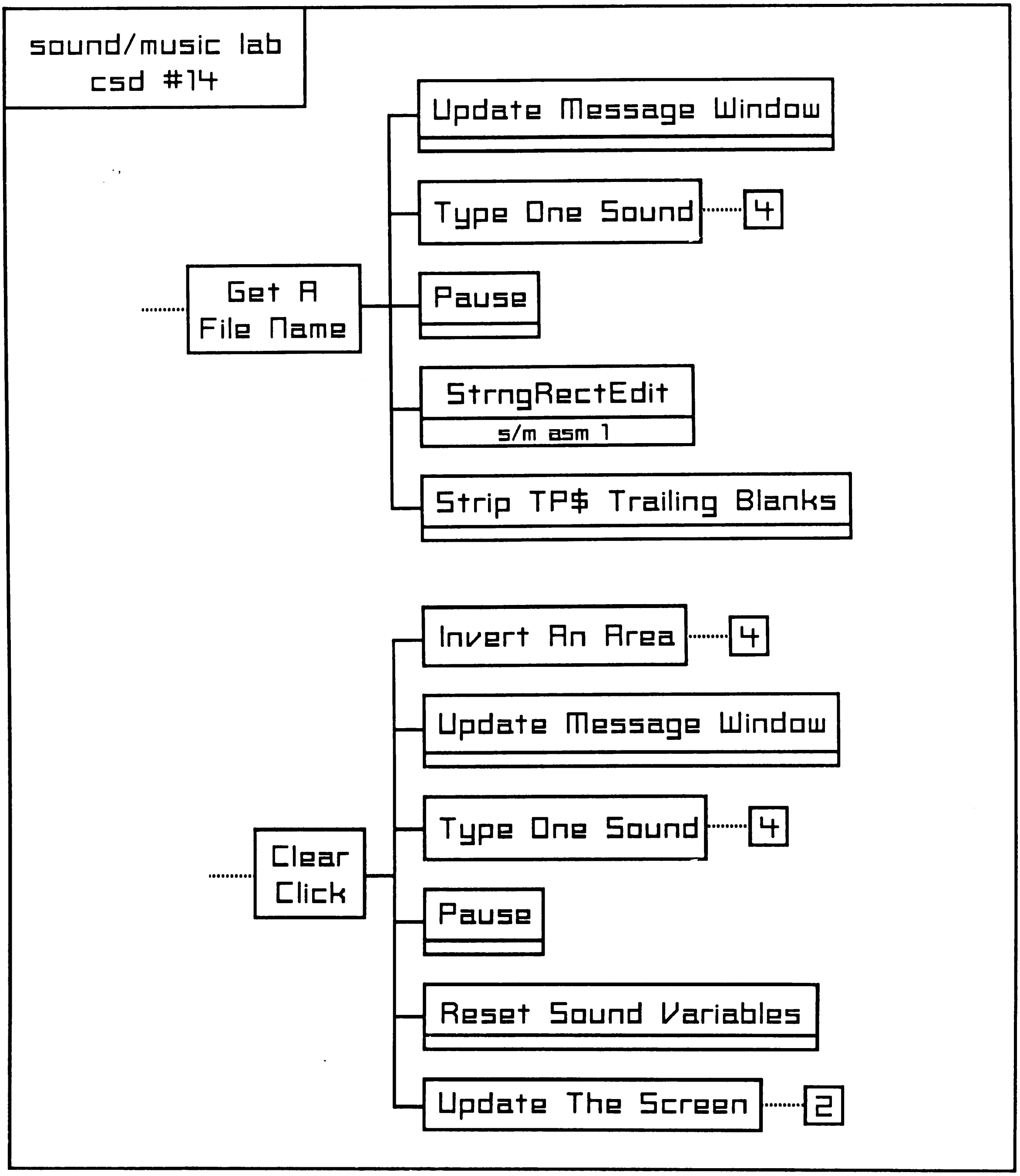

sound/music lab
csd #9
Tempo Click
Invert An Area
4
Update Message Window
Type One Sound
4
Fetch A Parameter
4
Record A Sound Frame
5
Update Tempo Window

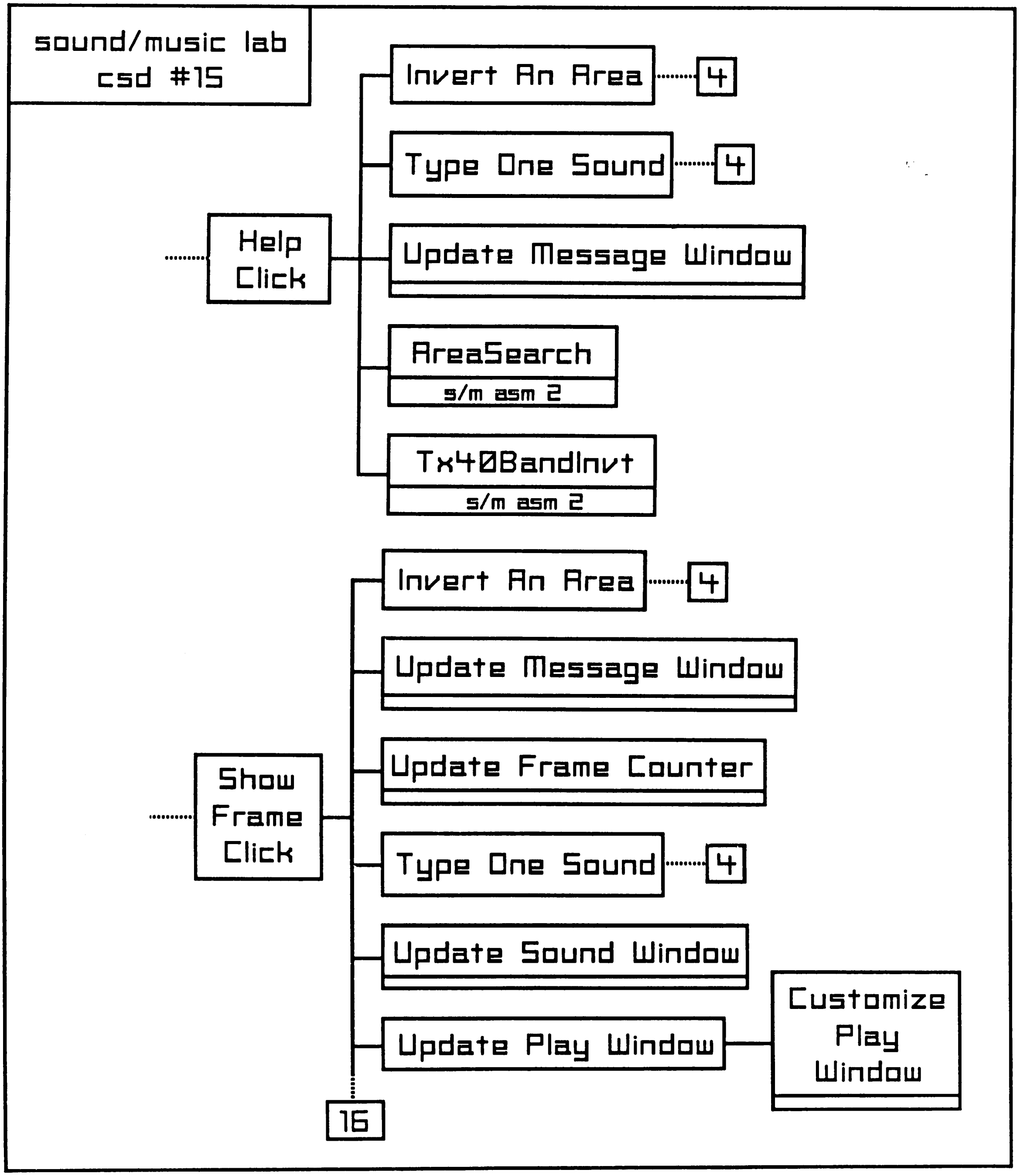

sound/music lab
csd #10
Invert An Area
4
Update Message Window
Type One Sound
4
HRBandInvt
s/m asm 2
Fetch A Parameter
4
Filter
Click
Update Message Window
Set
Current
Filter
Type One Sound
4
Pause
Record A Sound Frame
5
Update Filter Window

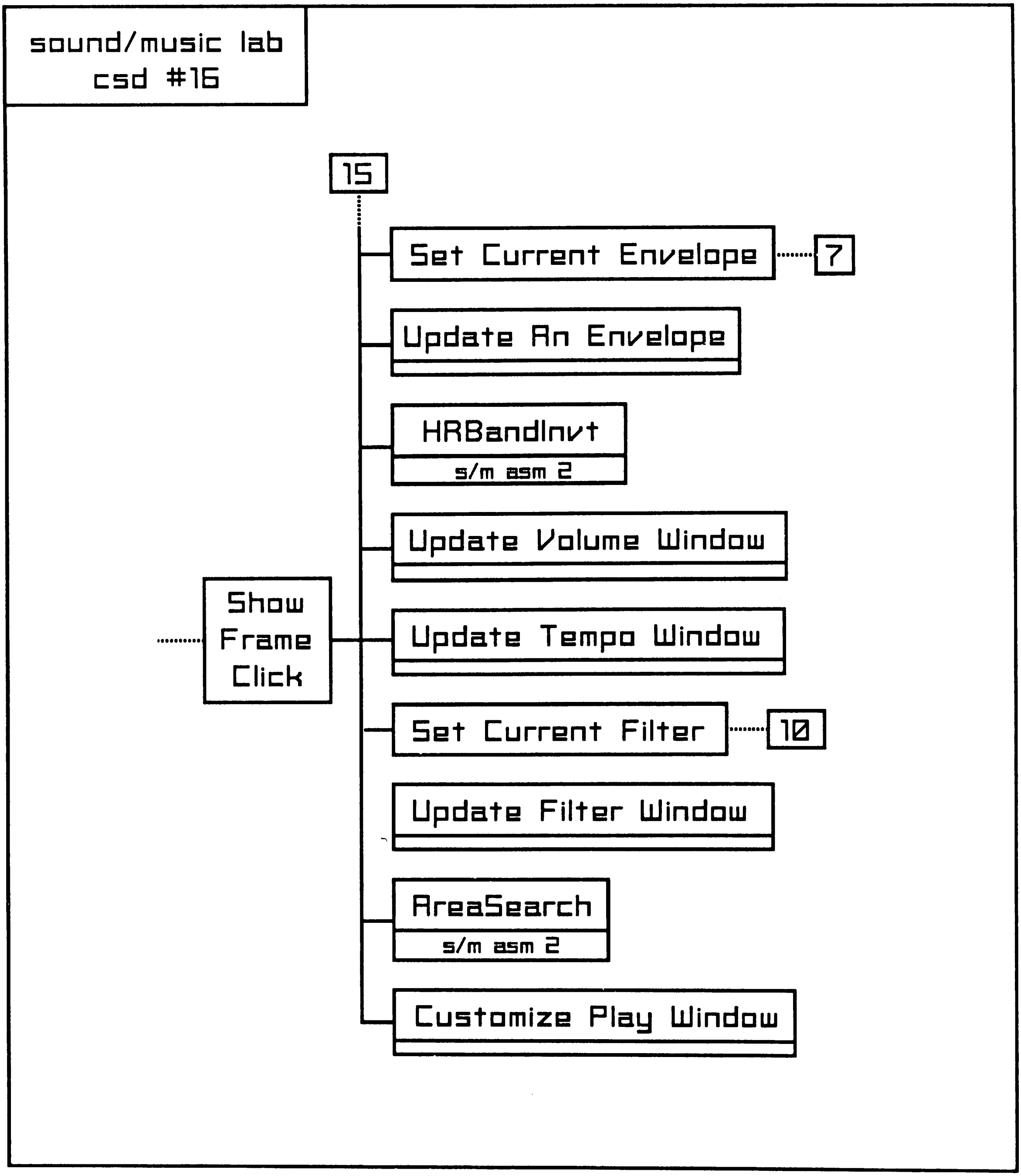

sound/music lab
csd #11
Frame Click
Invert An Area
4
Update Message Window
Type One Sound
4
Fetch A Parameter
4
Show Frame
Update Message Window
Type One Sound
4
Record A Sound Frame
5
Update Frame Counter


sound/music lab
csd #12
Go Click
Invert An Area
4
Update Message Window
Type One Sound
4
AreaSearch
s/m asm 2
Update Frame Counter
Forward Click
Invert An Area
4
Update Message Window
Type One Sound
4
Update Frame Counter
Show Frame, With Recorded Check
Update Message Window
Type Three Sound
5
Show Frame
11

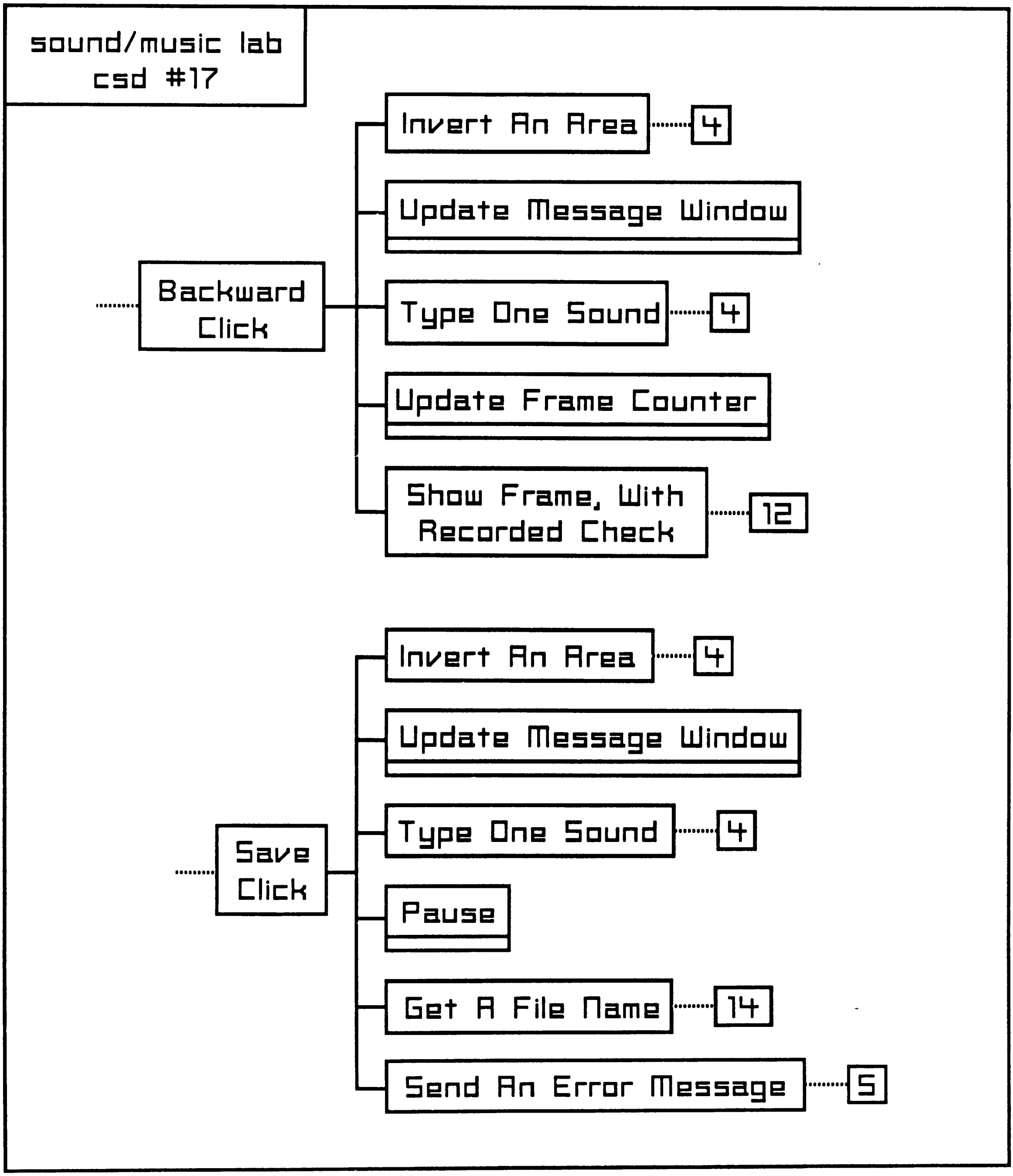

sound/music lab
csd #13
Load Click
Invert An Area
4
Update Message Window
Type One Sound
4
Get A File Name
14
Reset Sound Variables
Update The Screen
2
Send An Error Message
5
Clear Click
14

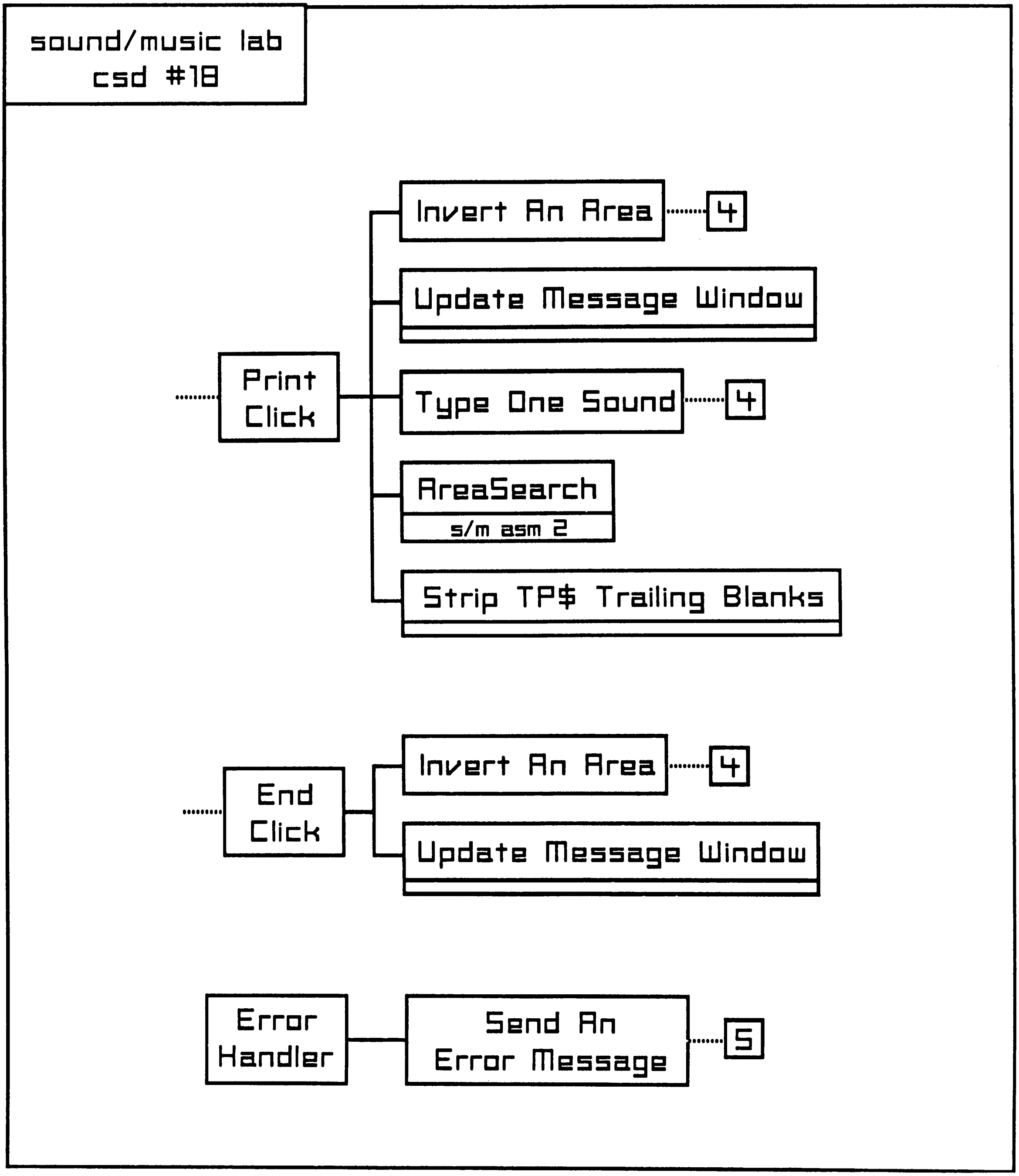

sound/music lab
csd #14
Get A File Name
Update Message Window
Type One Sound
4
Pause
StrngRectEdit
s/m asm 1
Strip TP$ Trailing Blanks
Clear Click
Invert An Area
4
Update Message Window
Type One Sound
4
Pause
Reset Sound Variables
Update The Screen
2


sound/music lab
csd #15
Help Click
Invert An Area
4
Type One Sound
4
Update Message Window
AreaSearch
s/m asm 2
Tx40BandInvt
s/m asm 2
Show Frame Click
Invert An Area
4
Update Message Window
Update Frame Counter
Type One Sound
4
Update Sound Window
Update Play Window
Customize Play Window
16



sound/music lab
csd #16
15
Set Current Envelope
7
Update An Envelope
HRBandInvt
s/m asm 2
Update Volume Window
Show
Frame
Click
Update Tempo Window
Set Current Filter
10
Update Filter Window
AreaSearch
s/m asm 2
Customize Play Window


sound/music lab
csd #17
Backward Click
Invert An Area
4
Update Message Window
Type One Sound
4
Update Frame Counter
Show Frame, With Recorded Check
12
Save Click
Invert An Area
4
Update Message Window
Type One Sound
4
Pause
Get A File Name
14
Send An Error Message
5


sound/music lab
csd #18
Print Click
Invert An Area
4
Update Message Window
Type One Sound
4
AreaSearch
s/m asm 2
Strip TP$ Trailing Blanks
End Click
Invert An Area
4
Update Message Window
Error Handler
Send An Error Message
5

Chapter 14: Subroutine Line Starts

This chapter consists of seven figures, as follows:

Fig. 14-1—subroutine line starts for S/M Asm 1 (1 sheet).
Fig. 14-2—subroutine line starts for S/M Asm 2 (1 sheet).
Fig. 14-3—subroutine line starts for S/M Help Packer (1 sheet).
Fig. 14-4—subroutine line starts for Make S/M Vars (1 sheet).
Fig. 14-5—subroutine line starts for Make 40 C Screens (1 sheet).
Fig. 14-6—subroutine line starts for 40C Edit (1 sheet).
Fig. 14-7—subroutine line starts for Sound/Music Lab (3 sheets).

S/M ASM 1 - Subroutine Line Starts	Sheet 1 Of 1
StrngRectEdit	A-288
:StorStuff	A-454
:InitEdVars	B-3
:SetStrgz	B-40
:DealKey	B-142
:DealMouse	B-281
:InvertCursor	B-439
:FigHotSpot	B-459
:OveRite	B-518
:Insert	B-569
:Delete	C-3
:DrwStrSec	C-82
:DrwStrChr	C-116
:CursRit	C-186
:CursLft	C-221
:CursDwn	C-256
:CursUp	C-284
DrawBMChar	C-378
CAsc2Pok1	C-512

Fig. 14-1. List of subroutine line starts for S/M Asm 1.

S/M ASM 2 - Subroutine Line Starts	Sheet 1 Of 1
Install	A-212
UnInstall	A-278
OurKeyChk	A-316
OurIIRQ	A-372
SetMoshn	B-3
AreaSearch	B-31
HRRectInvt	B-253
HRBandInvt	B-360
TX40BandInvt	B-448
FigOfs4025	B-531
BasBnk40	B-579

Fig. 14-2. List of subroutine line starts for S/M Asm 2.

S/M HELP PACKER - Subroutine Line Starts	Sheet 1 Of 1
Main Program Block	1220
Pack 'Em In	1290
Save It All	1530

Fig. 14-3. List of subroutine line starts for S/M Help Packer.

MAKE S/M VARS - Subroutine Line Starts	Sheet 1 Of 1
Main Program Block	1210
Open The File	1290
Write The Values	1370
Close The File	2800

Fig. 14-4. List of subroutine line starts for Make S/M Vars.

MAKE 40C SCREENS - Subroutine Line Starts	Sheet 1 Of 1
Main Program Block	1210
Get Ready	1290
Run It	1620
Clean Up	1730
Print Choices	1800
Bad Choice	1970
Edit Command	2080
Clear Command	2240
Save Command	2340
Fetch File Name And Device Number	2530
Load Command	2860
Print Command	3050
Quit Command	3130
Error Handler	3210

Fig. 14-5. List of subroutine line starts for Make 40C Screens.

40C EDIT - Subroutine Line Starts	Sheet 1 Of 1
40CEdit	82
InvertCursor	118
DealKey	141
PrintChar	308
Insert	353
SetPtr	405
Delete	421
CursRit	487
CursLft	520
CursDwn	552
CursUp	581
CAsc2Pokl	609

Fig. 14-6. List of subroutine line starts for 40C Edit.

SOUND/MUSIC LAB - Subroutine Line Starts	Sheet 1 Of 3
Main Program Block	1180
Set Up The Lab	1300
Lab Event Loop	1460
Clean Up The Lab	1620
Configure Memory	1730
Initialize Some Variables	1810
Reset Sound Variables	2600
Draw A Fresh Screen	2910
Update The Screen	3070
Load And Install Binary Files	3180
Initialize Cursor	3280
Draw Six Windows	3360
Customize Sound Window	3630
Customize Play Window	3800
Customize Envelope Window	3890
Customize Filter Window	4150
Draw Frame Counter	4310
Draw Nine Buttons & A Window	4380
Draw Help Button	4620
Update Sound Window	4750

Fig. 14-7. List of subroutine line starts for Sound/Music Lab.

SOUND/MUSIC LAB - Subroutine Line Starts	Sheet 2 Of 3
Update Play Window	4950
Update Envelopes Window	5080
Update Volume Window	5160
Update Tempo Window	5260
Update Filter Window	5360
Update Frame Counter	5550
Update Message Window	5650
Update An Envelope	5730
Sound Click	6020
Play Current Sound	6580
Invert An Area	6640
Play Click	6700
Envelope Click	7300
Set Current Envelope	7830
Volume Click	7930
Tempo Click	8350
Filter Click	8770
Set Current Filter	9190
Frame Click	9290
Go Click	9700
Forward Click	10310
Load Click	10490
Get A File Name	11080
Clear Click	11340
Help Click	11560
Show Frame Click	12190
Backward Click	13250
Save Click	13430
Strip TP$ Trailing Blanks	13980
Print Click	14080
End Click	14750
Fetch A Parameter	14890
Record A Sound Frame	15440
Type One Sound	15670
Type Three Sound	15740
Let Sound Finish	15810
Show Frame, With Recorded Check	15870
Show Frame	15980
Send An Error Message	16060
Error Handler	16240
Pause	16310

Chapter 15: Selected Algorithms

This chapter consists of seven figures, as follows:

Fig. 15-1—selected algorithms from S/M Asm 1 (10 sheets).
Fig. 15-2—selected algorithms from S/M Asm 2 (8 sheets).
Fig. 15-3—selected algorithms from S/M Help Packer (1 sheet).
Fig. 15-4—selected algorithms from Make S/M Vars (1 sheet).
Fig. 15-5—selected algorithms from Make 40C Screens (5 sheets).
Fig. 15-6—selected algorithms from 40C Edit (6 sheets).
Fig. 15-7—selected algorithms from Sound/Music Lab (43 sheets).

Selected Algorithms From S/M ASM 1 Sheet 1 Of 10

StrngRectEdit

store the function selector flag
store the pointer to the parameter block
call on :StorStuf to store items from the parameter block
call on :SetStrgz to set up strings
call on :InitEdVars to initialize some editing variables
call on :DrwStrSec to draw the exit string on the bit–map screen
IF
    the function selector flag says we're here just to draw the string
THEN
    RETURN
{ we're here to do some editing of the string }
REPEAT the following
    call on :InvertCursor to draw the editing cursor
    call on :FigHotSpot to determine the string's hot spot
        (the currently changeable character position)
    IF
        a call to the System routine GetIn shows there's a keypress
    THEN
        call on :DealKey to deal with the keypress
    ELSE IF
        there's been a click of the pseudo–mouse
    THEN
        call on :DealMouse to deal with the pseudo-mouse click
UNTIL
    there's a signal to end the editing session
call on :InvertCursor to erase the editing cursor
RETURN with an exit area code and the exit string's hot spot

Selected Algorithms From S/M ASM 1 Sheet 2 Of 10

:StorStuf

FOR
    each byte of information contained in the StrngRectEdit parameter block
DO the following
    grab that byte's destination address from the :WherTab table
    call on the System routine IndFet to grab the byte from the parameter block
    send it to its destination
RETURN

:InitEdVars

move the editing cursor to the upper–left corner of the editing rectangle
IF
    the editing cursor is supposed to start out somewhere else
THEN
    FOR
        each position to the right the editing cursor has to move to reach its
destination
    DO the following

Fig. 15-1. Selected algorithms from S/M Asm 1.

call on :CursRit to move it one position to the right
figure out the width of the editing rectangle
RETURN

:SetStrgz
call on the System routine IndFet to get the length of the entry and exit strings
(both have the same length)
call on the System routine IndFet to get pointers to the two strings
FOR
each byte in the strings (that's why we needed the length)
DO the following

call on the System routine IndFet to get the byte from the entry string
call on the System routine IndSta to copy that byte to the exit string
RETURN

:DealKey
IF
it's a printable character keypress
THEN
call on :OveRite to add the character to the exit string
RETURN, signalling for more editing
ELSE IF
it's a cursor–up keypress
THEN
call on :InvertCursor to erase the cursor
call on :CursUp to move the cursor up
RETURN, signalling for more editing
ELSE IF
it's a cursor–down keypress
THEN
call on :InvertCursor to erase the cursor
call on :CursDwn to move the cursor down
RETURN, signalling for more editing
ELSE IF
it's a cursor–left keypress
THEN
call on :InvertCursor to erase the cursor
call on :CursLft to move the cursor left
RETURN, signalling for more editing
ELSE IF

it's a cursor–right keypress
THEN
call on :InvertCursor to erase the cursor
call on :CursRit to move the cursor right
RETURN, signalling for more editing
ELSE IF
it's an insert keypress

THEN

call on :Insert to insert into the string

RETURN, signalling for more editing

ELSE IF

it's a delete keypress

THEN

call on :Delete to delete from the string

RETURN, signalling for more editing

ELSE { the keypress is not one we choose to deal with }

call on :InvertCursor to erase the cursor

RETURN, signalling for more editing

:DealMouse

call on AreaSearch (from S/M ASM 2) to find out the area the pseudo–mouse click occurred in

IF

we aren't looking for any specific area, just a p–m click

THEN

RETURN with a null exit area code and signalling that it's time to end the editing session

ELSE IF

the area isn't our editing area

THEN

RETURN with that area as the exit area code and signalling that it's time to end the editing session

ELSE { the mouse has been pressed in our editing area }

call on :InvertCursor to erase the cursor

convert the pseudo–mouse coordinates to text–screen coordinates, making sure the text–screen coordinates stay within our editing rectangle

move the cursor to those text–screen coordinates

RETURN, signalling for more editing

:InvertCursor

call on HRBandInvt (from S/M ASM 2) to invert the cursor

RETURN

:FigHotSpot

figure out the row the cursor is in relative to the editing rectangle

multiply that by the width of the editing rectangle

figure out the column the cursor is in relative to the editing rectangle

add that to the row–width product and we've got the hot spot

RETURN

:OveRite

set up a pointer to the exit string

call on the System routine IndSta to add the character to the exit string at the hot spot

call on :InvertCursor to erase the cursor

call on DrawBMChar to draw the character at the current cursor position

call on :CursRit to move the cursor to the right
RETURN

:Insert
set up a pointer to the exit string
IF
the hot spot is not at the exit string's last character position
THEN do the following
FOR
each character in the exit string from the next–to–last through to the hot spot
DO the following
call on the System routine IndFet to fetch the character
call on the System routine IndSta to store the
character one position to the right in the string
call on the System routine IndSta to store a space character at the exit string's hot spot
call on :DrwStrSec to redraw the exit string from the hot spot through to the last character
call on :InvertCursor to erase the cursor
RETURN

:Delete
set up a pointer to the exit string
call on :InvertCursor to erase the cursor
IF
the hot spot is not at the exit string's first character position
THEN do the following
FOR
each character in the exit string from the hot spot through to the last character
DO the following
call on the System routine IndFet to fetch the character

call on the System routine IndSta to store the
character one position to the left in the string
call on the System routine IndSta store a space character
at the exit string's last character position
call on :DrwStrSec to redraw the exit string from the hot spot through to the last character
call on :CursLft to move the cursor to the left
RETURN

:DrwStrSec
STARTING with
the first character of the string section
REPEAT
call on :DrwStrChr to draw the character
move on to the next character

UNTIL
    the last character of the string section has been drawn
RETURN

:DrwStrChr
    figure the row the character's in relative to the editing rectangle
    figure the column the character's in relative to the editing rectangle
    change the relative row to an absolute row
    change the relative column to an absolute column
    IF
        the character is inside the editing rectangle
    THEN
        call on the System routine IndFet to grab the character
        call on DrawBMChar to draw the character on the bit–map
            screen at its absolute row–column position

RETURN

:CursRit
    IF
        the cursor is at the editing rectangle's rightmost column
    THEN
        call on :CursDwn to move the cursor down a row
        move the cursor to the editing rectangle's leftmost column
    ELSE
        move the cursor one position to the right
    RETURN

:CursLft
    IF
        the cursor is at the editing rectangle's leftmost column
    THEN
        call on :CursUp to move the cursor up a row
        move the cursor to the editing rectangle's rightmost column
    ELSE
        move the cursor one position to the left
    RETURN

:CursDwn
    IF
        the cursor is at the bottom of the editing rectangle
    THEN
        move the cursor to the top of the editing rectangle
    ELSE
        move the cursor down a row

RETURN

:CursUp

IF

the cursor is at the top of the editing rectangle

THEN

move the cursor to the bottom of the editing rectangle

ELSE

move the cursor up a row

RETURN

DrawBMChar

save some registers

call on CAsc2Pok1 to transform the C-ASCII code to a character set 1 poke code

set a pointer to the address in the character ROM where the character's image data begins:

multiply the poke code by 8

add that to the base address of set 1 in the character ROM

set a pointer to the address in bit–map memory where the character's image data will begin :

multiply the character's absolute row by 320 bytes per row

add that to the base of the bit–map

multiply the character's absolute column by 8 bytes per column

add that to the previous result

save the current memory configuration

set the memory configuration to Bank 14

FOR

each of the 8 bytes of character image data

DO the following { with the pointers derived above }

grab the byte from the character ROM

store the byte into bit–map memory

restore the saved memory configuration

restore some registers

RETURN

CAsc2Pok1

convert the C-ASCII code to a character set 1 poke code as follows :

C-ASCII code range	poke code range
0..31	32
32..63	32..63
64..95	0..31
96..127	64..95
128..159	32
160..191	96..127
192..223	64..95
224..254	96..126
255	94

RETURN

Selected Algorithms From S/M ASM 2 Sheet 1 Of 8

Install
    disable interrupts
    save some registers
    save the current KeyChk vector
    point the KeyChk vector to OurKeyChk
    save the current IIRQ vector
    point the IIRQ vector to OurIIRQ
    set the sprite–in–motion flag to no motion
    set CIA #2 Timer A for use as a single/double click timer
        { this got left in the code even though I didn't have
        time to implement the click differentiation routines }
    restore some registers
    enable interrupts
    RETURN

UnInstall
    disable interrupts
    save some registers
    point the IIRQ vector back to its original routine
    point the KeyChk vector back to its original routine
    restore some registers
    enable interrupts
    RETURN

OurKeyChk
    save some registers
    FOR
        each of our test keycodes (upper cursor keys and return key)
    DO the following

Selected Algorithms From S/M ASM 2 Sheet 2 Of 8

        IF
            that's the keycode the System detected
        THEN
            hide that keypress from the System
            leave this FOR..DO loop
    restore some registers
    JUMP to the regular KeyChk routine

OurIIRQ
    save some registers
    save the entry memory configuration
    set the memory configuration to Bank 15 (System bank)
    check out the upper cursor keys :
        send signals out on the keyboard lines of interest
        read the resulting signal
        filter out noise from that result

Fig. 15-2. Selected algorithms from S/M Asm 2.

use the result to grab a cursor keys direction code
check out the joystick direction switches :
read the joystick data
filter out noise from that data
use the data to grab a joystick direction code
arbitrate between the two possible direction codes :
IF
the joystick direction code indicates movement is necessary
THEN
use the joystick direction code
ELSE
use the cursor keys direction code
IF

sprite #1 (our pseudo–mouse cursor) is currently in motion
THEN do the following
IF
the arbitrated direction code indicates a change in the direction of sprite motion
THEN do the following
stop sprite #1's motion
set the in–motion flag to stopped
ELSE
JUMP to where we check out the pseudo–mouse button
{ we get here if sprite #1 is not in motion }
IF
sprite #1 is to be put into motion
THEN
use the arbitrated direction code to set a pointer to the appropriate sprite motion data
call on SetMoshn to set sprite #1 into the appropriate motion
set the in–motion flag and record the new direction of sprite motion
{ this is where we check out the pseudo–mouse button }
IF
a test of the joystick data byte shows that the joystick button is being pressed
OR
a test of the keyboard circuitry shows that the RETURN key is being pressed
THEN
store sprite #1's current position
set the pseudo–mouse click state to 'clicked'
ELSE { neither the joystick button nor the RETURN key is being pressed by the User }
set the pseudo–mouse click state to 'not clicked'
restore the saved memory configuration

restore some registers
JUMP on to the regular IIrq handler

SetMoshn

save some registers
FOR
each of the bytes in the sprite motion data record
that's pointed to when we enter this routine
DO the following
store the byte in sprite #1's speed/direction table
restore some registers
RETURN

AreaSearch

initialize a search pointer to the first entry in a table of areas to be searched
save some registers
STARTING with
the first area in the table
REPEAT the following
IF
a grab of area data shows we're at the end of the table of areas
THEN
the pseudo–mouse point is not in any of this table's areas
ELSE IF
the pseudo–mouse vertical coordinate is above the area's top boundary
THEN
the pseudo–mouse point is not in this area
the pseudo–mouse point is not in any of this table's areas
ELSE IF

the pseudo–mouse vertical coordinate is below the area's bottom boundary
THEN
the pseudo–mouse point is not in this area
we need to move on to the next area
ELSE IF
the pseudo–mouse horizontal coordinate is to the left of the area's left boundary
THEN
the pseudo–mouse point is not in this area
we need to move on to the next area
ELSE IF
the pseudo–mouse horizontal coordinate is
to the right of the area's right boundary
THEN
the pseudo–mouse point is not in this area
we need to move on to the next area
ELSE
we've found an area the pseudo–mouse point is in
IF
we need to move on to the next area
THEN
move the search pointer to the next area in the table by adding
the length of an area's data record to it

UNTIL
we know the pseudo–mouse point is not in any of this table's areas
OR
we've found an area the pseudo–mouse point is in
IF

the pseudo–mouse point was not in any of the table's areas
THEN do the following
restore some registers
RETURN with a NULL result code
ELSE { we found an area }
restore some registers
RETURN with the area's ID number as a result code

HRRectInvt

set a pointer to the start of the table of area rectangle data
use the area's ID number to figure the offset of the area's rectangle's data in the table
add that offset to the pointer so we're pointing at the area's rectangle's data
grab the area's top row so we know where to start our loop
grab the area's bottom row so we know where to end our loop
grab the area's rectangle's left and right columns so we can figure out the width of a row
FOR
each row of the area's rectangle
DO the following
call on HRBandInvt to invert that row
RETURN

HRBandInvt

save some registers
set a pointer to the start of the band's row
move the pointer to the band's starting cell
FOR
each of the band's 8–pixel–by–8–pixel cells
DO the following
grab the cell's foreground/background byte

swap the foreground and background nibbles
store the cell's modified foreground/background byte
restore some registers
RETURN

TX40BandInvt

save some registers
call on FigOfs4025 to figure the band's leftmost column's
offset from the base of the text screen memory
call on BasBnk40 to get the base of the text screen memory
and the RAM memory bank it's in

add the fetched base and offset to set a pointer to the band's leftmost column
FOR
    each of the band's columns
DO the following
    call on the System routine IndFet to grab the column's current poke code
    flip–flop the poke code's hi–bit to invert the character
    call on the System routine IndSta to store the column's modified poke code
restore some registers
RETURN

FigOfs4025
park the parameters
get the lo–byte of the row's starting address
add in the column and park the result
get the hi–byte of the row's starting address
add in any Carry from the prior addition and park the result
grab the parked results
RETURN

BasBnk40
grab the current memory configuration byte
mask out all but bit 6, which represents the RAM bank that's in effect
    (0 for RAM bank 0, 1 for RAM bank 1)
move that bit around to bit 0 -- now we've got a byte
    holding the RAM bank number, 0 or 1
grab the hi–nibble of VIC register #24, which represent
    bits 10..13 of the text screen's base address
grab the contents of CIA #2 Port A
mask out all but bits 0 and 1, which represent bits 14
    and 15 of the text screen's base address
move those 6 fetched bits around, then add two zero bits,
    to form the hi–byte (bits 8..15) of the text screen's base address
{ bits 0..9 of the text screen's base address are 0,
    since text screens occur on one–K boundaries }
set the lo–byte of the text screen's base address to 0
RETURN with the bank number and base address in the registers

Selected Algorithms From S/M HELP PACKER — Sheet 1 Of 1

Main Program Block
    call on Pack 'Em In to load a block of memory with help screen data
    call on Save It All to save that block of memory to a disk file
    RETURN

Pack 'Em In
    FOR
        each of 22 help screens
    DO the following
        load the help screen into an area of RAM Bank 1
        poke a sprite data pointer into that area of RAM Bank 1
    load the finger cursor sprite image data into quadrant 3 of RAM Bank 1
    load the finger cursor sprite image data into quadrant 4 of RAM Bank 1
    RETURN

Save It All
    print a disk insertion prompt
    wait until the User presses a key
    save the block of RAM Bank 1 memory that holds the help screen goodies
    print the disk operation status info
    print a catalog of the disk
    RETURN

Fig. 15-3. Selected algorithms from S/M Help Packer.

Selected Algorithms From MAKE S/M VARS — Sheet 1 Of 1

Main Program Block
    Open The File
    Write The Values
    Close The File
    RETURN

Open The File
    fetch the device number from the User
    open a sequential file for writing on that device
    RETURN

Write The Values
    write a number of variables for SOUND/MUSIC LAB to that opened file
    RETURN

Close The File
    burp the disk buffer
    close the opened file
    RETURN

Fig. 15-4. Selected algorithms from Make S/M Vars.

Selected Algorithms From MAKE 40C SCREENS Sheet 1 Of 5

Main Program Block
    Get Ready
    Run It
    Clean Up
    RETURN

Get Ready
    slow down to 1 megahertz speed
    set up an error handler
    set screen to 80–column text, cleared, with a black background and border
    set finished flag to 'not finished'
    set up a command parsing string
    set up some character string constants
    load in the "40C EDIT" object code from the device this program was loaded from
    load in the "TEXT DUMPS" object code from the device this program was loaded from
    initialize the 40–column screen editing cursor to the upper left corner of the screen
    call on Print Choices to print out the menu of command choices
    RETURN

Run It
    REPEAT
        wait for a keypress
        figure out a command code by running the keypress through the parsing string
        based on the command code, call on Bad Choice, EditCommand, Clear Command,
            Save Command, LoadCommand, Print Command, or Quit Command
    UNTIL
        the finished flag says 'finished'

Selected Algorithms From MAKE 40C SCREENS Sheet 2 Of 5

    RETURN

Clean Up
    go to a cleared 80–column text screen
    speed up to 2 megahertz speed
    RETURN

Bad Choice
    print some feedback

Fig. 15-5. Selected algorithms from Make 40C Screens.

wait a second
clear the message area
print some advice
wait a second
clear the message area
RETURN

Edit Command
print some feedback
save the 80–column cursor position
set the 40–column cursor position
call on 40CEdit (from the file 40C EDIT) to edit the 40–column screen
save the 40–column cursor position
set the 80–column cursor position
clear the message area
RETURN

Clear Command
print some feedback
switch to a cleared 40–column text screen

switch to an undisturbed 80–column text screen
wait a second
clear the message area
RETURN

Save Command
print some feedback
IF
a call to Fetch File Name And Device Number is successful
THEN do the following
save the 40–column text screen as the named file on the chosen disk drive
clear the message area
print the disk drive status string as feedback
wait a second
clear the message area
RETURN

Fetch File Name And Device Number
get a file name from the User
IF
no file name was entered
THEN
clear the message area

```
            print some feedback
            RETURN with a result code of 'unsuccessful'
        ELSE  { a file name was entered }
            get a device number from the User
            IF
                the device number is not reasonable
            THEN
```

Selected Algorithms From MAKE 40C SCREENS   Sheet 4 Of 5

```
            clear the message area
            print some feedback
            RETURN with a result code of 'unsuccessful'
        ELSE  { the device number was reasonable }
            RETURN with a result code of 'successful'
```

Load Command

```
    print some feedback
    IF
        a call to Fetch File Name And Device Number is successful
    THEN do the following
        load the named 40–column text screen file from the chosen disk drive
        clear the message area
        print the disk drive status string as feedback
    wait a second
    clear the message area
    RETURN
```

Print Command

```
    print some feedback
    call on Dump40 ( from the file TEXT DUMPS ) to print the 40–column screen
    clear the message area
    RETURN
```

Quit Command

```
    print some feedback
    set finished flag to 'finished'
    RETURN
```

Selected Algorithms From MAKE 40C SCREENS   Sheet 5 Of 5

Error Handler

```
    clear the message area
    print out information about the error
    wait 2 seconds
    clear the message area
    RESUME execution at the next BASIC statement after the error
```

Selected Algorithms From 40C EDIT Sheet 1 Of 6

40CEdit

REPEAT the following
    call on InvertCursor to draw the editing cursor
    REPEAT the following
        call on the System routine GetIn to scan the keyboard
    UNTIL
        such a call reveals a key has been pressed
    call on DealKey to deal with the keypress
UNTIL
    DealKey returns with a signal to end editing
RETURN

InvertCursor

call on SetPtr to set a pointer to the start of the cursor's row
set an index to the cursor's column
using the pointer and the index, grab the cursor's poke code
flip–flop the poke code's hi–bit to invert it
store the modified poke code
RETURN

DealKey

call on InvertCursor to erase the cursor
IF
    the key's C–ASCII character code is in the range 32..127
        OR
    the key's C–ASCII character code is in the range 160..255
THEN do the following
    call on PrintChar to print the character
    RETURN with a signal to continue editing

Selected Algorithms From 40C EDIT Sheet 2 Of 6

ELSE IF
    it's a cursor–up keypress
THEN
    call on CursUp to move the cursor up
    RETURN with a signal to continue editing
ELSE IF
    it's a cursor–down keypress
THEN
    call on CursDwn to move the cursor down
    RETURN with a signal to continue editing
ELSE IF
    it's a cursor–left keypress
THEN
    call on CursLft to move the cursor to the left
    RETURN with a signal to continue editing
ELSE IF
    it's a cursor–right keypress

Fig. 15-6. Selected algorithms from 40C Edit.

```
    THEN
        call on CursRit to move the cursor to the right
        RETURN with a signal to continue editing
    ELSE IF
        it's an insert keypress
    THEN
        call on Insert to insert a space
        RETURN with a signal to continue editing
    ELSE IF
        it's a delete keypress
    THEN
        call on Delete to delete a character
```

Selected Algorithms From 40C EDIT Sheet 3 Of 6

```
        RETURN with a signal to continue editing
    ELSE IF
        it's a reverse–on keypress
    THEN
        set the reverse flag to 'on'
        RETURN with a signal to continue editing
    ELSE IF
        it's a reverse–off keypress
    THEN
        set the reverse flag to 'off'
        RETURN with a signal to continue editing
    ELSE IF
        it's a SHIFT–RETURN combination
    THEN
        RETURN with a signal to end editing
    ELSE
        { the keypress is not one we deal with }
        RETURN with a signal to continue editing

PrintChar
    call on CAsc2Pok1 to fetch the character's poke code
    IF
        the reverse flag is set to 'on'
    THEN
        reverse the character's poke code by setting bit 7
    call on SetPtr to set a pointer to the start of the cursor's row
    set an index to the cursor's column
    using the pointer and the index, store the character's poke code
    call on CursRit to move the cursor to the right
    RETURN
```

Selected Algorithms From 40C EDIT Sheet 4 Of 6

```
Insert
    call on SetPtr to set a pointer to the start of the cursor's row
    IF
        the cursor is not located in the rightmost column
```

THEN do the following
FOR
each column in the cursor's row from the next–to–last
column through to the cursor position
DO the following
fetch that row/column position's character
store it one position to the right
store a space character at the cursor position
RETURN

SetPtr
fetch the cursor's row
use it as an index into a table of row starting addresses
fetch the lo– and hi– bytes of the cursor's row's starting address
RETURN

Delete
call on SetPtr to set a pointer to the start of the cursor's row
IF
the cursor is in the leftmost column of its row
THEN do the following
call on CursLft to move the cursor to the left
{ it'll end up in the rightmost column of the previous row }
call on SetPtr to set a pointer to the start of the cursor's new row

Selected Algorithms From 40C EDIT Sheet 5 Of 6

set an index to the cursor's column
using the pointer and the index, store a space character at the cursor's position
RETURN
ELSE do the following { the cursor isn't in the leftmost column }
FOR
each column in the cursor's row from the cursor's
column through to the rightmost column
DO the following
fetch that row/column position's character
store it one position to the left
store a space character at the rightmost column
call on CursLft to move the cursor to the left
RETURN

CursRit
IF
the cursor's in the rightmost column
THEN
call on CursDwn to move the cursor down a row
move the cursor to the leftmost column
ELSE
move the cursor one column to the right
RETURN

CursLft
IF

the cursor's in the leftmost column
THEN
call on CursUp to move up a row

Selected Algorithms From 40C EDIT Sheet 6 Of 6

move the cursor to the rightmost column
ELSE
move the cursor one position to the left
RETURN

CursDwn
IF
the cursor's in the bottom row
THEN
move the cursor to the top row
ELSE
move the cursor down a row
RETURN

CursUp
IF
the cursor's in the top row
THEN
move the cursor to the bottom row
ELSE
move the cursor up a row
RETURN

Selected Algorithms From SOUND/MUSIC LAB Sheet 1 Of 43

Main Program Block
Set Up The Lab
REPEAT
CALL on the Lab Event Loop
UNTIL
the finished flag says 'finished'
Clean Up The Lab
RETURN

Set Up The Lab
speed up processor to 2 megahertz (VIC screen disappears)
set Bank 15 (System bank) for memory accesses
set up an error handler
Configure Memory
Initialize Some Variables
Reset Sound Variables
Draw A Fresh Screen
Update The Screen
Load And Install Binary Files

Fig. 15-7. Selected algorithms from Sound/Music Lab.

Initialize Cursor
slow down processor to 1 megahertz (VIC screen appears)
RETURN

Lab Event Loop
IF
the pseudo-mouse button has been clicked
THEN
CALL on AreaSearch (from S/M ASM 2) to find out where it's been clicked
IF

the pseudo–mouse was clicked in a valid area
THEN
JUMP to one of these routines, based on the area :
Sound Click, Play Click, Envelope Click, Volume Click,
Tempo Click, Filter Click, Frame Click, Go Click, Forward Click,
Load Click, Clear Click, Help Click, Show Frame Click,
Backward Click, Save Click, Print Click, End Click
RETURN

Clean Up The Lab
go to a cleared 40–column text screen
turn sprite #1 (the pseudo–mouse finger cursor) off
CALL on UnInstall (from S/M ASM 2) to un–install the pseudo–mouse routines
move RAM Bank 1 string storage back up to its normal position
RETURN

Configure Memory
move RAM Bank 1 string storage down to make room for the help screens
RETURN

Initialize Some Variables
set the default drive to the one last accessed (the one the program came from)
open the file S/M VARS on the default drive for reading
set the finished flag to 'not finished'
set the number of help screens to 22
set the current help screen to #1
set up some pointers
set up a string of blanks
set up a string of zeroes

set up sound frame arrays and pointers
read sound frame type titles from the disk file
set up an initial feedback message
read assembly language addresses from the disk file
read parameter fetching data from the disk file
read envelope parameter strings from the disk file
read the default sound array and sound parameter strings from the disk file
read the default envelopes from the disk file

read filter parameter strings from the disk file
read help screen inversion parameters from the disk file
read help screen K-boundaries and quadrants from the disk file
burp the buffer
close the disk file
RETURN

Reset Sound Variables
set the current sound array to the default
set the current play string to NULL
set the current envelopes array to the default
set the current envelopes
set the current volume to the default
set the current tempo to the default
zero out the current filter array
set the current filter
set the current frame to 1
set the frame pointers to 0
RETURN

Draw A Fresh Screen
set a black border and black background

go to an undisturbed 40–column text screen
set a green 40–column text pen
go to a cleared 40–column text screen
set a black bit–map pen
go to a cleared 40–column bit–map screen
Draw Six Windows
Draw Frame Counter
Draw Nine Buttons & A Window
Draw Help Button
RETURN

Update The Screen
Update Envelopes Window
Update Volume Window
Update Tempo Window
Update Filter Window
Update Frame Counter
Update Message Window
RETURN

Load And Install Binary Files
load the sprite data file FINGER CURSOR from the default drive
load the assembly language file S/M ASM 1 from the default drive
load the assembly language file S/M ASM 2 from the default drive
load the help screen data file S/M HELP PACK from the default drive
CALL on Install (from S/M ASM 2) to install the pseudo–mouse routines
RETURN

Initialize Cursor

move sprite #1 to the middle of the screen
turn sprite #1 on
set sprite #1's foreground color to black
set sprite #1 to appear in front of screen objects
set sprite #1 to normal (un–expanded) size
set sprite #1 to be a multi–color sprite
set sprite multi–color 1 to white
RETURN

Draw Six Windows
set a data pointer
FOR
each of six windows (sound, play, envelopes, volume, filter, and tempo)
DO the following
read in the window's data
draw the window's outline in red
draw the window's title in light blue
CALL ON the window's customization routine (if any) :
Customize Sound Window, Customize Play Window,
Customize Envelope Window, or Customize Filter Window
RETURN

Update Sound Window
set a data pointer
FOR
each of 8 sound parameters
DO the following
turn the parameter's current value into a string
read the parameter's display area width, display color,

and display area starting column
fit the string to the display area width
draw the string in the display color, starting at the display area starting column
RETURN

Update Envelopes Window
FOR
each of 9 envelopes
DO the following
CALL on Update An Envelope
RETURN

Update An Envelope
set a data pointer
read the two envelope parameter colors
adjust colors for this envelope
FOR
each of the envelope's six parameters

DO the following
set a color for the parameter
FOR
each of the envelope's six parameters
DO the following
turn the parameter's current value into a string
read the parameter's display area width and display area starting column
adjust the string to fit display area width
draw the string in the parameter's color, starting at the display area starting column
RETURN

Sound Click
CALL on Invert An Area to invert the Sound window title
CALL on Update Sound Window to draw the current sound array
set target area to where the pseudo-mouse click occurred
IF
target area's the window title
THEN
move the target area to the voice parameter
send some feedback via Update Message Window
{ this next unit is referred to as the REPEAT..UNTIL
parameter-fetching loop }
REPEAT the following
CALL on Type One Sound for a beep
set up so Fetch A Parameter will act on the target area and its parameter
invert the target area's label via HRBandInvt (from S/M ASM 2)
CALL on Fetch A Parameter to get a parameter value for the target area
{ Fetch A Parameter returns via a pseudo-mouse click,
which we'll call the FAP return click }
{ Fetch A Parameter returns a value for the parameter
it was to fetch, which we'll call the FAP exit value }
normalize the target area's label via HRBandInvt (from S/M ASM 2)
IF
the FAP return click is not in the Sound window
AND
the FAP return click is not in the Go button
THEN
save the area where the pseudo-mouse click occurred
JUMP down to this routine's exit block
UNTIL

the FAP exit value is valid
IF
the FAP exit value is different from the target area's current parameter
THEN
change the target area's current parameter
IF
the FAP return click is in the Sound window title

THEN do the following
give some feedback via Update Message Window
invert the frame counter via Invert An Area
Play Current Sound
Let Sound Finish
set frame type to 'sound'
set frame data to contents of the current sound array
IF
a CALL to Record A Sound Frame is successful
THEN
JUMP back to the top of the REPEAT..UNTIL parameter–fetching loop
ELSE
set the pseudo–mouse click area to NONE
JUMP to this routine's exit block
ELSE IF
the FAP return click is in the Sound window
THEN do the following
make that area the target area
JUMP back to the top of the REPEAT..UNTIL parameter–fetching loop
ELSE { the FAP return click is in the Go Button }
give some feedback via Update Message Window
invert the Go button via Invert An Area
Play Current Sound

Let Sound Finish
normalize the Go button via Invert An Area
JUMP back to the top of the REPEAT..UNTIL parameter–fetching loop
{ this is the exit block referred to above }
CALL on Update Message Window to clear the message area
clear the Sound window parameter data area
CALL on Invert An Area to normalize the sound window title
JUMP back into the Lab Event Loop to check out the pseudo–mouse click area

Invert An Area

CALL on HRRectInvt (from S/M ASM 2) with a pointer to
the table of lab area inversion rectangles and an area number
RETURN

Play Click

CALL on Invert An Area to invert the Play window title
initialize the play string editing cursor to the upper left corner of the Play window data area
CALL on Type One Sound for a beep
{ this is where we come to edit the play string }
send some feedback via Update Message Window
set up for a CALL to StrngRectEdit
CALL on StrngRectEdit (from S/M ASM 1) to edit the current
play string within the Play window's data area
{ StrngRectEdit returns via a pseudo–mouse click, which we'll call the SRE return click }

remember the editing cursor's position for further editing
IF
the SRE return click is in the Go button

THEN do the following
invert the Go button via Invert An Area
give some feedback via Update Message Window
play the current play string
normalize the Go button via Invert An Area
JUMP back up to edit the play string some more
ELSE IF
the SRE return click is in the Play window title
THEN do the following
IF
there's no room to store the play string in the string storage array
THEN do the following
Send An Error Message
set the pseudo–mouse click area to NONE
JUMP to this routine's exit block
{ it's at the end of this routine's pseudo–code }
ELSE { there's room to store the play string }
give some feedback via Update Message Window
invert the frame counter via Invert An Area
play the current play string
set frame type to 'play'
set frame data to the string storage array index
store the play string in the string storage array
IF
a CALL to Record A Sound Frame is successful
THEN do the following
increment the string storage array index
JUMP back up to edit the play string some more
ELSE { the recording failed }

Selected Algorithms From SOUND/MUSIC LAB Sheet 10 Of 43

set the pseudo–mouse click area to NONE
JUMP to this routine's exit block
ELSE { the SRE return click is neither in the Play window nor the Go button }
set the pseudo–mouse click area to the SRE return click's area
FALL THRU to this routine's exit block
{ this is the exit block referred to above }
CALL on Update Message Window to clear the message area
CALL on Customize Play Window to clear the play window data area
CALL on Invert An Area to normalize the play window title
JUMP back into the Lab Event Loop to check out the pseudo–mouse click area

Envelope Click
CALL on Invert An Area to invert the Envelope window title
set target area to where the pseudo–mouse click occurred
IF

the target area is set to the Envelope window's title
THEN
move the target area to the first envelope's first parameter
use the target area to determine which envelope we're working with
IF
the target area is set to an envelope's number
THEN
move the target area to that envelope's first parameter
use the target area to determine which envelope parameter we're working with
{ this next unit is referred to as the parameter–fetching block }
send some feedback via Update Message Window
CALL on Type One Sound for a beep
set up so Fetch A Parameter will act on the target area and its parameter
invert the target area's envelope number via HRBandInvt (from S/M ASM 2)
invert the target area's parameter label via HRBandInvt (from S/M ASM 2)

Selected Algorithms From SOUND/MUSIC LAB Sheet 12 Of 43

CALL on Fetch A Parameter to get a parameter value
{Fetch A Parameter returns via a pseudo–mouse click,
which we'll call the FAP return click}
{Fetch A Parameter returns a value for the parameter
it was to fetch, which we'll call the FAP exit value }
normalize the target area's parameter label via HRBandInvt (from S/M ASM 2)
normalize the target area's envelope number via HRBandInvt (from S/M ASM 2)
IF
the FAP return click is outside the Envelope window
THEN
save the area where the pseudo–mouse click occurred
JUMP to the exit block { at the end of this routine's pseudo–code }
IF
the FAP exit value is invalid
THEN
JUMP back up to the top of the parameter–fetching block
IF
the FAP exit value is different from the target area's current parameter value
THEN
change the target area's parameter value to the FAP exit value
CALL on Set Current Envelope to change the target area's envelope
IF
the FAP return click isn't in the Envelope window title
THEN
set the target area to the FAP return click's area
JUMP back up to the top of the parameter fetching block
ELSE { the FAP return click was in the title }
give some feedback via Update Message Window
give a beep via a CALL to Type One Sound

Selected Algorithms From SOUND/MUSIC LAB Sheet 13 Of 43

invert the frame counter via Invert An Area
set frame type to 'envelope'

set frame data to the envelope number and its parameters
IF
a CALL to Record A Sound Frame is successful
THEN do the following
JUMP back up to the top of the parameter fetching block
ELSE { the recording failed }
set the pseudo–mouse click area to NONE
FALL THRU to this routine's exit block
{ this is the exit block referred to above }
CALL on Update Message Window to clear the message area
CALL on Update An Envelope to redraw the last envelope fiddled with
CALL on Invert An Area to normalize the Envelope window title
JUMP back into the Lab Event Loop to check out the pseudo–mouse click area

Set Current Envelope
CALL on Update Message Window to print some feedback
CALL on Type One Sound for a beep
do a short Pause
CALL on Update Message Window to clear the feedback
set the envelope to the current envelope array values
RETURN

Volume Click
CALL on Invert An Area to invert the Volume window title
{ this next unit is referred to as the parameter–fetching block }
CALL on Update Message Window to send some feedback
CALL on Type One Sound for a beep

set up for a CALL to Fetch A Parameter
CALL on Fetch A Parameter to get a new volume value
{ Fetch A Parameter returns via a pseudo–mouse click,
which we'll call the FAP return click }
{ Fetch A Parameter returns a value for the parameter
it was to fetch, which we'll call the FAP exit value }
IF
the FAP return click is outside the Volume window
THEN
set the pseudo–mouse click area to the FAP return click area
JUMP to the exit block { at the end of this routine's pseudo–code }
IF
the FAP exit value is invalid
THEN
JUMP back up to the top of the parameter–fetching block
IF
the FAP exit value is a change from the current volume
THEN do the following
set the volume to the FAP exit value
send some feedback via Update Message Window
CALL Type One Sound for a beep
IF
the FAP return click isn't in the Volume window title

THEN
JUMP back up to the parameter–fetching block
ELSE { the FAP return click was in the title }
give some feedback via Update Message Window
give a beep via Type One Sound
invert the frame counter via Invert An Area
set frame type to 'volume'

set frame data to the new volume
IF
a CALL to Record A Sound Frame is successful
THEN
JUMP back up to the parameter–fetching block
ELSE
set the pseudo–mouse click area to NONE
FALL THRU to this routine's exit block
{ this is the exit block referred to above }
CALL Update Message Window to clear the message area
CALL Update Volume Window to make 'er pretty again
CALL Invert An Area to normalize the Volume window title
JUMP back into the Lab Event Loop to check out the pseudo–mouse click area

Tempo Click
CALL Invert An Area to invert the Tempo window title
{ this next unit is referred to as the parameter–fetching block }
CALL Update Message Window to send some feedback
CALL Type One Sound for a beep
set up for a CALL to Fetch A Parameter
CALL Fetch A Parameter to get a new tempo value
{ Fetch A Parameter returns via a pseudo–mouse click,
which we'll call the FAP return click }
{ Fetch A Parameter returns a value for the parameter
it was to fetch, which we'll call the FAP exit value }
IF
the FAP return click is outside the Tempo window
THEN
set the pseudo–mouse click area to the FAP return click area

JUMP to the exit block { at the end of this routine's pseudo–code }
IF
the FAP exit value is invalid
THEN
JUMP back up to the top of the parameter–fetching block
IF
the FAP exit value is a change from the current tempo
THEN do the following
set the tempo to the FAP exit value
send some feedback via Update Message Window
CALL Type One Sound for a beep

IF
    the FAP return click isn't in the Tempo window title
THEN
    JUMP back up to the parameter–fetching block
ELSE { the FAP return click was in the title }
    give some feedback via Update Message Window
    give a beep via Type One Sound
    invert the frame counter via Invert An Area
    set frame type to 'tempo'
    set frame data to the new tempo
    IF
        a CALL to Record A Sound Frame is successful
    THEN
        JUMP back up to the parameter–fetching block
    ELSE
        set the pseudo–mouse click area to NONE
        FALL THRU to this routine's exit block
{ this is the exit block referred to above }
CALL Update Message Window to clear the message area

CALL Update Tempo Window to make 'er pretty again
CALL Invert An Area to normalize the Tempo window title
JUMP back into the Lab Event Loop to check out the pseudo–mouse click area

Filter Click
CALL Invert An Area to invert the Filter window title
set target area to where the pseudo–mouse click occurred
IF
    the target area is set to the Filter window title
THEN
    move the target area to the first filter parameter
{ this next unit is referred to as the parameter–fetching block }
CALL Update Message Window to send some feedback
CALL Type One Sound for a beep
set up so Fetch A Parameter will act on the target area and its parameter
invert the target area's parameter label via HRBandInvt (from S/M ASM 2)
CALL Fetch A Parameter to get a parameter value
{ Fetch A Parameter returns via a pseudo–mouse click,
    which we'll call the FAP return click }
{ Fetch A Parameter returns a value for the parameter
    it was to fetch, which we'll call the FAP exit value }
normalize the target area's parameter label via HRBandInvt (from S/M ASM 2)
IF
    the FAP return click is outside the Filter window
THEN
    set the pseudo–mouse click area to the FAP return click area
    JUMP to the exit block { at the end of this routine's pseudo–code }
IF
    the FAP exit value is invalid

THEN
    JUMP back up to the top of the parameter–fetching block
IF
    the FAP exit value is different from the target area's current parameter value
THEN
    change the target area's parameter value to the FAP exit value
    CALL Set Current Filter to change the target area's envelope
IF
    the FAP return click isn't in the Filter window title
THEN
    set the target area to the FAP return click's area
    JUMP back up to the top of the parameter fetching block
ELSE { the FAP return click was in the title }
    give some feedback via Update Message Window
    give a beep via a CALL to Type One Sound
    invert the frame counter via Invert An Area
    set frame type to 'filter'
    set frame data to the five filter parameters
    IF
        a CALL to Record A Sound Frame is successful
    THEN do the following
        JUMP back up to the top of the parameter fetching block
    ELSE { the recording failed }
        set the pseudo–mouse click area to NONE
        FALL THRU to this routine's exit block
{ this is the exit block referred to above }
CALL Update Filter Window to redraw the Filter window
CALL Update Message Window to clear the message area
CALL Invert An Area to normalize the Filter window title

JUMP back into the Lab Event Loop to check out the pseudo–mouse click area

Set Current Filter
    CALL Update Message Window to print some feedback
    CALL Type One Sound for a beep
    do a short Pause
    CALL Update Message Window to clear the feedback
    set the filter to the current filter array values
    RETURN

Frame Click
    CALL on Invert An Area to invert the frame counter label
    { this is the top of the parameter fetching block }
    CALL on Update Message Window to announce the click
    CALL on Type One Sound for a beep
    set up for a CALL to Fetch A Parameter
    CALL Fetch A Parameter to get a frame counter value
    { Fetch A Parameter returns via a pseudo–mouse click,
        which we'll call the FAP return click }

{ Fetch A Parameter returns a value for the parameter
it was to fetch, which we'll call the FAP exit value }

IF
    the FAP return click is NOT in the frame counter
        AND
    the FAP return click is NOT in the frame counter label
THEN
    set the pseudo–mouse click area to the FAP return click area
    JUMP to this routine's exit block
IF
    the FAP exit value is invalid

        OR
    the FAP exit value equals the current frame counter value
THEN
    JUMP back up to the top of the parameter–fetching block
{ we have a valid and novel frame counter value }
save the current frame counter value
set the frame counter to the FAP exit value
CALL Show Frame to show the new frame counter value
IF
    the FAP return click is in the frame counter
THEN
    JUMP back up to the top of the parameter–fetching block
ELSE { the FAP return click is in the frame counter label }
    CALL on Update Message Window to announce we'll record
    set up to record it at the saved frame counter value
    CALL on Invert An Area to invert the frame counter
    set frame data type to 'frame'
    set frame data to the new frame counter value
    IF
        a CALL to Record A Sound Frame is successful
    THEN
        JUMP back up to the top of the parameter fetching block
    ELSE { the recording attempt was unsuccessful }
        set the pseudo–mouse click area to NONE
        FALL THRU to this routine's exit block
{ this is the exit block referred to above }
CALL on Update Message Window to clear any messages
CALL on Update Frame Counter to show the final frame counter value
CALL on Invert An Area to normalize the frame counter label

JUMP back into the Lab Event Loop to check out the pseudo–mouse click area

Go Click

CALL on Invert An Area to invert the Go button
CALL on Update Message Window to announce the button
CALL on Type One Sound to announce the button

```
save the current frame counter value
{ this is where we check the frame counter }
IF
      the current frame is not recorded yet
THEN
      JUMP down to this routine's last frame handler
{ the current frame is recorded }
IF
      there's a pseudo-mouse click
THEN do the following
      CALL on AreaSearch ( from S/M ASM 2 ) to determine the p-m click's area
      IF
            the user clicked the pseudo-mouse outside the Go button
      THEN
            JUMP down to this routine's user break handler
fetch the frame's data offset
fetch the frame's type
CALL on Update Message Window to tell about the frame
CALL on Update Frame Counter to show the current frame counter value
CASE OUT on the frame type to do a frame :
IF
      the frame type is 'sound'
THEN
```



```
      use the frame's data to make a sound
ELSE IF
      the frame type is 'play'
THEN
      use the frame's data to play a string
ELSE IF
      the frame type is 'envelope'
THEN
      use the frame's data to set an envelope
ELSE IF
      the frame type is 'volume'
THEN
      use the frame's data to set a volume
ELSE IF
      the frame type is 'tempo'
THEN
      use the frame's data to set a tempo
ELSE IF
      the frame type is "filter'
THEN
      use the frame's data to set a filter
ELSE IF
      the frame type is 'frame'
THEN
      use the frame's data to change the frame counter
IF
      the frame type is anything other than 'frame'
```

THEN

increment the frame counter

JUMP back up to where we check the frame counter

{ this is the last frame handler mentioned above }

CALL on Update Message Window to announce last recorded frame

CALL on Type One Sound for a beep

set pseudo–mouse click area to NONE

JUMP to this routine's exit block

{ this is the user break handler mentioned above }

CALL on Update Message Window to announce that the user has stopped the playback

CALL on Type One Sound for a beep

set the pseudo–mouse click area to where the user clicked

FALL THRU to this routine's exit block

{ this is the exit block referred to above }

CALL on Update Message Window to clear any messages

restore the frame counter to the saved entry value

CALL on Invert An Area to normalize the Go button

JUMP back into the Lab Event Loop to check out the pseudo–mouse click area

Forward Click

CALL on Invert An Area to invert the Forward button

CALL on Update Message Window to announce the button

CALL on Type One Sound to announce the button

REPEAT the following

increment the frame counter, with wraparound

CALL on Update Frame Counter to show the changed counter

UNTIL

the pseudo–mouse button gets let up

CALL on Show Frame, With Recorded Check to announce the changed frame counter

CALL on Invert An Area to normalize the Forward button

CALL on Update Message Window to clear all messages

JUMP back to the top of Lab Event Loop

Load Click

CALL on Invert An Area to invert the Load button

CALL on Update Message Window to announce the button

CALL on Type One Sound to announce the button

Get A File Name, returning with a name and a pseudo–mouse area

IF

the returned pseudo–mouse area IS NOT the Load button

THEN

set the pseudo–mouse click area to the returned area

JUMP to this routine's exit block

IF

the returned file name is 0 characters long

```
THEN
      set the pseudo-mouse click area to NONE
      JUMP to this routine's exit block
CALL on Update Message Window to announce file opening
IF
      an attempt to open a fresh file with the returned name for sequential reading fails
THEN
      JUMP to this subroutine's disk problems block
{ the file opened successfully }
Reset Sound Variables
CALL on Update Message Window to announce data loading
read important frame data variables from the file
IF
      the frame data stack will hold some data
THEN
```

Selected Algorithms From SOUND/MUSIC LAB Sheet 25 Of 43

```
      FOR
            each data element in the frame data stack
      DO the following
            read the data element from the file into the stack
IF
      the array of frame data offsets will hold some values
THEN
      FOR
            each value in the array of frame data offsets
      DO the following
            read the value from the file into the array
IF
      there will be any strings in the array of frame strings
THEN
      FOR
            each string in the array of frame strings
      DO the following
            read the string from the file into the array
IF
      there were reported disk problems
THEN
      JUMP to this routine's disk problems block
ELSE  { the file reading worked }
      speed up to 2 megahertz  { VIC screen disappears }
      set up some feedback
      Update The Screen
      slow down to 1 megahertz  { VIC screen appears }
      pause 2 seconds
      close the file
```

Selected Algorithms From SOUND/MUSIC LAB Sheet 26 Of 43

```
      set the pseudo-mouse click area to NONE
      JUMP to this subroutine's exit block
{ this is the disk problems block referred to above }
```

CALL on Send An Error Message to announce the problem
CALL on Clear Click to clear and redraw the screen
close the file
set the pseudo–mouse click area to NONE
FALL THRU to this subroutine's exit block
{ this is the exit block referred to above }
CALL on Update Message Window to clear any messages
CALL on Invert An Area to normalize the Load button
JUMP back into the Lab Event Loop to check out the pseudo–mouse click area

Get A File Name
CALL on Update Message Window to print a prompt
CALL on Type One Sound for a beep
Pause
CALL on StrngRectEdit (from S/M ASM 1) to fetch a file name
save the edit–ending pseudo–mouse click area
CALL on Strip TP$ Trailing Blanks to clean the file name
RETURN

Clear Click
CALL on Invert An Area to invert the Clear button
CALL on Update Message Window to announce clearing
CALL on Type One Sound for a beep
Pause
speed up to 2 megahertz speed { VIC screen disappears }
Reset Sound Variables

set a message for upcoming message window update
Update The Screen { our message will show up }
slow down to 1 megahertz speed { VIC screen appears }
CALL on Type One Sound for a beep
Pause
CALL on Update Message Window to clear messages
CALL on Invert An Area to normalize the Clear button
JUMP back to the top of the Lab Event Loop

Help Click
CALL on Invert An Area to invert the Help button
CALL on Type One Sound for a beep
CALL on Update Message Window to clear the message window
save VM1, which holds screen and character locations
speed up to 2 megahertz { VIC screen disappears }
CALL on Invert An Area to normalize the Help button
set GraphM, which sets the VIC display mode, to 40–column text
move the finger cursor sprite so it'll show up on the help screen's Quit button
set VIC's sights on RAM bank 1 by setting bit 6 of the MMU's
RAM configuration register
set VIC's sights on this help screen's memory quadrant by
fiddling with bits 0 and 1 of CIA #2 Port A
set VIC's sights on this help screen's text by fiddling with VM1
slow down to 1 megahertz { VIC screen appears }

{ we're now looking at a help screen }
{ this is the button scanner }
REPEAT
WAIT UNTIL
there's a pseudo–mouse click
CALL on AreaSearch (from S/M ASM 2) to find the p–m click's area

UNTIL
the click is inside one of the help screen's areas
{ we've got a click inside one of the help screen's areas,
each of which represents a button }
CALL on Tx40BandInvt (from S/M ASM 2) to invert the clicked button's text
CASE OUT on the clicked button
IF
it's a click of the First button
THEN
set the help screen to the first help screen
ELSE IF
it's a click of the Previous button
THEN
set the help screen to the previous help screen, with wraparound
ELSE IF
it's a click of the Next button
THEN
set the help screen to the next help screen, with wraparound
ELSE IF
it's a click of the Last button
THEN
set the help screen to the last help screen
ELSE IF
it's a click of the Quit button
THEN
JUMP to this subroutine's exit block
CALL on Type One Sound for a beep
CALL on Tx40BandInvt (from S/M ASM 2) to normalize the
clicked button's text

set VIC's sights on the help screen's text by fiddling with VM1
set VIC's sights on the help screen's memory quadrant by
fiddling with bits 0 and 1 of CIA #2 Port A
JUMP back up to this routine's button scanner
{ this is the exit block referred to above }
CALL on Type One Sound for a beep
speed up to 2 megahertz { VIC screen disappears }
CALL on Tx40BandInvt (from S/M ASM 2) to normalize the clicked button's text
set VIC's sights on RAM bank 0 by clearing bit 6 of the MMU's
RAM configuration register
set VIC's sights on the 0th memory quadrant by setting bits

0 and 1 of CIA #2 Port A
restore VM1 to the saved entry value
set GraphM to 40–column bit–map
move the finger cursor sprite so it'll show up on the lab screen's Help button
slow down to 1 megahertz { VIC screen appears }
JUMP back to the top of the Lab Event Loop

Show Frame Click
CALL on Invert An Area to invert the Show Frame button
CALL on Invert An Area to invert the frame counter
IF
there are no frames recorded
THEN
Send An Error Message
set the pseudo–mouse click area to NONE
JUMP to this routine's exit block
{ there are frames recorded }
IF

the current frame is not recorded yet
THEN do the following
set the current frame to the highest recorded frame
CALL on Invert An Area to normalize the frame counter
CALL on Update Frame Counter to display the new frame counter value
CALL on Invert An Area to invert the frame counter
{ this is the top of the frame display loop }
CALL on Update Message Window to announce the frame
CALL on Type One Sound for a beep
fetch the frame's data offset
fetch the frame's type
CALL on Invert An Area to invert the frame's type's title area
CASE OUT on the frame type as follows to show the frame:
IF
the frame type is 'sound'
THEN
set the current sound array to the frame's data
CALL on Update Sound Window to show it
ELSE IF
the frame type is 'play'
THEN
set the current play string to the frame's stored string
CALL on Update Play Window to show it
ELSE IF
the frame type is 'envelope'
THEN
figure the envelope number
set that envelope's parameters to the frame's stored data
CALL on Set Current Envelope to set the envelope
CALL on Update An Envelope to display it

CALL on HRBandInvt (from S/M ASM 2) to invert the envelope's number
ELSE IF
the frame type is 'volume'
THEN
set the current volume to the frame's data
CALL on Update Volume Window to display the new volume level
ELSE IF
the frame type is 'tempo'
THEN
set the current tempo to the frame's data
CALL on Update Tempo Window to display the new tempo level
ELSE IF
the frame type is "filter'
THEN
set the current filter array to the frame's data
CALL on Set Current Filter to set that array
CALL on Update Filter Window to display the current filter array
ELSE IF
the frame type is 'frame'
THEN
CALL on Invert An Area to normalize the frame counter
set the frame counter to the frame's data
CALL on Update Frame Counter to display the changed frame counter
WAIT UNTIL
there's a pseudo–mouse click
save the pseudo–mouse click's area for a later test
CALL on AreaSearch (from S/M ASM 2) to figure the click's area
CALL on Invert An Area to normalize the frame's type's title area
CASE OUT on the frame type as follows to un–show the frame:

IF
the frame type is 'sound'
THEN
clear the Sound window's data area
ELSE IF
the frame type is 'play'
THEN
CALL on Customize Play Window to clear the Play window's data area
ELSE IF
the frame type is 'envelope'
THEN
CALL on HRBandInvt (from S/M ASM 2) to normalize the envelope's
number
ELSE IF
the frame type is 'frame'
THEN
set the frame counter back to what it was
CALL on Update Frame Counter to display the changed frame counter value
CALL on Invert An Area to invert the frame counter

{ here's where we test the saved pseudo–mouse click area }

IF

    the saved pseudo–mouse click area is not the Print button

THEN

    set the pseudo–mouse click area to the saved area

    JUMP to this routine's exit block

move the current frame to the next frame, with wraparound

CALL on Invert An Area to normalize the frame counter

CALL on Update Frame Counter to display the changed current frame

CALL on Invert An Area to invert the frame counter

JUMP back up to the top of the frame display loop

{ this is the exit block referred to above }

CALL on Update Message Window to clear messages

CALL on Invert An Area to normalize the frame counter

CALL on Invert An Area to normalize the Show Frame button

JUMP back into the Lab Event Loop to check out the pseudo–mouse click area

Backward Click

CALL on Invert An Area to invert the Backward button

CALL on Update Message Window to announce the button

CALL on Type One Sound to announce the button

REPEAT the following

    decrement the frame counter, with wraparound

    CALL on Update Frame Counter to show the changed counter

UNTIL

    the pseudo–mouse button gets let up

CALL on Show Frame, With Recorded Check to announce the

    changed frame counter

CALL on Update Message Window to clear all messages

CALL on Invert An Area to normalize the Backward button

JUMP back to the top of Lab Event Loop

Save Click

CALL on Invert An Area to invert the Save button

CALL on Update Message Window to announce the button

CALL on Type One Sound to announce the button

Pause

Get A File Name, returning with a name and a pseudo–mouse area

IF

    the returned pseudo–mouse area IS NOT the Save button

THEN

    set the pseudo–mouse click area to the returned area

    JUMP to this routine's exit block

IF

    the returned file name is 0 characters long

THEN

set the pseudo–mouse click area to NONE
JUMP to this routine's exit block
CALL on Update Message Window to announce file opening
IF
an attempt to open a fresh file with the returned name
for sequential writing fails
THEN
JUMP to this subroutine's disk problems block
{ the file opened successfully }
CALL on Update Message Window to announce data saving
file–print important frame data variables
IF
the frame data stack holds some data
THEN
FOR
each data element in the frame data stack
DO the following
file–print the data element
IF
the array of frame data offsets holds some values
THEN
FOR
each value in the array of frame data offsets

DO the following
file–print the value
IF
there are any strings in the array of frame strings
THEN
FOR
each string in the array of frame strings
DO the following
file–print the string
burp the file buffer
IF
there were reported disk problems
THEN
JUMP to this subroutine's disk problems block
ELSE { the file–printing probably worked }
CALL on Update Message Window to announce the successful save
Pause
close the file
set the mouse–click area to NONE
JUMP to this subroutine's exit block
{ this is the disk problems block referred to above }
CALL on Send An Error Message to announce the problem
close the file
set the mouse–click area to NONE
FALL THRU to this subroutine's exit block
{ this is the exit block referred to above }
CALL on Update Message Window to clear any messages

CALL on Invert An Area to normalize the Save button
JUMP back into the Lab Event Loop to check out the pseudo–mouse click area

Strip TP$ Trailing Blanks
figure out the length of TP$
use that length to set an array index to the final character in TP$
WHILE
the indexed character of TP$ is a space
AND
the array index's value is greater than 0
DO the following
move the array index one character to the left
set TP$ to its leftmost value–of–the–index characters
RETURN

Print Click
call on Invert An Area to invert the Print button
call on Update Message Window to send some feedback
call on Type One Sound for a beep
STARTING with
the first frame
FOR
each recorded frame
DO the following
IF
there's a pseudo–mouse click
THEN do the following
call on AreaSearch (from S/M ASM 2) to find out where the click occurred
IF
the pseudo–mouse click occurred outside the Print button
THEN

call on Update Message Window to send some feedback
call on Type One Sound for a beep
set the pseudo–mouse click area to where the click occurred
JUMP to this routine's exit block
call on Update Message Window for some feedback
get the frame's data offset
get the frame's type
open the printer for output
printer–print the frame number
IF
the frame type is 'sound'
THEN do the following
printer–print the frame type and the stored sound command
ELSE IF
the frame type is 'play'

```
        THEN do the following
            grab the stored play string
            call on Strip TP$ Trailing Blanks to clean up the stored play string
            printer-print the frame type and the cleaned up play string
        ELSE IF
            the frame type is 'envelope'
        THEN do the following
            printer-print the frame type and the stored envelope command
        ELSE IF
            the frame type is 'volume'
        THEN do the following
            printer-print the frame type and the stored volume command
        ELSE IF
            the frame type is 'tempo'
```



```
        THEN do the following
            printer-print the frame type and the stored tempo command
        ELSE IF
            the frame type is 'filter'
        THEN do the following
            printer-print the frame type and the stored filter command
        ELSE IF
            the frame type is 'frame'
        THEN do the following
            printer-print the frame type and the stored frame-to-jump-to
        IF
            the printer's printed 60 lines ( a page's worth )
        THEN
            move to the next sheet of paper by printing 6 carriage returns
        burp the printer buffer
        close the printer
        renew the default disk drive device number
    set the pseudo-mouse click area to NONE
    { this is the exit block referred to above }
    call on Update Message Window to clear any feedback
    call on Invert An Area to normalize the Print button
    JUMP back into the Lab Event Loop to check out the pseudo-mouse click area
```

End Click

```
    call on Invert An Area to invert the End button
    call on Update Message Window for some feedback
    make a beeping sound
    call on Update Message Window for some feedback
    make a beeping sound
    call on Update Message Window for some feedback
```



```
    make a beeping sound
    call on Invert An Area to normalize the End button
    set the finished flag to 'finished'
```

JUMP back to the top of Lab Event Loop

Fetch A Parameter

IF
    the entry value is out of bounds
THEN
    drag it in
make a string version of the entry value
figure out the length of the stringized entry value
figure out the width of the parameter's display band
pad the stringized entry value with zeroes to fill the parameter's display band
set up for a CALL to StrngRectEdit
CALL on StrngRectEdit (from S/M ASM 1) to edit the
    entry value string within its display band
set an exit value by making a real number version of the edited entry value string
IF
    the exit value is an invalid value
THEN
    set the result code to 'exit value is invalid'
    set the exit value to the entry value
    Send An Error Message
ELSE { the exit value is a valid value }
    set the result code 'exit value is valid'
make a string version of the exit value
figure out the length of the stringized exit value
pad the stringized exit value with zeroes to fill the parameter's display band

set up for a printing–only CALL to StrngRectEdit
CALL on StrngRectEdit (from S/M ASM 1) to print the exit
    value string within its display band
RETURN

Record A Sound Frame

IF
    there's no room on the frame data stack
        OR
    there are no frames left to work with
        OR
    there's no room on the frame string stack
THEN
    CALL on Invert An Area to normalize the frame counter
    Send An Error Message
    RETURN with a result code of 'failure'
{ we have enough resources to record the frame }
FOR
    each of the frame's data elements
DO the following
    push the frame data element onto the frame data stack
    store the frame's data block offset into the array of frame
        data block offsets
IF

the current frame's number is higher than the topmost
recorded frame number
THEN
set the topmost recorded frame number to the current frame's number
CALL on Update Message Window to send some feedback
CALL on Type One Sound for a beep

CALL on Invert An Area to normalize the frame counter
increment the frame counter, with wraparound
show the new frame counter value with a CALL to Update Frame Counter
RETURN with a result code of 'success'

Type One Sound
set maximum volume
initiate the particular sound
Let Sound Finish
restore current lab volume
RETURN

Type Three Sound
set maximum volume
initiate the particular sound
Let Sound Finish
restore current lab volume
RETURN

Let Sound Finish
WAIT UNTIL
the System sound variable SoundTime resets
RETURN

Show Frame, With Recorded Check
IF
the frame hasn't been recorded yet
THEN do the following
CALL Update Message Window to send some feedback

CALL Type Three Sound for a beep
CALL on Show Frame to announce the frame
RETURN

Show Frame
CALL Update Message Window to announce the frame
CALL Type One Sound for a beep
count from 1 to 250 for a pause
RETURN

Send An Error Message

```
    set maximum volume
    WHILE
        the error message is at least one character in length
    DO the following
        grab the leftmost 16 characters of the error message
        DO the following 2 TIMES
            CALL Update Message Window to show the grabbee
            CALL Type Three Sound for a beep
            count from 1 to 120 for a pause
            CALL Update Message Window to clear the show
            count from 1 to 60 for a pause
        remove the leftmost 16 characters from the error message
    restore current lab volume
    RETURN

Error Handler
    build up an error message from the error descriptor string
        and the error site's line number
```



```
    Send An Error Message
    RESUME the program at the line following the error site's line

Pause
    count from 1 to 100
    RETURN
```

Chapter 16: Program Listings

This chapter consists of eight figures, as follows:

Fig. 16-1—code for S/M Asm 1.
Fig. 16-2—code for S/M Asm 2.
Fig. 16-3—code for S/M Help Packer.
Fig. 16-4—code for Make S/M Vars.
Fig. 16-5—code for Make 40C Screens.
Fig. 16-6—code for 40C Edit.
Fig. 16-7—list of variables for Sound/Music Lab.
Fig. 16-8—code for Sound/Music Lab.

```


*------------------ Program Identification --------------*
*                                                        *
*                         S/M ASM 1                      *
*                                                        *
* Assembly language tools for the BASIC 7.0 program      *
* ... SOUND/MUSIC LAB                                    *
*                                                        *
* Divided into three source files :                      *
*                                    S/M ASM 1 A.S       *
*                                    S/M ASM 1 B.S       *
*                                    S/M ASM 1 C.S       *
*                                                        *
* Lives in RAM Bank 0 at addresses $0400-$07E3           *
*                                                        *
*                                                        *
* To call the StrngRectEdit routine :                    *
*                                                        *
*      you need to set up a parameter block, then        *
*      ... call the routine                              *
*                                                        *
*      take a look at the comments at the head of        *
*      ... StrngRectEdit for more detail about each      *
*      ... item in the parameter block                   *
*                                                        *
*      the parameter block can be implemented from       *
*      ... BASIC 7.0 as an array of 8 integer values     *
*                                                        *
*      in the examples that follow,                      *
*                                                        *
*                 SRE% () is an array of 8 integer values *
*                                                        *
*                 EN$     is an entry string which       *
*                         ... provides the characters to *
*                         ... be edited                  *
*                                                        *
*                 EX$     is an exit string which will   *
*                         ... come out of the routine    *
*                         ... with the edited string     *
*                                                        *
*                 TR      is the topmost row of the      *
*                         ... editing rectangle [0..24]  *
*                                                        *
*                 BR      is the bottommost row of the   *
*                         ... editing rectangle [0..24]  *
*                                                        *
*                 LC      is the leftmost column of the  *
*                         ... editing rectangle [0..39]  *
*                                                        *
*                 RC      is the rightmost column of the *
*                         ... editing rectangle [0..39]  *
*                                                        *
*                 TR      is the topmost row of the      *
*                         ... editing rectangle [0..39]  *
*                                                        *
*                 AC      is an area identification code *
*                         ... for the editing rectangle  *
*                                                        *
*                 CI      is the initial 0-based offset  *
*                         ... of the editing cursor      *
*                         ... within the string          *
*                                                        *
*                 CF      is the final 0-based offset    *
```

Fig. 16-1. Source code for S/M Asm 1.

```
*                       ... of the editing cursor        *
*                       ... within the string            *
*                                                        *
*   okay, here's what each element of the parameter      *
*   ... block needs to get, along with an example of     *
*   ... BASIC 7.0 code.                                  *
*                                                        *
*     0th item   gets a pointer to an entry editing      *
*                ... string                              *
*     example :                                          *
*                SRE% (0) = POINTER (EN$)                *
*                                                        *
*                                                        *
*     1st item   gets a pointer to an exit editing       *
*                ... string                              *
*     example :                                          *
*                SRE% (1) = POINTER (EX$)                *
*                                                        *
*                                                        *
*     2nd item   gets the topmost row of the editing     *
*                ... rectangle [0..24]                   *
*     example :                                          *
*                SRE% (2) = TR                           *
*                                                        *
*                                                        *
*     3rd item   gets the bottommost row of the          *
*                ... editing rectangle [0..24]           *
*     example :                                          *
*                SRE% (3) = BR                           *
*                                                        *
*                                                        *
*     4th item   gets the leftmost column of the         *
*                ... editing rectangle [0..39]           *
*     example :                                          *
*                SRE% (4) = LC                           *
*                                                        *
*                                                        *
*     5th item   gets the rightmost column of the        *
*                ... editing rectangle [0..39]           *
*     example :                                          *
*                SRE% (5) = TR                           *
*                                                        *
*                                                        *
*     6th item   gets an area identification code        *
*                ... for the rectangle                   *
*     example :                                          *
*                SRE% (6) = AC                           *
*                                                        *
*     if the id code sent is 0, the routine will         *
*     return on ANY pseudo-mouse click                   *
*                                                        *
*                                                        *
*     7th item   gets a pointer to an area data table    *
*                                                        *
*         example :                                      *
*                   SRE% (7) = DEC ("15A3")              *
*                                                        *
*                                                        *
*     8th item   gets an initial 0-based position for    *
*                ... the editing cursor                  *
*                                                        *
*         example :                                      *
*                   SRE% (8) = CI                        *
*                                                        *
*                                                        *
```

```
*         when calling the routine :                         *
*                                                            *
*              the address of the parameter block gets       *
*              ... passed to the routine in the A- and X-    *
*              ... registers                                 *
*                                                            *
*              the Y- register holds a function selector :   *
*                        0   just print the string           *
*                        1   print and edit the string       *
*                                                            *
*         after setting up the parameter block, here's       *
*         ... how you can call StrngRectEdit :               *
*                                                            *
*         SYS 1024, POINTER ( SRE% (0) )  AND  255,          *
*               INT ( POINTER ( SRE% (0) )  /  256 ), 1      *
*                                                            *
*         when the routine returns control to BASIC,         *
*         ... here's how you can find out which area         *
*         ... the pseudo-mouse was in when the user          *
*         ... chose to exit and where the editing cursor     *
*         ... ended up                                       *
*                                                            *
*         RREG AC, CF                                        *
*         PRINT "THE PSEUDO-MOUSE WAS IN AREA" AC            *
*         PRINT "THE EDITING CURSOR ENDED UP AT" CF          *
*                                                            *
*                                                            *
* Version :      1.00                                        *
* Timestamp :    5:55 PM PST     September 16, 1986          *
*                                                            *
* Programmed by Stan Krute                                   *
* Copyright (C) 1986 by Stan Krute's Hacker & Nerd           *
*                        18617 Camp Creek Road               *
*                        Hornbrook, California   96044       *
*                        [916] 475-3428                      *
* All rights reserved                                        *
* Call or write for bug reports, help, licensing, etc.       *
*                                                            *
*------------------------------------------------------------*

*--------------------- Constants -----------------------*


* bit-map stuff

BMBase    =     $2000       ; start of the standard hi-res
                            ; ... bit-map


* character ROM stuff

CharRom   =     $D000       ; starting address for character
                            ; ... ROM


* Commodore ASCII codes

CrsrDnCAC =     17          ; code for cursor down
CrsrLfCAC =     157         ; code for cursor left
CrsrRtCAC =     29          ; code for cursor right
CrsrUpCAC =     145         ; code for cursor up
DeleteCAC =     20          ; code for delete
InsertCAC =     148         ; code for insert
LeftArrowCAC =  95          ; code for a left arrow
SpaceCAC =      32          ; code for a space
```

```
* cursor stuff

CursWidth =    1           ; width in character positions
                           ; ... of the editing cursor


* data structure stuff

SREPrmBlkSiz = 18          ; size of a parameter block
                           ; ... record for StrngRectEdit


* low-memory system variables

StaVec    =    $2B9        ; points to a zero-page pointer
                           ; ... for IndSta ROM call


* memory configuration stuff

Bank14    =    %00000001   ; configuration byte for Bank 14
MmuCR     =    $FF00       ; the always-available memory
                           ; ... configuration register


* ROM routines -- documented

GetIn     =    $FFE4       ; read buffered data from
                           ; ... current input device
IndFet    =    $FF74       ; fetch data from any bank
IndSta    =    $FF77       ; store data to any bank


* routines from S/M ASM 2.0

AreaSearch =   $1468       ; see if a mouse click is in
                           ; ... a defined screen area
HRBandInvt =   $1512       ; hi-res band inversion


* sprites

SprHzAdj =     24-8        ; horizontal adjustment factor
                           ; ... for sprite-screen
                           ; ... coordinate conversions
SprVtAdj =     50          ; vertical adjustment factor
                           ; ... for sprite-screen
                           ; ... coordinate conversions


* variables from S/M ASM 2.0

ButnStat =     $1B14       ; status of the pseudo-mouse
                           ; ... button
ClikHzHi =     $1B16       ; hi-byte of horizontal
                           ; ... location of a pseudo-mouse
                           ; ... click
ClikHzLo =     $1B15       ; lo-byte of horizontal
                           ; ... location of a pseudo-mouse
                           ; ... click
ClikVt   =     $1B17       ; vertical location of a
                           ; ... pseudo-mouse click
```

```
* zero-page variables

OurPtr1  =     $FA        ; a general-purpose pointer
OurPtr2  =     $FC        ; a general-purpose pointer
OurPtr3  =     $C8        ; a general-purpose pointer
OurPtr4  =     $CA        ; a general-purpose pointer


* --------------------- Macros -------------------------*

* a nice pseudo-unconditional branch

BRA      MAC
         CLV
         BVC   ]1
         <<<


*----------------- Set Program Origin ------------------*

* since SOUND/MUSIC LAB operates in graphics mode 1
* ... ( hi-res. bit-map) we are able to use $0400-$07F7
* ... for this group of routines

         ORG   $0400      ; that's 1024 in decimal, folks


*------------------- StrngRectEdit --------------------*

* edits a character string that fills a row/column
* ... rectangle on the standard hi-res bit-map screen

* can also be used to just print the string
* ... in its rectangle

* the character string must have a length of 1..255
* ... characters

* the string length should be equal to the area of the
* ... row/column rectangle ( numberRows * numberColumns)

* actually, you send the routine two strings of equal
* ... length : an entry string containing the character
* ... information to edit, and an exit string that the
* ... routine will actually work on

* upon exit from the procedure, the exit string will
* ... contain an edited version of the entry string,
* ... and the entry string will be unchanged

* also, the A- register will contain an area code
* ... identifying where the pseudo-mouse was when
* ... the user chose to exit the routine via a
* ... pseudo mouse click

* also, the X- register will contain a 0-based offset
* ... indicating where the editing cursor finished
* ... up at in the string

* allows the user to type characters, use Insert and
* ... Delete keys, use the cursor keys, and use the
* ... joystick-controlled mouse as she/he edits

* upon entry, A- (lo-byte) and X- (hi-byte) point to a
```

```
                * ... parameter block record of the following form :

                *  offset      contents
                *  ------      -----------------------------------------
                *    0         hi-byte of entry string record
                *    1         lo-byte of entry string record
                *    2         hi-byte of exit string record
                *    3         lo-byte of exit string record
                *    4         filler byte
                *    5         topmost row of editing rectangle
                *    6         filler byte
                *    7         bottommost row of editing rectangle
                *    8         filler byte
                *    9         leftmost row of editing rectangle
                *   10         filler byte
                *   11         rightmost row of editing rectangle
                *   12         filler byte
                *   13         area identifier for the rectangle
                *   14         hi-byte of area data list
                *   15         lo-byte of area data list
                *   16         filler byte
                *   17         initial position for editing cursor
                *              ... in the string

                * the string records have the following form :

                *  offset      contents
                *  ------      -----------------------------------------
                *    0         length of string
                *    1         lo-byte of address of actual string bytes
                *    2         hi-byte of address of actual string bytes

                * upon entry, the Y- register contains a function selector

                *              0   just print the string
                *              1   print and edit the string


                * see the Program Identification block for information on
                * ... setting up for and calling this routine from BASIC

                * parameters aren't checked for wackitude, so send
                * ... 'em in correct

                * all registers are trashed

                * to make things a bit more comprehensible, yet still
                * ... easily self-contained, I've used a liberal
                * ... sprinkling of local subroutines

                StrngRectEdit
                * store function flag
0400: 8C F2 06          STY   :FuncFlag

                * store pointer to the parameter block
0403: 85 C8             STA   OurPtr3
0405: 86 C9             STX   OurPtr3+1

                * store some items from the parameter block
0407: 20 44 04          JSR   :StorStuf

                * set up strings
040A: 20 8E 04          JSR   :SetStrgz

                * initialize some editing variables
```

```
040D: 20 6A 04           JSR   :InitEdVars

                * draw the exit string on the bit-map screen
0410: A9 00              LDA   #0          ; start at 0th char
0412: AE FD 06           LDX   :Length     ; end at last char
0415: CA                 DEX
0416: 20 5B 06           JSR   :DrwStrSec ; draw that section

                * see if we're here just to print, or to edit, too
0419: AC F2 06           LDY   :FuncFlag  ; grab the flag
041C: F0 25              BEQ   :ByeTwo     ; branch on it

                :EditLoopTop
                * this is the top of the editing loop

                * draw the editing cursor in its current position
041E: 20 99 05           JSR   :InvertCursor

                * figure the string's hot spot
0421: 20 A5 05           JSR   :FigHotSpot

                :LookKey
                * look for a keypress
0424: 20 E4 FF           JSR   GetIn       ; look for keyboard input
0427: F0 07              BEQ   :LookMouse ; no keypress, so next test

                * we got a keypress, so go deal with it
0429: 20 DE 04           JSR   :DealKey

                * if we come back with Carry set, do an exit
                * otherwise, back up to top of loop
042C: B0 0C              BCS   :ByeOne
042E: 90 EE              BCC   :EditLoopTop ; then back up to top of loop

                :LookMouse
                * look for a pseudo-mouse button click
0430: AD 14 1B           LDA   ButnStat    ; check the p-m button state
0433: F0 EF              BEQ   :LookKey    ; no p-m click, so look at
                                           ; ... keyboard again

                * we got a pseudo-mouse click, so go deal with it
0435: 20 32 05           JSR   :DealMouse

                * if we come back with Carry set, do an exit
                * otherwise, back up to top of loop
0438: 90 E4              BCC   :EditLoopTop ; then back up to top of loop

                :ByeOne
                * erase the editing cursor in its current position
043A: 20 99 05           JSR   :InvertCursor

                * pick up an exit area code
043D: AD 01 07           LDA   :AreaID

                * pick up cursor position within string
0440: AE FE 06           LDX   :HotSpot

                :ByeTwo
                * return from StrngRectEdit
0443: 60                 RTS


                *-------- local subroutines ------*

                *------- :StorStuf ------*
```

```
                * store some items from the parameter block
                * upon exit, all registers trashed
                :StorStuf
                * initialize Y- register to index into the param. block
                * ... and count thru the coming loop
0444: A0 11               LDY    #SREPrmBlkSiz-1
                * initialize X- register to index into a table of
                * ... destinations for the param. block info
0446: A2 23               LDX    #SREPrmBlkSiz*2-1
                :StSLoop
                * okay, we can unload that block in a sweet loop
                * set up next byte's destination address
0448: BD 04 07            LDA    :WherTab,X  ; set the hi-byte of a
044B: 8D 64 04            STA    :Store+2    ; ... destination address ---
                                             ; WARNING ---
                                             ; ... we actually modify the
                                             ; ... storage address that's
                                             ; ... in the code, breaking a
                                             ; ... generally important
                                             ; ... programming convention :
                                             ; ... NO SELF-MODIFYING CODE !!!
                                             ; ... but it works, is safe,
                                             ; ... and I'm not the one
                                             ; ... who designed a non-
                                             ; ... orthogonal instruction
                                             ; ... set anyways
                                             ; ... --- Defensive Stan
044E: CA                  DEX
044F: BD 04 07            LDA    :WherTab,X ; lo-byte of dest. address
0452: 8D 63 04            STA    :Store+1
                * go grab a byte of the parameter block
                * ... ( which is in the other RAM bank)
0455: 8E F8 06            STX    :Temp1      ; save X- register
0458: A9 C8               LDA    #OurPtr3    ; point to zero-page pointer
045A: A2 01               LDX    #1          ; indicate bank 1
045C: 20 74 FF            JSR    IndFet      ; fetch a byte from Bank 1
045F: AE F8 06            LDX    :Temp1      ; restore X- register
                * store the byte from the parameter block
0462: 8D F9 06  :Store    STA    :Temp2      ; this command is the one we
                                             ; ... modify by changing the
                                             ; ... storage address
                * bottom of the loop
0465: CA                  DEX                ; down those index/counters
0466: 88                  DEY
0467: 10 DF               BPL    :StSLoop    ; loop until finished
                * return from :StorStuf
0469: 60                  RTS
                          TTL    "S/M ASM 1 B.S"
                *------------ Here Comes The Second Source File ----------
                          PUT    "S/M ASM 1 B.S"
                *-------- :InitEdVars --------*
```

```
                   * initialize some editing variables

                   * upon exit, A- and Y- registers trashed

                   :InitEdVars

                   * get the editing cursor to upper-left corner
                   * ... of the editing rectangle
046A: AD F5 06             LDA   :Left
046D: 8D FA 06             STA   :EdCrsHz
0470: AD F3 06             LDA   :Top
0473: 8D FB 06             STA   :EdCrsVt

                   * now move it to where it's to start
0476: AC FC 06             LDY   :InCrsPs    ; grab the goal
0479: F0 06                BEQ   :FigWid     ; if 0 we're already there

047B: 20 A8 06     :ICPLup JSR   :CursRit    ; move one position to the right
047E: 88                   DEY               ; down the count
047F: D0 FA                BNE   :ICPLup     ; go 'til there

                   :FigWid
                   * figure out the width of the editing rectangle
                   * this'll be used to update the string's hot spot
0481: 38                   SEC               ; straightforward 1-byte
0482: AD F6 06             LDA   :Right      ; ... [m6[m6ubtraction
0485: ED F5 06             SBC   :Left
0488: AA                   TAX
0489: E8                   INX
048A: 8E F7 06             STX   :Width

                   * return from :InitEdVars
048D: 60                   RTS


                   *-------- :SetStrgz --------*

                   * gets the length of the strings, copies the entry
                   * ... string to the exit string, and returns a
                   * ... pointer to the exit string (which is the string
                   * ... we'll be working with)

                   * upon entry, OurPtr1 and OurPtr2 point
                   * to the entry and exit string records respectively

                   * upon exit, OurPtr2 points to the actual exit string

                   * trashes A- and Y- registers


                   * remember :  the string records have the following form :

                   *  offset      contents
                   *  ------      -----------------------------------------
                   *    0         length of string
                   *    1         lo-byte of address of actual string bytes
                   *    2         hi-byte of address of actual string bytes


                   :SetStrgz

                   * initialize Y- for indexing
048E: A0 00                LDY   #0
```

```
                  * get string lengths
                  * remember, both strings have same length
0490: A9 FA                LDA   #OurPtr1    ; point to a pointer
0492: A2 01                LDX   #1          ; indicate Bank 1
0494: 20 74 FF             JSR   IndFet      ; grab length of entry string
0497: 8D FD 06             STA   :Length      ; store it

                  * set pointers to actual strings
                  * grab lo-bytes of string addresses first, and stack 'em
049A: C8                   INY               ; move index to lo-byte
                                             ; ... address of string
049B: A9 FA                LDA   #OurPtr1    ; point to a pointer
049D: A2 01                LDX   #1          ; indicate Bank 1
049F: 20 74 FF             JSR   IndFet      ; grab lo-byte address of
                                             ; ... entry string
04A2: 48                   PHA               ; park on stack

04A3: A9 FC                LDA   #OurPtr2    ; point to a pointer
04A5: A2 01                LDX   #1          ; indicate Bank 1
04A7: 20 74 FF             JSR   IndFet      ; grab lo-byte address of
                                             ; ... exit string
04AA: 48                   PHA               ; park on stack

                  * now grab the hi-bytes of the string addresses,
                  * ... and store 'em
04AB: C8                   INY               ; move index to hi-byte
                                             ; ... address of string
04AC: A9 FA                LDA   #OurPtr1    ; point to a pointer
04AE: A2 01                LDX   #1          ; indicate Bank 1
04B0: 20 74 FF             JSR   IndFet      ; grab hi-byte address of
                                             ; ... entry string
04B3: 85 FB                STA   OurPtr1+1   ; and store it

04B5: A9 FC                LDA   #OurPtr2    ; point to a pointer
04B7: A2 01                LDX   #1          ; indicate Bank 1
04B9: 20 74 FF             JSR   IndFet      ; grab hi-byte address of
                                             ; ... exit string
04BC: 85 FD                STA   OurPtr2+1    ; and store it

                  * pull lo-bytes off the stack, and store 'em
                  * (we do this juggling to economize on pointers)
04BE: 68                   PLA               ; get lo-byte address of
                                             ; ... exit string
04BF: 85 FC                STA   OurPtr2      ; and store it
04C1: 68                   PLA               ; do same for entry string
04C2: 85 FA                STA   OurPtr1      ; and store it


                  * okay, we've got some pointers, so it's copy time
                  * set up for a copying loop
04C4: A0 00                LDY   #0          ; Y- will count and index
04C6: A9 FC                LDA   #OurPtr2    ; set up inter-bank storage
                                             ; ... vector
04C8: 8D B9 02             STA   StaVec

                  * loop to copy bytes of entry string to exit string
04CB: A9 FA       :SSLoop  LDA   #OurPtr1    ; point to zero-page pointer
04CD: A2 01                LDX   #$01        ; indicate Bank 1
04CF: 20 74 FF             JSR   IndFet      ; grab a byte of entry string
                                             ; byte comes back in A- register

04D2: A2 01                LDX   #1          ; indicate Bank 1
04D4: 20 77 FF             JSR   IndSta      ; store a byte of entry string

                  * loop bottom
```

```
04D7: C8                INY                ; up the counter
04D8: CC FD 06          CPY   :Length      ; done yet ?
04DB: 90 EE             BCC   :SSLoop      ; loop 'til done
                * return from :SetStrgz
04DD: 60                RTS
                *-------- :DealKey --------*
                * deal with a keypress
                * upon entry, A- holds the keypress' C-ASCII code
                * upon exit, all registers trashed
                * upon exit, Carry flag set   signals exit time
                *              Carry flag clear signals edit some more
                :DealKey
                :PrChTest
                * is it a printable character keypress ?
04DE: C9 20             CMP   #SpaceCAC         ; we want it in the range
04E0: 90 09             BCC   :CrsUpTest        ; ... Space..LeftArrow
04E2: C9 60             CMP   #LeftArrowCAC+1
04E4: B0 05             BCS   :CrsUpTest        ; if not in range, next
                                           ; ... test
                * yes, it's a printable character keypress
                * go deal with it
04E6: 20 CA 05          JSR   :OveRite
                * signal for more editing and leave
04E9: 18                CLC
04EA: 60                RTS
                :CrsUpTest
                * is it a cursor-up keypress ?
04EB: C9 91             CMP   #CrsrUpCAC ; check it out
04ED: D0 08             BNE   :CrsDnTest ; if not, next test
                * yes, it's a cursor-up keypress
                * erase the cursor at its present position
04EF: 20 99 05          JSR   :InvertCursor
                * adjust the cursor position upwards
04F2: 20 E1 06          JSR   :CursUp
                * signal for more editing and leave
04F5: 18                CLC
04F6: 60                RTS
                :CrsDnTest
                * is it a cursor-down keypress ?
04F7: C9 11             CMP   #CrsrDnCAC ; check it out
04F9: D0 08             BNE   :CrsLfTest ; if not, next test
                * yes, it's a cursor-down keypress
                * erase the cursor at its present position
04FB: 20 99 05          JSR   :InvertCursor
```

```
               * adjust the cursor position downwards
04FE: 20 D0 06          JSR   :CursDwn

               * signal for more editing and leave
0501: 18                CLC
0502: 60                RTS


               :CrsLfTest
               * is it a cursor-left keypress ?
0503: C9 9D             CMP   #CrsrLfCAC ; check it out
0505: D0 08             BNE   :CrsRtTest ; if not, next test

               * yes, it's a cursor-left keypress
               * erase the cursor at its present position
0507: 20 99 05          JSR   :InvertCursor

               * adjust the cursor position leftwards
050A: 20 BC 06          JSR   :CursLft

               * signal for more editing and leave
050D: 18                CLC
050E: 60                RTS


               :CrsRtTest
               * is it a cursor-right keypress ?
050F: C9 1D             CMP   #CrsrRtCAC ; check it out
0511: D0 08             BNE   :InsrTest  ; if not, next test

               * yes, it's a cursor-right keypress
               * erase the cursor at its present position
0513: 20 99 05          JSR   :InvertCursor

               * adjust the cursor position rightwards
0516: 20 A8 06          JSR   :CursRit

               * signal for more editing and leave
0519: 18                CLC
051A: 60                RTS


               :InsrTest
               * is it an insert keypress ?
051B: C9 94             CMP   #InsertCAC ; check it out
051D: D0 05             BNE   :DeleTest  ; if not, next test

               * yes, it's an insert keypress
               * go deal with it
051F: 20 E9 05          JSR   :Insert

               * signal for more editing and leave
0522: 18                CLC
0523: 60                RTS


               :DeleTest
               * is it a delete keypress ?
0524: C9 14             CMP   #DeleteCAC ; check it out
0526: D0 05             BNE   :KPNoGood  ; if not, keypress is no good

               * yes, it's a delete keypress
               * go deal with it
0528: 20 1F 06          JSR   :Delete

```

```
                 * signal for more editing and leave
052B: 18                CLC
052C: 60                RTS


                 :KPNoGood
                 * the keypress is not one we deal with

                 * erase the cursor at its present location
                 * ( so the calling loop can redraw it )
052D: 20 99 05          JSR   :InvertCursor

                 * signal for more editing and leave
0530: 18                CLC
0531: 60                RTS


                 *-------- :DealMouse --------*

                 * deal with a pseudo-mouse click

                 * upon exit, all registers trashed

                 * upon exit, Carry flag set   signals exit time
                 *           Carry flag clear signals edit some more
                 *                 if it's exit time, A- register holds
                 *                 ... an area id code

                 :DealMouse

                 * where'd it happen ? call the AreaSearch routine
0532: AD 02 07          LDA   :ArDtLo    ; point to the area data table
0535: AE 03 07          LDX   :ArDtHi
0538: 20 68 14          JSR   AreaSearch ; call the routine

                 * are we looking for an area ?
                 * a non-zero area identification number says "yes"
                 * ... and zero says "no"
053B: AE 01 07          LDX   :AreaID    ; check it out
053E: F0 54             BEQ   :NotLookin

                 * we are looking for an area
                 * if result is not our rectangle's area id, it's
                 * ... time to exit
0540: CD 01 07          CMP   :AreaID    ; check it out
0543: D0 4F             BNE   :DMExitBye ; if not ours

                 * the mouse has been pressed in our rectangle's area
                 * so we'll move the cursor to the mouse click

                 * invert the cursor at its current location, thereby
                 * ... erasing it
0545: 20 99 05          JSR   :InvertCursor

                 * convert horizontal point of mouse click, which is in
                 * ... sprite coordinates, to a text-screen-like 0..39
                 * ... horizontal coordinate

                 * we first convert the sprite horizontal position to
                 * ... a screen horizontal position
0548: 38                SEC              ; prepare to subtract
0549: AD 15 1B          LDA   ClikHzLo   ; get lo-byte of sprite
                                         ; ... horizontal position
054C: E9 10             SBC   #<SprHzAdj ; subtract lo-byte of sprite
                                         ; ... horizontal adjustment
```

```
                                             ; ... factor
054E: 48                  PHA                ; save the result
054F: AD 16 1B            LDA   ClikHzHi     ; get hi-byte of sprite
                                             ; ... horizontal position
0552: E9 00               SBC   #>SprHzAdj   ; subtract hi-byte of sprite
                                             ; ... horizontal adjustment
                                             ; ... factor

         * now, just divide the screen horizontal position by 8
         * ... via repeated right shifts
0554: 4A                  LSR                ; shift hi-byte, putting its
                                             ; ... vital bit into Carry
0555: 68                  PLA                ; get back lo-byte of screen
                                             ; ... horizontal position
0556: 6A                  ROR                ; shift it, including effect
                                             ; ... of hi-byte
0557: 4A                  LSR                ; two more shifts will do it
0558: 4A                  LSR

         * make sure this new horizontal coordinate is in bounds

         :DMLfTst
         * test against editing rectangle's left coordinate
0559: CD F5 06            CMP   :Left        ; want to be greater than or
                                             ; ... equal to left coordinate
055C: B0 06               BCS   :DMRiTst     ; okay, so test against right
                                             ; ... coordinate

         * outside left boundary, so rope 'er in
055E: AD F5 06            LDA   :Left        ; replace with left side
                                             ; ... coordinate
                          BRA   :DMHzStr     ; go store it
0561: B8                  CLV
0562: 50 0A               BVC   :DMHzStr
                          <<<

         :DMRiTst
         * test against editing rectangle's right coordinate
0564: CD F6 06            CMP   :Right       ; want to be less than or
                                             ; ... equal to right coordinate
0567: 90 05               BCC   :DMHzStr     ; we're okay, go store
0569: F0 03               BEQ   :DMHzStr     ; we're okay, go store

         * outside right boundary, so rope 'er in
056B: AD F6 06            LDA   :Right

         :DMHzStr
         * store the editing cursor's new horizontal coordinate
056E: 8D FA 06            STA   :EdCrsHz

         * convert vertical point of mouse click, which is in
         * ... sprite coordinates, to a text-screen-like 0..24
         * ... vertical coordinate

         * we first convert the sprite vertical position to
         * ... a screen vertical position
0571: 38                  SEC                ; prepare to subtract
0572: AD 17 1B            LDA   ClikVt       ; get sprite vertical position
0575: E9 32               SBC   #SprVtAdj    ; subtract sprite vertical
                                             ; ... adjustment factor

         * now, just divide the screen vertical position by 8
         * ... via repeated right shifts
0577: 4A                  LSR                ; three right shifts will do it
0578: 4A                  LSR
```

```
0579: 4A                  LSR

                  * make sure this new vertical coordinate is in bounds

                  :DMTopTst
                  * test against editing rectangle's top coordinate
057A: CD F3 06            CMP   :Top         ; want to be greater than or
                                             ; ... equal to top coordinate
057D: B0 06               BCS   :DMBtmTst    ; okay, so test against bottom
                                             ; ... coordinate

                  * outside top boundary, so rope 'er in
057F: AD F3 06            LDA   :Top         ; replace with top coordinate
                          BRA   :DMVtStr     ; go store it
0582: B8                  CLV
0583: 50 0A               BVC   :DMVtStr
                          <<<

                  :DMBtmTst
                  * test against editing rectangle's bottom coordinate
0585: CD F4 06            CMP   :Bottom      ; want to be less than or
                                             ; ... equal to bottom coordinate
0588: 90 05               BCC   :DMVtStr     ; we're okay, go store
058A: F0 03               BEQ   :DMVtStr     ; we're okay, go store

                  * outside bottom boundary, so rope 'er in
058C: AD F4 06            LDA   :Bottom

                  :DMVtStr
                  * store the editing cursor's new vertical coordinate
058F: 8D FB 06            STA   :EdCrsVt

                  :DMMorBye
                  * signal more editing and return from :DealMouse
0592: 18                  CLC
0593: 60                  RTS

                  :NotLookin
                  * we're not looking for a specific area, just a
                  * ... pseudo-mouse click
                  * when we get one, it's time to exit

                  :DMExitBye
                  * store the exit area's id
0594: 8D 01 07            STA   :AreaID

                  * signal exit time and return from :DealMouse
0597: 38                  SEC
0598: 60                  RTS


                  *---- :InvertCursor ----*

                  * inverts the cursor at its present position
                  * ... by calling our hi-res band inversion routine

                  * all registers trashed

                  :InvertCursor
                  * set up for inversion function
0599: AD FA 06            LDA   :EdCrsHz     ; A- holds starting column
059C: AE FB 06            LDX   :EdCrsVt     ; X- holds row
059F: A0 01               LDY   #CursWidth   ; Y- holds band width

                  * call it
```

```
05A1: 20 12 15        JSR   HRBandInvt ; invert away

                * return from :InvertCursor
05A4: 60              RTS


                *---- :FigHotSpot ----*

                * figure the string's hot spot
                * ... based on current cursor position

                * as the cursor moves, the hot spot follows

                * the formula is as follows :

                * :HotSpot = ( ( :EdCrsVt - :Top ) * :Width )
                *              + ( :EdCrsHz - :Left )

                * remember, strings are limited to 255 characters

                * trashes all registers

                :FigHotSpot
                * figure what relative row we're in  :EdCrsVt  -  :Top
05A5: 38              SEC
05A6: AD FB 06        LDA   :EdCrsVt
05A9: ED F3 06        SBC   :Top
05AC: AA              TAX              ; we'll use relative row
                                       ; ... for loop counting

                * initialize product to 0 and go to the bottom
                * ... of the multiplication loop
05AD: A9 00           LDA   #0
05AF: F0 04           BEQ   :FHSLpBt   ; branch always

                * multiply :Width by relative row using repeated additions
05B1: 18        :FHSLpTp CLC
05B2: 6D F7 06        ADC   :Width     ; add another width to sum

                * bottom of the multiplication loop
05B5: CA        :FHSLpBt DEX           ; down the loop counter
05B6: 10 F9           BPL   :FHSLpTp   ; back up if not finished

                * at this point, A- register holds
                * ... ( :EdCrsVt - :Top ) * :Width
                * let's park it for a moment ...
05B8: 8D F9 06        STA   :Temp2

                * okay, now we'll figure relative column
                * figure what relative column we're in with the formula
                * ... :EdCrsHz  -  :Left
05BB: 38              SEC              ; standard subtraction
05BC: AD FA 06        LDA   :EdCrsHz
05BF: ED F5 06        SBC   :Left

                * now, add that offset to our earlier result
                * ... and we're done
05C2: 18              CLC              ; prep to add
05C3: 6D F9 06        ADC   :Temp2     ; add
05C6: 8D FE 06        STA   :HotSpot   ; store the result

                * return from :FigHotSpot
05C9: 60              RTS
```

```
                *-------- :OveRite --------*
                * overwrites a printable character
                * puts it into the exit string
                * draws it on the screen
                * erases the cursor
                * moves the cursor one string position to the right
                * upon entry, the character's C-ASCII code is in
                * ... the A- register
                * upon exit, all registers trashed
                :OveRite
                * save a copy of the character
05CA: 48                 PHA
                * let's add the character to the exit string
                * set a pointer to the exit string pointer for
                * ... inter-bank storage
05CB: A2 FC              LDX   #OurPtr2
05CD: 8E B9 02           STX   StaVec
                * add the character to the exit string
                * remember, the character's sitting in the A- register
05D0: AC FE 06           LDY   :HotSpot    ; get index into string
05D3: A2 01              LDX   #1          ; indicate Bank 1
05D5: 20 77 FF           JSR   IndSta      ; store the character
                * erase cursor at current cursor position
05D8: 20 99 05           JSR   :InvertCursor
                * draw the new character at current cursor position
05DB: 68                 PLA               ; get the character back
05DC: AE FB 06           LDX   :EdCrsVt    ; X- holds a vertical
                                           ; ... coordinate (row)
05DF: AC FA 06           LDY   :EdCrsHz    ; Y- holds a horizontal
                                           ; ... coordinate (column)
05E2: 20 28 07           JSR   DrawBMChar  ; call char drawing routine
                * move cursor location to the right, dealing
                * ... with any necessary wraparound issues
05E5: 20 A8 06           JSR   :CursRit
                * return from :OveRite
05E8: 60                 RTS
                *-------- :Insert --------*
                * deal with a press of the insertion key
                * moves all characters from the hotspot thru to the
                * ... next-to-last character one position to the right
                * ... ( the last character gets bumped into its next
                * ...   existence )
                * puts a space character at the hotspot
                * upon exit, all registers trashed
                :Insert
```

```
                * set a pointer to the exit string pointer for
                * ... inter-bank storage
05E9: A9 FC              LDA   #OurPtr2
05EB: 8D B9 02           STA   StaVec

                :SlideRite
                * make room for the insertion by sliding all the characters
                * ... from :TheSpot thru to the next-to-last char
                * ... one position to the right

                * initialize for the loop
05EE: AC FD 06           LDY   :Length     ; we'll start at string end

                * enter the slide loop at the bottom
05F1: D0 0E              BNE   :SRBotm

                * slide an exit string character to the right
05F3: 88        :SRTop   DEY               ; point to target

05F4: A9 FC              LDA   #OurPtr2    ; point to our pointer
05F6: A2 01              LDX   #$01        ; indicate Bank 1
05F8: 20 74 FF           JSR   IndFet      ; grab a char

05FB: A2 01              LDX   #$01        ; indicate Bank 1
05FD: C8                 INY               ; move one position right
05FE: 20 77 FF           JSR   IndSta      ; store the char

                * bottom of the :SlideRite loop
0601: 88        :SRBotm  DEY               ; down the index
0602: CC FE 06           CPY   :HotSpot    ; are we done sliding ?
0605: D0 EC              BNE   :SRTop      ; no, so back to loop top

                * okay, we've slid stuff over to make room, so now
                * ... we can add a spacey character to the string
0607: A9 20              LDA   #SpaceCAC   ; get that character code
0609: AC FE 06           LDY   :HotSpot    ; get index into string
060C: A2 01              LDX   #1          ; indicate Bank 1
060E: 20 77 FF           JSR   IndSta      ; store the character

                * let's redraw the exit string from the hotspot thru
                * ... to the end of the string
0611: AD FE 06           LDA   :HotSpot    ; set up A- reg. to index
                                           ; ... the hotspot
0614: AE FD 06           LDX   :Length     ; set up X- reg. to index
0617: CA                 DEX               ; ... the last char
0618: 20 5B 06           JSR   :DrwStrSec  ; go draw that section of
                                           ; ... the exit string

                * erase cursor at current cursor position
061B: 20 99 05           JSR   :InvertCursor

                * return from :Insert
061E: 60                 RTS

                         TTL   "S/M ASM 1 C.S"
                *------------ Here Comes The Third Source File -----------*

                         PUT   "S/M ASM 1 C.S"


                *-------- :Delete --------*

                * deal with a press of the deletion key

```

```
                  * moves all characters from the hotspot thru to the
                  * ... last character one position to the left
                  * ... ( the character to the left of the hotspot gets
                  * ...   bumped into its next existence )

                  * puts a space character at the last character

                  * moves cursor one spot to left

                  * upon exit, all registers trashed

                  :Delete

                  * set a pointer to the exit string pointer for
                  * ... inter-bank storage
061F: A9 FC                LDA   #OurPtr2
0621: 8D B9 02             STA   StaVec

                  * erase cursor at current cursor position
0624: 20 99 05             JSR   :InvertCursor

                  :SlideLeft
                  * perform a deletion by sliding all the characters from
                  * ... :TheSpot thru to last char one position to the left

                  * initialize for the loop
0627: AC FE 06             LDY   :HotSpot    ; we'll start at the hotspot

                  * check for the no-no case
062A: F0 2E                BEQ   :DBye       ; if hotspot is at position 0
                                             ; ... we've got nothing to delete

                  * the top of the sliding loop
                  * slide an exit string character to the left
062C: A9 FC       :SLTop   LDA   #OurPtr2    ; point to our pointer
062E: A2 01                LDX   #$01        ; indicate Bank 1
0630: 20 74 FF             JSR   IndFet      ; grab a char

0633: A2 01                LDX   #$01        ; indicate Bank 1
0635: 88                   DEY               ; move one position left
0636: 20 77 FF             JSR   IndSta      ; store the char

0639: C8                   INY               ; move back to target
063A: C8                   INY               ; move on to next target
                  * bottom of the :SlideLeft loop
063B: CC FD 06    :SLBotm  CPY   :Length     ; are we done sliding ?
063E: D0 EC                BNE   :SLTop      ; no, so back to loop top

                  * okay, we've slid stuff over to make room, so now
                  * ... we can add a spacey character to the string
0640: A9 20                LDA   #SpaceCAC   ; get that character code
0642: AC FD 06             LDY   :Length     ; get index into string
0645: 88                   DEY
0646: A2 01                LDX   #1          ; indicate Bank 1
0648: 20 77 FF             JSR   IndSta      ; store the character

                  * let's redraw the exit string from the hotspot-1 thru
                  * ... to the end of the string
064B: AE FE 06             LDX   :HotSpot    ; set up A- reg. to index
                                             ; ... the hotspot-1
064E: CA                   DEX
064F: 8A                   TXA
0650: AE FD 06             LDX   :Length     ; set up X- reg. to index
0653: CA                   DEX               ; ... the last char
0654: 20 5B 06             JSR   :DrwStrSec  ; go draw that section of
```

```
                                           ; ... the exit string

                * move cursor to the left, dealing with wraparound
0657: 20 BC 06          JSR   :CursLft

                :DBye
                * return from :Delete
065A: 60                RTS


                *-------- :DrwStrSec -------*

                * draws a section of the exit string in its rectangle
                * ... on the bit-map screen

                * upon entry, A- indexes the first char of the section
                *             X- indexes the last char of the section

                * upon exit, all registers are trashed

                :DrwStrSec
                * save the entry parameters
065B: 8D F8 06          STA   :Temp1     ; we'll use this as an index
065E: E8                INX              ; we'll use this as a stop value
065F: 8E F9 06          STX   :Temp2     ; ... so we up it for easitude

                :DSSDoIt
                * go draw a character
0662: 20 71 06          JSR   :DrwStrChr

                * up the index
0665: EE F8 06          INC   :Temp1

                * grab the index
0668: AD F8 06          LDA   :Temp1

                * are we done yet ?
066B: CD F9 06          CMP   :Temp2     ; at stop value yet ?
066E: 90 F2             BCC   :DSSDoIt   ; if not, draw again

                * return from :DrwStrSec
0670: 60                RTS


                *-------- :DrwStrChr --------*

                * draw a character from the exit string on the
                * ... bit-map screen

                * upon entry, A- holds the character's 0-based
                * ... position in the string  [ 0..stringLength-1]

                * upon exit, all registers trashed

                :DrwStrChr

                * store a copy of the character's position in Y- register
                * ... for upcoming IndFet call
0671: A8                TAY

                * we need to figure absolute row and column of char
                * first, figure the relative row
                * starting assumption : row 0
0672: A2 00             LDX   #0

```

```
                 * we'll do some repeated subtraction
                 * A- register holds index to first char
0674: 38                  SEC                ; prep to subtract
0675: ED F7 06   :DSCSub  SBC   :Width       ; takeaway a width
0678: 90 03               BCC   :GotRRow     ; if below 0, branch
067A: E8                  INX                ; up the row
067B: B0 F8               BCS   :DSCSub      ; subtract again

                 :GotRRow
                 * we've got the relative row,
                 * ... so now figure the relative column
067D: 18                  CLC                ; just add back a width
067E: 6D F7 06            ADC   :Width

                 * make that an absolute column and store it
0681: 18                  CLC
0682: 6D F5 06            ADC   :Left
0685: 8D 00 07            STA   :AbsCol

                 * make relative row into absolute row and store it
0688: 18                  CLC
0689: 8A                  TXA
068A: 6D F3 06            ADC   :Top
068D: 8D FF 06            STA   :AbsRow

                 * make sure that the indexed character will fit into
                 * ... our rectangle by checking the absolute row
0690: CD F4 06            CMP   :Bottom      ; can't be greater than this
0693: 90 02               BCC   :DSCFetch    ; less than :Bottom is okay
0695: D0 10               BNE   :DSCBye      ; greater than :Bottom isn't
                                             ; equal to :Bottom is

                 :DSCFetch
                 * now fetch the charater from RAM 1
0697: A9 FC               LDA   #OurPtr2     ; point to zero-page pointer
0699: A2 01               LDX   #$01         ; indicate Bank 1
069B: 20 74 FF            JSR   IndFet       ; grab a byte of entry string
                                             ; byte comes back in A- register

                 * draw that character on the bit-map screen
069E: AE FF 06            LDX   :AbsRow
06A1: AC 00 07            LDY   :AbsCol
06A4: 20 28 07            JSR   DrawBMChar

                 :DSCBye
                 * return from :DrwStrChr
06A7: 60                  RTS


                 *-------- :CursRit --------*

                 * move the cursor to the right, dealing with wraparound

                 * upon exit, X- register is trashed

                 :CursRit

                 * check current cursor horizontal position to see if
                 * ... we'll need to deal with wraparound
06A8: AE FA 06            LDX   :EdCrsHz     ; get current cursor horizontal
06AB: EC F6 06            CPX   :Right       ; do we need to wraparound to
                                             ; ... the editing rectangle's
                                             ; ... leftmost column ?
06AE: D0 07               BNE   :RtUpHz      ; no, so life is easy

```

```
                * we need to wrap around horizontally
                * that means we also have to move down the screen,
                * ... which is done by upping the vertical coordinate
06B0: 20 D0 06          JSR   :CursDwn

                * now let's horizontally wrap around to the
                * ... leftmost column
06B3: AE F5 06          LDX   :Left
06B6: CA                DEX              ; so we can slide thru
                                         ; ... the next instruction

                * move to the right by upping the horizontal coordinate
06B7: E8        :RtUpHz  INX
06B8: 8E FA 06          STX   :EdCrsHz   ; store new cursor horizontal

                * return from :CursRit
06BB: 60                RTS


                *-------- :CursLft --------*

                * move the cursor to the left, dealing with wraparound

                * upon exit, X- register is trashed

                :CursLft

                * check current cursor horizontal position to see if
                * ... we'll need to deal with wraparound
06BC: AE FA 06          LDX   :EdCrsHz   ; get current cursor horizontal
06BF: EC F5 06          CPX   :Left      ; do we need to wraparound to
                                         ; ... the editing rectangle's
                                         ; ... rightmost column ?
06C2: D0 07             BNE   :LftDnHz   ; no, so life is easy

                * we need to wrap around horizontally
                * that means we also have to move up the screen,
                * ... which is done by downing the vertical coordinate
06C4: 20 E1 06          JSR   :CursUp

                * now let's horizontally wrap around to the
                * ... rightmost column
06C7: AE F6 06          LDX   :Right
06CA: E8                INX              ; so we can slide thru
                                         ; ... the next instruction

                * move to the left by downing the horizontal coordinate
06CB: CA        :LftDnHz DEX
06CC: 8E FA 06          STX   :EdCrsHz   ; store new cursor horizontal

                * return from :CursLft
06CF: 60                RTS


                *-------- :CursDwn --------*

                * move the cursor down a line, dealing with wraparound

                * upon exit, X- register is trashed

                :CursDwn
                * check current cursor vertical to see if we'll have to
                * ... deal with wraparound
06D0: AE FB 06          LDX   :EdCrsVt   ; get current cursor vertical
06D3: EC F4 06          CPX   :Bottom    ; at bottom of rectangle ?
```

```
06D6: D0 04             BNE   :CDUpIt     ; no, so no need to wrap

                 * we need to vertically wrap up to the topmost row
06D8: AE F3 06          LDX   :Top
06DB: CA                DEX               ; so we can slide thru
                                          ; ... the next instruction

                 :CDUpIt
                 * move down the screen by upping the vertical coordinate
06DC: E8                INX               ; up, down, it's all so
                                          ; ... confusing, eh ?
06DD: 8E FB 06          STX   :EdCrsVt    ; store new cursor vertical

                 * return from :CursDwn
06E0: 60                RTS


                 *-------- :CursUp --------*

                 * move the cursor up a line, dealing with wraparound

                 * upon exit, X- register is trashed

                 :CursUp

                 * check current cursor vertical to see if we'll have to
                 * ... deal with wraparound
06E1: AE FB 06          LDX   :EdCrsVt    ; get current cursor vertical
06E4: EC F3 06          CPX   :Top        ; at top of rectangle ?
06E7: D0 04             BNE   :CUUpIt     ; no, so no need to wrap

                 * we need to vertically wrap to the bottommost row
06E9: AE F4 06          LDX   :Bottom
06EC: E8                INX               ; so we can slide thru
                                          ; ... the next instruction

                 :CUUpIt
                 * move up the screen by downing the vertical coordinate
06ED: CA                DEX               ; up, down, it's all so
                                          ; ... confusing, eh ?
06EE: 8E FB 06          STX   :EdCrsVt    ; store new cursor vertical

                 * return from :CursUp
06F1: 60                RTS


                 *-------- local variables --------*

06F2: 00         :FuncFlag DS  1          ; saved Y- register function
                                          ; ... selector
06F3: 00         :Top      DS  1          ; topmost row of edit rectangle
06F4: 00         :Bottom   DS  1          ; bottommost row of edit rect.
06F5: 00         :Left     DS  1          ; leftmost row of edit rect.
06F6: 00         :Right    DS  1          ; rightmost row of edit rect.
06F7: 00         :Width    DS  1          ; width of edit rectangle
06F8: 00         :Temp1    DS  1          ; general-purpose temporary
06F9: 00         :Temp2    DS  1          ; a cosmic trash bucket
06FA: 00         :EdCrsHz  DS  1          ; editing cursor horizontal
                                          ; ... position
06FB: 00         :EdCrsVt  DS  1          ; editing cursor vertical
                                          ; ... position
06FC: 00         :InCrsPs  DS  1          ; editing cursor initial position
06FD: 00         :Length   DS  1          ; length of the string we're
                                          ; ... editing
06FE: 00         :HotSpot  DS  1          ; character position in the exit
```

```
                                              ; ... string that the cursor is
                                              ; ... currently at
                                              ; ... [0..:Length-1]
06FF: 00         :AbsRow   DS    1            ; an absolute screen row 0..24
0700: 00         :AbsCol   DS    1            ; an absolute screen column
                                              ; ...  0..39
0701: 00         :AreaID   DS    1            ; area identification number for
                                              ; ... the editing rectangle
                                              ; ... ( 0 if not applicable )
0702: 00         :ArDtLo   DS    1            ; lo-byte of address of area
                                              ; ... data table
0703: 00         :ArDtHi   DS    1            ; hi-byte of address of area
                                              ; ... data table


                 *-------- local tables --------*

                 * this table tells the parameter block unpacking routine
                 * ... where to store the various pieces of entry info

0704: FB 00      :WherTab DA    OurPtr1+1     ; where we store pointer to
0706: FA 00               DA    OurPtr1       ; ... entry string record
0708: FD 00               DA    OurPtr2+1     ; where we store pointer to
070A: FC 00               DA    OurPtr2       ; ... exit string record
070C: F8 06               DA    :Temp1        ; throw away a filler byte
070E: F3 06               DA    :Top          ; where we store topmost
                                              ; ... row of editing rect.
0710: F8 06               DA    :Temp1        ; throw away a filler byte
0712: F4 06               DA    :Bottom       ; where we store bottommost
                                              ; ... row of editing rect.
0714: F8 06               DA    :Temp1        ; throw away a filler byte
0716: F5 06               DA    :Left         ; where we store leftmost
                                              ; ... row of editing rect.
0718: F8 06               DA    :Temp1        ; throw away a filler byte
071A: F6 06               DA    :Right        ; where we store rightmost
                                              ; ... row of editing rect.
071C: F8 06               DA    :Temp1        ; throw away a filler byte
071E: 01 07               DA    :AreaID       ; where we store the rectangle's
                                              ; ... area ID
0720: 03 07               DA    :ArDtHi       ; where we store pointer to
0722: 02 07               DA    :ArDtLo       ; ... area data table
0724: F8 06               DA    :Temp1        ; throw away a filler byte
0726: FC 06               DA    :InCrsPs      ; where we store editing cursor
                                              ; ... initial position



                 *-------------------- DrawBMChar -----------------------*

                 * draws a character on the standard hi-res bit-map screen

                 * upon entry, A- reg. holds the character's C-ASCII code
                 *             X- reg. holds the row [0..24]
                 *             Y- reg. holds the column [0..39]

                 * all registers are preserved

                 DrawBMChar
                 * save some registers
0728: 8D AC 07            STA   :TheCode
072B: 8E AD 07            STX   :TheRow
072E: 8C AE 07            STY   :TheColumn

                 * transform C-ASCII code into a set 1 poke code
0731: 20 B2 07            JSR   CAsc2Pok1

```

```
                * initialize character ROM offset hi-byte to 0
0734: A2 00             LDX   #0
0736: 8E AF 07          STX   :ROMOffHi
                * multiply poke code by 8 to get character ROM offset
0739: A2 03             LDX   #3          ; we'll do three left shifts
073B: 0A        :LoopOne ASL              ; each shift cycle multiplies
073C: 2E AF 07          ROL   :ROMOffHi   ; ... by 2
073F: CA                DEX               ; another 2-mult. done
0740: D0 F9             BNE   :LoopOne    ; go until finished
                * by the way, A- register now holds character ROM
                * ... offset lo-byte
                * now, add that 16-bit offset to the ROM's base address
                * ... and store as a pointer
0742: 18                CLC               ; straight-forward 16-bit
0743: 69 00             ADC   #<CharRom   ; ... addition
0745: 85 FA             STA   OurPtr1
0747: AD AF 07          LDA   :ROMOffHi
074A: 69 D0             ADC   #>CharRom
074C: 85 FB             STA   OurPtr1+1
                * we need to figure out where in bit-map memory this
                * ... character's eight data bytes will go
                * initialize hi-byte of row-caused offset to 0
074E: A9 00             LDA   #0
0750: 8D B0 07          STA   :RowOffHi
                * grab row & multiply by the # of bytes in a row : 320
                * that's equivalent to multiplying by 64 and 256, then
                * ... adding the results
                * so, we'll start by multiplying by 64 via 6 left shifts
0753: AD AD 07          LDA   :TheRow     ; grab the row
0756: A2 06             LDX   #6          ; we'll do six left shifts
0758: 0A        :LoopTwo ASL              ; each shift cycle multiplies
0759: 2E B0 07          ROL   :RowOffHi   ; ... by 2
075C: CA                DEX               ; another 2-mult. done
075D: D0 F9             BNE   :LoopTwo    ; go until finished
                * lo-byte of :TheRow * 64 is in the A- register
                * now, some heavy adding
                * start by adding that 64 * :TheRow to the bit-map base
075F: 18                CLC               ; standard addition
0760: 69 00             ADC   #<BMBase
0762: 85 CA             STA   OurPtr4
0764: AD B0 07          LDA   :RowOffHi
0767: 69 20             ADC   #>BMBase    ; don't store that hi-byte yet
                * now, add 256 * :TheRow
                * ( cheap trick : just pretend it's a hi-byte )
0769: 18                CLC
076A: 6D AD 07          ADC   :TheRow
076D: 85 CB             STA   OurPtr4+1   ; okay, store hi-byte
                * now, multiply the column by 8
                * initialize hi-byte of column-caused offset to 0
076F: A9 00             LDA   #0
0771: 8D B1 07          STA   :ColOffHi
                * another shifty little multiplication
0774: AD AE 07          LDA   :TheColumn  ; grab the column
0777: A2 03             LDX   #3          ; we'll do three left shifts
0779: 0A        :Loop3   ASL              ; each shift cycle multiplies
```

```
077A: 2E B1 07        ROL   :ColOffHi  ; ... by 2
077D: CA              DEX              ; another 2-mult. done
077E: D0 F9           BNE   :Loop3     ; go until finished
                * and, finally, add that in
0780: 18              CLC
0781: 65 CA           ADC   OurPtr4
0783: 85 CA           STA   OurPtr4
0785: AD B1 07        LDA   :ColOffHi
0788: 65 CB           ADC   OurPtr4+1
078A: 85 CB           STA   OurPtr4+1
                * set memory configuration so we can talk to the
                * ... character ROM and the standard bit-map screen
078C: AD 00 FF        LDA   MmuCR      ; save current configuration
078F: 48              PHA              ; ... on the stack
0790: A9 01           LDA   #Bank14    ; that'll do the trick
0792: 8D 00 FF        STA   MmuCR      ; configured !
                * transfer those eight bytes
0795: A0 07           LDY   #7           ; our counter/index
0797: B1 FA     :Loop4 LDA   (OurPtr1),Y; grab character ROM byte
0799: 91 CA           STA   (OurPtr4),Y; store in bit-map
079B: 88              DEY              ; down the counter/index
079C: 10 F9           BPL   :Loop4      ; branch until finished
                * restore memory configuration
079E: 68              PLA              ; we parked it there
079F: 8D 00 FF        STA   MmuCR
                * restore some registers and leave
07A2: AC AE 07        LDY   :TheColumn
07A5: AE AD 07        LDX   :TheRow
07A8: AD AC 07        LDA   :TheCode
07AB: 60              RTS              ; return from DrawBMChar
                *-------- local variables --------*
07AC: 00        :TheCode DS    1          ; the character's C-ASCII code
07AD: 00        :TheRow  DS    1          ; the row it'll go in
07AE: 00        :TheColumn DS  1          ; the column it'll go in
07AF: 00        :ROMOffHi DS   1          ; hi-byte of char. ROM offset
07B0: 00        :RowOffHi DS   1          ; hi-byte of row-caused offset
07B1: 00        :ColOffHi DS   1          ; hi-byte of column-caused offset
                *----------------------- CAsc2Pok1 ---------------------*
                * transform Commodore ASCII code to Set 1 screen poke code
                * obviously, this would be faster with a 256-byte table,
                * ... but we're a bit squeezed for space -- this code only
                * ... eats up 50 bytes, and ain't all THAT slow
                * upon entry, A- reg. holds a C-ASCII code [0..255]
                * upon exit,  A- reg. holds a poke code [0..255]
                * X- and Y- registers are preserved
                CAsc2Pok1
                * C-ASCIIs 0..31 transform to pocode 32
```

```
07B2: C9 20    :Test1   CMP   #32        ; test for 0..31
07B4: B0 03             BCS   :Test2     ; not in range, do next test
07B6: A9 20             LDA   #32        ; in range, so return 32
07B8: 60                RTS              ; outta here
               * C-ASCIIs 32..63 transform to pocodes 32..63
07B9: C9 40    :Test2   CMP   #64        ; test for 32..63
07BB: B0 01             BCS   :Test3     ; not in range, do next test
07BD: 60                RTS              ; in range, so return as is
               * C-ASCIIs 64..95 transform to pocodes 0..31
07BE: C9 60    :Test3   CMP   #96        ; test for 64..95
07C0: B0 03             BCS   :Test4     ; not in range, do next test
07C2: E9 3F             SBC   #63        ; in range, transform 64..95
                                         ; ... to 0..31 by subtacting 64
                                         ; ... ( a clear Carry lets us
                                         ; ...   skip a SEC step if we
                                         ; ...   just subtract 63 )
07C4: 60                RTS              ; bye bye
               * C-ASCIIs 96..127 transform to pocodes 64..95
07C5: C9 80    :Test4   CMP   #128       ; test for 96..127
07C7: B0 03             BCS   :Test5     ; not in range, do next test
07C9: E9 1F             SBC   #31        ; in range, transform 96..127 to
                                         ; ... 64..95 by subtacting 32
                                         ; ... ( a clear Carry lets us
                                         ; ...   skip a SEC step if we
                                         ; ...   just subtract 31 )
07CB: 60                RTS              ; bye bye
               * C-ASCIIs 128..159 transform to pocode 32
07CC: C9 A0    :Test5   CMP   #160       ; test for 128..159
07CE: B0 03             BCS   :Test6     ; not in range, do next test
07D0: A9 20             LDA   #32        ; in range, so return 32
07D2: 60                RTS              ; bye bye
               * C-ASCIIs 160..191 transform to pocodes 96..127
07D3: C9 C0    :Test6   CMP   #192       ; test for 160..191
07D5: B0 03             BCS   :Test7     ; not in range, do next test
07D7: E9 3F             SBC   #63        ; in range, transform 160..191
                                         ; ... to 96..127 by subtacting 64
                                         ; ... ( a clear Carry lets us
                                         ; ...   skip a SEC step if we
                                         ; ...   just subtract 63 )
07D9: 60                RTS              ; bye bye
               * C-ASCIIs 192..223 transform to pocodes 64..95
               * C-ASCIIs 224..254 transform to pocodes 96..126
               * ( notice that both ranges transform down by 128 )
07DA: C9 FF    :Test7   CMP   #255       ; test for only remaining value
                                         ; ... that's not in one of these
                                         ; ... ranges
07DC: F0 03             BEQ   :Final     ; got it, so go transform it
07DE: E9 7F             SBC   #127       ; in range, transform 192..223
                                         ; ... to 64..95 and 224..255 to
                                         ; ... 96..126 by subtracting 128
                                         ; ... ( a clear Carry lets us
                                         ; ...   skip a SEC step if we
```

```
                                                ; ...   just subtract 127 )
07E0: 60                  RTS                  ; git gone

                 * C-ASCII 255 transforms to pocode 94
07E1: A9 5E      :Final   LDA    #94           ; transform 255 to 94
07E3: 60                  RTS                  ; that's all she wrote


--End assembly, 996 bytes, Errors: 0
```

```

*--------------- program identification ----------------*
*                                                       *
*                                                       *
*                     S/M ASM 2                         *
*                                                       *
*                                                       *
* Assembly language tools for the BASIC 7.0 program     *
* ... SOUND/MUSIC LAB.                                  *
*                                                       *
* Sits in Bank 0 RAM memory at $1300 - $1B17            *
*                                                       *
* Divided into three source files :                     *
*                     S/M ASM 2 A.S                     *
*                     S/M ASM 2 B.S                     *
*                     S/M ASM 2 C.S                     *
*                                                       *
*                                                       *
* To install the helper :                               *
*                                                       *
*                     BLOAD "S/M ASM 2"                 *
*                     SYS 4864                          *
*                                                       *
*                                                       *
* To un-install the helper :                            *
*                                                       *
*                     SYS 4933                          *
*                                                       *
*                                                       *
* The pseudo-mouse button state record begins at        *
* ... memory location 6757                              *
*                                                       *
*                                                       *
* To call the AreaSearch routine for the lab screen :   *
*                                                       *
*                     SYS 5224, 47, 22                  *
*                                                       *
*                                                       *
* To call the AreaSearch routine for the help screen :  *
*                                                       *
*                     SYS 5224, 15, 25                  *
*                                                       *
*                                                       *
* To call the HRRectInvt routine for the lab screen :   *
*                                                       *
*                     SYS 5318, 51, 25, areaID#         *
*                                                       *
*    where     areaID# is [1..105]                      *
*                                                       *
*                                                       *
* To call the HRBandInvt routine :                      *
*                                                       *
*                     SYS 5394, width, row, startColumn *
```

Fig. 16-2. Source code for S/M Asm 2.

```
*                                                          *
*    where       width is [1..39]                          *
*                  row is [0..24]                          *
*          startColumn is [0..39]                          *
*                                                          *
*
* To call the TXBandInvt routine :                         *
*                                                          *
*                     SYS 5459, width, row, startColumn    *
*                                                          *
*    where       width is [1..39]                          *
*                  row is [0..24]                          *
*          startColumn is [0..39]                          *
*                                                          *
*                                                          *
* Version :      1:00                                      *
* Timestamp :    5:15 PM PST     September 16, 1986        *
*                                                          *
* Programmed by Stan Krute                                 *
* Copyright (C) 1986 by Stan Krute's Hacker & Nerd         *
*                      18617 Camp Creek Road               *
*                      Hornbrook, California   96044       *
*                      [916] 475-3428                      *
* All rights reserved                                      *
* Call or write for bug reports, help, licensing, etc.     *
*                                                          *
*----------------------------------------------------------*


*---------------------- Constants ----------------------*

* CIAs

D1PrA     =     $DC00       ; CIA # 1 port A
D1PrB     =     $DC01       ; CIA # 1 port B
D2PrA     =     $DD00       ; CIA # 2 port A
D2T1L     =     $DD04       ; CIA # 2 timer A lo-byte
D2T1H     =     $DD05       ; CIA # 2 timer A hi-byte
D2ICR     =     $DD0D       ; CIA # 2 interrupt control reg.
D2CRA     =     $DD0E       ; CIA # 2 control register A


* Commodore ASCII codes

RtrnCAC   =     13          ; code for a carriage return


* joystick

JoyDirMsk =     %00001111   ; bit setup to mask in joystick
                            ; ... direction switches
JoyBtnMsk =     %00010000   ; bit setup to mask in joystick
                            ; ... button switch


* keyboard

KeyD      =     $034A       ; the keyboard buffer
Ndx       =     $D0         ; index to the keyboard buffer
K2Chek    =     %11111011   ; bit setup to test K2 line
UpCrsMsk  =     %01111000   ; bit setup to mask in upper
                            ; ... cursor keys
RtrnMsk   =     %00000010   ; bit setup to mask in return
                            ; ... key
```

```
* keycodes

UpKyCd    =     83          ; the upper cursor-up key
DwnKyCd   =     84          ; the upper cursor-down key
LftKyCd   =     85          ; the upper cursor-left key
RitKyCd   =     86          ; the upper cursor-right key
RtrnKyCd  =     01          ; the return key


* low-memory system variables

StaVec    =     $2B9        ; points to a zero-page pointer
                            ; ... for IndSta ROM Kernel call


* memory management

MmuCR     =     $FF00       ; configuration register
MmuRCR    =     $D506       ; ram configuration register
Bank15    =     %00000000   ; configuration byte


* pseudo-mouse

NoClik    =     0           ; code for no click
Clik      =     1           ; code for a click


* ROM routines - documented

IndFet    =     $FF74       ; fetch data from any bank
IndSta    =     $FF77       ; store data to any bank


* screen area stuff

AreaRcSz  =     7           ; size of an area record


* sprite stuff

NoDir     =     255         ; code for no directional info
Sp1Speed  =     $117E       ; sprite #1 speed parameter
Sp1MvRec  =     $117E       ; start of sprite #1 motion
                            ; ... record
MvRecSiz  =     7           ; size of sprite motion record
                            ; ... data that we set
NoMotion  =     0           ; MoveFlag code for no motion
YesMove   =     %10000000   ; MoveFlag mask for keyboard-
                            ; ... powered motion
NoMove    =     %01111111   ; MoveFlag mask for no keyboard-
                            ; ... powered motion


* vectors

IIRQ      =     $0314       ; IRQ interrupt routine
KeyChk    =     $033C       ; store a keypress


* VIC stuff

VICSave   =     $11D6       ; location of VIC shadow regs.
Sp1Vrt    =     VICSave+1
Sp1HrzLo  =     VICSave
```

```
                SprHrzHi =      VICSave+16
                Sp1HHMsk =      %00000001
                VicReg24 =      $D018
                VicReg47 =      $D02F


                * zero-page variables

                OurPtr   =      $FA           ; a pointer we use


                *---------------------- Macros -------------------------*

                * A nice pseudo-unconditional branch

                BRA      MAC
                         CLV
                         BVC    ]1
                         <<<


                *----------------- Set Program Origin ------------------*

                         ORG    $1300         ; for maximum area beneath
                                              ; ... the bit map


                *---------------------- Install ------------------------*

                * Installs the aids

                Install
                * disable interrupts
1300: 78                 SEI

                * save some registers
1301: 48                 PHA

                :Task1
                * install our KeyChk detour
                * save the current vector
1302: AD 3C 03           LDA    KeyChk
1305: 8D 0F 1B           STA    RegKeyChk     ; we'll need it for jumping
1308: AD 3D 03           LDA    KeyChk+1      ; ... and restoration
130B: 8D 10 1B           STA    RegKeyChk+1

                * vectorize to our routine
130E: A9 62              LDA    #<OurKeyChk; set the vector to its address
1310: 8D 3C 03           STA    KeyChk
1313: A9 13              LDA    #>OurKeyChk
1315: 8D 3D 03           STA    KeyChk+1

                :Task2
                * install our IIRQ detour
                * save the current vector
1318: AD 14 03           LDA    IIRQ
131B: 8D 11 1B           STA    RegIIRQ       ; we'll need it for jumping
131E: AD 15 03           LDA    IIRQ+1        ; ... and restoration
1321: 8D 12 1B           STA    RegIIRQ+1

                * vectorize to our routine
1324: A9 82              LDA    #<OurIIRQ     ; set the vector to its address
1326: 8D 14 03           STA    IIRQ
1329: A9 13              LDA    #>OurIIRQ
132B: 8D 15 03           STA    IIRQ+1
```

```
                    :Task3
                    * initialize the sprite-in-motion flag
132E: A9 00                 LDA   #NoMotion
1330: 8D 13 1B              STA   MoveFlag

                    :Task4
                    * initialize the pseudo-mouse single/double click timer
                    * turn off CIA #2 Timer A interrupt ability
1333: A9 01                 LDA   #%00000001
1335: 8D 0D DD              STA   D2ICR
                    * set the latched value
1338: A9 00                 LDA   #%0000000
133A: 8D 04 DD              STA   D2T1L
133D: A9 FF                 LDA   #%11111111
133F: 8D 05 DD              STA   D2T1H

                    :Bye
                    * restore some registers
1342: 68                    PLA

                    * enable interrupts
1343: 58                    CLI

                    * and leave
1344: 60                    RTS               ; return from Install


                    *--------------------- UnInstall -----------------------*

                    * Un-installs the aids

                    UnInstall
                    * disable interrupts
1345: 78                    SEI

                    * save some registers
1346: 48                    PHA

                    :Task1
                    * remove our IIRQ detour
                    * just vectorize to the original routine
1347: AD 11 1B              LDA   RegIIRQ     ; set the vector to its address
134A: 8D 14 03              STA   IIRQ
134D: AD 12 1B              LDA   RegIIRQ+1
1350: 8D 15 03              STA   IIRQ+1

                    :Task2
                    * remove our KeyChk detour
                    * just vectorize to the original routine
1353: AD 0F 1B              LDA   RegKeyChk   ; set the vector to its address
1356: 8D 3C 03              STA   KeyChk
1359: AD 10 1B              LDA   RegKeyChk+1
135C: 8D 3D 03              STA   KeyChk+1

                    :Bye
                    * restore some registers
135F: 68                    PLA

                    * enable interrupts
1360: 58                    CLI

                    * and leave
1361: 60                    RTS               ; return from Install

```

```

               *--------------------- OurKeyChk -----------------------*

               * Looks for keycodes that represent the upper set of
               * ... cursor keys, and keeps them from being put in buffer

               * Upon entry, Y- register holds the keycode of currently
               * ... pressed key

               OurKeyChk
               * save some registers
1362: 08                PHP
1363: 48                PHA

               * A- register will hold keycode in loop
1364: 98                TYA               ; initialize with keycode

               * run the screening tests
1365: A0 04             LDY   #:End-:TstCodz-1 ; it'll count the loop &
                                          ; ... index into the list
                                          ; ... of test keycodes
1367: D9 7D 13 :Test    CMP   :TstCodz,Y        ; is keycode one of our
                                          ; ... test codes ?
136A: F0 09             BEQ   :Bye2             ; yup, so bag it
136C: 88       :Next    DEY               ; nope, so down the
                                          ; ... loop counter
136D: 10 F8             BPL   :Test             ; if more to do, loop up

               :Bye
               * restore some registers
136F: A8                TAY
1370: 68                PLA
1371: 28                PLP

               * slide into the regular KeyChk routine
1372: 6C 0F 1B          JMP   (RegKeyChk)

               :Bye2
               * restore some registers
1375: A8                TAY
1376: 68                PLA
1377: A9 00             LDA   #0          ; this'll hide the keypress
1379: 28                PLP

               * slide into the regular KeyChk routine
137A: 6C 0F 1B          JMP   (RegKeyChk)


               * local data : keycodes to test for
137D: 53       :TstCodz DFB   UpKyCd
137E: 54                DFB   DwnKyCd
137F: 55                DFB   LftKyCd
1380: 56                DFB   RitKyCd
1381: 01                DFB   RtrnKyCd
               :End


               *---------------------- OurIIRQ ------------------------*

               * Our detour to the system heartbeat

               * Looks for pseudo-mouse commands from the upper cursor
               * ... keys, return key, and joystick

               * Controls the motion of sprite #1, the pseudo-mouse
               * ... cursor, based on what it finds
```

```

                * See the text for somewhat intense pseudo-code logic

                OurIIRQ
                * Save some registers
1382: 48                PHA
1383: 8A                TXA
1384: 48                PHA

                * save entry memory configuration
1385: AD 00 FF          LDA   MmuCR       ; go grab it
1388: 48                PHA               ; park it on the stack

                * set for Bank 15 memory configuration
1389: A9 00             LDA   #Bank15     ; the magic number
138B: 8D 00 FF          STA   MmuCR       ; do it to it

                * let's check out the upper cursor keys
138E: A9 FF             LDA   #%11111111 ; not interested in C0-C7 lines
1390: 8D 00 DC          STA   D1PrA       ; this bags them
1393: A9 FB             LDA   #K2Chek     ; send signal on K2 line
1395: 8D 2F D0          STA   VicReg47    ; nice kludge, Commodore
1398: AD 01 DC          LDA   D1PrB       ; read the result
139B: 29 78             AND   #UpCrsMsk   ; mask out noise
139D: 4A                LSR               ; move bits into lo nibble
139E: 4A                LSR
139F: 4A                LSR
13A0: AA                TAX               ; for table indexing
13A1: BD 33 14          LDA   :DirTab,X   ; grab a cursor key code
13A4: 8D 53 14          STA   :CrsDir     ; and store it

                * let's check out the joystick direction switches
                * a clear bit indicates a switch that's been pressed
13A7: AD 00 DC          LDA   D1PrA       ; grab joystick data
13AA: 29 0F             AND   #JoyDirMsk ; mask out noise
13AC: AA                TAX               ; for table indexing
13AD: BD 33 14          LDA   :DirTab,X   ; grab a joystick direction
                                          ; ... code
13B0: 8D 54 14          STA   :JoyDir     ; and store it

                * now, let's arbitrate the directional information
                * if cursor keys AND joystick are giving a valid
                * ... direction, we'll let the joystick win
13B3: C9 FF             CMP   #NoDir      ; if joystick has something,
13B5: D0 03             BNE   :StorArb    ; ... use it
13B7: AD 53 14          LDA   :CrsDir     ; ... else use cursor keys
13BA: 8D 55 14  :StorArb STA  :ArbDir

                * are we currently moving sprite #1 ?
13BD: 2C 13 1B          BIT   MoveFlag    ; currently moving ?
                                          ; (flagged by bit 7)
13C0: 10 12             BPL   :NotMovin   ; no, so jump on

                :Movin
                * sprite #1 is currently in motion
                * do we have a directional command to continue THAT
                * ... motion ?  if so, we can jump ahead, and
                * ... if not, we'll stop the sprite
13C2: CD 56 14          CMP   :MovDir     ; do they match ?
13C5: F0 2D             BEQ   :ChekButn   ; yes, so we can jump ahead

                :Stop
                * we get here if we need to stop key-controlled
                * ... sprite #1 motion
13C7: A9 00             LDA   #NoMotion   ; stop the motion
```

```
13C9: 8D 7E 11          STA   Sp1Speed
13CC: AD 13 1B          LDA   MoveFlag    ; fix the flag
13CF: 29 7F             AND   #NoMove     ; clears bit 7
13D1: 8D 13 1B          STA   MoveFlag

                :NotMovin
                * we get here if the sprite isn't moving
13D4: 2C 55 14          BIT   :ArbDir     ; was a direction specified
13D7: 30 1B             BMI   :ChekButn   ; no, so we can jump ahead

                * a directed motion WAS specified
                * use the code to grab motion data
13D9: AE 55 14          LDX   :ArbDir     ; grab that code
13DC: BD 43 14          LDA   :MDLo,X     ; index into address tables
13DF: 85 FA             STA   OurPtr
13E1: BD 4B 14          LDA   :MDHi,X
13E4: 85 FB             STA   OurPtr+1

                * go set sprite #1 into motion
13E6: 20 57 14          JSR   SetMoshn

                * set MoveFlag and :MovDir appropriately
13E9: AD 13 1B          LDA   MoveFlag
13EC: 09 80             ORA   #YesMove
13EE: 8D 13 1B          STA   MoveFlag
13F1: 8E 56 14          STX   :MovDir

                :ChekButn
                * see if the joystick button is being pressed
                * a clear bit indicates a switch that's been pressed
13F4: AD 00 DC          LDA   D1PrA       ; grab joystick data
13F7: 29 10             AND   #JoyBtnMsk  ; isolate the button bit
13F9: F0 11             BEQ   :GotClik    ; pressed, so jump on

                :ChekRtrn
                * see if the return key is being pressed
                * we check the hardware ourselves
13FB: A9 FE             LDA   #%11111110  ; we're interested in C0 line
13FD: 8D 00 DC          STA   D1PrA       ; this sends test out on it
1400: A9 FF             LDA   #%11111111  ; not interested in K0-K2
1402: 8D 2F D0          STA   VicReg47    ; this takes care of that
1405: AD 01 DC          LDA   D1PrB       ; read the result
1408: 29 02             AND   #RtrnMsk    ; clear bit means pressed
140A: D0 18             BNE   :NoClik     ; not pressed, so jump on

                :GotClik
                * we get here if joystick or return key pressed
                * store current sprite # 1 position
140C: AD D6 11          LDA   Sp1HrzLo
140F: 8D 15 1B          STA   ClikHzLo
1412: AD E6 11          LDA   SprHrzHi
1415: 29 01             AND   #Sp1HHMsk
1417: 8D 16 1B          STA   ClikHzHi
141A: AD D7 11          LDA   Sp1Vrt
141D: 8D 17 1B          STA   ClikVt

                * grab click code and jump
1420: A9 01             LDA   #Clik       ; it's a non-zero value
1422: D0 02             BNE   :StorClik

                :NoClik
                * we get here if neither joystick nor return key pressed
1424: A9 00             LDA   #NoClik

                :StorClik
```

```
                * store button state of click or no-click
1426: 8D 14 1B          STA   ButnStat

                :Bye
                * restore memory configuration
1429: 68                PLA
142A: 8D 00 FF          STA   MmuCR

                * restore some registers and leave
142D: 68                PLA
142E: AA                TAX
142F: 68                PLA

1430: 6C 11 1B          JMP   (RegIIRQ)


                * local constant data

                * table for translating raw joystick and upper cursor
                * ... key data into a direction code
1433: FF 04 00  :DirTab DFB   255,4,0
1436: FF 02 03          DFB   255,2,3
1439: 01 02 06          DFB   1,2,6
143C: 05 07 06          DFB   5,7,6
143F: FF 04 00          DFB   255,4,0
1442: FF                DFB   255

                * tables for getting the address of a direction's
                * ... sprite motion data (lo and hi bytes)
1443: D7        :MDLo   DFB   #<North
1444: DE                DFB   #<NorthEast
1445: E5                DFB   #<East
1446: EC                DFB   #<SouthEast
1447: F3                DFB   #<South
1448: FA                DFb   #<SouthWest
1449: 01                DFB   #<West
144A: 08                DFB   #<NorthWest

144B: 1A        :MDHi   DFB   #>North
144C: 1A                DFB   #>NorthEast
144D: 1A                DFB   #>East
144E: 1A                DFB   #>SouthEast
144F: 1A                DFB   #>South
1450: 1A                DFb   #>SouthWest
1451: 1B                DFB   #>West
1452: 1B                DFB   #>NorthWest

                * local variables

1453: 00        :CrsDir DS    1         ; direction code from read of
                                        ; ... upper cursor keys
1454: 00        :JoyDir DS    1         ; direction code from read of
                                        ; .... joystick switches
1455: 00        :ArbDir DS    1         ; direction code arbitrated
                                        ; ... from upper cursor and
                                        ; ... joystick direction codes
1456: 00        :MovDir DS    1         ; direction code for current
                                        ; ... sprite motion


                        TTL   "S/M ASM 2 B.S"
                *---------- Here Comes The Second Source File -----------*

                        PUT   "S/M ASM 2 B.S"
>1
```

```
                *-------------------- SetMoshn -------------------------*

                * Set sprite #1 into some kind of motion

                * on entry, OurPtr points to a motion data record

                SetMoshn

                * save some registers
1457: 48                 PHA
1458: 98                 TYA
1459: 48                 PHA

                * just store the motion data record
145A: A0 06              LDY   #MvRecSiz-1; count those bytes
145C: B1 FA     :Loop    LDA   (OurPtr),Y  ; grab a byte of record
145E: 99 7E 11           STA   Sp1MvRec,Y  ; store that byte
1461: 88                 DEY               ; down the counter
1462: 10 F8              BPL   :Loop       ; continue 'til done

                * restore some registers
1464: 68                 PLA
1465: A8                 TAY
1466: 68                 PLA

                * return from SetMoshn
1467: 60                 RTS

                *------------------- AreaSearch ------------------------*

                * uses the current sprite pseudo-mouse position to
                * ... determine which rectangular area (if any) of a set of
                * ... such areas the hot-point of the pseudo-mouse is in
                * ... ( the areas must be non-overlapping )

                * upon entry, A- (lo) and X- (hi) point to the data
                * ... representing the set of areas to be searched,
                * ... and the coordinates of the clicked pseudo-mouse
                * ... are in ClikVt, ClikHzLo, and ClikHzHi

                * upon exit, a byte-length identifier for the area the p-m
                * ... coordinates are in is in the A- register, with a
                * ... value of 0 indicating that the pseudo-mouse was not
                * ... found to be in a defined area

                * due to the register usage, only the Y- register is
                * ... preserved by the routine

                * the area data is organized as an array of 7-byte records

                * each of these records is set up as follows :
                *              top boundary
                *              bottom boundary
                *              left boundary - hi byte
                *              left boundary - lo byte
                *              right boundary - hi byte
                *              right boundary - lo byte
                *              area identifier

                * the areas are arranged in top-bottom, left-right
                * ... order within a table to facilitate the search

                * that's also why the two bytes of each horizontal
                * ... boundary are in non-6502-conventional order

```

```
                * the area coordinates are in a coordinate system based
                * ... on the hot-point of the finger-cursor sprite, so
                * ... no sprite-coordinate adjustments are necessary when
                * ... using that sprite image

                * the arrangement of the data allows the following
                * ... pseudo-code search algorithm :

                * if
                *    mouseVert < areaTop
                * then
                *    notInAnArea

                * else-if
                *    mouseVert > areaBottom
                * then
                *    chekNextArea

                * else-if
                *    mouseHorz < areaLeft
                * then
                *    chekNextArea

                * else-if
                *    mouseHorz > areaRight
                * then
                *    chekNextArea

                * else
                *    weHaveFoundTheAreaJack

                AreaSearch

                * initialize table pointer
1468: 85 FA              STA   OurPtr      ; remember, A- and X- come
146A: 86 FB              STX   OurPtr+1    ; ... in pointing to table

                * save a register
146C: 98                 TYA
146D: 48                 PHA

                * set Y for indexing into a fresh area record
146E: A0 00              LDY   #0          ; move to top boundary byte

                :LupTop
                * the top of our search loop

                * see if we're at the end of the table of areas
                * if so, get on out
1470: B1 FA              LDA   (OurPtr),Y ; get area top
1472: F0 46              BEQ   :NoFind2    ; 0 marks end of areas table

                * we still have areas to check
                * grab pseudo-mouse vertical coordinate
1474: AD 17 1B           LDA   ClikVt

                * compare with area's top boundary
1477: D1 FA              CMP   (OurPtr),Y

                * if it's less than that, we're outta here
1479: 90 3D              BCC   :NoFind1

                * passed that test
                * now, compare mouse vertical with bottom boundary
147B: C8                 INY               ; move to area bottom byte
147C: D1 FA              CMP   (OurPtr),Y ; check it out
```

```
147E: 90 02           BCC   :GetHorz    ; less sez passed, so next test
1480: D0 27           BNE   :NextArea   ; greater sez failed, so try
                                        ; ... next area
                                        ; equal sez passed, so next test

             :GetHorz
             * grab pseudo-mouse horizontal coordinate bytes
1482: AD 16 1B        LDA   ClikHzHi
1485: AE 15 1B        LDX   ClikHzLo

             * compare with area's left boundary hi-byte
1488: C8              INY               ; move to area left hi-byte
1489: D1 FA           CMP   (OurPtr),Y  ; check it out
148B: 90 1C           BCC   :NextArea   ; less sez hi-byte failed, so
                                        ; ... try next area
148D: D0 0A           BNE   :ChekRite   ; greater sez hi-byte passed,
                                        ; ... so on to next test
                                        ; equal sez hi-byte inconclusive,
                                        ; ... so check lo-byte

             * we get here if we need to check left boundary lo-byte
             * save that pseudo-mouse horizontal hi-byte on the stack
148F: 48              PHA

             * grab pseudo-mouse horizontal coordinate lo-byte
1490: 8A              TXA

             * compare with area's left boundary lo-byte
1491: C8              INY               ; move to area left lo-byte
1492: D1 FA           CMP   (OurPtr),Y  ; check it out

             * no matter what the result, we can re-sync things
1494: AA              TAX               ; mouse lo back into X-
1495: 68              PLA               ; mouse hi back into A-

             * now, let's branch on that last test's results
1496: 90 11           BCC   :NextArea   ; less sez lo-byte failed,
                                        ; ... so try next area
                                        ; greater or equal sez
                                        ; lo-byte passed, so continue
                                        ; on

             * get index back in sync after that lo-byte testing
1498: 88              DEY               ; back to area left hi-byte

             :ChekRite
             * time to check the right boundary
             * start with the hi-byte
1499: C8              INY               ; move to area left hi-byte
149A: C8              INY               ; move to area right hi-byte
149B: D1 FA           CMP   (OurPtr),Y  ; check it out
149D: 90 1E           BCC   :GotCha1    ; less sez hi-byte passed, so
                                        ; ... we've found the area
149F: D0 08           BNE   :NextArea   ; greater sez hi-byte failed,
                                        ; ... so on to next area
                                        ; equal sez hi-byte inconclusive,
                                        ; ... so check lo-byte

             * grab pseudo-mouse horizontal coordinate lo-byte
14A1: 8A              TXA

             * compare with area's right boundary lo-byte
14A2: C8              INY               ; move to area right lo-byte
14A3: D1 FA           CMP   (OurPtr),Y  ; check it out
14A5: 90 17           BCC   :GotCha2    ; less sez lo-byte passed, so
```

```
                                        ; ... we've found the area
14A7: F0 15          BEQ   :GotCha2     ; equal sez lo-byte passed, so
                                        ; ... we've found the area
                                        ; greater sez lo-byte failed,
                                        ; ... so try next area

              :NextArea
              * we get here when it's time to move on to the next
              * ... area in the table of areas
              * move the pointer along
14A9: 18             CLC                ; just add the size of a record
14AA: A5 FA          LDA   OurPtr       ; ... to the table pointer
14AC: 69 07          ADC   #AreaRcSz
14AE: 85 FA          STA   OurPtr
14B0: 90 02          BCC   :SetWhy
14B2: E6 FB          INC   OurPtr+1

              * set Y to index into a new record and branch
14B4: A0 00   :SetWhy LDY   #0          ; start at record top
14B6: F0 B8          BEQ   :LupTop      ; branch always

              * we get to these next two points if the pseudo-mouse
              * ... point is NOT in one of the areas

              :NoFind1
              * we get here if A- isn't guaranteed to contain a 0
14B8: A9 00          LDA   #0           ; not-found area ID code is 0

              :NoFind2
              * we get here when the point's not in an area
              * ... and A- is guaranteed to contain a 0
14BA: AA             TAX                ; park area ID in X- register

              * and branch on out
14BB: F0 05          BEQ   :Bye         ; always

              * we get here when we've found the area the pseudo-mouse
              * ... point is in
14BD: C8      :GotCha1 INY              ; move to area right lo-byte
14BE: C8      :GotCha2 INY              ; move to area identifier

14BF: B1 FA          LDA   (OurPtr),Y ; grab area's identifier
14C1: AA             TAX                ; ... and store it in X-

              :Bye
              * restore a register, grab result code, and leave
14C2: 68             PLA                ; restore a register
14C3: A8             TAY

14C4: 8A             TXA                ; remember, we put result in X-

14C5: 60             RTS                ; return from AreaSearch


              *------------------------ HRRectInvt --------------------*

              * switches foreground and background colors in a
              * ... rectangular area of the hi-res bit mapped screen

              * remember, such changes are carried out in a screen grid
              * ... that's 40 columns wide and 25 rows high

              * in other words, each grid element controls an area that's
              * ... 8 pixels wide and 8 pixels high

```

```
                * upon entry, A- (lo) and X- (hi) point to a table of
                *               area rectangle data
                *            Y- contains the area's ID number, which
                *               ( when put into 0-based form and
                *                 multiplied by the table's record size )
                *               will serve as an offset into the array

                * routine does no error checking, so get those parameters
                * ... right

                * upon exit, A- X- and Y- are trashed

                HRRectInvt

                * set pointer to start of area table
 4C6: 85 FA              STA   OurPtr
 4C8: 86 FB              STX   OurPtr+1

                * use the ID number to get an array offset
                * make it 0-based
14CA: 88                 DEY

                * then multiply by 4 ( the size of each data record
                * ... in the table ), allowing 2 bytes for the result
14CB: A9 00              LDA   #0          ; init. offset hi-byte to 0
14CD: 8D 0F 15           STA   :OfsHi
14D0: 98                 TYA               ; move ID# into A-
14D1: 0A                 ASL               ; just multiply by 2 twice
14D2: 2E 0F 15           ROL   :OfsHi
14D5: 0A                 ASL               ; so A- holds offset lo
14D6: 2E 0F 15           ROL   :OfsHi      ; ... and this guy holds hi

                * now add that offset to the start of the table
14D9: 18                 CLC               ; prepare to add
14DA: 65 FA              ADC   OurPtr      ; add lo-bytes
14DC: 85 FA              STA   OurPtr      ; store result
14DE: AD 0F 15           LDA   :OfsHi      ; add hi-bytes
14E1: 65 FB              ADC   OurPtr+1
14E3: 85 FB              STA   OurPtr+1    ; store result

                * so our pointer's now looking at the identified
                * ... area's rectangular data record, which
                * ... contains ( in this order ) :
                *                       topmost row        (byte)
                *                    bottommost row        (byte)
                *                      leftmost column     (byte)
                *                     rightmost column     (byte)

                * let's grab some of that data for a loop that'll call
                * ... on HRBandInvt for each row of the rectangle

                * prepare Y- for indexing into area record
14E5: A0 00              LDY   #0

                * grab the topmost row for first call to HRBandInvt
14E7: B1 FA              LDA   (OurPtr),Y ; grab topmost row
14E9: AA                 TAX               ; store it

                * grab the bottommost row for loop-testing purposes
14EA: C8                 INY               ; on to bottommost row item
14EB: B1 FA              LDA   (OurPtr),Y ; grab bottommost row
14ED: 8D 10 15           STA   :BtRwPs1    ; store it
14F0: EE 10 15           INC   :BtRwPs1    ; up it by 1 for easy testing

                * figure out the width of each band
```

```
14F3: C8                    INY              ; on to leftmost column item
14F4: B1 FA                 LDA   (OurPtr),Y ; grab leftmost column
14F6: 8D 11 15              STA   :LeftCol   ; park it
14F9: C8                    INY              ; on to next item
14FA: B1 FA                 LDA   (OurPtr),Y ; grab rightmost column
14FC: 38                    SEC              ; prepare to subtract
14FD: ED 11 15              SBC   :LeftCol   ; this gives the width less 1
1500: A8                    TAY              ; store it
1501: C8                    INY              ; get it up to par

                   * get the leftmost column back into A-
1502: AD 11 15              LDA   :LeftCol

                   * now we're all set up for our loop that'll call HRBandInvt
1505: 20 12 15     :Loop    JSR   HRBandInvt ; invert that band
1508: E8                    INX              ; up the row number
1509: EC 10 15              CPX   :BtRwPs1   ; are we done yet ?
150C: 90 F7                 BCC   :Loop      ; if not, do another band
                                             ; otherwise, leave

                   * leave
150E: 60                    RTS              ; return from HRRectInvt

                   * local variables

150F: 00           :OfsHi   DS    1          ; holds hi-byte of array offset
1510: 00           :BtRwPs1 DS    1          ; holds bottommost row number
                                             ; ... plus 1
1511: 00           :LeftCol DS    1          ; holds leftmost column number


                   *------------------------ HRBandInvt ---------------------*

                   * switches foreground and background colors in a supplied
                   * ... horizontal band of the hi-res bit-mapped screen

                   * remember, such changes are carried out in a screen grid
                   * ... that's 40 columnswide and 25 rows high

                   * in other words, each grid element controls an area that's
                   * ... 8 pixels wide and 8 pixels high

                   * upon entry, A- contains the leftmost column (0..39)
                   *             X- contains the row (0..24)
                   *             Y- contains the width of the band (1..39)

                   * routine does no error checking, so get those parameters
                   * ... right

                   * upon exit, A- X- and Y- are same as when they entered

                   HRBandInvt

                   * park the starting column
1512: 8D 52 15              STA   :Temp

                   * save some registers
1515: 48                    PHA
1516: 8A                    TXA
1517: 48                    PHA
1518: 98                    TYA
1519: 48                    PHA

                   * set pointer to start of the row
151A: BD E4 15              LDA   HRRowsLo,X ; index table of row lo-bytes
151D: 85 FA                 STA   OurPtr
```

```
151F: BD FD 15          LDA   HRRowsHi,X ; index table of row hi-bytes
1522: 85 FB             STA   OurPtr+1

                * move pointer to start of the band
1524: AD 52 15          LDA   :Temp      ; fetch the starting column
1527: 18                CLC              ; add it to row start to get
1528: 65 FA             ADC   OurPtr     ; ... start of the band
152A: 85 FA             STA   OurPtr
152C: 90 02             BCC   :LooPrep
152E: E6 FB             INC   OurPtr+1

                * loop to swap foreground and background nibbles
1530: 88        :LooPrep DEY             ; make width 0-based

1531: B1 FA     :LoopTop LDA  (OurPtr),Y ; grab a byte

1533: AA                TAX              ; store a copy of byte
1534: 29 0F             AND   #%00001111 ; isolate lo-nibble
1536: 0A                ASL              ; move it into hi-nibble
1537: 0A                ASL
1538: 0A                ASL
1539: 0A                ASL
153A: 8D 52 15          STA   :Temp      ; save it

153D: 8A                TXA              ; get original back
153E: 29 F0             AND   #%11110000 ; isolate hi-nibble
1540: 4A                LSR              ; move it into lo-nibble
1541: 4A                LSR
1542: 4A                LSR
1543: 4A                LSR

1544: 0D 52 15          ORA   :Temp      ; combine the two nibbles

1547: 91 FA             STA   (OurPtr),Y ; store worked-over byte

1549: 88                DEY              ; see if there's more to do
154A: 10 E5             BPL   :LoopTop   ; if so, do it

                * restore some registers
154C: 68                PLA
154D: A8                TAY
154E: 68                PLA
154F: AA                TAX
1550: 68                PLA

                * return from HRBandInvt
1551: 60                RTS

                * local variables

1552: 00        :Temp   DS    1          ; various local activities


                *--------------------- TX40BandInvt --------------------*

                * inverts a band of characters drawn on the standard
                * ... 40-column text screen

                * the text screen can be located anywhere VIC allows it
                * the routine is smart enough to find it

                * upon entry, A- contains the leftmost column (0..39)
                *             X- contains the row (0..24)
                *             Y- contains the width of the band (1..39)

                * routine does no error checking, so get those
```

```
                 * ... entry parameters right

                 * upon exit, A- X- and Y- are same as when they entered

                 TXBandInvt

                 * save the registers / park the parameters
1553: 8D A4 15            STA   :LeftCol
1556: 8E A5 15            STX   :Row
1559: 8C A6 15            STY   :Width

                 * figure the leftmost column's offset from the screen base
155C: 8A                  TXA               ; send row in A-
155D: AE A4 15            LDX   :LeftCol    ; send column in X-
1560: 20 AA 15            JSR   FigOfs4025  ; call offset figuring function
1563: 8D A8 15            STA   :OfstLo     ; store the result
1566: 8E A9 15            STX   :OfstHi

                 * get base and bank of current 40-column screen
1569: 20 BF 15            JSR   BasBnk40    ; comes back with A- (lo) and
156C: 86 FB               STX   OurPtr+1    ; ... X- (hi) holding base
156E: 8C A7 15            STY   :Bank       ; ... address, Y- holding bank #

                 * set pointer to leftmost column by adding base and offset
1571: 18                  CLC               ; prepare to add
1572: 6D A8 15            ADC   :OfstLo     ; A- already holds base-lo
1575: 85 FA               STA   OurPtr      ; store lo-byte result
1577: A5 FB               LDA   OurPtr+1    ; add hi-bytes
1579: 6D A9 15            ADC   :OfstHi
157C: 85 FB               STA   OurPtr+1    ; store hi-byte result

                 * set up for upcoming loop's IndSta banked storage call
157E: A9 FA               LDA   #OurPtr     ; get pointer to our pointer
1580: 8D B9 02            STA   StaVec      ; load the IndSta vector

                 * Y- will serve as a loop counter and screen byte index
1583: AC A6 15            LDY   :Width      ; grab band width
1586: 88                  DEY               ; make it a 0-based count

                 * loop to grab, invert, and store changed poke code
1587: AE A7 15   :LupTop  LDX   :Bank       ; setup for IndFet
158A: A9 FA               LDA   #OurPtr
158C: 20 74 FF            JSR   IndFet      ; fetch a poke code

158F: 49 80               EOR   #%10000000  ; flip-flop hi-bit

1591: AE A7 15            LDX   :Bank       ; setup for IndSta
1594: 20 77 FF            JSR   IndSta      ; store inverted poke code

1597: 88         :LupTest DEY               ; see if there's more to do
1598: 10 ED               BPL   :LupTop     ; if so, do it

                 * restore some registers
159A: AD A4 15            LDA   :LeftCol
159D: AE A5 15            LDX   :Row
15A0: AC A6 15            LDY   :Width

                 * return from TX40BandInvt
15A3: 60                  RTS

                 * ----- local variables ----- *

15A4: 00         :LeftCol DS    1           ; the band's leftmost column
15A5: 00         :Row     DS    1           ; the band's row
15A6: 00         :Width   DS    1           ; the band's width
```

```
15A7: 00         :Bank     DS    1          ; VIC's working bank
15A8: 00         :OfstLo   DS    1          ; offset of the band's leftmost
15A9: 00         :OfstHi   DS    1          ; ... column from the screen base


                 *---------------------- FigOfs4025 ----------------------*

                 * figure a character position's offset from the start of
                 * ... a 40-column by 25-row VIC screen buffer

                 * ( or any 1000 byte chunk of memory organized as
                 *    ... an array [0..24, 0..39]  )

                 * upon entry, A- contains the position's row (0..24)
                 *             X- contains the position's column (0..39)

                 * upon exit,  A- contains the lo-byte of the offset
                 *             X- contains the hi-byte of the offset

                 FigOfs4025

                 * store the parameters
15AA: 8E BE 15           STX   :Column      ; column gets a local spot
15AD: AA                 TAX                ; row into X-

                 * get the lo-byte of the row's starting address
15AE: BD E4 15           LDA   Rows4025Lo,X; indexed from a table

                 * add in the column and park the result
15B1: 18                 CLC                ; prep to add
15B2: 6D BE 15           ADC   :Column      ; do it
15B5: 48                 PHA                ; stack holds lo-byte of offset

                 * grab the hi-byte of the row's starting address
15B6: BD 16 16           LDA   Rows4025Hi,X; indexed from a table

                 * add in any carry
15B9: 69 00              ADC   #0

                 * move hi-byte into X-
15BB: AA                 TAX

                 * get lo-byte into A- from stack
15BC: 68                 PLA

                 * return from FigOfs4025
15BD: 60                 RTS

                 * ----- local variables ----- *

15BE: 00         :Column  DS    1          ; the column


                 *----------------------- BasBnk40 ---------------------*

                 * returns the base address and bank of current 40-column
                 * ... VIC text screen

                 * upon exit, A- contains lo-byte of base address
                 *            X- contains hi-byte of base address
                 *            Y- contains VIC bank ( 0 or 1 )

                 * the text screen can be located anywhere VIC allows it
                 * the routine is smart enough to find it

```

```
                BasBnk40
                * grab byte holding VIC bank state
15BF: AD 06 D5          LDA   MmuRCR
                * mask out all but bit 6
15C2: 29 40             AND   #%01000000
                * move that standout around to bit 0
15C4: 0A                ASL                ; move it into bit 7
15C5: 0A                ASL                ; move it into Carry flag
15C6: 2A                ROL                ; move it into bit 0
                * move that bank # to its register
15C7: A8                TAY
                * now for the base address
                * since the screen can only start on one-K boundaries,
                * ... the low 10 bits of this 16-bit address are 0
                * ... ( that's bits 0..9 )
                * first, we'll get the VIC one-K indicator nibble
                * these become bits 10..13 later on
15C8: AD 18 D0          LDA   VicReg24
                * clear out unwanted nibble
15CB: 29 F0             AND   #%11110000
                * store result for a moment
15CD: 8D E3 15          STA   :BaseHi
                * grab the byte holding future address bits 14..15
15D0: AD 00 DD          LDA   D2PrA
                * flip-flop those two bits
15D3: 49 03             EOR   #%00000011
                * move those two bits into position
                * at the same time, also get bits 10..13 into place
15D5: 4A                LSR
15D6: 6E E3 15          ROR   :BaseHi
15D9: 4A                LSR
15DA: 6E E3 15          ROR   :BaseHi
                * pack up base address bytes
15DD: A9 00             LDA   #0           ; lo-byte is always this
15DF: AE E3 15          LDX   :BaseHi      ; as figured above
                * return from BasBnk40
15E2: 60                RTS
                * ----- local variables -----*
15E3: 00        :BaseHi DS    1            ; hi-byte of screen base address
                        TTL   "S/M ASM 2 C.S"
                *---------- Here Comes The Third Source File -------------*
                        PUT   "S/M ASM 2 C.S"
                *---------------- screen address data ------------------*
                * a few tables are intertwined here
```

```
                * HRRowsLo holds the lo-bytes of row start absolute
                * ... addresses for  a stock hi-res bit-map color
                * ... screen buffer

                * HRRowsHi holds the hi-bytes for the same

                * Rows4025Lo holds the lo-bytes of row start relative
                * ... addresses for any 40h by 25v screen buffer

                * Rows4025Hi holds the hi-bytes for the same

                HRRowsLo
                Rows4025Lo
15E4: 00 28 50           HEX   00,28,50
15E7: 78 A0 C8           HEX   78,A0,C8
15EA: F0 18 40           HEX   F0,18,40
15ED: 68 90 B8           HEX   68,90,B8
15F0: E0 08 30           HEX   E0,08,30
15F3: 58 80 A8           HEX   58,80,A8
15F6: D0 F8 20           HEX   D0,F8,20
15F9: 48 70 98           HEX   48,70,98
15FC: C0                 HEX   C0

                HRRowsHi
15FD: 1C 1C 1C           HEX   1C,1C,1C
1600: 1C 1C 1C           HEX   1C,1C,1C
1603: 1C 1D 1D           HEX   1C,1D,1D
1606: 1D 1D 1D           HEX   1D,1D,1D
1609: 1D 1E 1E           HEX   1D,1E,1E
160C: 1E 1E 1E           HEX   1E,1E,1E
160F: 1E 1E 1F           HEX   1E,1E,1F
1612: 1F 1F 1F           HEX   1F,1F,1F
1615: 1F                 HEX   1F

                Rows4025Hi
1616: 00 00 00           HEX   00,00,00
1619: 00 00 00           HEX   00,00,00
161C: 00 01 01           HEX   00,01,01
161F: 01 01 01           HEX   01,01,01
1622: 01 02 02           HEX   01,02,02
1625: 02 02 02           HEX   02,02,02
1628: 02 02 03           HEX   02,02,03
162B: 03 03 03           HEX   03,03,03
162E: 03                 HEX   03


                *------------- area data for the sound/music lab ---------*

                * coordinates are in a coordinate system based on the
                * ... finger cursor sprite's hot point

                * here's the format for each area's data record :

                *       DFB    ???      top boundary
                *       DFB    ???      bottom boundary
                *       DDB    ???      left boundary -   hi byte, then lo
                *       DDB    ???      right boundary -  hi byte, then lo
                *       DFB    ???      area identifier

                LabAreas

162F: 36                 DFB   54          ; sound : title
1630: 4C                 DFB   76
1631: 00 15              DDB   21
1633: 00 33              DDB   51
```

```
1635: 01        >72           DFB   1
                >73
1636: 36        >74           DFB   54         ; sound : voice
1637: 4C        >75           DFB   76
1638: 00 35     >76           DDB   53
163A: 00 45     >77           DDB   69
163C: 02        >78           DFB   2
                >79
163D: 36        >80           DFB   54         ; sound : frequency
163E: 4C        >81           DFB   76
163F: 00 46     >82           DDB   70
1641: 00 76     >83           DDB   118
1643: 03        >84           DFB   3
                >85
1644: 36        >86           DFB   54         ; sound : duration
1645: 4C        >87           DFB   76
1646: 00 77     >88           DDB   119
1648: 00 A5     >89           DDB   165
164A: 04        >90           DFB   4
                >91
164B: 36        >92           DFB   54         ; sound : step direction
164C: 4C        >93           DFB   76
164D: 00 A6     >94           DDB   166
164F: 00 B5     >95           DDB   181
1651: 05        >96           DFB   5
                >97
1652: 36        >98           DFB   54         ; sound : minimum frequency
1653: 4C        >99           DFB   76
1654: 00 B6     >100          DDB   182
1656: 00 E5     >101          DDB   229
1658: 06        >102          DFB   6
                >103
1659: 36        >104          DFB   54         ; sound : sweep step value
165A: 4C        >105          DFB   76
165B: 00 E6     >106          DDB   230
165D: 01 14     >107          DDB   276
165F: 07        >108          DFB   7
                >109
1660: 36        >110          DFB   54         ; sound : waveform
1661: 4C        >111          DFB   76
1662: 01 15     >112          DDB   277
1664: 01 26     >113          DDB   294
1666: 08        >114          DFB   8
                >115
1667: 36        >116          DFB   54         ; sound : pulse width
1668: 4C        >117          DFB   76
1669: 01 27     >118          DDB   295
166B: 01 4C     >119          DDB   332
166D: 09        >120          DFB   9
                >121
166E: 56        >122          DFB   86         ; play : title
166F: 7C        >123          DFB   124
1670: 00 15     >124          DDB   21
1672: 00 3C     >125          DDB   60
1674: 0A        >126          DFB   10
                >127
1675: 56        >128          DFB   86         ; play : data
1676: 7C        >129          DFB   124
1677: 00 3E     >130          DDB   62
1679: 01 4C     >131          DDB   332
167B: 0B        >132          DFB   11
                >133
167C: 86        >134          DFB   134        ; envelope : title
167D: 94        >135          DFB   148
167E: 00 15     >136          DDB   21
1680: 00 94     >137          DDB   148
```

```
1682: 0C             DFB  12
1683: 86             DFB  134     ; volume : title
1684: 94             DFB  148
1685: 00 9D          DDB  157
1687: 00 BC          DDB  188
1689: 53             DFB  83
168A: 86             DFB  134     ; tempo : title
168B: 94             DFB  148
168C: 00 C5          DDB  197
168E: 00 E4          DDB  228
1690: 55             DFB  85
1691: 86             DFB  134     ; filter : title
1692: 94             DFB  148
1693: 00 ED          DDB  237
1695: 01 4C          DDB  332
1697: 57             DFB  87
1698: 96             DFB  150     ; envelope 0 - #
1699: A8             DFB  168
169A: 00 15          DDB  21
169C: 00 28          DDB  40
169E: 0D             DFB  13
169F: 96             DFB  150     ; envelope 0 - attack
16A0: A8             DFB  168
16A1: 00 29          DDB  41
16A3: 00 38          DDB  56
16A5: 0E             DFB  14
16A6: 96             DFB  150     ; envelope 0 - decay
16A7: A8             DFB  168
16A8: 00 39          DDB  57
16AA: 00 48          DDB  72
16AC: 0F             DFB  15
16AD: 96             DFB  150     ; envelope 0 - sustain
16AE: A8             DFB  168
16AF: 00 49          DDB  73
16B1: 00 58          DDB  88
16B3: 10             DFB  16
16B4: 96             DFB  150     ; envelope 0 - release
16B5: A8             DFB  168
16B6: 00 59          DDB  89
16B8: 00 68          DDB  104
16BA: 11             DFB  17
16BB: 96             DFB  150     ; envelope 0 - waveform
16BC: A8             DFB  168
16BD: 00 69          DDB  105
16BF: 00 70          DDB  112
16C1: 12             DFB  18
16C2: 96             DFB  150     ; envelope 0 - pulse width
16C3: A8             DFB  168
16C4: 00 71          DDB  113
16C6: 00 94          DDB  148
16C8: 13             DFB  19
16C9: 96             DFB  150     ; volume - data
16CA: A4             DFB  164
16CB: 00 9D          DDB  157
```

```
16CD: 00 BC     DDB  188
16CF: 54        DFB  84

16D0: 96        DFB  150     ; tempo - data
16D1: A4        DFB  164
16D2: 00 C5     DDB  197
16D4: 00 E4     DDB  228
16D6: 56        DFB  86

16D7: 96        DFB  150     ; filter - frequency
16D8: AC        DFB  172
16D9: 00 ED     DDB  237
16DB: 01 15     DDB  277
16DD: 58        DFB  88

16DE: 96        DFB  150     ; filter - low-pass
16DF: AC        DFB  172
16E0: 01 16     DDB  278
16E2: 01 21     DDB  289
16E4: 59        DFB  89

16E5: 96        DFB  150     ; filter - band-pass
16E6: AC        DFB  172
16E7: 01 22     DDB  290
16E9: 01 29     DDB  297
16EB: 5A        DFB  90

16EC: 96        DFB  150     ; filter - high-pass
16ED: AC        DFB  172
16EE: 01 2A     DDB  298
16F0: 01 35     DDB  309
16F2: 5B        DFB  91

16F3: 96        DFB  150     ; filter - resonance
16F4: AC        DFB  172
16F5: 01 36     DDB  310
16F7: 01 4C     DDB  332
16F9: 5C        DFB  92

16FA: A6        DFB  166     ; frame counter - title
16FB: B4        DFB  180
16FC: 00 96     DDB  150
16FE: 00 C5     DDB  197
1700: 5D        DFB  93

1701: A6        DFB  166     ; frame counter - data
1702: B4        DFB  180
1703: 00 C6     DDB  198
1705: 00 EB     DDB  235
1707: 5E        DFB  94

1708: A9        DFB  169     ; envelope 1 - #
1709: B0        DFB  176
170A: 00 15     DDB  21
170C: 00 28     DDB  40
170E: 14        DFB  20

170F: A9        DFB  169     ; envelope 1 - attack
1710: B0        DFB  176
1711: 00 29     DDB  41
1713: 00 38     DDB  56
1715: 15        DFB  21

1716: A9        DFB  169     ; envelope 1 - decay
1717: B0        DFB  176
```

```
1718: 00 39        DDB   57
171A: 00 48        DDB   72
171C: 16           DFB   22

171D: A9           DFB   169      ; envelope 1 - sustain
171E: B0           DFB   176
171F: 00 49        DDB   73
1721: 00 58        DDB   88
1723: 17           DFB   23

1724: A9           DFB   169      ; envelope 1 - release
1725: B0           DFB   176
1726: 00 59        DDB   89
1728: 00 68        DDB   104
172A: 18           DFB   24

172B: A9           DFB   169      ; envelope 1 - waveform
172C: B0           DFB   176
172D: 00 69        DDB   105
172F: 00 70        DDB   112
1731: 19           DFB   25

1732: A9           DFB   169      ; envelope 1 - pulse width
1733: B0           DFB   176
1734: 00 71        DDB   113
1736: 00 94        DDB   148
1738: 1A           DFB   26

1739: B1           DFB   177      ; envelope 2 - #
173A: B8           DFB   184
173B: 00 15        DDB   21
173D: 00 28        DDB   40
173F: 1B           DFB   27

1740: B1           DFB   177      ; envelope 2 - attack
1741: B8           DFB   184
1742: 00 29        DDB   41
1744: 00 38        DDB   56
1746: 1C           DFB   28

1747: B1           DFB   177      ; envelope 2 - decay
1748: B8           DFB   184
1749: 00 39        DDB   57
174B: 00 48        DDB   72
174D: 1D           DFB   29

174E: B1           DFB   177      ; envelope 2 - sustain
174F: B8           DFB   184
1750: 00 49        DDB   73
1752: 00 58        DDB   88
1754: 1E           DFB   30

1755: B1           DFB   177      ; envelope 2 - release
1756: B8           DFB   184
1757: 00 59        DDB   89
1759: 00 68        DDB   104
175B: 1F           DFB   31

175C: B1           DFB   177      ; envelope 2 - waveform
175D: B8           DFB   184
175E: 00 69        DDB   105
1760: 00 70        DDB   112
1762: 20           DFB   32

1763: B1           DFB   177      ; envelope 2 - pulse width
1764: B8           DFB   184
```

```
1765: 00 71        DDB   113
1767: 00 94        DDB   148
1769: 21           DFB   33
176A: B6           DFB   182       ; go
176B: C4           DFB   196
176C: 00 9D        DDB   157
176E: 00 BC        DDB   188
1770: 5F           DFB   95
1771: B6           DFB   182       ; forward
1772: C4           DFB   196
1773: 00 C5        DDB   197
1775: 00 E4        DDB   228
1777: 60           DFB   96
1778: B6           DFB   182       ; load
1779: C4           DFB   196
177A: 00 ED        DDB   237
177C: 01 0C        DDB   268
177E: 61           DFB   97
177F: B6           DFB   182       ; clear
1780: C4           DFB   196
1781: 01 15        DDB   277
1783: 01 34        DDB   308
1785: 62           DFB   98
1786: B6           DFB   182       ; help
1787: DC           DFB   220
1788: 01 3D        DDB   317
178A: 01 4C        DDB   332
178C: 63           DFB   99
178D: B9           DFB   185       ; envelope 3 - #
178E: C0           DFB   192
178F: 00 15        DDB   21
1791: 00 28        DDB   40
1793: 22           DFB   34
1794: B9           DFB   185       ; envelope 3 - attack
1795: C0           DFB   192
1796: 00 29        DDB   41
1798: 00 38        DDB   56
179A: 23           DFB   35
179B: B9           DFB   185       ; envelope 3 - decay
179C: C0           DFB   192
179D: 00 39        DDB   57
179F: 00 48        DDB   72
17A1: 24           DFB   36
17A2: B9           DFB   185       ; envelope 3 - sustain
17A3: C0           DFB   192
17A4: 00 49        DDB   73
17A6: 00 58        DDB   88
17A8: 25           DFB   37
17A9: B9           DFB   185       ; envelope 3 - release
17AA: C0           DFB   192
17AB: 00 59        DDB   89
17AD: 00 68        DDB   104
17AF: 26           DFB   38
17B0: B9           DFB   185       ; envelope 3 - waveform
```

```
17B1: C0        >399          DFB   192
17B2: 00 69     >400          DDB   105
17B4: 00 70     >401          DDB   112
17B6: 27        >402          DFB   39
                >403
17B7: B9        >404          DFB   185        ; envelope 3 - pulse width
17B8: C0        >405          DFB   192
17B9: 00 71     >406          DDB   113
17BB: 00 94     >407          DDB   148
17BD: 28        >408          DFB   40
                >409
17BE: C1        >410          DFB   193        ; envelope 4 - #
17BF: C8        >411          DFB   200
17C0: 00 15     >412          DDB   21
17C2: 00 28     >413          DDB   40
17C4: 29        >414          DFB   41
                >415
17C5: C1        >416          DFB   193        ; envelope 4 - attack
17C6: C8        >417          DFB   200
17C7: 00 29     >418          DDB   41
17C9: 00 38     >419          DDB   56
17CB: 2A        >420          DFB   42
                >421
17CC: C1        >422          DFB   193        ; envelope 4 - decay
17CD: C8        >423          DFB   200
17CE: 00 39     >424          DDB   57
17D0: 00 48     >425          DDB   72
17D2: 2B        >426          DFB   43
                >427
17D3: C1        >428          DFB   193        ; envelope 4 - sustain
17D4: C8        >429          DFB   200
17D5: 00 49     >430          DDB   73
17D7: 00 58     >431          DDB   88
17D9: 2C        >432          DFB   44
                >433
17DA: C1        >434          DFB   193        ; envelope 4 - release
17DB: C8        >435          DFB   200
17DC: 00 59     >436          DDB   89
17DE: 00 68     >437          DDB   104
17E0: 2D        >438          DFB   45
                >439
17E1: C1        >440          DFB   193        ; envelope 4 - waveform
17E2: C8        >441          DFB   200
17E3: 00 69     >442          DDB   105
17E5: 00 70     >443          DDB   112
17E7: 2E        >444          DFB   46
                >445
17E8: C1        >446          DFB   193        ; envelope 4 - pulse widt
17E9: C8        >447          DFB   200
17EA: 00 71     >448          DDB   113
17EC: 00 94     >449          DDB   148
17EE: 2F        >450          DFB   47
                >451
17EF: C9        >452          DFB   201        ; envelope 5 - #
17F0: D0        >453          DFB   208
17F1: 00 15     >454          DDB   21
17F3: 00 28     >455          DDB   40
17F5: 30        >456          DFB   48
                >457
17F6: C9        >458          DFB   201        ; envelope 5 - attack
17F7: D0        >459          DFB   208
17F8: 00 29     >460          DDB   41
17FA: 00 38     >461          DDB   56
17FC: 31        >462          DFB   49
                >463
17FD: C9        >464          DFB   201        ; envelope 5 - decay
```

```
17FE: D0        >465        DFB   208
17FF: 00 39     >466        DDB   57
1801: 00 48     >467        DDB   72
1803: 32        >468        DFB   50
                >469
1804: C9        >470        DFB   201       ; envelope 5 - sustain
1805: D0        >471        DFB   208
1806: 00 49     >472        DDB   73
1808: 00 58     >473        DDB   88
180A: 33        >474        DFB   51
                >475
180B: C9        >476        DFB   201       ; envelope 5 - release
180C: D0        >477        DFB   208
180D: 00 59     >478        DDB   89
180F: 00 68     >479        DDB   104
1811: 34        >480        DFB   52
                >481
1812: C9        >482        DFB   201       ; envelope 5 - waveform
1813: D0        >483        DFB   208
1814: 00 69     >484        DDB   105
1816: 00 70     >485        DDB   112
1818: 35        >486        DFB   53
                >487
1819: C9        >488        DFB   201       ; envelope 5 - pulse width
181A: D0        >489        DFB   208
181B: 00 71     >490        DDB   113
181D: 00 94     >491        DDB   148
181F: 36        >492        DFB   54
                >493
1820: CE        >494        DFB   206       ; show
1821: DC        >495        DFB   220
1822: 00 9D     >496        DDB   157
1824: 00 BC     >497        DDB   188
1826: 64        >498        DFB   100
                >499
1827: CE        >500        DFB   206       ; backward
1828: DC        >501        DFB   220
1829: 00 C5     >502        DDB   197
182B: 00 E4     >503        DDB   228
182D: 65        >504        DFB   101
                >505
182E: CE        >506        DFB   206       ; save
182F: DC        >507        DFB   220
1830: 00 ED     >508        DDB   237
1832: 01 0C     >509        DDB   268
1834: 66        >510        DFB   102
                >511
1835: CE        >512        DFB   206       ; print
1836: DC        >513        DFB   220
1837: 01 15     >514        DDB   277
1839: 01 34     >515        DDB   308
183B: 67        >516        DFB   103
                >517
183C: D1        >518        DFB   209       ; envelope 6 - #
183D: D8        >519        DFB   216
183E: 00 15     >520        DDB   21
1840: 00 28     >521        DDB   40
1842: 37        >522        DFB   55
                >523
1843: D1        >524        DFB   209       ; envelope 6 - attack
1844: D8        >525        DFB   216
1845: 00 29     >526        DDB   41
1847: 00 38     >527        DDB   56
1849: 38        >528        DFB   56
                >529
```

```
184A: D1              DFB  209      ; envelope 6 - decay
184B: D8              DFB  216
184C: 00 39           DDB  57
184E: 00 48           DDB  72
1850: 39              DFB  57

1851: D1              DFB  209      ; envelope 6 - sustain
1852: D8              DFB  216
1853: 00 49           DDB  73
1855: 00 58           DDB  88
1857: 3A              DFB  58

1858: D1              DFB  209      ; envelope 6 - release
1859: D8              DFB  216
185A: 00 59           DDB  89
185C: 00 68           DDB  104
185E: 3B              DFB  59

185F: D1              DFB  209      ; envelope 6 - waveform
1860: D8              DFB  216
1861: 00 69           DDB  105
1863: 00 70           DDB  112
1865: 3C              DFB  60

1866: D1              DFB  209      ; envelope 6 - pulse width
1867: D8              DFB  216
1868: 00 71           DDB  113
186A: 00 94           DDB  148
186C: 3D              DFB  61

186D: D9              DFB  217      ; envelope 7 - #
186E: E0              DFB  224
186F: 00 15           DDB  21
1871: 00 28           DDB  40
1873: 3E              DFB  62

1874: D9              DFB  217      ; envelope 7 - attack
1875: E0              DFB  224
1876: 00 29           DDB  41
1878: 00 38           DDB  56
187A: 3F              DFB  63

187B: D9              DFB  217      ; envelope 7 - decay
187C: E0              DFB  224
187D: 00 2A           DDB  42
187F: 00 48           DDB  72
1881: 40              DFB  64

1882: D9              DFB  217      ; envelope 7 - sustain
1883: E0              DFB  224
1884: 00 49           DDB  73
1886: 00 58           DDB  88
1888: 41              DFB  65

1889: D9              DFB  217      ; envelope 7 - release
188A: E0              DFB  224
188B: 00 59           DDB  89
188D: 00 68           DDB  104
188F: 42              DFB  66

1890: D9              DFB  217      ; envelope 7 - waveform
1891: E0              DFB  224
1892: 00 69           DDB  105
1894: 00 70           DDB  112
1896: 43              DFB  67
```

```
1897: D9            DFB   217      ; envelope 7 - pulse width
1898: E0            DFB   224
1899: 00 71         DDB   113
189B: 00 94         DDB   148
189D: 44            DFB   68

189E: E1            DFB   225      ; envelope 8 - #
189F: E8            DFB   232
18A0: 00 15         DDB   21
18A2: 00 28         DDB   40
18A4: 45            DFB   69

18A5: E1            DFB   225      ; envelope 8 - attack
18A6: E8            DFB   232
18A7: 00 29         DDB   41
18A9: 00 38         DDB   56
18AB: 46            DFB   70

18AC: E1            DFB   225      ; envelope 8 - decay
18AD: E8            DFB   232
18AE: 00 39         DDB   57
18B0: 00 48         DDB   72
18B2: 47            DFB   71

18B3: E1            DFB   225      ; envelope 8 - sustain
18B4: E8            DFB   232
18B5: 00 49         DDB   73
18B7: 00 58         DDB   88
18B9: 48            DFB   72

18BA: E1            DFB   225      ; envelope 8 - release
18BB: E8            DFB   232
18BC: 00 59         DDB   89
18BE: 00 68         DDB   104
18C0: 49            DFB   73

18C1: E1            DFB   225      ; envelope 8 - waveform
18C2: E8            DFB   232
18C3: 00 69         DDB   105
18C5: 00 70         DDB   112
18C7: 4A            DFB   74

18C8: E1            DFB   225      ; envelope 8 - pulse width
18C9: E8            DFB   232
18CA: 00 71         DDB   113
18CC: 00 94         DDB   148
18CE: 4B            DFB   75

18CF: E6            DFB   230      ; message window
18D0: F4            DFB   244
18D1: 00 9D         DDB   157
18D3: 01 24         DDB   292
18D5: 68            DFB   104

18D6: E6            DFB   230      ; end
18D7: F4            DFB   244
18D8: 01 2E         DDB   302
18DA: 01 4C         DDB   332
18DC: 69            DFB   105

18DD: E9            DFB   233      ; envelope 9 - #
18DE: F4            DFB   244
18DF: 00 15         DDB   21
18E1: 00 28         DDB   40
```

```
18E3: 4C           DFB   76
18E4: E9           DFB   233       ; envelope 9 - attack
18E5: F4           DFB   244
18E6: 00 29        DDB   41
18E8: 00 38        DDB   56
18EA: 4D           DFB   77
18EB: E9           DFB   233       ; envelope 9 - decay
18EC: F4           DFB   244
18ED: 00 39        DDB   57
18EF: 00 48        DDB   72
18F1: 4E           DFB   78
18F2: E9           DFB   233       ; envelope 9 - sustain
18F3: F4           DFB   244
18F4: 00 49        DDB   73
18F6: 00 58        DDB   88
18F8: 4F           DFB   79
18F9: E9           DFB   233       ; envelope 9 - release
18FA: F4           DFB   244
18FB: 00 59        DDB   89
18FD: 00 68        DDB   104
18FF: 50           DFB   80
1900: E9           DFB   233       ; envelope 9 - waveform
1901: F4           DFB   244
1902: 00 69        DDB   105
1904: 00 70        DDB   112
1906: 51           DFB   81
1907: E9           DFB   233       ; envelope 9 - pulse width
1908: F4           DFB   244
1909: 00 71        DDB   113
190B: 00 94        DDB   148
190D: 52           DFB   82
190E: 00           DFB   0         ; marks end of this table
             *-------------- area data for the help screen ------------*
             * here's the format :
             *      DFB    ???     top boundary
             *      DFB    ???     bottom boundary
             *      DDB    ???     left boundary -   hi byte, then lo
             *      DDB    ???     right boundary -  hi byte, then lo
             *      DFB    ???     area identifier
             HlpAreas
190F: E5           DFB   229       ; first
1910: F5           DFB   245
1911: 00 15        DDB   21
1913: 00 45        DDB   69
1915: 01           DFB   1
1916: E5           DFB   229       ; previous
1917: F5           DFB   245
1918: 00 55        DDB   85
191A: 00 9D        DDB   157
191C: 02           DFB   2
191D: E5           DFB   229       ; next
```

```
191E: F5                 DFB   245
191F: 00 AD              DDB   173
1921: 00 D5              DDB   213
1923: 03                 DFB   3

1924: E5                 DFB   229        ; last
1925: F5                 DFB   245
1926: 00 E5              DDB   229
1928: 01 0D              DDB   269
192A: 04                 DFB   4

192B: E5                 DFB   229        ; lab
192C: F5                 DFB   245
192D: 01 2D              DDB   301
192F: 01 4D              DDB   333
1931: 05                 DFB   5

1932: 00                 DFB   0          ; marks end of this table


               *------ inversion rectangles for lab areas ---------------*

               * an array of area inversion rectangle data, arranged in
               * ... order of each area's identifying number

               * coordinates are in the standard 40-column text screen
               * ... coordinate system, since bit-map color info is
               * ... stored in such a system

               * here's the format for each area's inversion rectangle :

               *       DFB     ??,??    topMostRow,bottomMostRow
               *       DFB     ??,??    leftMostColumn,rightMostColumn

               LabInvRex
1933: 02 02              DFB   2,2        ; sound : title
1935: 01 03              DFB   1,3

1937: 01 02              DFB   1,2        ; sound : voice
1939: 05 05              DFB   5,5

193B: 01 02              DFB   1,2        ; sound : frequency
193D: 07 0B              DFB   7,11

193F: 01 02              DFB   1,2        ; sound : duration
1941: 0D 11              DFB   13,17

1943: 01 02              DFB   1,2        ; sound : step direction
1945: 13 13              DFB   19,19

1947: 01 02              DFB   1,2        ; sound : minimum frequency
1949: 15 19              DFB   21,25

194B: 01 02              DFB   1,2        ; sound : sweep step value
194D: 1B 1F              DFB   27,31

194F: 01 02              DFB   1,2        ; sound : waveform
1951: 21 21              DFB   33,33

1953: 01 02              DFB   1,2        ; sound : pulse width
1955: 23 26              DFB   35,38

1957: 07 07              DFB   7,7        ; play : title
1959: 01 04              DFB   1,4

```

```
195B: 05 08       DFB   5,8        ; play : data
195D: 06 26       DFB   6,38

195F: 0B 0B       DFB   11,11      ; envelope : title
1961: 04 0C       DFB   4,12

1963: 0E 0E       DFB   14,14      ; envelope 0 - #
1965: 01 01       DFB   1,1

1967: 0E 0E       DFB   14,14      ; envelope 0 - attack
1969: 03 04       DFB   3,4

196B: 0E 0E       DFB   14,14      ; envelope 0 - decay
196D: 05 06       DFB   5,6

196F: 0E 0E       DFB   14,14      ; envelope 0 - sustain
1971: 07 08       DFB   7,8

1973: 0E 0E       DFB   14,14      ; envelope 0 - release
1975: 09 0A       DFB   9,10

1977: 0E 0E       DFB   14,14      ; envelope 0 - waveform
1979: 0B 0B       DFB   11,11

197B: 0E 0E       DFB   14,14      ; envelope 0 - pulse width
197D: 0C 0F       DFB   12,15

197F: 0F 0F       DFB   15,15      ; envelope 1 - #
1981: 01 01       DFB   1,1

1983: 0F 0F       DFB   15,15      ; envelope 1 - attack
1985: 03 04       DFB   3,4

1987: 0F 0F       DFB   15,15      ; envelope 1 - decay
1989: 05 06       DFB   5,6

198B: 0F 0F       DFB   15,15      ; envelope 1 - sustain
198D: 07 08       DFB   7,8

198F: 0F 0F       DFB   15,15      ; envelope 1 - release
1991: 09 0A       DFB   9,10

1993: 0F 0F       DFB   15,15      ; envelope 1 - waveform
1995: 0B 0B       DFB   11,11

1997: 0E 0E       DFB   14,14      ; envelope 1 - pulse width
1999: 0C 0F       DFB   12,15

199B: 10 10       DFB   16,16      ; envelope 2 - #
199D: 01 01       DFB   1,1

199F: 10 10       DFB   16,16      ; envelope 2 - attack
19A1: 03 04       DFB   3,4

19A3: 10 10       DFB   16,16      ; envelope 2 - decay
19A5: 05 06       DFB   5,6

19A7: 10 10       DFB   16,16      ; envelope 2 - sustain
19A9: 07 08       DFB   7,8

19AB: 10 10       DFB   16,16      ; envelope 2 - release
19AD: 09 0A       DFB   9,10

19AF: 10 10       DFB   16,16      ; envelope 2 - waveform
19B1: 0B 0B       DFB   11,11
```

```
19B3: 10 10          DFB   16,16     ; envelope 2 - pulse width
19B5: 0C 0F          DFB   12,15
19B7: 11 11          DFB   17,17     ; envelope 3 - #
19B9: 01 01          DFB   1,1
19BB: 11 11          DFB   17,17     ; envelope 3 - attack
19BD: 03 04          DFB   3,4
19BF: 11 11          DFB   17,17     ; envelope 3 - decay
19C1: 05 06          DFB   5,6
19C3: 11 11          DFB   17,17     ; envelope 3 - sustain
19C5: 07 08          DFB   7,8
19C7: 11 11          DFB   17,17     ; envelope 3 - release
19C9: 09 0A          DFB   9,10
19CB: 11 11          DFB   17,17     ; envelope 3 - waveform
19CD: 0B 0B          DFB   11,11
19CF: 11 11          DFB   17,17     ; envelope 3 - pulse width
19D1: 0C 0F          DFB   12,15
19D3: 12 12          DFB   18,18     ; envelope 4 - #
19D5: 01 01          DFB   1,1
19D7: 12 12          DFB   18,18     ; envelope 4 - attack
19D9: 03 04          DFB   3,4
19DB: 12 12          DFB   18,18     ; envelope 4 - decay
19DD: 05 06          DFB   5,6
19DF: 12 12          DFB   18,18     ; envelope 4 - sustain
19E1: 07 08          DFB   7,8
19E3: 12 12          DFB   18,18     ; envelope 4 - release
19E5: 09 0A          DFB   9,10
19E7: 12 12          DFB   18,18     ; envelope 4 - waveform
19E9: 0B 0B          DFB   11,11
19EB: 12 12          DFB   18,18     ; envelope 4 - pulse width
19ED: 0C 0F          DFB   12,15
19EF: 13 13          DFB   19,19     ; envelope 5 - #
19F1: 01 01          DFB   1,1
19F3: 13 13          DFB   19,19     ; envelope 5 - attack
19F5: 03 04          DFB   3,4
19F7: 13 13          DFB   19,19     ; envelope 5 - decay
19F9: 05 06          DFB   5,6
19FB: 13 13          DFB   19,19     ; envelope 5 - sustain
19FD: 07 08          DFB   7,8
19FF: 13 13          DFB   19,19     ; envelope 5 - release
1A01: 09 0A          DFB   9,10
1A03: 13 13          DFB   19,19     ; envelope 5 - waveform
1A05: 0B 0B          DFB   11,11
1A07: 13 13          DFB   19,19     ; envelope 5 - pulse width
1A09: 0C 0F          DFB   12,15
```

```
1A0B: 14 14     DFB   20,20     ; envelope 6 - #
1A0D: 01 01     DFB   1,1

1A0F: 14 14     DFB   20,20     ; envelope 6 - attack
1A11: 03 04     DFB   3,4

1A13: 14 14     DFB   20,20     ; envelope 6 - decay
1A15: 05 06     DFB   5,6

1A17: 14 14     DFB   20,20     ; envelope 6 - sustain
1A19: 07 08     DFB   7,8

1A1B: 14 14     DFB   20,20     ; envelope 6 - release
1A1D: 09 0A     DFB   9,10

1A1F: 14 14     DFB   20,20     ; envelope 6 - waveform
1A21: 0B 0B     DFB   11,11

1A23: 14 14     DFB   20,20     ; envelope 6 - pulse width
1A25: 0C 0F     DFB   12,15

1A27: 15 15     DFB   21,21     ; envelope 7 - #
1A29: 01 01     DFB   1,1

1A2B: 15 15     DFB   21,21     ; envelope 7 - attack
1A2D: 03 04     DFB   3,4

1A2F: 15 15     DFB   21,21     ; envelope 7 - decay
1A31: 05 06     DFB   5,6

1A33: 15 15     DFB   21,21     ; envelope 7 - sustain
1A35: 07 08     DFB   7,8

1A37: 15 15     DFB   21,21     ; envelope 7 - release
1A39: 09 0A     DFB   9,10

1A3B: 15 15     DFB   21,21     ; envelope 7 - waveform
1A3D: 0B 0B     DFB   11,11

1A3F: 15 15     DFB   21,21     ; envelope 7 - pulse width
1A41: 0C 0F     DFB   12,15

1A43: 16 16     DFB   22,22     ; envelope 8 - #
1A45: 01 01     DFB   1,1

1A47: 16 16     DFB   22,22     ; envelope 8 - attack
1A49: 03 04     DFB   3,4

1A4B: 16 16     DFB   22,22     ; envelope 8 - decay
1A4D: 05 06     DFB   5,6

1A4F: 16 16     DFB   22,22     ; envelope 8 - sustain
1A51: 07 08     DFB   7,8

1A53: 16 16     DFB   22,22     ; envelope 8 - release
1A55: 09 0A     DFB   9,10

1A57: 16 16     DFB   22,22     ; envelope 8 - waveform
1A59: 0B 0B     DFB   11,11

1A5B: 16 16     DFB   22,22     ; envelope 8 - pulse width
1A5D: 0C 0F     DFB   12,15

1A5F: 17 17     DFB   23,23     ; envelope 9 - #
```

```
1A61: 01 01          DFB   1,1

1A63: 17 17          DFB   23,23      ; envelope 9 - attack
1A65: 03 04          DFB   3,4

1A67: 17 17          DFB   23,23      ; envelope 9 - decay
1A69: 05 06          DFB   5,6

1A6B: 17 17          DFB   23,23      ; envelope 9 - sustain
1A6D: 07 08          DFB   7,8

1A6F: 17 17          DFB   23,23      ; envelope 9 - release
1A71: 09 0A          DFB   9,10

1A73: 17 17          DFB   23,23      ; envelope 9 - waveform
1A75: 0B 0B          DFB   11,11

1A77: 17 17          DFB   23,23      ; envelope 9 - pulse width
1A79: 0C 0F          DFB   12,15

1A7B: 0B 0B          DFB   11,11      ; volume : title
1A7D: 12 14          DFB   18,20

1A7F: 0D 0D          DFB   13,13      ; volume - data
1A81: 12 14          DFB   18,20

1A83: 0B 0B          DFB   11,11      ; tempo : title
1A85: 17 19          DFB   23,25

1A87: 0D 0D          DFB   13,13      ; tempo - data
1A89: 17 19          DFB   23,25

1A8B: 0B 0B          DFB   11,11      ; filter : title
1A8D: 1E 23          DFB   30,35

1A8F: 0D 0E          DFB   13,14      ; filter - frequency
1A91: 1C 1F          DFB   28,31

1A93: 0D 0E          DFB   13,14      ; filter - low-pass
1A95: 21 21          DFB   33,33

1A97: 0D 0E          DFB   13,14      ; filter - band-pass
1A99: 22 22          DFB   34,34

1A9B: 0D 0E          DFB   13,14      ; filter - high-pass
1A9D: 23 23          DFB   35,35

1A9F: 0D 0E          DFB   13,14      ; filter - resonance
1AA1: 25 26          DFB   37,38

1AA3: 0F 0F          DFB   15,15      ; frame counter - title
1AA5: 11 15          DFB   17,21

1AA7: 0F 0F          DFB   15,15      ; frame counter - data
1AA9: 17 1A          DFB   23,26

1AAB: 11 11          DFB   17,17      ; go
1AAD: 12 13          DFB   18,19

1AAF: 11 11          DFB   17,17      ; forward
1AB1: 17 19          DFB   23,25

1AB3: 11 11          DFB   17,17      ; load
1AB5: 1C 1E          DFB   28,30

```

```
1AB7: 11 11          DFB   17,17      ; clear
1AB9: 21 23          DFB   33,35

1ABB: 11 14          DFB   17,20      ; help
1ABD: 26 26          DFB   38,38

1ABF: 14 14          DFB   20,20      ; show
1AC1: 12 14          DFB   18,20

1AC3: 14 14          DFB   20,20      ; backward
1AC5: 17 19          DFB   23,25

1AC7: 14 14          DFB   20,20      ; save
1AC9: 1C 1E          DFB   28,30

1ACB: 14 14          DFB   20,20      ; print
1ACD: 21 23          DFB   33,35

1ACF: 17 17          DFB   23,23      ; message window
1AD1: 12 21          DFB   18,33

1AD3: 17 17          DFB   23,23      ; end
1AD5: 24 26          DFB   36,38


                 *------------------ sprite motion data -------------------*

                 * data for moving north

                 North
1AD7: 03         :Speed    HEX   03
1AD8: 00         :Unknown  HEX   00
1AD9: 00         :Quadrnt  HEX   00
1ADA: 00 00      :DeltaX   HEX   00,00
1ADC: FF 7F      :DeltaY   HEX   FF,7F


                 * data for moving northeast

                 NorthEast
1ADE: 03         :Speed    HEX   03
1ADF: 00         :Unknown  HEX   00
1AE0: 00         :Quadrnt  HEX   00
1AE1: 2B 5A      :DeltaX   HEX   2B,5A
1AE3: 2B 5A      :DeltaY   HEX   2B,5A


                 * data for moving east

                 East
1AE5: 04         :Speed    HEX   04
1AE6: 00         :Unknown  HEX   00
1AE7: 01         :Quadrnt  HEX   01
1AE8: FF 7F      :DeltaX   HEX   FF,7F
1AEA: 00 00      :DeltaY   HEX   00,00


                 * data for moving southeast

                 SouthEast
1AEC: 03         :Speed    HEX   03
1AED: 00         :Unknown  HEX   00
1AEE: 01         :Quadrnt  HEX   01
1AEF: 2B 5A      :DeltaX   HEX   2B,5A
1AF1: 2B 5A      :DeltaY   HEX   2B,5A
```

```
                * data for moving south

                South
1AF3: 03        :Speed    HEX    03
1AF4: 00        :Unknown  HEX    00
1AF5: 02        :Quadrnt  HEX    02
1AF6: 00 00     :DeltaX   HEX    00,00
1AF8: FF 7F     :DeltaY   HEX    FF,7F


                * data for moving southwest

                SouthWest
1AFA: 03        :Speed    HEX    03
1AFB: 00        :Unknown  HEX    00
1AFC: 02        :Quadrnt  HEX    02
1AFD: 2B 5A     :DeltaX   HEX    2B,5A
1AFF: 2B 5A     :DeltaY   HEX    2B,5A


                * data for moving west

                West
1B01: 04        :Speed    HEX    04
1B02: 00        :Unknown  HEX    00
1B03: 03        :Quadrnt  HEX    03
1B04: FF 7F     :DeltaX   HEX    FF,7F
1B06: 00 00     :DeltaY   HEX    00,00


                * data for moving northwest

                NorthWest
1B08: 03        :Speed    HEX    03
1B09: 00        :Unknown  HEX    00
1B0A: 03        :Quadrnt  HEX    03
1B0B: 2B 5A     :DeltaX   HEX    2B,5A
1B0D: 2B 5A     :DeltaY   HEX    2B,5A


                *------------- Miscellaneous Global Variables ------------*

1B0F: 00 00     RegKeyChk DS     2            ; the pre-detour KeyChk vector
1B11: 00 00     RegIIRQ   DS     2            ; the pre-detour IIrq vector
1B13: 00        MoveFlag  DFB    0            ; tells whether we're still or
                                              ; ... moving, under keyboard or
                                              ; ... joystick power
1B14: 00        ButnStat  DFB    0            ; holds state of pseudo-mouse
                                              ; ... button
1B15: 00        ClikHzLo  DS     1            ; pseudo-mouse horizontal
                                              ; ... position lo-byte at time
                                              ; ... of a click
1B16: 00        ClikHzHi  DS     1            ; pseudo-mouse horizontal
                                              ; ... position hi-bit at time
                                              ; ... of a click
1B17: 00        ClikVt    DS     1            ; pseudo-mouse vertical
                                              ; ... position at time of a
                                              ; ... click

--End assembly, 2072 bytes, Errors: 0
```

```
1000 REM ----- PROGRAM IDENTIFICATION
1010 :
1020 REM          S/M HELP PACKER
1030 :
1040 REM   CREATES THE FILE S/M HELP PACK
1050 :
1060 REM   S/M HELP PACK CONTAINS DATA FOR SOUND/MUSIC LAB'S
1070 REM   ... HELP SCREENS, READY TO BLOAD INTO POSITION
1080 REM   ... IN RAM BANK 1, 32768-64575  ( $8000-$FC3F )
1090 :
1100 REM   VERSION :    1.00
1110 REM   TIMESTAMP :  11:06 PM     NOVEMBER 23, 1986
1120 :
1130 REM   PROGRAMMED BY STAN KRUTE
1140 REM   COPYRIGHT (C) 1986 BY STAN KRUTE'S HACKER & NERD
1150 REM                          18617 CAMP CREEK ROAD
1160 REM                          HORNBROOK, CALIFORNIA   96044
1170 REM                          [916] 475-3428
1180 REM   ALL RIGHTS RESERVED
1190 REM   CALL OR WRITE FOR HELP, BUG REPORTS, LICENSING, ETC.
1200 :
1210 :
1220 REM ----- MAIN PROGRAM BLOCK -----
1230 :
1240 GOSUB 1310    :REM   PACK 'EM IN
1250 GOSUB 1570    :REM   SAVE IT ALL
1260 END
1270 :
1280 :
1290 REM ----- PACK 'EM IN -----
1300 :
1310 RESTORE 1400                               :REM  PREP FOR ADDRESS DATA
1320 :
1330 FOR N = 1 TO 22                            :REM  READ IN 22 HELP SCREENS
1340 :   N$ = STR$ (N)                          :REM    STRINGIZE THE NUMBER
1350 :   READ AD$ : AD = DEC ( AD$ )            :REM    GET ADDRESS & DECIMIZE
1360 :   BLOAD ("S/M HELP" + N$),U9,B1,P(AD)    :REM    LOAD NTH SCREEN THERE
1370 :   BANK 1 : POKE AD + 1016, 240 : BANK 15 :REM    ADD SPRITE DATA POINTR
1380 :   NEXT
1390 :
1400 DATA "8000", "8400", "8800", "8C00"        :REM  ADDRESSES TO LOAD THE
1410 DATA "A000", "A400", "A800", "AC00"        :REM  ... 22 HELP SCREENS
1420 DATA "B000", "B400", "B800"
1430 DATA "C000", "C400", "C800", "CC00"
1440 DATA "E000", "E400", "E800", "EC00"
1450 DATA "F000", "F400", "F800"
1460 :
1470 BLOAD "FINGER CURSOR", U9, B1, P48128      :REM  SPRITE TO QUADRANT 3
1480 BLOAD "FINGER CURSOR", U9, B1, P64512      :REM  SPRITE TO QUADRANT 4
1490 :
1500 RETURN
1510 :
1520 :
1530 REM ----- SAVE IT ALL -----
1540 :
1550 REM   THAT'S FROM $18000 THRU $1FC3F
1560 :
1570 PRINT "INSERT RECEIVING DISK. PRESS A KEY TO CONTINUE ... " ;  :REM  PROMPT
1580 GETKEY DM$                                         :REM  WAIT FOR KEYPRESS
1590 :
1600 BSAVE "@S/M HELP PACK", U9, B1, P32768 TO P64576   :REM  SAVE THE BLOCK
```

Fig. 16-3. Source code for S/M Help Packer.

```
1610 :
1620 PRINT "DS"DS"   DS$"DS$           :REM  PRINT DISK STATUS
1630 CATALOG U9                        :REM  PRINT A CATALOG
1640 :
1650 RETURN
```

```
1000 REM ----- PROGRAM IDENTIFICATION -----
1010 :
1020 REM            MAKE S/M VARS
1030 :
1040 REM   CREATES THE FILE S/M VARS
1050 :
1060 REM   S/M VARS IS A FILE OF VARIABLE INITIALIZERS
1070 REM   ... FOR THE PROGRAM SOUND/MUSIC LAB
1080 :
1090 REM   VERSION :     1.00
1100 REM   TIMESTAMP :  10:07 AM PST    SEPTEMBER 18, 1986
1110 :
1120 REM   PROGRAMMED BY STAN KRUTE
1130 REM   COPYRIGHT (C) 1986 BY STAN KRUTE'S HACKER & NERD
1140 REM                           18617 CAMP CREEK ROAD
1150 REM                           HORNBROOK, CALIFORNIA   96044
1160 REM                           [916] 475-3428
1170 REM   ALL RIGHTS RESERVED
1180 REM   CALL OR WRITE FOR HELP, BUG REPORTS, LICENSING, ETC.
1190 :
1200 :
1210 REM ----- MAIN PROGRAM BLOCK -----
1220 :
1230 GOSUB 1310       :REM  OPEN THE FILE
1240 GOSUB 1390       :REM  WRITE THE VALUES
1250 GOSUB 2820       :REM  CLOSE THE FILE
1260 END              :REM  BYE BYE
1270 :
1280 :
1290 REM ----- OPEN THE FILE -----
1300 :
1310 INPUT "DEVICE NUMBER "; DN$            :REM  GET DEVICE NUMBER
1320 DN = VAL ( DN$ )                       :REM  VALUIZE ENTERED STRING
1330 OPEN 12, (DN), 12, "@:S/M VARS,S,W"    :REM  OPEN A FRESH OUTPUT FILE
1340 RETURN
1350 :
1360 :
1370 REM ----- WRITE THE VALUES -----
1380 :
1390 ZR$ = "0000000000"  :REM  10 ZEROES
1400 PRINT# 12, ZR$      :REM  TO DISK
1410 :
1420 RESTORE 1500       :REM  PREP TO READ
1430 FOR N = 1 TO 7     :REM  SIZE OF FRAME DATA BLOCK, TITLE AREA #, AND
1440 :   READ DZ (N), TT (N), FT$ (N)  :REM  ... DESCRIPTIVE STRING FOR
1450 :   PRINT# 12, DZ (N)             :REM  ... SEVEN FRAME TYPES
1460 :   PRINT# 12, TT (N)      :REM  TO DISK
1470 :   PRINT# 12, FT$ (N)
1480 :   NEXT                          :REM  ... SEVEN FRAME TYPES
1490 :
1500 DATA 8,   1, "SOUND",       1,  10, "PLAY"
1510 DATA 7,  12, "ENVELOPE",    1,  83, "VOLUME"
1520 DATA 1,  85, "TEMPO",       5,  87, "FILTER"
1530 DATA 1,  93, "FRAME"
1540 :
```

Fig. 16-4. Source code for Make S/M Vars.

```
1550 RESTORE 1620       :REM   PREP TO READ
1560 DIM HA (13)        :REM   A LARGE ARRAY
1570 FOR N = 0 TO 13    :REM   SET ASSEMBLY LANGUAGE HELPER ADDRESSES
1580 :    READ HA (N)          :REM   READ DATA ELEMENT
1590 :    PRINT# 12, HA (N)    :REM   WRITE DATA ELEMENT TO DISK
1600 :    NEXT
1610 :
1620 DATA 4864, 4933, 6932, 5224   :REM  HELPER ADDRESS DATA ELEMENTS
1630 DATA   47,   22,   15,   25
1640 DATA 5394, 5318,   51,   25
1650 DATA 5679, 5459
1660 :
1670 RESTORE 1760       :REM   PREP TO READ
1680 DIM PF ( 21, 3 )   :REM   A LARGE ARRAY
1690 FOR N = 0 TO 21    :REM   SET PARAMETER FETCH ARRAY
1700 :    FOR P = 0 TO 3
1710 :       READ PF ( N, P )        :REM  READ DATA ELEMENT
1720 :       PRINT# 12, PF ( N, P )  :REM  WRITE DATA ELEMENT TO DISK
1730 :       NEXT
1740 :    NEXT
1750 :
1760 DATA  1,      3,  5, 1   :REM   PARAMETER FETCH DATA
1770 DATA  0, 65535,  7, 5   :REM   76 4-TUPLES OF THE FORM
1780 DATA  0, 32767, 13, 5   :REM   ... MINIMUM VALUE, MAXIMUM VALUE,
1790 DATA  0,      2, 19, 1   :REM   ... START COLUMN,  WIDTH
1800 DATA  0, 65535, 21, 5
1810 DATA  0, 32767, 27, 5   :REM   FIRST GROUP IS FOR SOUND PARAMETERS
1820 DATA  0,      3, 33, 1   :REM   ( 0..7 )
1830 DATA  0,  4095, 35, 4
1840 :
1850 DATA  0,     15,  3, 2   :REM   ENVELOPE PARAMETERS
1860 DATA  0,     15,  5, 2   :REM   ( 8..13 )
1870 DATA  0,     15,  7, 2
1880 DATA  0,     15,  9, 2
1890 DATA  0,      4, 11, 1
1900 DATA  0,  4095, 12, 4
1910 :
1920 DATA  0,     15, 18, 3   :REM   VOLUME   ( 14 )
1930 DATA  1,    255, 23, 3   :REM   TEMPO    ( 15 )
1940 :
1950 DATA  0,  2047, 28, 4   :REM   FILTER PARAMETERS
1960 DATA  0,      1, 33, 1   :REM   ( 16..20 )
1970 DATA  0,      1, 34, 1
1980 DATA  0,      1, 35, 1
1990 DATA  0,     15, 37, 2
2000 :
2010 DATA  1,  1000, 23, 4   :REM   FRAME NUMBER   ( 21 )
2020 :
2030 RESTORE 2090     :REM   PREP TO READ
2040 FOR N = 0 TO 5   :REM   ENVELOPE PARAMETER STRINGS
2050 :    READ EW$ (N)         :REM   READ VALUE
2060 :    PRINT# 12, EW$ (N)   :REM   TO DISK
2070 :    NEXT
2080 :
2090 DATA  "ATTACK",     "DECAY",    "SUSTAIN"
2100 DATA "RELEASE", "WAVEFORM", "PULS WDTH"
2110 :
2120 RESTORE 2190
2130 FOR N = 0 TO 7   :REM  SOUND PARAMETER STRINGS
2140 :    READ DS ( N ) , SW$ ( N ) :REM  READ VALUES
2150 :    PRINT# 12, DS ( N )   :REM  TO DISK
2160 :    PRINT# 12, SW$ ( N )
2170 :    NEXT
2180 :
2190 DATA 1, "VOICE",     1500, "FREQUENCY",    10, "DURATION",     0, "DIRECTION"
```

```
2200 DATA 0, "MINM FRQCY",   0, "SWEEPSTEP",   1, "WAVEFORM", 2048, "PULSEWIDTH"
2210 :
2220 RESTORE 2300          :REM  PREP TO READ
2230 FOR N = 0 TO 9        :REM  SET DEFAULT ENVELOPES
2240 :   FOR P = 0 TO 5    :REM  GET IT
2250 :      READ DE% ( N, P )       :REM  READ VALUE
2260 :      PRINT# 12, DE% ( N, P )  :REM  TO DISK
2270 :      NEXT
2280 :   NEXT
2290 :
2300 DATA  0,  9,  0,  0, 2, 1536   :REM   DEFAULT ENVELOPE 0
2310 DATA 12,  0, 12,  0, 1,    0   :REM   E1
2320 DATA  0,  0, 15,  0, 0,    0   :REM   E2
2330 DATA  0,  5,  5,  0, 3,    0   :REM   E3
2340 DATA  9,  4,  4,  0, 0,    0   :REM   E4
2350 DATA  0,  9,  2,  1, 1,    0   :REM   E5
2360 DATA  0,  9,  0,  0, 2,  512   :REM   E6
2370 DATA  0,  9,  9,  0, 2, 2048   :REM   E7
2380 DATA  8,  9,  4,  1, 2,  512   :REM   E8
2390 DATA  0,  9,  0,  0, 0,    0   :REM   E9
2400 :
2410 RESTORE 2470           :REM  PREP TO READ
2420 FOR N = 0 TO 4
2430 :   READ FW$ (N)       :REM  PICK UP A STRING
2440 :   PRINT# 12, FW$ (N) :REM  TO DISK
2450 :   NEXT
2460 :
2470 DATA "FREQUENCY", "LO-PASS", "BAND-PASS", "HI-PASS", "RESONANCE"
2480 :
2490 RESTORE 2570           :REM  PREP TO READ
2500 FOR N = 1 TO 5
2510 :   FOR P = 1 TO 3
2520 :      READ HP ( N, P )        :REM  GET A HELP SCREEN INVERSION PARAMETER
2530 :      PRINT# 12, HP ( N, P )  :REM  TO DISK
2540 :      NEXT
2550 :   NEXT
2560 :
2570 DATA  1, 23, 5,    9, 23, 8    :REM  FOR EACH OF 5 HELP SCREEN BUTTONS
2580 DATA 20, 23, 4,   27, 23, 4    :REM  ... THERE'S A LEFTMOST COLUMN, ROW,
2590 DATA 36, 23, 3                 :REM  ... AND WIDTH
2600 :
2610 RESTORE 2690                :REM  PREP TO READ
2620 DIM HK (22), HQ (22)        :REM  BIG ARRAYS
2630 FOR N = 1 TO 22             :REM  FOR 22 HELP SCREENS,
2640 :   READ HK (N), HQ (N)     :REM  ... GRAB K-BOUNDARY AND QUADRANT
2650 :   PRINT# 12, HK (N)       :REM  TO DISK
2660 :   PRINT# 12, HQ (N)
2670 :   NEXT
2680 :
2690 DATA  0, 1,   1, 1,   2, 1,   3, 1     :REM  K-BOUNDARY, QUADRANT
2700 DATA  8, 1,   9, 1,  10, 1,  11, 1
2710 DATA 12, 1,  13, 1,  14, 1
2720 DATA  0, 0,   1, 0,   2, 0,   3, 0
2730 DATA  8, 0,   9, 0,  10, 0,  11, 0
2740 DATA 12, 0,  13, 0,  14, 0
2750 :
2760 :
2770 RETURN
2780 :
2790 :
2800 REM ----- CLOSE THE FILE -----
2810 :
2820 PRINT# 12       :REM  BURP THE BUFFER
2830 CLOSE  12       :REM  CLOSE THE FILE
2840 RETURN
```

```
1000 REM ----- PROGRAM IDENTIFICATION -----
1010 :
1020 REM           MAKE 40C SCREENS
1030 :
1040 REM   MAKES 40-COLUMN SCREEN FILES
1050 :
1060 REM   DESIGNED TO BE USED ON A TWO-MONITOR  ( ONE 40-COLUMN,
1070 REM   ... ONE 80-COLUMN )  SYSTEM
1080 :
1090 REM   VERSION :      1.00
1100 REM   TIMESTAMP :    2:48 PM PST    SEPTEMBER 19, 1986
1110 :
1120 REM   PROGRAMMED BY STAN KRUTE
1130 REM   COPYRIGHT (C) 1986 BY STAN KRUTE'S HACKER & NERD
1140 REM                         18617 CAMP CREEK ROAD
1150 REM                         HORNBROOK, CALIFORNIA   96044
1160 REM                         [916] 475-3428
1170 REM   ALL RIGHTS RESERVED
1180 REM   CALL OR WRITE FOR HELP, BUG REPORTS, LICENSING, ETC.
1190 :
1200 :
1210 REM ----- MAIN PROGRAM BLOCK -----
1220 :
1230 GOSUB 1310    :REM  GET READY
1240 GOSUB 1640    :REM  RUN IT
1250 GOSUB 1750    :REM  CLEAN UP
1260 END           :REM  WE GONE
1270 :
1280 :
1290 REM ----- GET READY -----
1300 :
1310 SLOW                    :REM  MAKE SURE WE'RE VISIBLE
1320 :
1330 TRAP 3230               :REM  SET ERROR HANDLER
1340 GRAPHIC 5,1             :REM  80-COLUMN TEXT, CLEARED
1350 COLOR 0,1 : COLOR 4,1   :REM  BLACK BACKGROUND & BORDER
1360 :
1370 FINISHED = 0            :REM  NOT DONE YET
1380 CM$ = "ECSLPQ"          :REM  FOR PARSING COMMANDS
1390 :
1400 RN$ = CHR$ (18)         :REM  REVERSE ON
1410 RF$ = CHR$ (146)        :REM  REVERSE OFF
1420 :
1430 CR$ = CHR$ (13)         :REM  CARRIAGE RETURN
1440 UP$ = CHR$ (145)        :REM  CURSOR UP
1450 RT$ = CHR$ (29)         :REM  CURSOR RIGHT
1460 :
1470 MR$ = RT$ + RT$ + RT$ + RT$
1480 MR$ = MR$ + MR$ + MR$ + MR$   :REM  STRING OF 16 CURSOR RIGHTS
1490 :
1500 EE$ = CHR$ (27) + "@"   :REM  ERASES TO END OF SCREEN
1510 :
1520 DN = PEEK (186)              :REM  DEVICE WE LOADED FROM
1530 BLOAD "40C EDIT", U (DN)     :REM  THE EDITING ROUTINE
1540 BLOAD "TEXT DUMPS", U (DN) :REM  THE SCREEN PRINTING ROUTINE
1550 :
1560 R40 = 0 : C40 = 0          :REM  INIT 40-COLUMN CURSOR POSITION
1570 GOSUB 1820                 :REM  PRINT CHOICES
1580 :
1590 RETURN
1600 :
1610 :
1620 REM ----- RUN IT -----
1630 :
```

Fig. 16-5. Source code for Make 40C Screens.

```
1640 GETKEY KP$                                   :REM  GET KEYPRESS
1650 :
1660 CM = INSTR ( CM$, KP$ ) + 1                  :REM  FIGURE COMMAND CODE
1670 :
1680 ON CM GOSUB 1990, 2100, 2260, 2360, 2880, 3070, 3150  :REM  BRANCH ON CODE
1690 :
1700 IF FINISHED THEN RETURN : ELSE 1640    :REM  LEAVE OR DO MORE
1710 :
1720 :
1730 REM ----- CLEAN UP -----
1740 :
1750 GRAPHIC 5,0      :REM   BACK TO 80-COL SCREEN
1760 FAST             :REM   BACK UP TO SPEED
1770 RETURN
1780 :
1790 :
1800 REM ----- PRINT CHOICES -----
1810 :
1820 GRAPHIC 5, 0  :REM  80-COLUMN SCREEN
1830 :
1840 PRINT "MAKE 40C SCREENS" : PRINT
1850 PRINT ,, RN$ "E" RF$ "DIT THE SCREEN"   : PRINT
1860 PRINT ,, RN$ "C" RF$ "LEAR THE SCREEN"  : PRINT
1870 PRINT ,, RN$ "S" RF$ "AVE THE SCREEN"   : PRINT
1880 PRINT ,, RN$ "L" RF$ "OAD THE SCREEN"   : PRINT
1890 PRINT ,, RN$ "P" RF$ "RINT THE SCREEN"  : PRINT
1900 PRINT ,, RN$ "Q" RF$ "UIT THE PROGRAM"  : PRINT
1910 PRINT
1920 PRINT "YOUR CHOICE : " ;
1930 :
1940 RETURN
1950 :
1960 :
1970 REM ----- BAD CHOICE -----
1980 :
1990 PRINT "BAD CHOICE"     :REM  FEEDBACK
2000 SLEEP 1                :REM  PAUSE
2010 CHAR , 14, 15, EE$     :REM  ERASE FEEDBACK
2020 PRINT "PLEASE TRY ONE OF THE LIT-UP LETTERS"   :REM  FEEDBACK
2030 SLEEP 1                :REM  PAUSE
2040 CHAR , 14, 15, EE$     :REM  ERASE FEEDBACK
2050 RETURN
2060 :
2070 :
2080 REM ----- EDIT COMMAND -----
2090 :
2100 PRINT "EDIT THE SCREEN  --  PRESS SHIFT-RETURN TO END"   :REM  FEEDBACK
2110 :
2120 RW = PEEK (235) : CM = PEEK (236)       :REM  SAVE 80-COLUMN CURSOR
2130 POKE 235, R40 : POKE 236, C40           :REM  SET 40-COLUMN CURSOR
2140 :
2150 SYS 4864                                :REM  CALL THE EDITING ROUTINE
2160 :
2170 R40 = PEEK (235) : C40 = PEEK (236)     :REM  SAVE 40-COLUMN CURSOR
2180 POKE 235, RW : POKE 236, CM             :REM  RESTORE 80-COLUMN CURSOR
2190 :
2200 CHAR , 14, 15, EE$      :REM  ERASE FEEDBACK
2210 RETURN
2220 :
2230 :
2240 REM ----- CLEAR COMMAND -----
2250 :
2260 PRINT "CLEAR THE SCREEN"   :REM  FEEDBACK
2270 GRAPHIC 0,1                :REM  CLEAR 40-COLUMN TEXT
2280 GRAPHIC 5,0                :REM  BACK TO 80-COLUMN
```

```
2290 SLEEP 1                    :REM  PAUSE
2300 CHAR , 14, 15, EE$         :REM  CLEAR FEEDBACK
2310 RETURN
2320 :
2330 :
2340 REM ----- SAVE COMMAND -----
2350 :
2360 PRINT "SAVE THE SCREEN" : PRINT     :REM  FEEDBACK
2370 :
2380 GOSUB 2570                :REM  FETCH FILE NAME AND DEVICE NUMBER
2390 :
2400 IF FD = 0 THEN 2470       :REM  IF PROBLEMS, JUST LEAVE
2410 :
2420 BSAVE ("@" + NM$), U(DN), P1024 TO P2024  :REM  SAVE THAT SCREEN
2430 :
2440 CHAR , 14, 15, EE$        :REM  CLEAR FEEDBACK
2450 PRINT DS$                 :REM  PRINT DISK STATUS
2460 :
2470 SLEEP 1                   :REM  PAUSE
2480 CHAR , 14, 15, EE$        :REM  CLEAR FEEDBACK
2490 :
2500 RETURN
2510 :
2520 :
2530 REM ----- FETCH FILE NAME AND DEVICE NUMBER -----
2540 :
2550 REM   RETURNS A RESULT CODE IN FD :  -1 IF OK, 0 IF NOT
2560 :
2570 REM  GET A FILE NAME, DEFAULTING TO LAST NAME USED
2580 PRINT "NAME ? "  NM$  CR$  UP$  LEFT$ ( MR$, 5 ) ;
2590 INPUT NM$
2600 :
2610 IF NM$ <> "" THEN 2700    :REM  CONTINUE IF SOME NAME ENTERED
2620 :
2630 REM  DEAL WITH NO FILE NAME
2640 CHAR , 14, 15, EE$                   :REM  CLEAR FEEDBACK
2650 PRINT "NO NAME. FILE SAVE ABORTED"  :REM  NEW FEEDBACK
2660 FD = 0                               :REM  RESULT CODE NOT OKAY
2670 RETURN                               :REM  GET GONE
2680 :
2690 REM  GET A DEVICE NUMBER, DEFAULTING TO LAST DEVICE NUMBER USED
2700 PRINT "DEVICE NUMBER ?" STR$ ( DN )  CR$ UP$ LEFT$ ( MR$, 14) ;
2710 INPUT DN$
2720 :
2730 DN = VAL (DN$)                        :REM  VALUIZE THE DEVICE NUMBER
2740 IF DN >= 8 AND DN <= 11 THEN 2820     :REM  CONTINUE IF REASONABLE
2750 :
2760 REM  DEAL WITH LOUSY DEVICE NUMBER
2770 CHAR , 14, 15, EE$                    :REM  CLEAR FEEDBACK
2780 PRINT "BAD DEVICE NUMBER. FILE SAVE ABORTED"    :REM  NEW FEEDBACK
2790 FD = 0                                :REM  RESULT CODE NOT OKAY
2800 RETURN                                :REM  GET GONE
2810 :
2820 FD = -1           :REM  RESULT CODE OKAY
2830 RETURN            :REM  BYE BYE
2840 :
2850 :
2860 REM ----- LOAD COMMAND -----
2870 :
2880 PRINT "LOAD THE SCREEN" : PRINT        :REM  FEEDBACK
2890 :
2900 GOSUB 2570               :REM  FETCH FILE NAME AND DEVICE NUMBER
2910 :
2920 IF FD = 0 THEN 2990      :REM  IF PROBLEMS, JUST LEAVE
2930 :
```

```
2940 BLOAD (NM$), U(DN)        :REM  LOAD THAT SCREEN
2950 :
2960 CHAR , 14, 15, EE$        :REM  CLEAR FEEDBACK
2970 PRINT DS$                 :REM  PRINT DISK STATUS
2980 :
2990 SLEEP 1                   :REM  PAUSE
3000 CHAR , 14, 15, EE$        :REM  CLEAR FEEDBACK
3010 :
3020 RETURN
3030 :
3040 :
3050 REM ----- PRINT COMMAND -----
3060 :
3070 PRINT "PRINT THE 40-COLUMN SCREEN"  :REM  FEEDBACK
3080 SYS 2975                            :REM  CALL DUMP40 ROUTINE
3090 CHAR , 14, 15, EE$                  :REM  ERASE FEEDBACK
3100 RETURN
3110 :
3120 :
3130 REM ---- QUIT COMMAND -----
3140 :
3150 PRINT "QUIT THE PROGRAM" : PRINT
3160 PRINT "THANKS FOR THE WORKOUT"
3170 FINISHED = -1
3180 RETURN
3190 :
3200 :
3210 REM ----- ERROR HANDLER -----
3220 :
3230 CHAR , 14, 15, EE$                  :REM  CLEAR FEEDBACK
3240 PRINT ERR$ (ER)  " IN" STR$ (EL)    :REM  PRINT ERROR MESSAGE
3250 SLEEP 2                             :REM  PAUSE
3260 CHAR , 14, 15, EE$                  :REM  CLEAR FEEDBACK
3270 RESUME NEXT                         :REM  RESUME EXECUTION
```

```

*------------------- Program Identification --------------*
*                                                         *
*                        40C EDIT                         *
*                                                         *
* Lets you edit the standard 40-column text screen        *
*                                                         *
* Lives in RAM Bank 0 at addresses $1300-$1484            *
*                                                         *
* To call the routine                                     *
*                       SYS 4864                          *
*                                                         *
* To leave the routine                                    *
*                       press SHIFT/RETURN                *
*                                                         *
*                                                         *
* Version :     1.00                                      *
* Timestamp :   2:53 PM PST     September 19, 1986        *
*                                                         *
* Programmed by Stan Krute                                *
* Copyright (C) 1986 by Stan Krute's Hacker & Nerd        *
*                     18617 Camp Creek Road               *
*                     Hornbrook, California   96044       *
*                     [916] 475-3428                      *
* All rights reserved                                     *
* Call or write for help, bug reports, licensing, etc.    *
*                                                         *
```

Fig. 16-6. Source code for 40C Edit.

```
*-----------------------------------------------------------*


*--------------------- Constants ----------------------------*

* Commodore ASCII codes

CrsrDnCAC =    17          ; code for cursor down
CrsrLfCAC =    157         ; code for cursor left
CrsrRtCAC =    29          ; code for cursor right
CrsrUpCAC =    145         ; code for cursor up
DeleteCAC =    20          ; code for delete
InsertCAC =    148         ; code for insert
ShftRtrnCAC =  141         ; code for a shifted return
RvsOnCAC =     18          ; code for reverse on
RvsOffCAC =    146         ; code for reverse off


* editor variables

CurRow    =    $EB         ; current cursor row
CurCol    =    $EC         ; current cursor column
RvsFlag   =    $F3         ; reverse mode flag


* poke codes

SpacePC   =    32          ; code for a space


* ROM routines -- documented

GetIn     =    $FFE4       ; read buffered data from
                           ; ... current input device


* screen stuff

Top       =    0           ; topmost row
Bottom    =    24          ; bottommost row
Left      =    0           ; leftmost column
Right     =    39          ; rightmost column


* zero-page variables

OurPtr    =    $FA         ; a general-purpose pointer


*----------------- Set Program Origin ------------------*

          ORG  $1300       ; that's 4864 in decimal


*------------------- 40CEdit -------------------------*

* lets a user edit the 40-column text screen

* allows the user to type characters and use the following
* ... special keys : insert, delete, cursor keys,
* ... reverse on, and reverse off

* the user leaves the editor by pressing shift-return

* all registers are trashed
```

```
                40CEdit
                :LoopTop
                * this is the top of the editing loop
                * draw the editing cursor in its current position
1300: 20 0E 13          JSR   InvertCursor
                :LookKey
                * look for a keypress
1303: 20 E4 FF          JSR   GetIn       ; look for keyboard input
1306: F0 FB             BEQ   :LookKey    ; no keypress, so keep looking
                * we got a keypress, so go deal with it
1308: 20 1A 13          JSR   DealKey
                * if we come back with Carry set, do an exit
                * otherwise, back up to top of loop
130B: 90 F3             BCC   :LoopTop
                * return from 40CEdit
130D: 60                RTS
                *-------------------- InvertCursor ---------------------*
                * invert the cursor at its current location
                InvertCursor
                * set pointer to start of cursor row
130E: 20 B6 13          JSR   SetPtr
                * set index to cursor column
1311: A4 EC             LDY   CurCol      ; grab cursor column
                * grab poke code, invert it, store it back
1313: B1 FA             LDA   (OurPtr),Y ; grab poke code
1315: 49 80             EOR   #%10000000 ; flip-flop hi bit to invert
1317: 91 FA             STA   (OurPtr),Y ; store it back
                * return from InvertCursor
1319: 60                RTS
                *---------------------- DealKey ------------------------*
                * deal with a keypress
                * upon entry, A- holds the keypress' C-ASCII code
                * upon exit, all registers trashed
                *          cursor is erased
                *          Carry flag set   signals exit time
                *          Carry flag clear signals edit some more
                DealKey
                * park the character code
131A: 48                PHA
                * erase the cursor at its present position
```

```
131B: 20 0E 13          JSR   InvertCursor
                  * get the code back for testing
131E: 68                PLA
                  :TestOne
                  * is it in printable range 32..127 ?
131F: C9 20             CMP   #32
1321: 90 09             BCC   :TestTwo    ; jump if 0..31
1323: C9 80             CMP   #128
1325: B0 05             BCS   :TestTwo    ; jump if > 127
                  :Print
                  * yes, it's in printable range 32..127 or 160..255
                  * go print it
1327: 20 86 13          JSR   PrintChar
                  * signal for more editing and leave
132A: 18                CLC
132B: 60                RTS
                  :TestTwo
                  * is it in printable range 160..255 ?
132C: C9 A0             CMP   #160        ; check it out
132E: B0 F7             BCS   :Print      ; if so, print it
                  :CrsUpTest
                  * is it a cursor-up keypress ?
1330: C9 91             CMP   #CrsrUpCAC ; check it out
1332: D0 05             BNE   :CrsDnTest ; if not, next test
                  * yes, it's a cursor-up keypress
                  * adjust the cursor position upwards
1334: 20 16 14          JSR   CursUp
                  * signal for more editing and leave
1337: 18                CLC
1338: 60                RTS
                  :CrsDnTest
                  * is it a cursor-down keypress ?
1339: C9 11             CMP   #CrsrDnCAC ; check it out
133B: D0 05             BNE   :CrsLfTest ; if not, next test
                  * yes, it's a cursor-down keypress
                  * adjust the cursor position downwards
133D: 20 09 14          JSR   CursDwn
                  * signal for more editing and leave
1340: 18                CLC
1341: 60                RTS
                  :CrsLfTest
                  * is it a cursor-left keypress ?
1342: C9 9D             CMP   #CrsrLfCAC ; check it out
1344: D0 05             BNE   :CrsRtTest ; if not, next test
                  * yes, it's a cursor-left keypress
                  * adjust the cursor position leftwards
1346: 20 FB 13          JSR   CursLft
                  * signal for more editing and leave
1349: 18                CLC
134A: 60                RTS
```

```
                 :CrsRtTest
                 * is it a cursor-right keypress ?
134B: C9 1D              CMP   #CrsrRtCAC ; check it out
134D: D0 05              BNE   :InsrTest  ; if not, next test

                 * yes, it's a cursor-right keypress
                 * adjust the cursor position rightwards
134F: 20 EB 13           JSR   CursRit

                 * signal for more editing and leave
1352: 18                 CLC
1353: 60                 RTS

                 :InsrTest
                 * is it an insert keypress ?
1354: C9 94              CMP   #InsertCAC ; check it out
1356: D0 05              BNE   :DeleTest  ; if not, next test

                 * yes, it's an insert keypress
                 * go deal with it
1358: 20 9C 13           JSR   Insert

                 * signal for more editing and leave
135B: 18                 CLC
135C: 60                 RTS

                 :DeleTest
                 * is it a delete keypress ?
135D: C9 14              CMP   #DeleteCAC ; check it out
135F: D0 05              BNE   :RvsOnTest ; if not, next test

                 * yes, it's a delete keypress
                 * go deal with it
1361: 20 C3 13           JSR   Delete

                 * signal for more editing and leave
1364: 18                 CLC
1365: 60                 RTS

                 :RvsOnTest
                 * is it a reverse-on keypress ?
1366: C9 12              CMP   #RvsOnCAC  ; check it out
1368: D0 08              BNE   :RvsOffTest; if not, next test

                 * adjust the reverse flag
136A: A5 F3              LDA   RvsFlag    ; grab it
136C: 09 80              ORA   #%10000000 ; set hi bit
136E: 85 F3              STA   RvsFlag    ; store it back

                 * signal for more editing and leave
1370: 18                 CLC
1371: 60                 RTS

                 :RvsOffTest
                 * is it a reverse-off keypress ?
1372: C9 92              CMP   #RvsOffCAC ; check it out
1374: D0 08              BNE   :ExiTest   ; if not, next test

                 * adjust the reverse flag
1376: A5 F3              LDA   RvsFlag    ; grab it
1378: 29 7F              AND   #%01111111 ; clear hi bit
137A: 85 F3              STA   RvsFlag    ; store it back

                 * signal for more editing and leave
137C: 18                 CLC
```

```
137D: 60                       RTS

                       :ExiTest
                       * is it a shift-return combination
137E: C9 8D                    CMP   #ShftRtrnCAC; check it out
1380: D0 02                    BNE   :KPNoGood  ; if not, jump

                       * yes, it's a shift-return combination
                       * signal an exit and leave
1382: 38                       SEC
1383: 60                       RTS

                       :KPNoGood
                       * the keypress is not one we deal with

                       * signal for more editing and leave
1384: 18                       CLC
1385: 60                       RTS


                       *--------------------- PrintChar -----------------------*

                       * prints a character to the screen

                       * moves the cursor one position right, with wraparound

                       * upon entry, the character's C-ASCII code is in
                       * ... the A- register

                       * upon exit, all registers trashed

                       PrintChar

                       * fetch the character's poke code
1386: 20 21 14                 JSR   CAsc2Pok1

                       * if reverse flag is set, reverse the image
1389: A6 F3                    LDX   RvsFlag    ; non-zero says set
138B: F0 02                    BEQ   :StorPoke  ; jump if not set

                       * it's set, so reverse by setting hi bit
138D: 09 80                    ORA   #%10000000

                       :StorPoke
                       * store poke code on stack
138F: 48                       PHA

                       * set pointer to start of cursor row
1390: 20 B6 13                 JSR   SetPtr

                       * set index to cursor column
1393: A4 EC                    LDY   CurCol

                       * store that poke code
1395: 68                       PLA              ; back from stack
1396: 91 FA                    STA   (OurPtr),Y ; store it

                       * move cursor location to the right, dealing
                       * ... with any necessary wraparound issues
1398: 20 EB 13                 JSR   CursRit

                       * return from PrintChar
139B: 60                       RTS
```

```
                *---------------------- Insert -----------------------*

                * deal with a press of the insertion key

                * moves all characters from the cursor thru to the end
                * ... of its row one position to the right
                * ... ( the last character gets bumped into its next
                * ...   existence )

                * puts a space character at the cursor position

                * upon exit, all registers trashed

                Insert

                * set pointer to start of cursor row
139C: 20 B6 13          JSR   SetPtr

                * set index to next-to-rightmost column
139F: A0 26             LDY   #Right-1

                * enter loop at bottom test
13A1: D0 09             BNE   :LoopTest  ; always

                :LoopTop
                * move a character to the right
13A3: B1 FA             LDA   (OurPtr),Y ; grab it
13A5: C8                INY              ; change position
13A6: 91 FA             STA   (OurPtr),Y ; store it

                * adjust our index
13A8: 88                DEY              ; end up one to left
13A9: 88                DEY

                * take care of the leftmost case
13AA: 30 04             BMI   :StorSpace

                :LoopTest
                * see if we're done yet
13AC: C4 EC             CPY   CurCol     ; to left of cursor yet ?
13AE: B0 F3             BCS   :LoopTop   ; if not, back up

                :StorSpace
                * store a space at cursor position
13B0: A9 20             LDA   #SpacePC   ; code for a space
13B2: C8                INY              ; aim back at cursor column
13B3: 91 FA             STA   (OurPtr),Y ; store that space

                * return from Insert
13B5: 60                RTS


                *------------------ SetPtr --------------------------*

                * set OurPtr to start of cursor's row

                * trashes A- and X- registers

                SetPtr
13B6: A6 EB             LDX   CurRow     ; grab row
13B8: BD 53 14          LDA   RowsLo,X   ; index into table
13BB: 85 FA             STA   OurPtr     ; store lo-byte
13BD: BD 6C 14          LDA   RowsHi,X
13C0: 85 FB             STA   OurPtr+1   ; store hi-byte
```

```
13C2: 60                RTS                ; return from SetPtr
                *-------------------- Delete --------------------------*
                * deal with a press of the deletion key
                * moves all characters from the cursor position thru to
                * ... the end of the cursor's row one position to the left
                * ... ( the character to the left of the cursor gets
                * ...   bumped into its next existence )
                * puts a space character at the end of the row
                * moves cursor one spot to left
                * upon exit, all registers trashed
                Delete
                * set pointer to start of cursor row
13C3: 20 B6 13          JSR   SetPtr
                :SlideLeft
                * perform a deletion by sliding all the characters from
                * ... cursor position to last char one position to the left
                * initialize for the loop
13C6: A4 EC             LDY   CurCol      ; we'll start at the cursor
                * check for a leftmost cursor
13C8: D0 0D             BNE   :SLTop      ; jump if not in leftmost column
                * special case a leftmost cursor
13CA: 20 FB 13          JSR   CursLft     ; move cursor to left
13CD: 20 B6 13          JSR   SetPtr      ; set pointer to start of that ro
w
13D0: A4 EC             LDY   CurCol      ; index row's last char
13D2: A9 20             LDA   #SpacePC    ; get that character code
13D4: 91 FA             STA   (OurPtr),Y  ; store the space
13D6: 60                RTS               ; return from Delete
                :SLTop
                * the top of the sliding loop
                * slide an exit string character to the left
13D7: B1 FA             LDA   (OurPtr),Y  ; grab a char
13D9: 88                DEY               ; move one position left
13DA: 91 FA             STA   (OurPtr),Y  ; store a char
13DC: C8                INY               ; move back to target
13DD: C8                INY               ; move on to next target
                * bottom of the :SlideLeft loop
13DE: C0 28     :SLBotm CPY   #Right+1    ; are we done sliding ?
13E0: D0 F5             BNE   :SLTop      ; no, so back to loop top
                * okay, we've slid stuff over to make room, so now
                * ... we can add a spacey character to the row
13E2: 88                DEY               ; back to target
13E3: A9 20             LDA   #SpacePC    ; get that character code
13E5: 91 FA             STA   (OurPtr),Y  ; store the space
                * move cursor to the left, dealing with wraparound
13E7: 20 FB 13          JSR   CursLft
```

```
                :Bye
                * return from Delete
13EA: 60                RTS


                *----------------------- CursRit -----------------------*

                * move the cursor to the right, dealing with wraparound

                * upon exit, X- register is trashed

                CursRit

                * check current cursor column to see if
                * ... we'll need to deal with wraparound
13EB: A6 EC             LDX   CurCol      ; get current cursor column
13ED: E0 27             CPX   #Right      ; do we need to wraparound to the
                                          ; ... screen's leftmost column ?
13EF: D0 06             BNE   :RtUpHz     ; no, so life is easy

                * we need to wrap around horizontally
                * that means we also have to move down the screen
13F1: 20 09 14          JSR   CursDwn     ; get down

                * now let's horizontally wrap around to the
                * ... leftmost column
13F4: A2 00             LDX   #Left
13F6: CA                DEX               ; so we can slide thru
                                          ; ... the next instruction

                * move to the right by upping the horizontal coordinate
13F7: E8        :RtUpHz INX
13F8: 86 EC             STX   CurCol      ; store new cursor horizontal

                * return from CursRit
13FA: 60                RTS


                *----------------------- CursLft -----------------------*

                * move the cursor to the left, dealing with wraparound

                * upon exit, X- register is trashed

                CursLft

                * check current cursor column to see if
                * ... we'll need to deal with wraparound
13FB: A6 EC             LDX   CurCol      ; get current cursor column
13FD: D0 06             BNE   :LftDnHz    ; no wrap if non-zero

                * we need to wrap around horizontally
                * that means we also have to move up the screen,
                * ... which is done by downing the vertical coordinate
13FF: 20 16 14          JSR   CursUp      ; get up

                * now let's horizontally wrap around to the
                * ... rightmost column
1402: A2 27             LDX   #Right
1404: E8                INX               ; so we can slide thru
                                          ; ... the next instruction

                * move to the left by downing the column
1405: CA        :LftDnHz DEX
1406: 86 EC             STX   CurCol      ; store new cursor column
```

```
            * return from CursLft
1408: 60            RTS


            *-------------------- CursDwn --------------------------*

            * move the cursor down a line, dealing with wraparound

            * upon exit, X- register is trashed

            CursDwn

            * check current cursor row to see if we'll have to
            * ... deal with wraparound
1409: A6 EB         LDX   CurRow      ; get current cursor row
140B: E0 18         CPX   #Bottom     ; at bottom of rectangle ?
140D: D0 03         BNE   :CDUpIt     ; no, so no need to wrap

            * we need to vertically wrap up to the topmost row
140F: A2 00         LDX   #Top
1411: CA            DEX               ; so we can slide thru
                                      ; ... the next instruction

            :CDUpIt
            * move down the screen by upping the cursor row
1412: E8            INX               ; up, down, it's all so
                                      ; ... confusing, eh ?
1413: 86 EB         STX   CurRow      ; store new cursor row

            * return from CursDwn
1415: 60            RTS


            *------------------------ CursUp ----------------------*

            * move the cursor up a line, dealing with wraparound

            * upon exit, X- register trashed

            CursUp

            * check current cursor vertical to see if we'll have to
            * ... deal with wraparound
1416: A6 EB         LDX   CurRow      ; get current cursor vertical
1418: D0 03         BNE   :CUUpIt     ; no need to wrap if not at top

            * we need to vertically wrap to the bottommost row
141A: A2 18         LDX   #Bottom
141C: E8            INX               ; so we can slide thru
                                      ; ... the next instruction

            :CUUpIt
            * move up the screen by downing the vertical coordinate
141D: CA            DEX               ; up, down, it's all so
                                      ; ... confusing, eh ?
141E: 86 EB         STX   CurRow      ; store new cursor vertical

            * return from :CursUp
1420: 60            RTS


            *---------------------- CAsc2Pok1 ---------------------*

            * transform Commodore ASCII code to Set 1 screen poke code
```

```

            * obviously, this would be faster with a 256-byte table,
            * ... but we're a bit squeezed for space -- this code only
            * ... eats up 50 bytes, and ain't all THAT slow

            * upon entry, A- reg. holds a C-ASCII code [0..255]

            * upon exit,  A- reg. holds a poke code [0..255]

            * X- and Y- registers are preserved

            CAsc2Pok1

            * C-ASCIIs 0..31 transform to pocode 32
1421: C9 20  :Test1   CMP   #32        ; test for 0..31
1423: B0 03           BCS   :Test2     ; not in range, do next test

1425: A9 20           LDA   #32        ; in range, so return 32
1427: 60              RTS              ; outta here

            * C-ASCIIs 32..63 transform to pocodes 32..63
1428: C9 40  :Test2   CMP   #64        ; test for 32..63
142A: B0 01           BCS   :Test3     ; not in range, do next test

142C: 60              RTS              ; in range, so return as is

            * C-ASCIIs 64..95 transform to pocodes 0..31
142D: C9 60  :Test3   CMP   #96        ; test for 64..95
142F: B0 03           BCS   :Test4     ; not in range, do next test

1431: E9 3F           SBC   #63        ; in range, transform 64..95
                                       ; ... to 0..31 by subtacting 64
                                       ; ... ( a clear Carry lets us
                                       ; ...   skip a SEC step if we
                                       ; ...   just subtract 63 )
1433: 60              RTS              ; bye bye

            * C-ASCIIs 96..127 transform to pocodes 64..95
1434: C9 80  :Test4   CMP   #128       ; test for 96..127
1436: B0 03           BCS   :Test5     ; not in range, do next test

1438: E9 1F           SBC   #31        ; in range, transform 96..127 to
                                       ; ... 64..95 by subtacting 32
                                       ; ... ( a clear Carry lets us
                                       ; ...   skip a SEC step if we
                                       ; ...   just subtract 31 )
143A: 60              RTS              ; bye bye

            * C-ASCIIs 128..159 transform to pocode 32
143B: C9 A0  :Test5   CMP   #160       ; test for 128..159
143D: B0 03           BCS   :Test6     ; not in range, do next test

143F: A9 20           LDA   #32        ; in range, so return 32
1441: 60              RTS              ; bye bye

            * C-ASCIIs 160..191 transform to pocodes 96..127
1442: C9 C0  :Test6   CMP   #192       ; test for 160..191
1444: B0 03           BCS   :Test7     ; not in range, do next test

1446: E9 3F           SBC   #63        ; in range, transform 160..191
                                       ; ... to 96..127 by subtacting 64
                                       ; ... ( a clear Carry lets us
                                       ; ...   skip a SEC step if we
                                       ; ...   just subtract 63 )
1448: 60              RTS              ; bye bye
```

```
                 * C-ASCIIs 192..223 transform to pocodes 64..95
                 * C-ASCIIs 224..254 transform to pocodes 96..126
                 * ( notice that both ranges transform down by 128 )
1449: C9 FF      :Test7    CMP    #255        ; test for only remaining value
                                              ; ... that's not in one of these
                                              ; ... ranges
144B: F0 03                BEQ    :Final      ; got it, so go transform it

144D: E9 7F                SBC    #127        ; in range, transform 192..223
                                              ; ... to 64..95 and 224..255 to
                                              ; ... 96..126 by subtracting 128
                                              ; ... ( a clear Carry lets us
                                              ; ...   skip a SEC step if we
                                              ; ...   just subtract 127 )
144F: 60                   RTS                ; git gone

                 * C-ASCII 255 transforms to pocode 94
1450: A9 5E      :Final    LDA    #94         ; transform 255 to 94
1452: 60                   RTS                ; that's all she wrote


                 *----------------- screen address data ------------------*

                 * addresses of standard 40-column text row starts

                 RowsLo
1453: 00 28 50             HEX    00,28,50
1456: 78 A0 C8             HEX    78,A0,C8
1459: F0 18 40             HEX    F0,18,40
145C: 68 90 B8             HEX    68,90,B8
145F: E0 08 30             HEX    E0,08,30
1462: 58 80 A8             HEX    58,80,A8
1465: D0 F8 20             HEX    D0,F8,20
1468: 48 70 98             HEX    48,70,98
146B: C0                   HEX    C0

                 RowsHi
146C: 04 04 04             HEX    04,04,04
146F: 04 04 04             HEX    04,04,04
1472: 04 05 05             HEX    04,05,05
1475: 05 05 05             HEX    05,05,05
1478: 05 06 06             HEX    05,06,06
147B: 06 06 06             HEX    06,06,06
147E: 06 06 07             HEX    06,06,07
1481: 07 07 07             HEX    07,07,07
1484: 07                   HEX    07


--End assembly, 389 bytes, Errors: 0
```

SOUND/MUSIC LAB Variables — Sheet 1 Of 3

AD . . . . an area number
AN . . . . an area number
BL$ . . . string of blanks
CE% (9,5) . current envelope arrays
CF (5) . . . current filter array
CH . . . . current help screen number

Fig. 16-7. List of variables for Sound/Music Lab.

CM$ . . . current message
CN . . . . a column
CP . . . . cursor position
CP$ . . . . current play string
CS (7) . . . current sound array
CT . . . . current tempo
CV . . . . current volume
D (8) . . . data to go onto stack of frame data
DE% (9,5) . array of default envelope parameters
DN . . . . drive number the program was loaded from
DS (7) . . . array of default sound command parameters
DZ (7) . . size of frame data for 7 command types
EM$ . . . error message
EN . . . . envelope number
EV . . . . entry value for parameter fetching
EV$ . . . string version of EV
EW$ (6) . . array of envelope parameter strings
FD (MD) . frame data stack
FF . . . . former sound frame
FF% (MF) . for each frame, offset into frame data stack
FINISHED . boolean flags if we're ready to quit or not
FR . . . . current sound frame

SOUND/MUSIC LAB Variables Sheet 2 Of 3

FS$ (MS) . array of frame strings
FT$ . . . . array of frame data type name strings
FW$ (5) . . array of filter parameter label strings
H1 . . . . a color
H2 . . . . a color
HA (14) . . array of S/M LAB HELP.O addresses
HB . . . . high byte, for pointer work
HK (NH) . . array of help screen memory K-boundaries
HP (5,3) . . array of help screen button inversion parameters
HQ (NH) . . array of help screen memory quadrants
HU . . . . a color
KP$ . . . keypress
LB . . . . low byte, for pointer work
LB$ . . . a label
MA . . . . area returned by a mouse click
MD . . . . maximum selector for array of frame data FD ()
MF . . . . maximum number of frames, maximum selector for array of FD() pointers FF%()
MM . . . . area returned by a mouse click
MS . . . . maximum selector for array of frame strings FS$ ()
N . . . . . looping index
NH . . . . number of help screens
P . . . . . looping index
PF (21,3) . array of parameter fetch data
PH . . . . play window cursor horizontal position
PN . . . . parameter number for parameter fetching
PS . . . . parameter selector

PV . . . . play window cursor vertical position
PW . . . . parameter width

SOUND/MUSIC LAB Variables — Sheet 3 Of 3

RR . . . . boolean recording result
RW . . . . row for parameter fetching
SF . . . . start frame
SH . . . . string hot spot
SM . . . . a strawman swapping tool
SR . . . . starting row
SR% (9) . . array of parameters for StrngRectEdit
SW$ (7) . . array of sound parameter label strings
T . . . . . temporary real
T$ . . . . temporary string
T1 . . . . temporary real
T2 . . . . temporary real
T3 . . . . temporary real
T4 . . . . temporary real
TA . . . . target area
TD . . . . pointer to top of FD ()
TF . . . . highest recorded frame and pointer to top of FF% ()
TP . . . . temporary real
TP$ . . . . temporary string
TS . . . . pointer to top of FS$ ()
TT (7) . . . array of title area numbers for recordable commands
VC . . . . fetched parameter validity code
W . . . . temporary width of a string
WK$ . . . local working string
XV . . . . exit value from parameter fetching
XV$ . . . string version of XV
YY . . . . during development, flag for loading S/M LAB HELP.O
ZR$ . . . string of zeroes

```
1000 REM ----- PROGRAM IDENTIFICATION -----
1010 :
1020 REM          SOUND/MUSIC LAB
1030 :
1040 REM   LETS YOU PLAY WITH BASIC SOUND AND MUSIC COMMANDS
1050 :
1060 REM   VERSION :     1.00
1070 REM   TIMESTAMP :  11:33 PM PST    SEPTEMBER 17, 1986
1080 :
1090 REM   PROGRAMMED BY STAN KRUTE
1100 REM   COPYRIGHT (C) 1986 BY STAN KRUTE'S HACKER & NERD
1110 REM                         18617 CAMP CREEK ROAD
1120 REM                         HORNBROOK, CALIFORNIA   96044
1130 REM                         [916] 475-3428
1140 REM   ALL RIGHTS RESERVED
1150 REM   CALL OR WRITE FOR ASSISTANCE
1160 :
1170 :
1180 REM ----- MAIN PROGRAM BLOCK -----
```

Fig. 16-8. Source code for Sound/Music Lab.

```
1190 :
1200 GOSUB 1320   :REM  SET UP THE LAB
1210 :
1220 DO
1230 :   GOSUB 1480       :REM  LAB EVENT LOOP
1240 LOOP UNTIL FINISHED
1250 :
1260 GOSUB 1640   :REM  CLEAN UP THE LAB
1270 END          :REM  BYE BYE
1280 :
1290 :
1300 REM ----- SET UP THE LAB -----
1310 :
1320 FAST       :REM  MOVE RIGHT ALONG
1330 BANK 15    :REM  SYSTEM BANK
1340 TRAP 16240 :REM  SET ERROR HANDLER
1350 GOSUB 1760 :REM  CONFIGURE MEMORY
1360 GOSUB 1830 :REM  INITIALIZE SOME VARIABLES
1370 GOSUB 2620 :REM  RESET SOUND VARIABLES
1380 GOSUB 2930 :REM  DRAW A FRESH SCREEN
1390 GOSUB 3090 :REM  UPDATE THE SCREEN
1400 GOSUB 3200 :REM  LOAD AND INSTALL BINARY FILES
1410 GOSUB 3300 :REM  INITIALIZE CURSOR
1420 SLOW       :REM  BACK INTO SIGHT
1430 RETURN
1440 :
1450 :
1460 REM ----- LAB EVENT LOOP -----
1470 :
1480 IF  PEEK ( HA (2) ) = 0 THEN 1590    :REM  SCAN FOR P-M BUTTON PRESS
1490 :
1500 SYS HA (3), HA (4), HA (5) : RREG MA :REM  FIND WHERE IT'S PRESSED
1510 :
1520 IF MA = 0   THEN 1480  :REM  IF NOT IN A VALID AREA
1530 IF MA < 10  THEN 6040  :REM  SOUND CLICK
1540 IF MA < 12  THEN 6720  :REM  PLAY CLICK
1550 IF MA < 83  THEN 7320  :REM  ENVELOPE CLICK
1560 ON (MA - 82) GOTO 7950,7950,8370,8370,8790,8790,8790,8790,8790,8790,9310,
     9310,9720,10330,10510,11360,11580,12210,13270,13450,14100,1580,14770
1570 REM  BRANCHES FOR VOLUME, TEMPO, FILTER, FRAME, GO, FORWARD, LOAD,
        CLEAR, HELP, SHOW FRAME, BACKWARD, SAVE, PRINT, & END CLICKS
1580 :
1590 RETURN
1600 :
1610 :
1620 REM ----- CLEAN UP THE LAB -----
1630 :
1640 GRAPHIC 0, 1  :REM  40C TEXT, CLEAR
1650 SPRITE 1,0  :REM  FINGER CURSOR OFF
1660 SYS HA (1)  :REM  UNINSTALL ASM STUFF
1670 REM  MOVE BANK 1 TOP BACK UP
1680 POKE 53, 00 : POKE 54, 255
1690 POKE 57, 00 : POKE 58, 255
1700 RETURN
1710 :
1720 :
1730 REM ----- CONFIGURE MEMORY -----
1740 :
1750 REM  MOVE BANK 1 TOP DOWN FOR HELP SCREENS
1760 POKE 53, 00 : POKE 54, 128
1770 POKE 57, 00 : POKE 58, 128
1780 RETURN
1790 :
1800 :
1810 REM ----- INITIALIZE SOME VARIABLES -----
```

```
1820 :
1830 DN = PEEK (186)  :REM   DRIVE # WE ENTERED VIA
1840 OPEN 12, (DN), 12, "S/M VARS,S,R"  :REM  OPEN FILE
1850 :
1860 FINISHED = 0   :REM   FINISHED IS FALSE
1870 NH = 22 :REM  NUMBER OF HELP SCREENS
1880 CH = 1  :REM  CURRENT HELP SCREEN
1890 :
1900 LB = 1 : HB = 1  :REM  LO AND HI BYTES FOR POINTER WORK
1910 :
1920 BL$ = "                    "
1930 BL$ = BL$ + BL$ + BL$ + BL$
1940 BL$ = BL$ + BL$     :REM  160 BLANKS
1950 :
1960 INPUT# 12, ZR$ :REM  10 ZEROES
1970 :
1980 MD = 3000       :REM   MAXIMUM FD () SELECTOR
1990 DIM FD (MD)     :REM   STACK FOR SOUND FRAME DATA
2000 MF = 1000       :REM   MAXIMUM FF% () SELECTOR
2010 DIM FF% (MD)    :REM   ARRAY OF POINTERS INTO FD ()
2020 :               :REM   ... ONE FOR EACH FRAME
2030 MS = 100        :REM   MAXIMUM FS$ () SELECTOR
2040 DIM FS$ (MS)    :REM   ARRAY OF FRAME DATA STRINGS
2050 :
2060 FOR N = 1 TO 7    :REM  SIZE OF FRAME DATA BLOCK, TITLE AREA #, AND
2070 :   INPUT# 12, DZ (N), TT (N), FT$ (N)  :REM  ... DESCRIPTIVE STRING FOR
2080 :   NEXT          :REM  ... SEVEN FRAME TYPES
2090 :
2100 CM$ = "READY TO ROLL..."  :REM  INITIAL MESSAGE
2110 :
2120 DIM HA (13)       :REM   A LARGE ARRAY
2130 FOR N = 0 TO 13   :REM  SET ASM HELP ADDRESSES
2140 :   INPUT# 12, HA (N)
2150 :   NEXT
2160 :
2170 DIM PF ( 21, 3 )   :REM  A LARGE ARRAY
2180 FOR N = 0 TO 21    :REM  SET PARAMETER FETCH ARRAY
2190 :   FOR P = 0 TO 3
2200 :      INPUT# 12, PF ( N, P )
2210 :      NEXT
2220 :   NEXT
2230 :
2240 FOR N = 0 TO 5
2250 :   INPUT# 12, EW$ (N)  :REM  ENVELOPE PARAM. STRINGS
2260 :   NEXT
2270 :
2280 FOR N = 0 TO 7
2290 :   INPUT# 12, DS (N)   :REM  DEFAULT SOUND ARRAY
2300 :   INPUT# 12, SW$ (N)  :REM  SOUND PARAM. STRINGS
2310 :   NEXT
2320 :
2330 FOR N = 0 TO 9
2340 :   FOR P = 0 TO 5
2350 :      INPUT# 12, DE% (N,P)  :REM  DEFAULT ENVELOPES
2360 :      NEXT
2370 :   NEXT
2380 :
2390 FOR N = 0 TO 4
2400 :   INPUT# 12, FW$ (N)  :REM  FILTER PARAM. STRINGS
2410 :   NEXT
2420 :
2430 FOR N = 1 TO 5
2440 :   FOR P = 1 TO 3
2450 :      INPUT# 12, HP ( N, P )  :REM  HELP SCREEN INVERSION PARAMETERS
2460 :      NEXT
```

```
2470 :   NEXT
2480 :
2490 DIM HK (NH), HQ (NH)
2500 FOR N = 1 TO NH
2510 :   INPUT# 12, HK (N)   :REM  HELP SCREEN K-BOUNDARY & QUADRANT
2520 :   INPUT# 12, HQ (N)
2530 :   NEXT
2540 :
2550 PRINT# 12 : CLOSE 12  :REM  CLOSE FILE
2560 :
2570 RETURN
2580 :
2590 :
2600 REM ----- RESET SOUND VARIABLES -----
2610 :
2620 FOR N = 0 TO 7  :REM  SET SOUND ARRAY TO DEFAULT
2630 :   CS (N) = DS (N)
2640 :   NEXT
2650 :
2660 CP$ = ""     :REM   CURRENT PLAY STRING
2670 :
2680 FOR N = 0 TO 9     :REM  DEFAULT ENVELOPES
2690 :   FOR P = 0 TO 5 :REM  5 PARAMS
2700 :      CE% (N,P) = DE% (N,P)
2710 :      NEXT
2720 :   ENVELOPE N, CE% (N,0), CE% (N,1), CE% (N,2),
                     CE% (N,3), CE% (N,4), CE% (N,5)   :REM  SET IT
2730 :   NEXT
2740 :
2750 CV = 15 : VOL CV    :REM  SET CURRENT VOLUME
2760 CT = 8  : TEMPO CT  :REM  SET CURRENT TEMPO
2770 :
2780 FOR N = 0 TO 4  :REM  CURRENT FILTER ARRAY IS ZEROED
2790 :   CF(N) = 0
2800 :   NEXT
2810 FILTER CF (0), CF (1), CF (2), CF (3), CF (4)  :REM   SET IT
2820 :
2830 FR = 1  :REM  CURRENT SOUND FRAME
2840 TF = 0  :REM  HIGHEST RECORDED FRAME & TOP OF FF% ()
2850 TD = 0  :REM  POINTER TO TOP OF FD ()
2860 TS = 0  :REM  POINTER TO TOP OF FS$ ()
2870 :
2880 RETURN
2890 :
2900 :
2910 REM ----- DRAW A FRESH SCREEN -----
2920 :
2930 COLOR 4,1   :REM  BLACK BORDER
2940 COLOR 0,1   :REM  BLACK BACKGROUND
2950 GRAPHIC 0,0 :REM  40-COLUMN TEXT, NO CLEAR
2960 COLOR 5,6   :REM  GREEN TEXT FOR HELP
2970 GRAPHIC 0,1 :REM  40-COLUMN TEXT, CLEAR (TO GREEN)
2980 COLOR 1,1   :REM  BLACK BIT MAP PEN
2990 GRAPHIC 1,1 :REM  BIT MAP, CLEARED
3000 GOSUB 3380  :REM  DRAW SIX WINDOWS
3010 GOSUB 4330  :REM  DRAW FRAME COUNTER
3020 GOSUB 4400  :REM  DRAW NINE BUTTONS & A WINDOW
3030 GOSUB 4640  :REM  DRAW HELP BUTTON
3040 RETURN
3050 :
3060 :
3070 REM ----- UPDATE THE SCREEN -----
3080 :
3090 GOSUB 5100 :REM  UPDATE ENVELOPES WINDOW
3100 GOSUB 5180 :REM  UPDATE VOLUME WINDOW
```

```
3110 GOSUB 5280 :REM  UPDATE TEMPO WINDOW
3120 GOSUB 5380 :REM  UPDATE FILTER WINDOW
3130 GOSUB 5570 :REM  UPDATE FRAME COUNTER
3140 GOSUB 5670 :REM  UPDATE MESSAGE WINDOW
3150 RETURN
3160 :
3170 :
3180 REM ----- LOAD AND INSTALL BINARY FILES -----
3190 :
3200 BLOAD "FINGER CURSOR", B15, U(DN)  :REM   LAB SPRITE DATA
3210 BLOAD "S/M ASM 1",  B15, U(DN)     :REM   ASSEM. ROUTINES
3220 BLOAD "S/M ASM 2",  B15, U(DN)     :REM   ASSEM. ROUTINES
3230 BLOAD "S/M HELP PACK", B1, U(DN)   :REM   HELP SCREEN DATA
3240 SYS HA (0) :REM   INSTALL ASSEM. ROUTINES
3250 RETURN
3260 :
3270 :
3280 REM ----- INITIALIZE CURSOR -----
3290 :
3300 MOVSPR  1, 190, 96   :REM  MOVE IT INTO PLACE
3310 SPRITE  1, 1, 1, 0, 0, 0, 1  :REM  SET IT UP
3320 SPRCOLOR 2   :REM  MORE SET UP
3330 RETURN
3340 :
3350 :
3360 REM ----- DRAW SIX WINDOWS -----
3370 :
3380 RESTORE 3480       :REM  PREP TO READ
3390 FOR N = 1 TO 6  :REM  SIX TO DO
3400 :   READ T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, TP$  :REM  GET THAT DATA
3410 :   COLOR  1, 3              :REM  RED
3420 :   BOX   , T0, T1, T2, T3    :REM  BOX
3430 :   DRAW  , T4, T5 TO T6, T7  :REM  LINE
3440 :   COLOR  1, 15             :REM  LIGHT BLUE
3450 :   CHAR   , T8, T9, TP$      :REM  TITLE
3460 :   NEXT
3470 :
3480 DATA    3,   3, 315,  27,  35,  3,  35, 27,  1,  2, "SND"
3490 DATA    3,  35, 315,  75,  43, 35,  43, 75,  1,  7, "PLAY"
3500 DATA    3,  83, 131, 195,   3, 99, 131, 99,  4, 11, "ENVELOPES"
3510 DATA  139,  83, 171, 115, 139, 99, 171, 99, 18, 11, "VOL"
3520 DATA  219,  83, 315, 123, 219, 99, 315, 99, 30, 11, "FILTER"
3530 DATA  179,  83, 211, 115, 179, 99, 211, 99, 23, 11, "TMP"
3540 :
3550 FOR P = 1 TO 6
3560 :   REM  CUSTOMIZE WINDOWS
3570 :   ON P GOSUB 3650, 3820, 3910, 3600, 3600, 4170
3580 :   NEXT
3590 :
3600 RETURN
3610 :
3620 :
3630 REM ----- CUSTOMIZE SOUND WINDOW -----
3640 :
3650 RESTORE 3720    :REM  PREP TO READ DATA
3660 FOR N = 1 TO 8  :REM  DRAW HORIZONTAL LABELS
3670 :   READ HU,CN,LB$  :REM  GRAB LABEL'S SPECS
3680 :   COLOR 1,HU      :REM  SET PEN COLOR
3690 :   CHAR ,CN,1,LB$  :REM  DRAW LABEL
3700 :   NEXT
3710 :
3720 DATA  10,  5, "V",       12,  7, "FRQCY"   :REM  HORIZONTAL LABEL DATA
3730 DATA  10, 13, "DURTN",   12, 19, "D"       :REM  COLOR, COLUMN, STRING
3740 DATA  10, 21, "MINFR",   12, 27, "SWSTP"
3750 DATA  10, 33, "W",       12, 36, "PW"
```

```
3760 :
3770 RETURN
3780 :
3790 :
3800 REM ----- CUSTOMIZE PLAY WINDOW -----
3810 :
3820 COLOR 1, 15      :REM  LIGHT BLUE
3830 FOR N = 5 TO 8   :REM  CLEAR EACH ROW
3840 :   CHAR , 6, N, LEFT$ ( BL$, 33)
3850 :   NEXT N
3860 RETURN
3870 :
3880 :
3890 REM ----- CUSTOMIZE ENVELOPE WINDOW -----
3900 :
3910 RESTORE 3980     :REM  PREP TO READ
3920 FOR N = 1 TO 7   :REM  SEVEN HORIZONTAL LABELS
3930 :   READ HU,CN,LB$    :REM    GRAB LABEL SPECS
3940 :   COLOR 1,HU        :REM    SET COLOR
3950 :   CHAR ,CN,13,LB$   :REM    DRAW IT
3960 :   NEXT
3970 :
3980 DATA  10,  1, "#",    12,  3, "AT"   :REM  HORIZONTAL LABEL DATA
3990 DATA  10,  5, "DC",   12,  7, "SS"   :REM  COLOR, COLUMN, STRING
4000 DATA  10,  9, "RL",   12, 11, "W"
4010 DATA  10, 13, "PW"
4020 :
4030 READ H1, H2       :REM  READ VERT. LABEL COLORS
4040 FOR N = 0 TO 9    :REM  DRAW VERTICAL LABELS
4050 :   COLOR 1, H1   :REM  SET PEN COLOR
4060 :   CHAR , 1, 14+N, RIGHT$ ( STR$(N), 1 )   :REM  THE NUMBERS 0..9
4070 :   SM = H2 : H2 = H1 : H1 = SM     :REM  SWAP COLORS
4080 :   NEXT
4090 :
4100 DATA 12, 10  :REM   VERT. LABEL COLORS
4110 :
4120 RETURN
4130 :
4140 :
4150 REM ----- CUSTOMIZE FILTER WINDOW -----
4160 :
4170 RESTORE 4240              :REM  PREP FOR DATA READS
4180 FOR N = 1 TO 5            :REM  DRAW HORIZONTAL LABELS
4190 :   READ HU, CN, LB$      :REM    GRAB LABEL'S SPECS
4200 :   COLOR 1, HU           :REM    SET PEN COLOR
4210 :   CHAR , CN, 13, LB$    :REM    DRAW LABEL
4220 :   NEXT
4230 :
4240 DATA  10, 28, "FREQ",  12, 33, "L"    :REM  HORIZ. LABEL DATA
4250 DATA  10, 34, "B",     12, 35, "H"    :REM  COLOR, COLUMN, STRING
4260 DATA  10, 37, "RS"
4270 :
4280 RETURN
4290 :
4300 :
4310 REM ----- DRAW FRAME COUNTER -----
4320 :
4330 COLOR  1, 15             :REM   LIGHT BLUE
4340 CHAR  , 17, 15, "FRAME"  :REM   DRAW TITLE
4350 RETURN
4360 :
4370 :
4380 REM ----- DRAW NINE BUTTONS & A WINDOW -----
4390 :
4400 RESTORE 4500        :REM  PREP TO READ
```

```
4410 FOR N = 1 TO 10    :REM  TEN ITEMS TO DRAW
4420 :   READ T1, T2, T3, T4, T5, T6, TP$   :REM  GET BUTTON DATA
4430 :   COLOR  1, 3               :REM  RED
4440 :   BOX  , T1, T2, T3, T4     :REM  NICE BOX
4450 :   COLOR  1, 15              :REM  LIGHT BLUE
4460 :   CHAR  , T5, T6, TP$       :REM  NICE TITLE
4470 :   NEXT
4480 RETURN
4490 :
4500 DATA  139, 131, 171, 147, 18, 17, "GO"
4510 DATA  179, 131, 211, 147, 23, 17, "FWD"
4520 DATA  219, 131, 251, 147, 28, 17, "LOD"
4530 DATA  259, 131, 291, 147, 33, 17, "CLR"
4540 DATA  139, 155, 171, 171, 18, 20, "SHO"
4550 DATA  179, 155, 211, 171, 23, 20, "BKD"
4560 DATA  219, 155, 251, 171, 28, 20, "SAV"
4570 DATA  259, 155, 291, 171, 33, 20, "PRT"
4580 DATA  139, 179, 275, 195,  1,  1, ""     :REM  MESSAGE WINDOW
4590 DATA  283, 179, 315, 195, 36, 23, "END"
4600 :
4610 :
4620 REM ----- DRAW HELP BUTTON -----
4630 :
4640 COLOR 1,3             :REM  RED
4650 BOX , 299, 131, 315, 171   :REM  NICE BOX
4660 COLOR  1, 15          :REM  LIGHT BLUE
4670 CHAR  , 38, 17, "H"   :REM  TITLE
4680 CHAR  , 38, 18, "E"
4690 CHAR  , 38, 19, "L"
4700 CHAR  , 38, 20, "P"
4710 RETURN
4720 :
4730 :
4740 :
4750 REM ----- UPDATE SOUND WINDOW -----
4760 :
4770 RESTORE   4870           :REM   PREP FOR DATA READS
4780 FOR N =  0 TO 7          :REM   FOR 8 SOUND PARAMETERS
4790 :   TP$ = STR$ ( CS(N)) :REM      STRINGIZE A VALUE
4800 :   T = LEN (TP$)        :REM      GET LENGTH
4810 :   READ PW, HU, CN      :REM      GET WIDTH,COLOR,COLUMN
4820 :   TP$ = LEFT$ (ZR$, PW - T + 1) + RIGHT$ (TP$, T - 1)  :REM  PAD WITH 0'S
4830 :   COLOR 1, HU          :REM      SET PEN COLOR
4840 :   CHAR ,CN, 2, TP$     :REM      DRAW THAT VALUE
4850 :   NEXT
4860 :
4870 DATA  1, 7,  5,    5, 15,  7   :REM   SOUND WINDOW DATA
4880 DATA  5, 7, 13,    1, 15, 19   :REM   EIGHT 3-TUPLETS :
4890 DATA  5, 7, 21,    5, 15, 27   :REM   WIDTH, COLOR, COLUMN
4900 DATA  1, 7, 33,    4, 15, 35
4910 :
4920 RETURN
4930 :
4940 :
4950 REM ----- UPDATE PLAY WINDOW -----
4960 :
4970 COLOR 1,7               :REM  BLUE PEN
4980 GOSUB 3820              :REM  CUSTOMIZE PLAY WINDOW TO CLEAR DATA AREA
4990 IF CP$ = "" THEN 5050   :REM  DON'T DRAW NULL STRINGS
5000 :
5010 SR = 6 + ( LEN(CP$) > 99  )  :REM  FIGURE STARTING ROW
5020 FOR N = 0 TO 3               :REM  DRAW ROWS OF STRING
5030 :   CHAR , 6, SR + N,  MID$ (CP$, 1  +  33 * N, 33)
5040 :   NEXT
5050 RETURN
```

```
5060 :
5070 :
5080 REM ----- UPDATE ENVELOPES WINDOW -----
5090 :
5100 FOR EN = 0 TO 9    :REM  NINE ENVELOPES TO DO
5110 :   GOSUB 5750     :REM    UPDATE AN ENVELOPE
5120 :   NEXT
5130 RETURN
5140 :
5150 :
5160 REM ----- UPDATE VOLUME WINDOW -----
5170 :
5180 TP$ = STR$ (CV)        :REM   STRINGIZE CURRENT VOLUME
5190 T = LEN (TP$)          :REM   GET LENGTH
5200 TP$ = LEFT$ (ZR$, 4 - T ) + RIGHT$ (TP$, T - 1)  :REM  PAD WITH 0'S
5210 COLOR 1,7              :REM   BLUE PEN
5220 CHAR , 18, 13, TP$     :REM   DRAW THAT STRING
5230 RETURN
5240 :
5250 :
5260 REM ----- UPDATE TEMPO WINDOW -----
5270 :
5280 TP$ = STR$ (CT)        :REM   STRINGIZE CURRENT TEMPO
5290 T = LEN (TP$)          :REM   GET LENGTH
5300 TP$ = LEFT$ (ZR$, 4 - T ) + RIGHT$ (TP$, T - 1)  :REM  PAD WITH 0'S
5310 COLOR 1,7              :REM   BLUE PEN
5320 CHAR , 23, 13, TP$     :REM   DRAW THAT STRING
5330 RETURN
5340 :
5350 :
5360 REM ----- UPDATE FILTER WINDOW -----
5370 :
5380 RESTORE 5480               :REM   PREP FOR DATA READS
5390 FOR N = 0 TO 4             :REM   DEAL WITH ALL FILTER PARAMS
5400 :   TP$ = STR$ ( CF(N) )   :REM      STRINGIZE A PARAMETER
5410 :   T = LEN (TP$)          :REM      GET LENGTH
5420 :   READ PW, HU, CN        :REM      GET WIDTH,COLOR,COLUMN
5430 :   TP$ = LEFT$ (ZR$, PW - T + 1) + RIGHT$ (TP$, T - 1)  :REM  PAD WITH 0'S
5440 :   COLOR 1, HU            :REM      SET PEN COLOR
5450 :   CHAR  , CN, 14, TP$    :REM      PAINT THAT PARAM
5460 :   NEXT
5470 :
5480 DATA  4, 7, 28,    1, 15, 33   :REM   SUBROUTINE DATA
5490 DATA  1, 7, 34,    1, 15, 35   :REM   FIVE WIDTH, COLUMN,
5500 DATA  2, 7, 37                 :REM   ... COLOR TRIPLETS
5510 :
5520 RETURN
5530 :
5540 :
5550 REM ----- UPDATE FRAME COUNTER -----
5560 :
5570 TP$ = STR$ (FR)      :REM   STRINGIZE CURRENT SOUND FRAME
5580 T = LEN (TP$)        :REM   GET LENGTH
5590 TP$ = LEFT$ (ZR$, 5 - T) + RIGHT$ (TP$, T - 1)  :REM   PAD WITH 0'S
5600 COLOR 1,7            :REM   BLUE PEN
5610 CHAR , 23, 15, TP$   :REM   DRAW THAT STRING
5620 RETURN
5630 :
5640 :
5650 REM ----- UPDATE MESSAGE WINDOW -----
5660 :
5670 COLOR 1,6                           :REM   GREEN PEN
5680 CHAR   , 18, 23, LEFT$ ( BL$, 16 )  :REM   BLANK OUT AREA
5690 CHAR   , 18, 23, LEFT$ ( CM$, 16 )  :REM   PRINT CURRENT MESSAGE
5700 RETURN
```

```
5710 :
5720 :
5730 REM ----- UPDATE AN ENVELOPE -----
5740 :
5750 RESTORE 5950                    :REM   PREP TO READ DATA
5760 :
5770 READ H1, H2                     :REM   READ THE TWO COLORS
5780 IF INT ( EN / 2 ) <> EN / 2 THEN 5810    :REM   ALTERNATE ROWS SWITCH HUES
5790 SM = H1 : H1 = H2 : H2 = SM     :REM      SWITCH 'EM
5800 :
5810 FOR N = 0 TO 5 STEP 2           :REM   SET COLORS FOR THE SIX
5820 :   H(N) = H1                   :REM   ... PARAMETERS
5830 :   H(N+1) = H2
5840 :   NEXT
5850 :
5860 FOR N = 0 TO 5                  :REM   DRAW THE SIX PARAMETERS
5870 :   TP$ = STR$ ( CE% (EN, N) )  :REM      STRINGIZE A PARAMETER
5880 :   T = LEN (TP$)               :REM      STRIP SPACE
5890 :   READ PW, CN                 :REM      READ WIDTH AND COLUMN
5900 :   TP$ = LEFT$ (ZR$, PW - T + 1) + RIGHT$ (TP$, T - 1)  :REM  PAD WITH 0'S
5910 :   COLOR 1,H(N)                :REM      SET PEN COLOR
5920 :   CHAR ,CN, 14 + EN, TP$      :REM      DRAW IT
5930 :   NEXT
5940 :
5950 DATA 15,  7                     :REM   SUBROUTINE DATA
5960 DATA  2,  3,   2,  5,   2,  7   :REM   TWO PEN COLORS
5970 DATA  2,  9,   1, 11,   4, 12   :REM   SIX WIDTH,COLUMN PAIRS
5980 :
5990 RETURN
6000 :
6010 :
6020 REM ----- SOUND CLICK -----
6030 :
6040 AN = 1 : GOSUB 6660   :REM   INVERT TITLE
6050 GOSUB 4770            :REM   DRAW CURRENT SOUND ARRAY
6060 TA = MA               :REM   SET TARGET AREA
6070 IF TA = 1 THEN TA = 2 :REM   MOVE FROM TITLE
6080 :
6090 CM$ = "SOUND " + SW$ ( TA - 2 ) : GOSUB 5670   :REM   FEEDBACK
6100 GOSUB 15690      :REM   TYPE ONE SOUND
6110 :
6120 RW = 2           :REM   SET PARAM. FETCH ROW
6130 PN = TA - 2      :REM   SET PARAM. FETCH SELECTOR
6140 EV = CS ( PN )   :REM   SET PARAM. FETCH ENTRY VALUE
6150 AD = 0           :REM   SET PARAM. FETCH AREA ID
6160 :
6170 SYS HA (8), PF (PN, 2), 1, PF (PN, 3)  :REM   INVERT TARGET LABEL
6180 GOSUB 15020            :REM   FETCH A PARAMETER
6190 SYS HA (8), PF (PN, 2), 1, PF (PN, 3)  :REM   NORMALIZE TARGET LABEL
6200 :
6210 REM   JUMP IF OUT OF WINDOW, UNLESS IT'S THE GO BUTTON
6220 IF ( MM < 1 OR MM > 9 ) AND MM <> 95 THEN MA = MM : GOTO 6520
6230 :
6240 IF VC = 0 THEN 6090  :REM   BACK UP IF INVALID PARAMETER
6250 :
6260 IF XV <> EV THEN CS ( TA - 2 ) = XV  :REM   STORE VALID PARAMETER
6270 :
6280 IF MM <> 1 THEN 6410      :REM   JUMP IF NOT IN TITLE
6290 CM$ = "RECRD SOUND CMND" : GOSUB 5670  :REM   FEEDBACK
6300 AN = 94 : GOSUB 6660      :REM      INVERT COUNTER
6310 GOSUB 6600                :REM      PLAY CURRENT SOUND
6320 GOSUB 15830               :REM      LET SOUND FINISH
6330 D (0) = 1                 :REM      SET UP SOUND FRAME DATA
6340 FOR N = 1 TO 8            :REM      FIRST THE TYPE, THEN THE VALUES
6350 :  D (N) = CS (N - 1)
```

```
6360 :  NEXT
6370 GOSUB 15460     :REM      RECORD A SOUND FRAME
6380 IF NOT RR THEN MA = 0 : GOTO 6520   :REM  LEAVE IF COULDN'T RECORD
6390 GOTO 6090       :REM      BACK UP FOR MORE
6400 :
6410 IF MM <2 OR MM >9 THEN 6450  :REM   JUMP IF NOT IN SOUND AREA ...
6420 TA = MM         :REM      MOVE TO THAT AREA
6430 GOTO 6090       :REM      FETCH ANOTHER PARAM.
6440 :
6450 CM$ = "TEST SOUND CMND" : GOSUB 5670   :REM   FEEDBACK
6460 AN = 95 : GOSUB 6660     :REM      INVERT GO BUTTON
6470 GOSUB 6600               :REM      PLAY CURRENT SOUND
6480 GOSUB 15830              :REM      LET SOUND FINISH
6490 AN = 95 : GOSUB 6660     :REM      NORMALIZE GO BUTTON
6500 GOTO 6090                :REM      BACK UP FOR MORE
6510 :
6520 CM$ = "" : GOSUB 5670    :REM   CLEAR FEEDBACK
6530 CHAR  , 5, 2, LEFT$ ( BL$, 34 )  :REM   CLEAR DATA AREA
6540 AN = 1 : GOSUB 6660      :REM   NORMALIZE "SND"
6550 GOTO 1520                :REM   BACK TO LAB EVENT LOOP
6560 :
6570 :
6580 REM ----- PLAY CURRENT SOUND -----
6590 :
6600 SOUND CS(0), CS(1), CS(2), CS(3), CS(4), CS(5), CS(6), CS(7)
6610 RETURN
6620 :
6630 :
6640 REM ----- INVERT AN AREA -----
6650 :
6660 SYS HA (9), HA (10), HA (11), AN  :REM   INVERT AN AREA
6670 RETURN
6680 :
6690 :
6700 REM ----- PLAY CLICK -----
6710 :
6720 AN = 10 : GOSUB 6660     :REM   INVERT TITLE
6730 CP = 0                   :REM   INIT EDITING CURSOR
6740 GOSUB 15690              :REM   TYPE ONE SOUND
6750 CM$ = "EDIT PLAY STRING" : GOSUB 5670 :REM   FEEDBACK
6760 :
6770 IF LEN (CP$) < 132 THEN CP$ = CP$ + LEFT$ ( BL$, 132 - LEN (CP$) )
6780 :                        :REM   PAD CP$ TO SIZE
6790 :                        :REM   SET UP FOR STRNGRECTEDIT
6800 SR% (0) = POINTER (CP$) :REM  ENTRY STRING
6810 SR% (1) = POINTER (CP$) :REM  EXIT STRING
6820 SR% (2) = 5              :REM  TOP
6830 SR% (3) = 8              :REM  BOTTOM
6840 SR% (4) = 6              :REM  LEFT
6850 SR% (5) = 38             :REM  RIGHT
6860 SR% (6) = 11             :REM  AREA ID
6870 SR% (7) = HA (12)        :REM  AREA DATA TABLE
6880 SR% (8) = CP             :REM  EDITING CURSOR POSITION
6890 N = POINTER ( SR% ( 0 ) )  :REM  ADDRESS OF ARRAY
6900 HB = INT ( N / 256 )     :REM  ADDRESS HI-BYTE
6910 LB = N - ( HB * 256 )    :REM  ADDRESS LO-BYTE
6920 SYS 1024, LB, HB, 1      :REM  CALL STRNGRECTEDIT
6930 RREG MM, CP              :REM  GET P-M EXIT AREA ID & CURSOR POSITION
6940 :
6950 IF MM <> 95 THEN 7030    :REM   IF NOT GO BUTTON, JUMP
6960 :
6970 AN = 95 : GOSUB 6660     :REM   INVERT GO TITLE
6980 CM$ = "TEST PLAY STRING" : GOSUB 5670 :REM   FEEDBACK
6990 PLAY CP$                 :REM   PLAY IT
7000 AN = 95 : GOSUB 6660     :REM   NORMALIZE GO TITLE
```

```
7010 GOTO 6750                   :REM   BACK FOR MORE
7020 :
7030 IF MM <> 10 THEN 7220      :REM   IF NOT PLAY TITLE, BYE
7040 :
7050 IF TS <= MS THEN 7100    :REM   JUMP IF ROOM FOR STRING
7060 :
7070 EM$ = "NO STRING SPACE" : GOSUB 16080   :REM   FEEDBACK
7080 MA = 0 : GOTO 7240       :REM   AND LEAVE
7090 :
7100 CM$ = "RECRD PLAY STRNG" : GOSUB 5670   :REM   FEEDBACK
7110 AN = 94 : GOSUB 6660    :REM   INVERT COUNTER
7120 PLAY CP$                 :REM   PLAY EDITED STRING
7130 :
7140 D (0) = 2        :REM   FRAME DATA IS PLAY TYPE
7150 D (1) = TS       :REM   DATA INDEXES PLAY STRING
7160 FS$ (TS) = CP$   :REM   STORE PLAY STRING
7170 GOSUB 15460      :REM   RECORD A PLAY FRAME
7180 IF NOT RR THEN MA = 0 : GOTO 7240   :REM  LEAVE IF COULDN'T RECORD
7190 TS = TS + 1      :REM   UP POINTER TO STRING ARRAY TOP
7200 GOTO 6750        :REM   BACK FOR MORE
7210 :
7220 MA = MM                  :REM   SET MOUSE AREA FOR RETURN
7230 :
7240 CM$ = "" : GOSUB 5670    :REM   CLEAR MESSAGE WINDOW
7250 GOSUB 3820               :REM   CLEAR PLAY DATA AREA
7260 AN = 10 : GOSUB 6660     :REM   NORMALIZE PLAY TITLE
7270 GOTO 1520                :REM   BACK TO LAB EVENT LOOP
7280 :
7290 :
7300 REM --- ENVELOPE CLICK -----
7310 :
7320 AN = 12 : GOSUB 6660      :REM   INVERT TITLE
7330 :
7340 TA = MA                   :REM   SET TARGET AREA
7350 IF TA = 12 THEN TA = 14 :REM   MOVE FROM TITLE
7360 :
7370 EN = INT ( (TA - 13) / 7 )     :REM   FIGURE ENVELOPE #
7380 IF (TA - 13) / 7  =  EN  THEN TA = TA + 1    :REM   MOVE FROM ENVELOPE #
7390 PS = TA - 14 - (EN * 7)        :REM   FIGURE PARAMETER SELECTOR
7400 :
7410 REM   FEEDBACK
7420 CM$ = "ENV #" + RIGHT$ ( STR$(EN), 1 ) + " " + EW$ (PS) : GOSUB 5670
7430 GOSUB 15690   :REM   TYPE ONE SOUND
7440 :
7450 RW = EN + 14         :REM  SET PARAM. FETCH ROW
7460 PN = PS + 8          :REM  SET PARAM. FETCH SELECTOR
7470 EV = CE% ( EN, PS) :REM  SET PARAM. FETCH ENTRY VALUE
7480 AD = 0               :REM  SET PARAM. FETCH AREA ID
7490 :
7500 SYS HA (8), 1, EN + 14, 1       :REM   INVERT ENVELOPE #
7510 SYS HA (8), PF (PS + 8, 2), 13, PF (PS + 8, 3)  :REM   INVERT PARAM. LABEL
7520 GOSUB 15020   :REM   FETCH A PARAMETER
7530 SYS HA (8), PF (PS + 8, 2), 13, PF (PS + 8, 3)  :REM NORMALIZE PARAM. LABEL
7540 SYS HA (8), 1, EN + 14, 1       :REM   NORMALIZE ENVELOPE #
7550 :
7560 IF MM < 12 OR MM > 82 THEN MA = MM : GOTO 7770 :REM  LEAVE IF OUT OF WINDOW
7570 :
7580 IF VC = 0 THEN 7420   :REM   BACK UP IF INVALID PARAMETER
7590 :
7600 IF XV <> EV THEN CE% (EN, PS) = XV : GOSUB 7850  :REM   STORE & SET
7610 :
7620 IF MM <> 12 THEN TA = MM : GOTO 7370  :REM   JUMP IF NOT IN TITLE
7630 :
7640 CM$ = "RECRD ENV" + STR$ (EN) + " CMND" : GOSUB 5670   :REM FEEDBACK
7650 GOSUB 15690              :REM   TYPE ONE SOUND
```

```
7660 AN = 94 : GOSUB 6660   :REM   INVERT COUNTER
7670 :
7680 D (0) = 3     :REM   DATA TYPE IS ENVELOPE
7690 D (1) = EN    :REM   FILL IN DATA FRAME VALUES
7700 FOR N = 2 TO 7
7710 :   D (N) = CE% ( EN, N - 2 )
7720 :   NEXT
7730 GOSUB 15460  :REM   RECORD A SOUND FRAME
7740 IF NOT RR THEN MA = 0 : GOTO 7770   :REM   LEAVE IF COULDN'T RECORD
7750 GOTO 7420     :REM   BACK UP FOR MORE
7760 :
7770 CM$ = "" : GOSUB 5670  :REM   CLEAR FEEDBACK
7780 GOSUB 5750             :REM   REDRAW LAST ENVELOPE
7790 AN = 12 : GOSUB 6660   :REM   NORMALIZE TITLE
7800 GOTO 1520              :REM   BACK TO LAB EVENT LOOP
7810 :
7820 :
7830 REM ----- SET CURRENT ENVELOPE -----
7840 :
7850 CM$ = "NEW ENVELOPE" + STR$ (EN) : GOSUB 5670  :REM   FEEDBACK
7860 GOSUB 15690   :REM   TYPE ONE SOUND
7870 GOSUB 16330   :REM   PAUSE
7880 CM$ = "" : GOSUB 5670     :REM   CLEAR FEEDBACK
7890 ENVELOPE EN, CE% (EN, 0), CE% (EN, 1), CE% (EN, 2),
                  CE% (EN, 3), CE% (EN, 4), CE% (EN, 5)   :REM   SET IT
7900 RETURN
7910 :
7920 :
7930 REM ----- VOLUME CLICK -----
7940 :
7950 AN = 83 : GOSUB 6660  :REM  INVERT TITLE
7960 :
7970 CM$ = "SET VOLUME LEVEL" : GOSUB 5670 :REM  FEEDBACK
7980 GOSUB 15690           :REM  TYPE ONE SOUND
7990 :
8000 RW = 13    :REM   SET PARAM. FETCH ROW
8010 PN = 14    :REM   SET PARAM. FETCH SELECTOR
8020 EV = CV    :REM   SET PARAM. FETCH ENTRY VALUE
8030 AD = 0     :REM   SET PARAM. FETCH AREA ID
8040 :
8050 GOSUB 15020  :REM   FETCH A PARAMETER
8060 :
8070 IF MM < 83 OR MM > 84 THEN MA = MM : GOTO 8290 :REM  LEAVE IF OUT OF WINDOW
8080 :
8090 IF VC = 0 THEN 7970    :REM   BACK UP IF INVALID PARAMETER
8100 :
8110 IF XV = EV THEN 8170   :REM   JUMP IF NO CHANGE TO VOLUME
8120 :
8130 CV = XV : VOL CV       :REM   SET NEW VOLUME
8140 CM$ = "VOLUME SET TO" + STR$(CV) : GOSUB 5670   :REM   FEEDBACK
8150 GOSUB 15690            :REM   TYPE ONE SOUND
8160 :
8170 IF MM = 84 THEN 7970   :REM   JUMP IF NOT IN TITLE
8180 :
8190 CM$ = "RECORD VOLUME" + STR$ (CV) : GOSUB 5670  :REM   FEEDBACK
8200 GOSUB 15690            :REM   TYPE ONE SOUND
8210 AN = 94 : GOSUB 6660   :REM   INVERT COUNTER
8220 :
8230 D(0) = 4     :REM   DATA TYPE IS VOLUME
8240 D(1) = CV    :REM   SEND THE NEW VOLUME
8250 GOSUB 15460 :REM   RECORD THE FRAME.
8260 IF RR THEN 7970  :REM   IF RECORDED OK, BACK FOR MORE
8270 MA = 0       :REM   IF COULDN'T RECORD, LEAVE
8280 :
8290 CM$ = "" : GOSUB 5670  :REM   CLEAR FEEDBACK
```

```
8300 GOSUB 5180               :REM   UPDATE VOLUME WINDOW
8310 AN = 83 : GOSUB 6660     :REM   NORMALIZE TITLE
8320 GOTO 1520                :REM   BACK TO LAB EVENT LOOP
8330 :
8340 :
8350 REM ----- TEMPO CLICK -----
8360 :
8370 AN = 85 : GOSUB 6660     :REM   INVERT TITLE
8380 :
8390 CM$ = "SET PLAY TEMPO" : GOSUB 5670  :REM   FEEDBACK
8400 GOSUB 15690              :REM   TYPE ONE SOUND
8410 :
8420 RW = 13    :REM   SET PARAM. FETCH ROW
8430 PN = 15    :REM   SET PARAM. FETCH SELECTOR
8440 EV = CT    :REM   SET PARAM. FETCH ENTRY VALUE
8450 AD = 0     :REM   SET PARAM. FETCH AREA ID
8460 :
8470 GOSUB 15020  :REM   FETCH A PARAMETER
8480 :
8490 IF MM < 85 OR MM > 86 THEN MA = MM : GOTO 8710 :REM  LEAVE IF OUT OF WINDOW
8500 :
8510 IF VC = 0 THEN 8390   :REM   BACK UP IF INVALID PARAMETER
8520 :
8530 IF XV = EV THEN 8590  :REM   JUMP IF NO CHANGE TO TEMPO
8540 :
8550 CT = XV : TEMPO CT    :REM   SET NEW TEMPO
8560 CM$ = "TEMPO SET TO" + STR$(CT) : GOSUB 5670   :REM   FEEDBACK
8570 GOSUB 15690           :REM   TYPE ONE SOUND
8580 :
8590 IF MM = 86 THEN 8390  :REM  JUMP IF NOT IN TITLE
8600 :
8610 CM$ = "RECORD TEMPO" + STR$ (CT) : GOSUB 5670  :REM   FEEDBACK
8620 GOSUB 15690           :REM   TYPE ONE SOUND
8630 AN = 94 : GOSUB 6660  :REM   INVERT COUNTER
8640 :
8650 D(0) = 5     :REM   DATA TYPE IS TEMPO
8660 D(1) = CT    :REM   SEND THE NEW TEMPO
8670 GOSUB 15460 :REM   RECORD THE FRAME
8680 IF RR THEN 8390  :REM   IF RECORDED OK, BACK FOR MORE
8690 MA = 0       :REM   IF COULDN'T RECORD, LEAVE
8700 :
8710 CM$ = "" : GOSUB 5670  :REM   CLEAR FEEDBACK
8720 GOSUB 5280             :REM   UPDATE TEMPO WINDOW
8730 AN = 85 : GOSUB 6660   :REM   NORMALIZE TITLE
8740 GOTO 1520              :REM   BACK TO LAB EVENT LOOP
8750 :
8760 :
8770 REM ----- FILTER CLICK -----
8780 :
8790 AN = 87 : GOSUB 6660    :REM   INVERT TITLE
8800 TA = MA                 :REM   SET TARGET AREA
8810 IF TA = 87 THEN TA = 88 :REM   MOVE FROM TITLE
8820 :
8830 CM$ = "FILTER " + FW$ ( TA - 88 ) : GOSUB 5670  :REM   FEEDBACK
8840 GOSUB 15690      :REM   TYPE ONE SOUND
8850 :
8860 RW = 14          :REM   SET PARAM. FETCH ROW
8870 PN = TA - 72     :REM   SET PARAM. FETCH SELECTOR
8880 EV = CF ( TA - 88 )  :REM   SET PARAM. FETCH ENTRY VALUE
8890 AD = 0           :REM   SET PARAM. FETCH AREA ID
8900 :
8910 SYS HA(8), PF (PN, 2), 13,  PF (PN, 3)  :REM   INVERT TARGET LABEL
8920 GOSUB 15020           :REM   FETCH A PARAMETER
8930 SYS HA(8), PF (PN, 2), 13,  PF (PN, 3)  :REM   NORMALIZE TARGET LABEL
8940 :
```

```
8950 IF MM < 87 OR MM > 92 THEN MA = MM : GOTO 9130 :REM  JUMP IF OUT OF WINDOW
8960 :
8970 IF VC = 0 THEN 8830       :REM   BACK UP IF INVALID PARAMETER
8980 :
8990 IF XV <> EV THEN CF ( TA - 88 ) = XV : GOSUB 9210  :REM  STORE & SET
9000 :
9010 IF MM <> 87 THEN  TA = MM : GOTO 8830    :REM   JUMP IF NOT IN TITLE
9020 CM$ = "RECRD FILTR CMND" : GOSUB 5670  :REM   FEEDBACK
9030 GOSUB 15690               :REM      TYPE ONE SOUND
9040 AN = 94 : GOSUB 6660      :REM      INVERT COUNTER
9050 D (0) = 6                 :REM   FRAME DATA TYPE IS FILTER
9060 FOR N = 1 TO 5            :REM   FIVE VALUES
9070 :  D (N) = CF (N - 1)
9080 :  NEXT
9090 GOSUB 15460             :REM  RECORD A SOUND FRAME
9100 IF RR THEN GOTO 8830    :REM  IF RECORDED OK, BACK UP FOR MORE
9110 MA = 0                  :REM  IF COULDN'T RECORD, LEAVE
9120 :
9130 GOSUB 5380              :REM  DRAW CURRENT FILTER
9140 CM$ = "" : GOSUB 5670   :REM  CLEAR FEEDBACK
9150 AN = 87 : GOSUB 6660    :REM  NORMALIZE TITLE
9160 GOTO 1520               :REM  BACK TO LAB EVENT LOOP
9170 :
9180 :
9190 REM ----- SET CURRENT FILTER -----
9200 :
9210 CM$ = "NEW FILTER SET" : GOSUB 5670  :REM   FEEDBACK
9220 GOSUB 15690   :REM   TYPE ONE SOUND
9230 GOSUB 16330   :REM   PAUSE
9240 CM$ = "" : GOSUB 5670   :REM   CLEAR FEEDBACK
9250 FILTER CF(0), CF(1), CF(2), CF(3), CF(4) :REM   SET IT
9260 RETURN
9270 :
9280 :
9290 REM ----- FRAME CLICK -----
9300 :
9310 AN = 93 : GOSUB 6660    :REM   INVERT TITLE
9320 :
9330 CM$ = "FRAME COUNTER" : GOSUB 5670 :REM   FEEDBACK
9340 GOSUB 15690              :REM   TYPE ONE SOUND
9350 :
9360 RW = 15    :REM   SET PARAM. FETCH ROW
9370 PN = 21    :REM   SET PARAM. FETCH SELECTOR
9380 EV = FR    :REM   SET PARAM. FETCH ENTRY VALUE
9390 AD = 0     :REM   SET PARAM. FETCH AREA ID
9400 :
9410 GOSUB 15020  :REM   FETCH A PARAMETER
9420 :
9430 IF MM < 93 OR MM > 94 THEN MA = MM : GOTO 9640 :REM LEAVE IF OUTTA WINDOW
9440 :
9450 IF VC = 0  THEN 9330   :REM   BACK UP IF INVALID PARAMETER
9460 :
9470 FF = FR                 :REM   SAVE FORMER FRAME
9480 IF XV = EV THEN 9330   :REM   JUMP IF NO CHANGE TO FRAME
9490 :
9500 FR = XV : GOSUB 16000   :REM   SET NEW FRAME
9510 :
9520 IF MM = 94 THEN 9330   :REM   BACK UP IF NOT IN TITLE
9530 :
9540 CM$ = "RCRD FRAME CHNGE" : GOSUB 5670   :REM   FEEDBACK
9550 FR = FF :               :REM   RECORD IT AT FORMER FRAME
9560 AN = 94 : GOSUB 6660    :REM   INVERT COUNTER
9570 :
9580 D (0) = 7      :REM   DATA TYPE IS FRAME
9590 D (1) = XV     :REM   DATA VALUE IS NEW FRAME
```

```
9600 GOSUB 15460   :REM   RECORD THE FRAME
9610 IF RR THEN 9330        :REM   IF RECORDED OK, BACK UP FOR MORE
9620 MA = 0         :REM   OTHERWISE, LEAVE
9630 :
9640 CM$ = "" : GOSUB 5670  :REM   CLEAR FEEDBACK
9650 GOSUB 5570             :REM   UPDATE FRAME COUNTER
9660 AN = 93 : GOSUB 6660   :REM   NORMALIZE TITLE
9670 GOTO 1520              :REM   BACK TO LAB EVENT LOOP
9680 :
9690 :
9700 REM ----- GO CLICK -----
9710 :
9720 AN = 95 : GOSUB 6660   :REM   INVERT TITLE
9730 CM$ = "GO BUTTON" : GOSUB 5670   :REM   FEEDBACK
9740 GOSUB 15690   :REM   TYPE ONE SOUND
9750 SF = FR       :REM   SAVE START FRAME
9760 :
9770 IF TF < FR THEN 10160  :REM   LEAVE IF FRAME NOT RECORDED
9780 :
9790 IF PEEK ( HA (2) ) = 0 THEN 9850   :REM   CONTINUE IF NO P-M CLICK
9800 :
9810 SYS HA (3), HA (4), HA (5) :REM   SEE WHERE CLICK WAS
9820 RREG MM
9830 IF MM <> 95 THEN 10210     :REM   LEAVE IF NOT IN GO BUTTON
9840 :
9850 T1 = FF% ( FR )  :REM  GET FRAME DATA OFFSET
9860 T2 = FD ( T1 )   :REM  GET FRAME'S TYPE
9870 CM$ = "#" + STR$ ( FR ) + ": " + FT$ (T2) : GOSUB 5670  :REM   SHOW 'EM
9880 GOSUB 5570       :REM  UPDATE COUNTER
9890 :
9900 REM   BRANCH TO CARRY OUT COMMAND TYPES
9910 ON T2 GOTO 9930, 9960, 9990, 10020, 10050, 10080, 10110
9920 :
9930 SOUND FD (T1+1), FD (T1+2), FD (T1+3), FD (T1+4), FD (T1+5), FD (T1+6),
          FD (T1+7)    :REM   SOUND A FRAME
9940 GOTO 10130        :REM   & CONTINUE
9950 :
9960 PLAY FS$ ( FD ( T1+1 ) )  :REM   PLAY A FRAME
9970 GOTO 10130        :REM   & CONTINUE
9980 :
9990 ENVELOPE FD (T1+1), FD (T1+2), FD (T1+3), FD (T1+4), FD (T1+5), FD (T1+6),
             FD (T1+7) :REM   ENVELOPE A FRAME
10000 GOTO 10130       :REM   & CONTINUE
10010 :
10020 VOL FD (T1 + 1)   :REM   VOLUME A FRAME
10030 GOTO 10130        :REM   & CONTINUE
10040 :
10050 TEMPO FD (T1 + 1) :REM   TEMPO A FRAME
10060 GOTO 10130        :REM   & CONTINUE
10070 :
10080 FILTER FD (T1+1), FD (T1+2), FD (T1+3), FD (T1+4), FD (T1+5)
               :REM   FILTER A FRAME
10090 GOTO 10130   :REM   & CONTINUE
10100 :
10110 FR = FD ( T1 + 1 ) :REM   FRAME A FRAME
10120 :
10130 IF T2 <> 7 THEN FR = FR + 1   :REM   INCREMENT FRAME COUNTER
10140 GOTO 9770                     :REM   BACK UP FOR MORE
10150 :
10160 CM$ = "LAST RCRDED FRAM" : GOSUB 5670  :REM   FEEDBACK
10170 GOSUB 15690                  :REM   TYPE ONE SOUND
10180 MA = 0      :REM   LEAVE CLEAN
10190 GOTO 10250  :REM   BYE
10200 :
10210 CM$ = "USER SEZ STOP" : GOSUB 5670   :REM   FEEDBACK
```

```
10220 GOSUB 15690    :REM   TYPE ONE SOUND
10230 MA = MM        :REM   ASSIGN MOUSE AREA
10240 :
10250 CM$ = "" : GOSUB 5670   :REM   CLEAR FEEDBACK
10260 FR = SF : GOSUB 5570    :REM   RESTORE FRAME COUNTER
10270 AN = 95 : GOSUB 6660    :REM   NORMALIZE TITLE
10280 GOTO 1520               :REM   BACK TO LAB EVENT LOOP
10290 :
10300 :
10310 REM ----- FORWARD CLICK -----
10320 :
10330 AN = 96 : GOSUB 6660    :REM   INVERT TITLE
10340 CM$ = "FORWARD BUTTON" : GOSUB 5670  :REM   FEEDBACK
10350 GOSUB 15690        :REM   TYPE ONE SOUND
10360 :
10370 DO                 :REM   FORWARD 'TIL NO BUTTON
10380 :   FR = FR + 1    :REM      INCREMENT FRAME COUNTER
10390 :   IF FR > MF THEN FR = 1  :REM   WRAPAROUND
10400 :   GOSUB 5570     :REM      UPDATE FRAME COUNTER
10410 LOOP WHILE PEEK ( HA (2) )
10420 :
10430 GOSUB 15890              :REM   SHOW THAT FRAME
10440 AN = 96 : GOSUB 6660     :REM   NORMALIZE TITLE
10450 CM$ = "" : GOSUB 5670    :REM   CLEAR FEEDBACK
10460 GOTO 1480                :REM   BACK TO EVENT LOOP
10470 :
10480 :
10490 REM ----- LOAD CLICK -----
10500 :
10510 AN = 97 : GOSUB 6660    :REM   INVERT TITLE
10520 :
10530 CM$ = "LOAD BUTTON" : GOSUB 5670  :REM   FEEDBACK
10540 GOSUB 15690             :REM   TYPE ONE SOUND
10550 :
10560 GOSUB 11100    :REM  GET A FILE NAME
10570 :
10580 IF MM <> 97 THEN MA = MM : GOTO 11030   :REM  LEAVE ON OUTSIDE CLICK
10590 :
10600 IF TP$ = "" THEN 11010    :REM  LEAVE ON NULL NAME
10610 :
10620 CM$ = "OPENING FILE ..." : GOSUB 5670   :REM  FEEDBACK
10630 OPEN 12, (DN), 12, TP$ + ",S,R"      :REM  OPEN FILE
10640 IF DS THEN 10960           :REM  JUMP IF PROBLEMS
10650 :
10660 GOSUB 2620    :REM   CLEAR SOUND VARIABLES
10670 :
10680 CM$ = "LOADING DATA ..." : GOSUB 5670 :REM  FEEDBACK
10690 :
10700 INPUT# 12, TD   :REM  GRAB VITALS
10710 INPUT# 12, TF
10720 INPUT# 12, TS
10730 INPUT# 12, FR
10740 :
10750 IF TD = 0 THEN 10800  :REM  SKIP IF NONE
10760 FOR N = 0 TO TD-1     :REM  GRAB FRAME DATA
10770 :   INPUT# 12, FD (N)
10780 :   NEXT
10790 :
10800 IF TF = 0 THEN 10850  :REM  SKIP IF NONE
10810 FOR N = 1 TO TF       :REM  GRAB FRAME DATA OFFSETS
10820 :   INPUT# 12, FF% (N)
10830 :   NEXT
10840 :
10850 IF TS = 0 THEN 10900  :REM  SKIP IF NONE
10860 FOR N = 0 TO TS-1     :REM  GRAB FRAME STRINGS
```

```
10870 :   INPUT# 12, FS$ (N)
10880 :   NEXT
10890 :
10900 IF DS THEN 10960        :REM  JUMP IF PROBLEMS
10910 FAST
10920 CM$ = "LOADED & READY" :REM  FEEDBACK
10930 GOSUB 3090 : SLOW      :REM  REDRAW THE SCREEN
10940 SLEEP 2 : GOTO 11000   :REM  PAUSE, THEN LEAVE
10950 :
10960 EM$ = "DISK PROBLEMS !!" : GOSUB 16080   :REM  FEEDBACK
10970 EM$ = DS$ : GOSUB 16080 :REM  FEEDBACK
10980 GOSUB 11360  :REM  CLEAR & REDRAW
10990 :
11000 CLOSE 12     :REM   CLOSE FILE
11010 MA = 0       :REM   NOWHERE MOUSE
11020 :
11030 CM$ = "" : GOSUB 5670  :REM   CLEAR FEEDBACK
11040 AN = 97 : GOSUB 6660   :REM   NORMALIZE TITLE
11050 GOTO 1520              :REM   BACK TO LAB EVENT LOOP
11060 :
11070 :
11080 REM ----- GET A FILE NAME -----
11090 :
11100 CM$ = "ENTER FILE NAME:" : GOSUB 5670  :REM   FEEDBACK
11110 GOSUB 15690   :REM   TYPE ONE SOUND
11120 GOSUB 16330   :REM   PAUSE
11130 :
11140 TP$ = LEFT$ ( BL$, 16 ) :REM   INIT FILE NAME
11150 :                       :REM   SET UP FOR STRNGRECTEDIT
11160 SR% (0) = POINTER (TP$) :REM   ENTRY STRING
11170 SR% (1) = POINTER (TP$) :REM   EXIT STRING
11180 SR% (2) = 23            :REM   TOP
11190 SR% (3) = 23            :REM   BOTTOM
11200 SR% (4) = 18            :REM   LEFT
11210 SR% (5) = 33            :REM   RIGHT
11220 SR% (6) = 104           :REM   AREA ID
11230 SR% (7) = HA (12)       :REM   AREA DATA TABLE
11240 SR% (8) = 0             :REM   EDITING CURSOR POSITION
11250 N = POINTER ( SR% ( 0 ) )  :REM  ADDRESS OF ARRAY
11260 HB = INT ( N / 256 )     :REM  ADDRESS HI-BYTE
11270 LB = N - ( HB * 256 )    :REM  ADDRESS LO-BYTE
11280 SYS 1024, LB, HB, 1      :REM  CALL STRNGRECTEDIT
11290 RREG MM       :REM   GET P-M EXIT AREA ID
11300 GOSUB 14000  :REM  STRIP TP$ TRAILING BLANKS
11310 RETURN
11320 :
11330 :
11340 REM ----- CLEAR CLICK -----
11350 :
11360 AN = 98 : GOSUB 6660   :REM   INVERT TITLE
11370 :
11380 CM$ = "CLEARING ..." : GOSUB 5670  :REM   FEEDBACK
11390 GOSUB 15690   :REM  TYPE ONE SOUND
11400 GOSUB 16330   :REM  PAUSE
11410 :
11420 FAST    :REM  MOVE IT
11430 GOSUB 2620   :REM   RESET SOUND VARIABLES
11440 CM$ = "   ... ALL CLEAR"   :REM   FOR UPDATING
11450 GOSUB 3090   :REM   UPDATE THE SCREEN
11460 SLOW    :REM  BACK INTO VIEW
11470 :
11480 GOSUB 15690   :REM   TYPE ONE SOUND
11490 GOSUB 16330   :REM   PAUSE
11500 :
11510 CM$ = "" : GOSUB 5670  :REM   CLEAR FEEDBACK
```

```
11520 AN = 98 : GOSUB 6660   :REM  NORMALIZE TITLE
11530 GOTO 1480              :REM  BACK TO LAB EVENT LOOP
11540 :
11550 :
11560 REM ----- HELP CLICK -----
11570 :
11580 AN = 99 : GOSUB 6660   :REM  INVERT TITLE
11590 GOSUB 15690            :REM  TYPE 1 SOUND
11600 CM$ = "" : GOSUB 5670  :REM  CLEAR FEEDBACK
11610 :
11620 T2 = PEEK (2604)  :REM  SAVE SCREEN & CHAR SETUP
11630 :
11640 FAST    :REM  HIDE
11650 AN = 99 : GOSUB 6660   :REM  NORMALIZE TITLE
11660 POKE 216, 0   :REM  TEXT
11670 MOVSPR  1, 330, 243  :REM  ADJUST SPRITE
11680 POKE 54534, PEEK (54534) OR 64  :REM  VIC BANK
11690 POKE 56576, PEEK (56576) AND 252 OR HQ (CH)  :REM  VIC QUADRANT
11700 POKE 2604, PEEK ( 2604 ) AND 15 OR ( HK (CH) * 16 )  :REM  VIC K-BOUNDARY
11710 SLOW    :REM  APPEAR
11720 :
11730 IF  PEEK ( HA (2) ) = 0 THEN 11730    :REM  SCAN FOR P-M BUTTON PRESS
11740 :
11750 SYS HA (3), HA (6), HA (7) : RREG MM  :REM  FIND WHERE PRESSED
11760 :
11770 IF MM = 0 THEN 11730  :REM  BACK IF NOWHERE
11780 :
11790 SYS HA (13), HP ( MM, 1 ), HP ( MM, 2 ), HP ( MM, 3 ) :REM INVERT BUTTON
11800 :
11810 REM  BUTTON BRANCH: FIRST, PREVIOUS, NEXT, LAST, QUIT
11820 ON MM GOTO 11840, 11870, 11910, 11950, 12060
11830 :
11840 CH = 1        :REM  1ST HELP SCREEN
11850 GOTO 11970    :REM  MERGE
11860 :
11870 CH = CH - 1   :REM  PREVIOUS HELP SCREEN
11880 IF CH = 0 THEN CH = NH    :REM  WRAPAROUND
11890 GOTO 11970    :REM  MERGE
11900 :
11910 CH = CH + 1   :REM  NEXT HELP SCREEN
11920 IF CH > NH THEN CH = 1   :REM  WRAPAROUND
11930 GOTO 11970    :REM  MERGE
11940 :
11950 CH = NH       :REM  LAST HELP SCREEN
11960 :
11970 GOSUB 15690   :REM  TYPE 1 SOUND
11980 :
11990 SYS HA (13), HP ( MM, 1 ), HP ( MM, 2 ), HP ( MM, 3 )  :REM  NORMAL BUTTON
12000 :
12010 POKE 2604, PEEK ( 2604 ) AND 15 OR ( HK (CH) * 16 )  :REM VIC K-BOUNDARY
12020 POKE 56576, PEEK ( 56576 ) AND 252 OR HQ (CH)  :REM VIC QUADRANT
12030 :
12040 GOTO 11730   :REM  BACK UP TO SCAN
12050 :
12060 GOSUB 15690   :REM  TYPE 1 SOUND
12070 FAST          :REM  HIDE
12080 SYS HA (13), HP ( MM, 1 ), HP ( MM, 2 ), HP ( MM, 3 )  :REM  NORMAL BUTTON
12090 POKE 54534, PEEK (54534) AND 191  :REM  VIC SEES RAM 0
12100 POKE 56576, PEEK ( 56576 ) AND 252 OR 3  :REM  QUADRANT 0
12110 POKE 2604, T2 :REM  RESTORE TEXT SCREEN BASE
12120 POKE 216, 32  :REM  BITMAP
12130 MOVSPR  1, 330, 218  :REM  ADJUST SPRITE
12140 SLOW        :REM  APPEAR
12150 :
12160 GOTO 1480   :REM  BACK TO LAB EVENT LOOP
```

```
12170 :
12180 :
12190 REM ----- SHOW FRAME CLICK -----
12200 :
12210 AN = 100 : GOSUB 6660  :REM  INVERT TITLE
12220 AN = 94  : GOSUB 6660  :REM  INVERT COUNTER
12230 :
12240 IF TF > 0 THEN 12280   :REM  CONTINUE IF FRAMES TO SHOW
12250 EM$ = "NO FRAMES 2 SHOW" : GOSUB 16080   :REM  FEEDBACK
12260 MA = 0 : GOTO 13190    :REM  LEAVE TO NOWHERE
12270 :
12280 IF FR <= TF THEN 12350 :REM  JUMP IF CURRENT FRAME RECORDED
12290 :
12300 FR = TF  :REM  DEFAULT TO TOPMOST FRAME
12310 AN = 94  : GOSUB 6660  :REM  NORMALIZE COUNTER
12320 GOSUB 5570   :REM  UPDATE COUNTER
12330 AN = 94  : GOSUB 6660  :REM  INVERT COUNTER
12340 :
12350 CM$ = "SHOWING FRM" + STR$ ( FR ) : GOSUB 5670  :REM   FEEDBACK
12360 GOSUB 15690    :REM   TYPE 1 SOUND
12370 :
12380 T3 = FF% ( FR )   :REM  GET FRAME DATA OFFSET
12390 T4 = FD ( T3 )    :REM  GET FRAME'S TYPE
12400 :
12410 AN = TT (T4) : GOSUB 6660   :REM  INVERT TYPE'S TITLE'S AREA
12420 :
12430 REM  BRANCH TO SHOW FRAME CONTENTS
12440 ON T4 GOTO 12460, 12520, 12560, 12650, 12700, 12750, 12820
12450 :
12460 FOR N = 1 TO 8     :REM  MAKE DATA NEW CURRENT SOUND ARRAY
12470 :   CS ( N - 1 ) = FD ( T3 + N )
12480 :   NEXT
12490 GOSUB 4770  :REM  UPDATE SOUND WINDOW
12500 GOTO 12860  :REM  REGROUP
12510 :
12520 CP$ = FS$ ( FD ( T3 + 1 ) )  :REM  MAKE DATA NEW CURRENT PLAY STRING
12530 GOSUB 4970  :REM  UPDATE PLAY WINDOW
12540 GOTO 12860  :REM  REGROUP
12550 :
12560 EN = FD ( T3 + 1 ) :REM  GET ENVELOPE NUMBER
12570 FOR N = 0 TO 5     :REM  MAKE DATA NEW CURRENT ENVELOPE ARRAY ENTRY
12580 :   CE% ( EN, N ) = FD ( T3 + 2 + N )
12590 :   NEXT
12600 GOSUB 7890  :REM  SET NEW ENVELOPE
12610 GOSUB 5750  :REM  UPDATE AN ENVELOPE
12620 SYS HA (8), 1, EN + 14, 15   :REM  INVERT ENV. #
12630 GOTO 12860  :REM  REGROUP
12640 :
12650 CV = FD ( T3 + 1 )   :REM  MAKE DATA NEW CURRENT VOLUME
12660 VOL CV      :REM  SET NEW VOLUME
12670 GOSUB 5180  :REM  UPDATE VOLUME WINDOW
12680 GOTO 12860  :REM  REGROUP
12690 :
12700 CT = FD ( T3 + 1 )   :REM  MAKE DATA NEW CURRENT TEMPO
12710 TEMPO CT    :REM  SET NEW TEMPO
12720 GOSUB 5280  :REM  UPDATE TEMPO WINDOW
12730 GOTO 12860
12740 :
12750 FOR N = 1 TO 5     :REM  MAKE DATA NEW CURRENT FILTER ARRAY
12760 :   CF ( N - 1 ) = FD ( T3 + N )
12770 :   NEXT
12780 GOSUB 9250   :REM  SET NEW FILTER
12790 GOSUB 5380   :REM  UPDATE FILTER WINDOW
12800 GOTO 12860   :REM  REGROUP
12810 :
```

```
12820 AN = 94  : GOSUB 6660  :REM  NORMALIZE FRAME COUNTER
12830 T2 = FR : FR = FD ( T3 + 1 )  :REM  SET NEW FRAME
12840 GOSUB 5570   :REM  UPDATE FRAME COUNTER
12850 :
12860 IF PEEK ( HA (2) ) = 0 THEN 12860   :REM  WAIT FOR P-M BUTTON CLICK
12870 :
12880 SYS HA (3), HA (4), HA (5)  :REM  FIGURE CLICK AREA
12890 RREG MM
12900 :
12910 AN = TT (T4) : GOSUB 6660   :REM  NORMALIZE TYPE'S TITLE'S AREA
12920 :
12930 REM  CASE OUT TO ERASE CURRENT DATA
12940 ON T4 GOTO 12960, 12990, 13020, 13100, 13100, 13100, 13050
12950 :
12960 CHAR , 5, 2, LEFT$ ( BL$, 34 )  :REM  CLEAR SOUND DATA AREA
12970 GOTO 13100   :REM  REGROUP
12980 :
12990 GOSUB 3820   :REM CLEAR PLAY DATA AREA
13000 GOTO 13100   :REM  REGROUP
13010 :
13020 SYS HA (8), 1, EN + 14, 15   :REM  NORMALIZE ENV. #
13030 GOTO 13100   :REM  REGROUP
13040 :
13050 FR = T2      :REM  SET NEW FRAME
13060 GOSUB 5570   :REM  UPDATE FRAME COUNTER
13070 AN = 94  : GOSUB 6660  :REM  INVERT FRAME COUNTER
13080 GOTO 13100   :REM  REGROUP
13090 :
13100 IF MM <> 100 THEN MA = MM : GOTO 13190  :REM  LEAVE IF NOT IN PRINT BUTTON
13110 :
13120 FR = FR + 1   :REM  INCREMENT FRAME
13130 IF FR > TF THEN FR = 1 :REM  FRAME WRAPAROUND
13140 AN = 94  : GOSUB 6660  :REM  NORMALIZE FRAME COUNTER
13150 GOSUB 5570    :REM  UPDATE FRAME COUNTER
13160 AN = 94  : GOSUB 6660  :REM  INVERT FRAME COUNTER
13170 GOTO 12350    :REM  BACK UP TO SHOW FRAME
13180 :
13190 CM$ = "" : GOSUB 5670  :REM  CLEAR FEEDBACK
13200 AN = 94  : GOSUB 6660  :REM  NORMALIZE FRAME COUNTER
13210 AN = 100 : GOSUB 6660  :REM  NORMALIZE TITLE
13220 GOTO 1520              :REM  BACK TO LAB EVENT LOOP
13230 :
13240 :
13250 REM ----- BACKWARD CLICK -----
13260 :
13270 AN = 101 : GOSUB 6660   :REM  INVERT TITLE
13280 CM$ = "BACKWARD BUTTON" : GOSUB 5670    :REM  FEEDBACK
13290 GOSUB 15690       :REM  TYPE ONE SOUND
13300 :
13310 DO                :REM  BACKWARD 'TIL NO BUTTON
13320 :   FR = FR - 1   :REM     DECREMENT FRAME COUNTER
13330 :   IF FR = 0 THEN FR = MF  :REM  WRAPAROUND
13340 :   GOSUB 5570    :REM     UPDATE FRAME COUNTER
13350 LOOP WHILE PEEK ( HA (2) )
13360 GOSUB 15890       :REM  SHOW THAT FRAME
13370 :
13380 CM$ = "" : GOSUB 5670   :REM  CLEAR FEEDBACK
13390 AN = 101 : GOSUB 6660   :REM  NORMALIZE BKD
13400 GOTO 1480               :REM  BACK TO EVENT LOOP
13410 :
13420 :
13430 REM ----- SAVE CLICK -----
13440 :
13450 AN = 102 : GOSUB 6660   :REM  INVERT TITLE
13460 :
```

```
13470 CM$ = "SAVE BUTTON" : GOSUB 5670  :REM   FEEDBACK
13480 GOSUB 15690   :REM   TYPE ONE SOUND
13490 GOSUB 16330   :REM   PAUSE
13500 :
13510 GOSUB 11100   :REM  GET A FILE NAME
13520 :
13530 IF MM <> 102 THEN MA = MM : GOTO 13930  :REM  LEAVE ON OUTSIDE CLICK
13540 :
13550 IF TP$ = "" THEN 13920    :REM  LEAVE ON NULL NAME
13560 :
13570 CM$ = "OPENING FILE ..." : GOSUB 5670  :REM  FEEDBACK
13580 OPEN 12, (DN), 12, "@:" + TP$ + ",S,W"      :REM  OPEN FILE
13590 IF DS THEN 13880          :REM  JUMP IF PROBLEMS
13600 :
13610 CM$ = "SAVING DATA ..." : GOSUB 5670 :REM  FEEDBACK
13620 :
13630 PRINT# 12, TD  :REM  STORE VITALS
13640 PRINT# 12, TF
13650 PRINT# 12, TS
13660 PRINT# 12, FR
13670 :
13680 IF TD = 0 THEN 13730   :REM  SKIP IF NONE
13690 FOR N = 0 TO TD-1      :REM  STORE FRAME DATA
13700 :   PRINT# 12, FD (N)
13710 :   NEXT
13720 :
13730 IF TF = 0 THEN 13780   :REM  SKIP IF NONE
13740 FOR N = 1 TO TF        :REM  STORE FRAME DATA OFFSETS
13750 :   PRINT# 12, FF% (N)
13760 :   NEXT
13770 :
13780 IF TS = 0 THEN 13830   :REM  SKIP IF NONE
13790 FOR N = 0 TO TS-1      :REM  STORE FRAME STRINGS
13800 :   PRINT# 12, FS$ (N)
13810 :   NEXT
13820 :
13830 PRINT# 12   :REM  CLEAR BUFFER
13840 IF DS THEN 13880         :REM  JUMP IF PROBLEMS
13850 CM$ = "ALL IS SAVED" : GOSUB 5670 :REM  FEEDBACK
13860 GOSUB 16330 : GOTO 13910  :REM   PAUSE, THEN LEAVE
13870 :
13880 EM$ = "DISK PROBLEMS !!" : GOSUB 16080   :REM  FEEDBACK
13890 EM$ = DS$ : GOSUB 16080 :REM  FEEDBACK
13900 :
13910 CLOSE 12     :REM   CLOSE FILE
13920 MA = 0       :REM   NOWHERE MOUSE
13930 CM$ = "" : GOSUB 5670  :REM  CLEAR FEEDBACK
13940 AN = 102 : GOSUB 6660  :REM  NORMALIZE TITLE
13950 GOTO 1520              :REM  BACK TO LAB EVENT LOOP
13960 :
13970 :
13980 REM ----- STRIP TP$ TRAILING BLANKS -----
13990 :
14000 N = LEN ( TP$ )
14010 DO WHILE MID$ ( TP$, N, 1 ) = " "
14020 :   N = N - 1 : IF N = 0 THEN EXIT
14030 :   LOOP
14040 TP$ = LEFT$ ( TP$, N )
14050 RETURN
14060 :
14070 :
14080 REM ----- PRINT CLICK -----
14090 :
14100 AN = 103 : GOSUB 6660   :REM   INVERT TITLE
14110 :
```

```
14120 CM$ = "PRINT BUTTON" : GOSUB 5670  :REM   FEEDBACK
14130 GOSUB 15690  :REM TYPE ONE SOUND
14140 :
14150 TP = 1    :REM   START WITH FIRST FRAME
14160 :
14170 IF TP > TF THEN MA = 0 : GOTO 14700 :REM   LEAVE WHEN DONE
14180 :
14190 IF PEEK ( HA (2) ) = 0 THEN 14250   :REM   CONTINUE IF NO P-M CLICK
14200 :
14210 SYS HA (3), HA (4), HA (5) :REM   SEE WHERE CLICK WAS
14220 RREG MM
14230 IF MM <> 103 THEN 14660     :REM   LEAVE IF NOT IN PRINT BUTTON
14240 :
14250 CM$ = "PRNTG FRAME" + STR$ (TP) : GOSUB 5670  :REM   FEEDBACK
14260 :
14270 T1 = FF% ( TP )  :REM  GET FRAME DATA OFFSET
14280 T2 = FD ( T1 )   :REM  GET FRAME'S TYPE
14290 :
14300 OPEN 4,4    :REM   OPEN PRINTER
14310 PRINT#4, "FRAME #" STR$ (TP) ": " ;
14320 :
14330 REM   BRANCH TO PRINT FRAME CONTENTS
14340 ON T2 GOTO 14360, 14390, 14430, 14460, 14490, 14520, 14550
14350 :
14360 PRINT#4, "SOUND" FD (T1+1) "," FD (T1+2) ","  FD (T1+3) "," FD (T1+4) ","
               FD (T1+5) "," FD (T1+6) "," FD (T1+7) "," FD (T1+8) ;
14370 GOTO 14570
14380 :
14390 TP$ = FS$ ( FD ( T1+1 ) ) : GOSUB 14000  :REM   STRIP TRAILING BLANKS
14400 PRINT#4, "PLAY " TP$ ;
14410 GOTO 14570
14420 :
14430 PRINT#4, "ENVELOPE" FD (T1+1) "," FD (T1+2) "," FD (T1+3) "," FD (T1+4)
                "," FD (T1+5) "," FD (T1+6) "," FD (T1+7) ;
14440 GOTO 14570
14450 :
14460 PRINT#4, "VOL" FD (T1 + 1) ;
14470 GOTO 14570
14480 :
14490 PRINT#4, "TEMPO" FD (T1 + 1) ;
14500 GOTO 14570
14510 :
14520 PRINT#4, "FILTER" FD (T1+1) "," FD (T1+2) "," FD (T1+3) "," FD (T1+4) ","
                FD (T1+5) ;
14530 GOTO 14570
14540 :
14550 PRINT#4, "JUMP TO FRAME" FD ( T1 + 1 ) ;
14560 :
14570 IF TP/60 <> INT (TP/60) THEN 14600  :REM   JUMP IF NOT PAGE END
14580 FOR N = 1 TO 6 : PRINT#4 : NEXT
14590 :
14600 PRINT#4 : CLOSE 4  :REM  CLOSE PRINTER
14610 POKE 186, DN       :REM  RENEW DISK DEVICE #
14620 :
14630 TP = TP + 1   :REM  NEXT FRAME
14640 GOTO 14170    :REM  BACK ON UP
14650 :
14660 CM$ = "USER SEZ STOP" : GOSUB 5670   :REM   FEEDBACK
14670 GOSUB 15690   :REM  TYPE 1 SOUND
14680 MA = MM       :REM  ASSIGN MOUSE AREA
14690 :
14700 CM$ = "" : GOSUB 5670  :REM  CLEAR FEEDBACK
14710 AN = 103 : GOSUB 6660  :REM  NORMALIZE TITLE
14720 GOTO 1520              :REM  BACK TO LAB EVENT LOOP
14730 :
```

```
14740 :
14750 REM ----- END CLICK -----
14760 :
14770 AN = 105 : GOSUB 6660      :REM   INVERT "END"
14780 CM$ = "SO" : GOSUB 5670    :REM   DRAW MESSAGE
14790 SOUND 1, 3000,  4,  0,  0,  0,   2, 200  :REM   BEEP
14800 CM$ = "SO LONG," : GOSUB 5670            :REM   DRAW MESSAGE
14810 SOUND 1, 2500,  4,  0,  0,  0,   2, 200  :REM   BOP
14820 CM$ = "SO LONG, PAL ..." : GOSUB 5670    :REM   DRAW MESSAGE
14830 SOUND 1, 2800,  4,  0,  0,  0,   2, 200  :REM   BLIP
14840 AN = 105 : GOSUB 6660      :REM   NORMALIZE "END"
14850 FINISHED = 1     :REM   WE DONE
14860 GOTO 1480        :REM   BACK TO LAB EVENT LOOP
14870 :
14880 :
14890 REM ----- FETCH A PARAMETER -----
14900 :
14910 REM  UPON ENTRY, RW CONTAINS THE ROW THE PARAMETER LIVES IN [0..24]
14920 REM              PN CONTAINS A PARAMETER SELECTOR [0..20]
14930 REM              EV CONTAINS A PARAMETER ENTRY VALUE
14940 REM              AD CONTAINS AN AREA ID NUMBER
14950 :
14960 REM  UPON EXIT,  MM CONTAINS AREA LOCATION OF AN EXITING MOUSECLICK
14970 REM              XV CONTAINS A PARAMETER EXIT VALUE
14980 REM              VC CONTAINS A PARAMETER VALIDITY CODE :
14990 REM                 0   INVALID
15000 REM                -1   VALID
15010 :
15020 IF EV < PF (PN, 0) THEN EV = PF (PN, 0)  :REM  INSURE VALID ENTRY VALUE
15030 IF EV > PF (PN, 1) THEN EV = PF (PN, 1)
15040 EV$ = STR$ (EV)                          :REM   STRINGIZE ENTRY VALUE
15050 T = LEN (EV$) : W = PF (PN, 3)           :REM   GET A COUPLA WIDTHS
15060 EV$ = LEFT$ (ZR$, W - T + 1) + RIGHT$ (EV$, T - 1) :REM   PAD EV$ WITH 0'S
15070 :
15080 :                             :REM   SET UP FOR STRNGRECTEDIT
15090 SR% (0) = POINTER (EV$)       :REM   ENTRY STRING
15100 SR% (1) = POINTER (EV$)       :REM   EXIT STRING
15110 SR% (2) = RW                  :REM   TOP
15120 SR% (3) = RW                  :REM   BOTTOM
15130 SR% (4) = PF (PN, 2)          :REM   LEFT
15140 SR% (5) = SR% (4) + W  - 1    :REM   RIGHT
15150 SR% (6) = AD                  :REM   AREA ID
15160 SR% (7) = HA (12)             :REM   AREA DATA TABLE
15170 SR% (8) = 0                   :REM   EDITING CURSOR AT START
15180 N = POINTER ( SR% ( 0 ) )     :REM  ADDRESS OF ARRAY
15190 HB = INT ( N / 256 )          :REM  ADDRESS HI-BYTE
15200 LB = N - ( HB * 256 )         :REM  ADDRESS LO-BYTE
15210 SYS 1024, LB, HB, 1           :REM  CALL STRNGRECTEDIT
15220 RREG MM                       :REM   GET P-M EXIT AREA ID
15230 :
15240 XV = VAL (EV$)                :REM   GET EXIT VALUE
15250 IF XV >= PF (PN, 0)  AND  XV <= PF (PN, 1)
      THEN VC = -1 : GOTO 15310     :REM  INDICATE & JUMP IF VALID VALUE
15260 :
15270 VC = 0    :REM   INDICATE INVALID
15280 XV = EV   :REM   RESTORE ENTRY VALUE
15290 EM$ = "BAD PARAMETER !" : GOSUB 16080  :REM FEEDBACK
15300 :
15310 XV$ = STR$ (XV)  :REM   STRINGIZE EXIT VALUE
15320 T = LEN (XV$)    :REM   LENGTH OF EXIT VALUE STRING
15330 XV$ = LEFT$ (ZR$, W - T + 1) + RIGHT$ (XV$, T - 1)    :REM   PAD WITH 0'S
15340 SR% (0) = POINTER (XV$)     :REM   ENTRY STRING
15350 SR% (1) = POINTER (XV$)     :REM   EXIT STRING
15360 N = POINTER ( SR% ( 0 ) )   :REM  ADDRESS OF ARRAY
15370 HB = INT ( N / 256 )    :REM  ADDRESS HI-BYTE
```

```
15380 LB = N - ( HB * 256 )    :REM  ADDRESS LO-BYTE
15390 SYS 1024, LB, HB, 0      :REM  CALL STRNGRECTEDIT TO UPDATE EXIT VALUE
15400 :
15410 RETURN
15420 :
15430 :
15440 REM ----- RECORD A SOUND FRAME -----
15450 :
15460 IF   TD + DZ ( D(0) ) < MD   AND   FR <= MF   AND   TS <= MS   THEN 15510
15470 :
15480 EM$ = "NO ROOM TO RECRD" : GOSUB 16080  :REM   NO-ROOM FEEDBACK
15490 RR = 0 : RETURN                         :REM   SET RESULT AND LEAVE
15500 :
15510 FOR N = 0 TO DZ ( D (0) )  :REM   IF ROOM, STORE DATA
15520 :   FD ( TD + N ) = D (N)  :REM      STORE EACH FRAME ELEMENT
15530 :   NEXT
15540 FF% ( FR ) = TD        :REM   STORE OFFSET TO THIS FRAME'S STACK DATA
15550 TD = TD + N            :REM   ADJUST TOP OF STACK
15560 IF FR > TF THEN TF = FR   :REM   IF NEEDED, UP HIGHEST RECORDED FRAME
15570 CM$ = "FRAME" + STR$ ( FR ) + " REC'D" : GOSUB 5670   :REM   FEEDBACK
15580 GOSUB 15690            :REM   TYPE ONE SOUND
15590 AN = 94 : GOSUB 6660   :REM   NORMALIZE COUNTER
15600 FR = FR + 1  :REM   UPDATE FRAME COUNTER
15610 IF FR > MF THEN FR = 1  :REM   WRAPAROUND
15620 GOSUB 5570   :REM   SHOW UPDATE ON SCREEN
15630 RR = -1      :REM   SET RESULT
15640 RETURN
15650 :
15660 :
15670 REM ----- TYPE ONE SOUND -----
15680 :
15690 VOL 15 : SOUND 1, 3000, 4, 0, 0, 0, 2, 200  :REM   BEEP LOUD
15700 GOSUB 15830      :REM  LET SOUND FINISH
15710 VOL CV : RETURN  :REM  RESTORE VOLUME & GIT BACK
15720 :
15730 :
15740 REM ----- TYPE THREE SOUND -----
15750 :
15760 VOL 15 : SOUND 1, 8000, 15, 1, 0, 200, 1   :REM   RAZZ LOUD
15770 GOSUB 15830             :REM   LET SOUND FINISH
15780 VOL CV : RETURN         :REM   RESTORE VOLUME & GIT BACK
15790 :
15800 :
15810 REM ----- LET SOUND FINISH -----
15820 :
15830 DO : LOOP UNTIL PEEK (4741) = 255 AND PEEK (4738) = 255
15840 RETURN
15850 :
15860 :
15870 REM ----- SHOW FRAME, WITH RECORDED CHECK -----
15880 :
15890 IF TF >= FR THEN 15940    :REM   IF FRAME'S BEEN RECORDED, JUMP
15900 :
15910 CM$ = "NOT RECORDED YET" : GOSUB 5670   :REM   FEEDBACK
15920 GOSUB 15760  :REM   TYPE 3 SOUND
15930 :
15940 GOSUB 16000  :REM   SHOW FRAME
15950 RETURN
15960 :
15970 :
15980 REM ----- SHOW FRAME -----
15990 :
16000 CM$ = "FRM GOES TO" + STR$ (FR) : GOSUB  5670  :REM   FEEDBACK
16010 GOSUB 15690              :REM   TYPE ONE SOUND
16020 FOR N = 1 TO 250 : NEXT   :REM   PAUSE
```

```
16030 RETURN
16040 :
16050 :
16060 REM ----- SEND AN ERROR MESSAGE -----
16070 :
16080 VOL 15                :REM  MAX NOISE
16090 DO WHILE EM$ <> ""    :REM  PRINT 'TIL PRINTED
16100 :  TP$ = LEFT$ (EM$, 16)     :REM  GRAB A HUNK
16110 :  FOR N = 1 TO 2            :REM  FLASH IT TWICE
16120 :    CM$ = TP$ : GOSUB 5670  :REM    FLASH IT
16130 :    GOSUB 15760             :REM    TYPE 3 SOUND
16140 :    FOR P = 1 TO 120 : NEXT :REM    PAUSE
16150 :    CM$ = "" : GOSUB 5670   :REM    BLANK IT
16160 :    FOR P = 1 TO 60 : NEXT  :REM    PAUSE
16170 :    NEXT
16180 :    EM$ = MID$ ( EM$, 17, 16 )  :REM  NEXT PIECE
16190 :    LOOP
16200 VOL CV                :REM   RESTORE VOLUME
16210 RETURN
16220 :
16230 :
16240 REM ----- ERROR HANDLER -----
16250 :
16260 EM$ = ERR$ ( ER ) + " IN" + STR$ (EL) :REM  BUILD ERROR MESSAGE
16270 GOSUB 16080    :REM  SEND IT
16280 RESUME NEXT    :REM  GET BACK
16290 :
16300 :
16310 REM ----- PAUSE -----
16320 :
16330 FOR N = 1 TO 100 : NEXT N : RETURN


READY.
```

Appendix A: Useful Conventions

By useful conventions, I mean: abbreviations, jargon, number formats, system terminology, etc. I try to: hold this stuff to a minimum; use the most natural forms of expression; keep to the terminology Commodore uses in their C-128 Programmer's Reference Guide; and be consistent in my usage. Here's a list:

Bits, Nibbles, Bytes, And Words

These four terms describe convenient chunks of computer number representation.

A **bit** is the smallest value a computer diddles with. A **bit** can take on either of the values 0 or 1.

A **nibble** is four **bits**. That's half a **byte**, or one-fourth of a **word**.

A **byte** is eight **bits**. That's two **nibbles**, or half a **word**.

A **word** is sixteen **bits**. That's four **nibbles**, or two **bytes**.

Books

I use abbreviations for the following book titles. When I refer to page numbers, they're from these specific editions.

C-128 Prg	*Commodore* 128 *Programmer's Reference Guide*, by Larry Greenley & others (Bantam Books, Inc. First edition. February, 1986.)
C-128 Ints	*Commodore* 128 *Internals*, by K. Gerits & others (Abacus Software. First edition. October, 1985.)
C64/128 S&GP	*Commodore* 64/128 *Graphics And Sound Programming, 2nd Edition*, by Stan Krute (TAB BOOKS Inc. Second edition. 1986.)

8502 Registers

I use A to indicate the 8502's accumulator, X for the X register, and Y for the Y register.

Kernel Functions

I use the kernal function names found on pages 414-457 of the C-128 Prg. In most cases, I'll have the function's jump table address nearby, in hexadecimal format. When I can come up with a reasonable set of words, I use that information to capitalize the kernel function name. And, like Commodore, I can never remember how to spell kernel, or is it kernal?

examples:	TkSA	$FF96	Talk Secondary Address
	MemTop	$FF99	Memory Top
	PrImm	$FF7D	Print Immediate

Memory Locations

I use the memory location names found on pages 502-540 of the C-128 Prg. As with kernel functions, you'll usually find the actual address nearby, in hexadecimal format. And when I can come up with a reasonable set of words that fit the name, I use that information to capitalize the names.

examples:	GarbFl	$0011	Garbage Flag
	AryTab	$0031	Array Table

Miscellaneous Terms

0-based, 1-based Sometimes in computer work we start counting with 0, sometimes we start with 1. These are adjectives I use to distinguish the two types of counting.

assembly language A language with a very low level of abstraction, assembly language allows/requires you to program a computer by direct use of memory locations and chip registers. Assembly language instructions translate directly and mechanically into equivalent machine language instructions for the computer's processor.

C-ASCII Short for Commodore ASCII. The code numbers for the set of text and control characters used by the C-128. Similar to but distinct from the standard ASCII codes.

CIA Complex Interface Adapter. The C-128 has two of these versatile input/output/timer chips, CIA 1 and CIA 2.

lo-byte, hi-byte I refer to bits 0..7 of a word as the lo-byte, bits 8..15 as the hi-byte.

lo-nibble, hi-nibble I refer to bits 0..3 of a byte as the lo-nibble, bits 4..7 as the hi-nibble.

memory quadrant A 16,384-byte piece of memory. That's one quarter of the processor's 65,536-byte memory map. The VIC chip does some of its work based on a default quadrant. In this book, I often qualify four quadrants with a numeric adjective, as follows:

zeroth quadrant	$0000-$3FFF
first quadrant	$4000-$7FFF
second quadrant	$8000-$BFFF
third quadrant	$C000-$FFFF

machine language

The actual numeric code that controls the operations of a computer's processor. An assembler takes a program written in assembly language and turns it into machine language.

MMU — Memory Management Unit. To the programmer, the MMU is a set of primary and secondary registers that control aspects of the C-128's memory mapping. Check out pages 458-471 of the C-128 Prg.

object code

Machine language produced by an assembler or compiler.

p-m — Short for pseudo-mouse. This is when we use a joystick and/or the keyboard to simulate a mouse.

Port A,

Port B — Each CIA chip has two byte-sized input/output ports. This is what we call them.

source code

The program instructions a programmer actually writes. An assembler, compiler, or interpreter is used to transform this code into machine language.

the system

Think of the operational C-128 computer and its peripherals as an entity you interract with. This phrase is the entity's name. May indicate particular aspects of same, depending on context.

Numbers

A number without a prefix character is decimal.

example: 22

A number with a $ prefix is hexadecimal.

example $F7D3

A number with a % prefix is binary.

example: %10110011

I've tried to use the number format that's most appropriate to a given situation. In general:

decimal numbers are used for:
register numbers, loop counts, hardware specifications

hexadecimal numbers are used for:
addresses

binary numbers are used for:
masks, flags

Processors

The C-128 has two processors, an 8502 and a Z-80. In this book we deal with the 8502. The Z-80 is used by CP/M software. It's quite powerful, actually, and could be used to do graphics and sound work, but the programming tools for such tasks aren't widely available. So I ignore it herein.

The 8502 is Commodore's slightly modified version of a 6502 chip. They did a similar thing with the C-64; in that machine, the modified 6502 is called a 6510. The noticeable part of the modification involves the use of memory locations $0000 and $0001 as I/O ports to control several hardware functions. All three chips use the same 6502 assembly language. In this book, I use "6502" and "8502" interchangeably. So, if you see one, think of the other.

VIC Registers

Depending upon the context, I use O-based decimal register numbers, capitalized versions of the register names from pages 524-527 of the C-128 Prg, and/or hexadecimal absolute addresses.

examples:	VIC register 0	VicReg0	$D000
	VIC register 22	VicReg22	$D016

VDC Registers

Depending upon the context, I use 0-based decimal register numbers and/or the register names from Fig. 10-1 (page 294) of the C-128 Prg.

examples:	VDC register 0	Horizontal Total
	VDC register 31	Data

Appendix B: Calling Structure Diagrams

CALLING STRUCTURE DIAGRAMS

The key to writing easily-debugged programs is modularity. Break the programming task up into a number of mostly-self-contained pieces of code. Now, depending on the context, these pieces may be called functions, routines, subroutines, procedures, modules, blocks, or something totally different. But the idea is the same. You build a piece of code, get it functional, then call on it as a unit from other pieces. That way you get to remove one more layer of detail from your thought processes. You can get tasks done at a higher level of abstraction.

The programs in this book are highly modular. That's because I detest debugging. It's nice to quickly see all a program's modules, and the lines of communication between them. Calling structure diagrams help.

All the programs in this book come with a complete set of calling structure diagrams. They're called that because they show how a program's modules call upon one another. In the discussion that follows, I'll describe the graphic vocabulary used in the diagrams.

In the diagrams, each routine is represented by a rectangle with an identifier. Here are some examples :

Install UnInstall Fetch A Parameter

If it's a BASIC 7.0 routine, the identifier matches the comment used in the source code, and indicates what the routine does, as in these examples :

Type One Sound Get A File Name

If it's a 6502 assembly language routine, the identifier is the one used in the source code, as in these examples :

OurKeyChk HRAreaInvt IEscPrDetor

If it's a documented routine from the C-128's ROM, the identifier is usually the one Commodore uses in the C-128 PRG. In addition, a solid vertical line on the left side of the rectangle indicates it's a documented ROM routine. Here are some examples :

IndSta IndFet BSOut

Undocumented (gasp) ROM routines get a dotted vertical line on the left side of the rectangle. The identifier is the one I use in the program's source code. Examples :

FndComTxt   FndTknTxt

Three types of routines don't get analyzed any further in the calling structure diagrams : ROM routines, terminal routines, and foreign routines.

ROM routines aren't analyzed any further -- that is, I don't show what routines they may call on -- because we're tracing through my code convolutions, not Commodore's.

A terminal, or leaf, routine is one that doesn't call on other routines. Such a routine is marked with a solid horizontal line on the bottom of the rectangle. Examples :

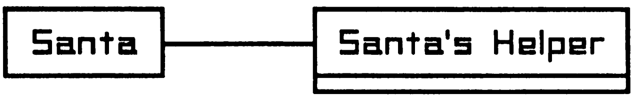
LodSprDat   InitPosits   InitHues

A foreign routine is one contained in another program. Since it's analyzed there, no sense doing so again. Such a routine is marked with a solid horizontal line on the bottom of the rectangle, and the name of its home program beneath that line. Examples :

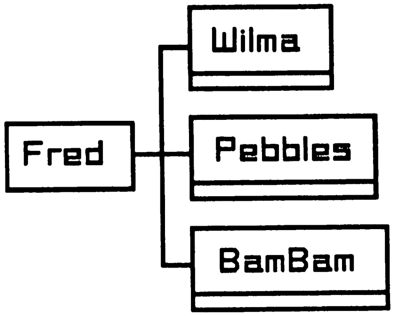
Some routines are called through vectors. I indicate such a routine by encasing its identifier in parentheses. The identifier will usually be the vector's name. Examples :

Okay, I've covered how the routine rectangles are set up. Now I'll explain how connections between routines are indicated.

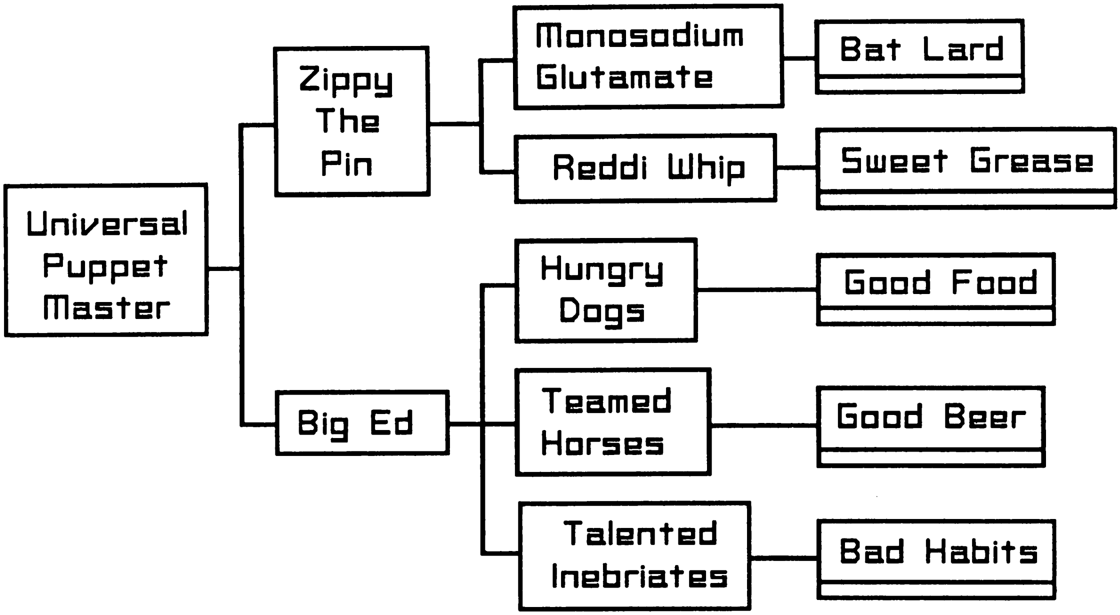
If one routine calls on another routine, the called routine is placed to the right of the caller, and the two routines are connected with a line. Example :



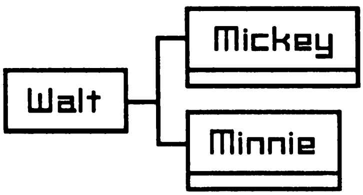
If a routine calls on several routines, the called routines are aligned vertically, and their links to the calling routine look something like this :

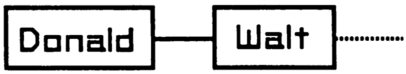


Of course, a called routine may call on other routines, which may call on others, and on and on. Example :



Sometimes a routine that calls on other routines appears more than once on a single page. Or there's no room to finish a string of calls in one horizontal band, but there's room to finish up elsewhere on the same page. Or there's not enough room to list all the routines called by a routine in one vertical column, but there's room elsewhere on the page to finish up. In all these cases, I use a dotted line to indicate that there's more to look for, and that it can be found on the same page. Examples :





And sometimes the continuation can't fit on the same page, or the routine's already analyzed on another page. In those cases, the dotted line connects to a little box, and the box contains the sheet number (of that program's set of calling structure diagram sheets) that contains the rest of the analysis. Example :

Finally : if a routine has been referred to previously, then its rectangle will have a dotted line on the left side. Here's an example :

Well, I think that covers all the features of the calling structure diagrams used in this book. I find them useful at all stages of my programming tasks, and hope they help your own understanding. All I've got to do now is come up with a way to automate their production.

Appendix C: Pinhead Pseudo-Code

Pseudo-code is a way of expressing computer algorithms in a language-independent, near-English form. It's particularly useful when documenting algorithms written in languages that lack modern structured features. I call the pseudo-code in this book the **Pinhead Pseudo Code (PPC)** in honor of its utter simplicity. You'll probably be able to read it without any fancy explanation, but completeness compels me to codify it.

While discussing the **PPC**, I'll have cause to mention common programming features and techniques. Though I may seem to refer to all types of programs, that's just a stylistic easement. The type of programs I have in mind are those that are well-structured, modular, and do one thing at a time.

Such a program consists of a collection of instructions. The instructions are grouped into subsets to ease the programmer's and computer's minds, and these groupings are called functions, subroutines, procedures, or some such name. In the **PPC**, a call to a subroutine is represented by a short phrase that describes the subroutine call. Here's an example:

```
CALL on Type One Sound for a beep
```

Note that the **PPC** keyword **CALL** appears in this short phrase, as does the capitalized name of the called subroutine (Type One Sound), and a short explanatory clause. By the way, **PPC** keywords come completely capitalized, subroutine names have the first letter of each word capitalized, and instructions are uncapitalized. Sometimes a subroutine name is underlined, if it aids comprehension. Here's another example of a subroutine call. Note that it also includes the three elements (**CALL**, subroutine name, short explanatory clause):

```
CALL Update Filter Window to redraw the Filter Window
```

As mentioned above, a subroutine is a collection of instructions, and it has a name. In the **PPC**, each subroutine's instructions are listed after its name, indented a tab position. Example:

```
Initialize Variables
    set all integer and real variables to 0
    set all strings to the empty string
    RETURN
```

This subroutine has two instructions. When it finishes, it returns control to wherever it was called from, as indicated by the **PPC** keyword **RETURN**.

A subroutine may not return control so neatly when it finishes. It may jump to some spot in the code other than where it was called from. The **PPC** keyword **JUMP** is then used. Here's an example:

JUMP to the System Error Handler

Within a subroutine, instructions are carried out one after another (sequence), repeatedly (loop), and/or conditionally (branching). The previous examples include instructions that are carried out sequentially. Here's an example of a **PPC** loop:

```
REPEAT
    check for a keypress
UNTIL
    the Spacebar is pressed
```

The **PPC** keyword construction **REPEAT** *some action* **UNTIL** *some condition is true* is one way to indicate a loop. It's used when the action must be carried out at least once. Notice how I use indentation to reinforce syntax. Another **PPC** keyword construction used to indicate looping is **WHILE** *some condition is true* **DO** *some action*. Here's an example:

```
WHILE
    there's a sound being produced
DO the following
    change the sprite's color to the next color
    move the sprite three positions to the left
```

Note a few things here. First, the action may never be taken, since the conditional test comes first. Second, the clarifying phrase the following is stuck onto the keyword **DO**. It's there just to get closer to the friendly english language. Third, the action taken can consist of more than one action. Finally, indentation again adds clarity to the syntax.

Another keyword construction I use for loops is **WAIT UNTIL** *some condition is true*. It's really just a convenient way to express a **REPEAT..UNTIL** loop whose action is no action. Here's an example:

```
set sprite 1 into motion
WAIT UNTIL
    the joystick button is pressed
stop sprite 1
```

Another loop construction used in the **PPC** is the **FOR** loop. The format is this: **FOR** *each member of a set* **DO** *some action*. Here's an example:

```
FOR
    each integer in the range 1 to 10
DO the following
    set the text character color to the integer's value
    print the integer
```

print the phrase "comes in this color"
print a carriage return

The workhorse **PPC** branching statement is **IF** *some condition is true* **THEN** *do some action*. As in most modern programming languages, this can be extended with **ELSE** and **ELSE IF** clauses, as in this example:

```
IF
    the up-cursor key is pressed
THEN
    move the sprite up one position
ELSE IF
    the down-cursor key is pressed
THEN
    move the sprite down one position
ELSE IF
    the left-cursor key is pressed
THEN
    move the sprite left one position
ELSE IF
    the right-cursor key is pressed
THEN
    move the sprite right one position
ELSE
    flash the message "move me, please"
```

Sometimes I'll give more detail on a single instruction. This is indicated in the **PPC** by ending the single instruction with a colon (:), then indenting the block of detail beneath it.

Example:

```
set screen attributes:
    set a black background
    set a black border for 40-column screen
    set a foreground character color
    make sure screen is in text mode and clear it
```

Finally: sometimes I'll include extra explanatory comments in the **PPC**. They're encased in parentheses (like this) or curly brackets {like this}.

Appendix D: System Interface Summary

System Interface Summary — Sheet 1 Of 10

G80 INSTALL.S

BASIC 7.0 : NEW command from assembly language	113–114
BASIC 7.0 : setting program text start	103–111
BASIC 7.0 : warm start from assembly language	146–147
Kernal routine : Load ($FFD5)	139–141
Kernal routine : SetBnk ($FF68)	118–121
Kernal routine : SetLFS ($FFBA)	129–137
Kernal routine : SetNam ($FFBD)	123–127
Memory Configuration	95–101, 149–151

GRAFIX 80 0.S

GRAFIX 80 1.S

BASIC 7.0 : cruising through commas in program text	294–312
Low–Memory Routine : ChrGet ($0380)	307
Low–Memory Routine : IndTxt ($03C9)	295–297
System Global : $000D (Count)	185–190
System Vectors : IError ($0300–$0301)	310–312
System Vectors : IEscLk ($030C–$030D)	11–21, 92–96, 147–200
System Vectors : IEscPr ($030E–$030F)	38–48, 113–117,203–234
System Vectors : IEscEx ($0310–$0311)	65–75, 134–138, 237–276
Tokens : crunching new commands	181–200
Tokens : detecting new commands	162–179, 211–217
Tokens : un–crunching new commands	220–227
Undocumented ROM Routine : FndComTxt ($43E2)	174–179
Undocumented ROM Routine : FndTknTxt ($516A)	174–179

System Interface Summary — Sheet 2 Of 10

GRAFIX 80 2.S

Low–Memory Routine : ChrGet ($0380)	49–50, 355
Low–Memory Routine : ChrGot ($0386)	84–86, 114–116
Memory Configuration :	200–204, 273–274, 419–423, 457–459
6502 Usage : deriving an absolute value	255–257
6502 Usage : two–stage masking	450–454

System Interface Summary Sheet 4 Of 10

System Interface Summary Sheet 5 Of 10

Appendix E: VIC Registers

VIC starting address is 53248 ($D000)

Register number Decimal	Hex	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	This register controls:
0	$00	S0 H7	S0 H6	S0 H5	S0 H4	S0 H3	S0 H2	S0 H1	S0 H0	Sprite #0 horizontal position
1	$01	S0 V7	S0 V6	S0 V5	S0 V4	S0 V3	S0 V2	S0 V1	S0 V0	Sprite #0 vertical position
2	$02	S1 H7	S1 H6	S1 H5	S1 H4	S1 H3	S1 H2	S1 H1	S1 H0	Sprite #1 horizontal position
3	$03	S1 V7	S1 V6	S1 V5	S1 V4	S1 V3	S1 V2	S1 V1	S1 V0	Sprite #1 vertical position
4	$04	S2 H7	S2 H6	S2 H5	S2 H4	S2 H3	S2 H2	S2 H1	S2 H0	Sprite #2 horizontal position
5	$05	S2 V7	S2 V6	S2 V5	S2 V4	S2 V3	S2 V2	S2 V1	S2 V0	Sprite #2 vertical position
6	$06	S3 H7	S3 H6	S3 H5	S3 H4	S3 H3	S3 H2	S3 H1	S3 H0	Sprite #3 horizontal position
7	$07	S3 V7	S3 V6	S3 V5	S3 V4	S3 V3	S3 V2	S3 V1	S3 V0	Sprite #3 vertical position
8	$08	S4 H7	S4 H6	S4 H5	S4 H4	S4 H3	S4 H2	S4 H1	S4 H0	Sprite #4 horizontal position
9	$09	S4 V7	S4 V6	S4 V5	S4 V4	S4 V3	S4 V2	S4 V1	S4 V0	Sprite #4 vertical position
10	$0A	S5 H7	S5 H6	S5 H5	S5 H4	S5 H3	S5 H2	S5 H1	S5 H0	Sprite #5 horizontal position
11	$0B	S5 V7	S5 V6	S5 V5	S5 V4	S5 V3	S5 V2	S5 V1	S5 V0	Sprite #5 vertical position
12	$0C	S6 H7	S6 H6	S6 H5	S6 H4	S6 H3	S6 H2	S6 H1	S6 H0	Sprite #6 horizontal position
13	$0D	S6 V7	S6 V6	S6 V5	S6 V4	S6 V3	S6 V2	S6 V1	S6 V0	Sprite #6 vertical position
14	$0E	S7 H7	S6 H6	S7 H5	S7 H4	S7 H3	S7 H2	S7 H1	S7 H0	Sprite #7 horizontal position
15	$0F	S7 V7	S7 V6	S7 V5	S7 V4	S7 V3	S7 V2	S7 V1	S7 V1	Sprite #7 vertical position
16	$10	S7 H8	S6 H8	S5 H8	S4 H8	S3 H8	S2 H8	S1 H8	S0 H8	Most significant bit of horizontal positions
17	$11	Raster bit 8	Extended color text mode	Bit map mode	Blank screen	24 or 25 rows of text	Vertical scroll bit 2	Vertical scroll bit 1	Vertical scroll bit 0	Miscellaneous functions

18	$12	Raster bit 7	Raster bit 6	Raster bit 5	Raster bit 4	Raster bit 3	Raster bit 2	Raster bit 1	Raster bit 0	Raster register
19	$13	LP H7	LP H6	LP H5	LP H4	LP H3	LP H2	LP H1	LP H0	Light pen horizontal position
20	$14	LP V7	LP V6	LP V5	LP V4	LP V3	LP V2	LP V1	LP V0	Light pen vertical position
21	$15	S7 On/off	S6 On/off	S5 On/off	S4 On/off	S3 On/off	S2 On/off	S1 On/off	S0 On/Off	Turn sprites on/off
22	$16	—	—	Reset-always set to 0	Multi-color mode	38 or 40 columns of text	Horizontal scroll bit 2	Horizontal scroll bit 1	Horizontal scroll bit 0	Miscellaneous functions
23	$17	S7 EV	S6 EV	S5 EV	S4 EV	S3 EV	S2 EV	S1 EV	S0 EV	Expand sprite (2x) vertically
24	$18	Text screen bit 3	Text screen bit 2	Text screen bit 1	Text screen bit 0	Char defs bit 2	Char defs bit 1	Char defs bit 0	—	Memory pointers for character display, bit map, & screen
25	$19	Interrupt from VIC	—	—	—	Light pen latched	Sprite to sprite collision	Sprite to bkgrnd collision	Raster count match	Interrupt register
26	$1A	—	—	—	—	Light pen latched	Sprite to sprite collision	Sprite to bkgrnd collision	Raster count match	Enable interrupts
27	$1B	S7 SBP	S6 SBP	S5 SBP	S4 SBP	S3 SBP	S2 SBP	S1 SBP	S0 SBP	Sprite to background priorities
28	$1C	S7 MCM	S6 MCM	S5 MCM	S4 MCM	S3 MCM	S2 MCM	S1 MCM	S0 MCM	Select multicolor mode for sprites
29	$1D	S7 EH	S6 EH	S5 EH	S4 EH	S3 EH	S2 EH	S1 EH	S0 EH	Expand sprite (2x) horizontally
30	$1E	S7 SSC	S6 SSC	S5 SSC	S4 SSC	S3 SSC	S2 SSC	S1 SSC	S0 SSC	Sprite to sprite collision
31	$1F	S7 SBC	S6 SBC	S5 SBC	S4 SBC	S3 SBC	S2 SBC	S1 SBC	S0 SBC	Sprite to back-ground collision

VIC starting address is 53248 ($D000)

Register number Decimal	Hex	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	This register controls:
32	$20	–	–	–	–	Border C3	Border C2	Border C1	Border C0	Border color
33	$21	–	–	–	–	Bkg 0 C3	Bkg 0 C2	Bkg 0 C1	Bkg 0 C0	Background #0 color
34	$22	–	–	–	–	Bkg 1 C3	Bkg 1 C2	Bkg 1 C1	Bkg 1 C0	Background #1 color
35	$23	–	–	–	–	Bkg 2 C3	Bkg 2 C2	Bkg 2 C1	Bkg 2 C0	Background #2 color
36	$24	–	–	–	–	Bkg 3 C3	Bkg C2	Bkg 3 C1	Bkg 3 C0	Background #3 color
37	$25	–	–	–	–	SMC 0 C3	SMC 0 C2	SMC 0 C1	SMC 0 C0	Sprite multicolor #0
38	$26	–	–	–	–	SMC 1 C3	SMC 1 C2	SMC 1 C1	SMC 1 C0	Sprite multicolor #1
39	$27	–	–	–	–	S0 C3	S0 C2	S0 C1	S0 C0	Sprite #0 color
40	$28	–	–	–	–	S1 C3	S1 C2	S1 C1	S1 C0	Sprite #1 color
41	$29	–	–	–	–	S2 C3	S2 C2	S2 C1	S2 C0	Sprite #2 color
42	$2A	–	–	–	–	S3 C3	S3 C2	S3 C1	S3 C0	Sprite #3 color
43	$2B	–	–	–	–	S4 C3	S4 C2	S4 C1	S4 C0	Sprite #4 color
44	$2C	–	–	–	–	S5 C3	S5 C2	S5 C1	S5 C0	Sprite #5 color
45	$2D	–	–	–	–	S6 C3	S6 C2	S6 C1	S6 C0	Sprite #6 color
46	$2E	–	–	–	–	S7 C3	S7 C2	S7 C1	S7 C0	Sprite #7 color

Appendix F:
VIC Screen Colors

0 - black	8 - orange
1 - white	9 - brown
2 - red	10 - light red
3 - cyan	11 - dark gray
4 - purple	12 - medium gray
5 - green	13 - light green
6 - blue	14 - light blue
7 - yellow	15 - light gray

Appendix G: Sprite Shadow Registers

Two sets of memory locations serve as sprite shadow registers. That is, each time the C-128's vertical retrace interrupt occurs, the values in these locations are used to update a number of sprite-related VIC chip registers. These registers are the easiest way to work with sprites from assembly language (so long as you haven't disabled the vertical retrace interrupt).

$11D6-$11EA VIC CHIP SHADOW REGISTERS

These 21 memory locations are used by the C-128 to update 21 VIC chip registers, triggered by the vertical retrace interrupt. By poking appropriate values directly into these locations, you can update the VIC registers.

The mapping of memory locations into VIC registers is as follows (addresses are given in hexadecimal and decimal):

memory location		VIC register location		VIC register number	brief description
$11D6	4566	$D000	53248	0	sprite 0,horizontal lo-byte
$11D7	4567	$D001	53249	1	sprite 0,vertical
$11D8	4568	$D002	53250	2	sprite 1,horizontal lo-byte
$11D9	4569	$D003	53251	3	sprite 1,vertical
$11DA	4570	$D004	53252	4	sprite 2,horizontal lo-byte
$11DB	4571	$D005	53253	5	sprite 2,vertical
$11DC	4572	$D006	53254	6	sprite 3,horizontal lo-byte
$11DD	4573	$D007	53255	7	sprite 3,vertical
$11DE	4574	$D008	53256	8	sprite 4,horizontal lo-byte
$11DF	4575	$D009	53257	9	sprite 4,vertical
$11E0	4576	$D00A	53258	10	sprite 5,horizontal lo-byte
$11E1	4577	$D00B	53259	11	sprite 5,vertical
$11E2	4578	$D00C	53260	12	sprite 6,horizontal lo-byte
$11E3	4579	$D00D	53261	13	sprite 6,vertical
$11E4	4580	$D00E	53262	14	sprite 7,horizontal lo-byte
$11E5	4581	$D00F	53263	15	sprite 7,vertical
$11E6	4582	$D010	53264	16	sprites 0-7, horizontal hi-bits
$11E7	4583	$D01E	53278	30	sprite to sprite collision latch
$11E8	4584	$D01F	53279	31	sprite to background collision latch
$11E9	4585	$D013	53267	19	light pen latch horizontal
$11EA	4586	$D014	53268	20	light pen latch vertical

$117E-$11D5 SPRITE SPEED AND DIRECTION TABLES

These 88 memory locations (11 for each sprite) are used by the C-128 to implement sprite motion. The BASIC 7.0 sprite motion commands plug them with values, then the vertical retrace interrupt routines use those values to adjust the VIC chip registers that position the sprites. You get sprites to move by poking appropriate values directly into these locations.

Although not documented, a little experimentation let me figure out enough to be able to use these tables. A Pascal declaration of this area of memory would look like this:

```
spriteSpdDirTables    = ARRAY [0..7] OF spriteSpdDirData
spriteSpdDirData      = RECORD
                            speed:      byte;       {offset 0}
                            unknown1:   byte;       {offset 1}
                            quadrant:   byte;       {offset 2}
                            deltaX:     word;       {offset 3}
                            deltaY:     word;       {offset 5}
                            unknown2:   longWord    {offset 7}
                        END;
```

Here's a little description of the spriteSpdDirData fields I've figured out:

spriteSpdDirData.speed—The speed at which the sprite will move. The higher the value, the faster the motion.

spriteSpdDirData.quadrant—The general direction of motion.

spriteSpdDirData.deltaX—A scaled representation of the absolute amount a sprite moves horizontally each interrupt. See samples below.

spriteSpdDirData.deltaY—A scaled representation of the absolute amount a sprite moves vertically each interrupt. See samples below.

If you turn a sprite on, then poke appropriate values into these memory locations, the C-128's vertical retrace interrupt mechanism will move the sprite for you. Very useful stuff. Here are some sample values that'll work; you should be able to find others via inspired inference and/or experimentation.

```
To move north:
    speed       =   $03
    unknown1    =   $00
    quadrant    =   $00
    deltaX      =   $0000
    deltaY      =   $FF7F

To move northeast:
    speed       =   $03
    unknown1    =   $00
    quadrant    =   $00
```

```
    deltaX          =   $2B5A
    deltaY          =   $2B5A

To move east:
    speed           =   $03
    unknown1        =   $00
    quadrant        =   $01
    deltaX          =   $FF7F
    deltaY          =   $0000

To move southeast:
    speed           =   $03
    unknown1        =   $00
    quadrant        =   $01
    deltaX          =   $2B5A
    deltaY          =   $2B5A

To move south:
    speed           =   $03
    unknown1        =   $00
    quadrant        =   $02
    deltaX          =   $0000
    deltaY          =   $FF7F

To move southwest:
    speed           =   $03
    unknown1        =   $00
    quadrant        =   $02
    deltaX          =   $2B5A
    deltaY          =   $2B5A

To move west:
    speed           =   $03
    unknown1        =   $00
    quadrant        =   $03
    deltaX          =   $FF7F
    deltaY          =   $0000

To move northwest:

    speed           =   $03
    unknown1        =   $00
    quadrant        =   $03
    deltaX          =   $2B5A
    deltaY          =   $2B5A
```

To save you a bit of arithmetic, here are the starting addresses for each sprite's spriteSpdDirData record:

Sprite 0	$117E	4478
Sprite 1	$1189	4489
Sprite 2	$1194	4500
Sprite 3	$119F	4511
Sprite 4	$11AA	4522
Sprite 5	$11B5	4533
Sprite 6	$11C0	4544
Sprite 7	$11CB	4555

Appendix H:
8563 VDC Registers

Register # Dec	Hex	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Short Description
0	$00	HT7	HT6	HT5	HT4	HT3	HT2	HT1	HT0	Horizontal Total
1	$01	HD7	HD6	HD5	HD4	HD3	HD2	HD1	HD0	Horizontal Displayed
2	$02	HP7	HP6	HP5	HP4	HP3	HP2	HP1	HP0	Horizontal Sync Position
3	$03	VW3	VW2	VW1	VW0	HW3	HW2	HW1	HW0	Vert/Horiz Sync Width
4	$04	VT7	VT6	VT5	VT4	VT3	VT2	VT1	VT0	Vertical Total
5	$05	-	-	-	VA4	VA3	VA2	VA1	VA0	Vertical Total Adjust
6	$06	VD7	VD6	VD5	VD4	VD3	VD2	VD1	VD0	Vertical Displayed
7	$07	VP7	VP6	VP5	VP4	VP3	VP2	VP1	VP0	Vertical Sync Position
8	$08	-	-	-	-	-	-	IM1	IM0	Interlace Mode
9	$09	-	-	-	CTV4	CTV3	CTV2	CTV1	CTV0	Character Total Vertical

Register # Dec	Register # Hex	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Short Description
10	$0A	-	CM1	CM0	CS4	CS3	CS2	CS1	CS0	Cursor Mode, Start Scan Line
11	$0B	-	-	-	CE4	CE3	CE2	CE1	CE0	Cursor End Scan Line
12	$0C	DS15	DS14	DS13	DS12	DS11	DS10	DS9	DS8	Display Start Address Hi
13	$0D	DS7	DS6	DS5	DS4	DS3	DS2	DS1	DS0	Display Start Address Lo
14	$0E	CP15	CP14	CP13	CP12	CP11	CP10	CP9	CP8	Cursor Position Hi
15	$0F	CP7	CP6	CP5	CP4	CP3	CP2	CP1	CP0	Cursor Position Lo
16	$10	LPV7	LPV6	LPV5	LPV4	LPV3	LPV2	LPV1	LPV0	Light Pen Vertical
17	$11	LPH7	LPH6	LPH5	LPH4	LPH3	LPH2	LPH1	LPH0	Light Pen Horizontal
18	$12	UA15	UA14	UA13	UA12	UA11	UA10	UA9	UA8	Update Address Hi
19	$13	UA7	UA6	UA5	UA4	UA3	UA2	UA1	UA0	Update Address Lo

Register # Dec	Hex	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Short Description
20	$14	AA15	AA14	AA13	AA12	AA11	AA10	AA9	AA8	Attribute Start Address Hi
21	$15	AA7	AA6	AA5	AA4	AA3	AA2	AA1	AA0	Attribute Start Address Lo
22	$16	CTH3	CTH2	CTH1	CTH0	CDH3	CDH2	CDH1	CDH0	Char Horz Total, Displayed
23	$17	-	-	-	CDV4	CDV3	CDV2	CDV1	CDV0	Char Vert Displayed
24	$18	COPY	RVS	CBRate	VSS4	VSS3	VSS2	VSS1	VSS0	Vertical Smooth Scroll
25	$19	TEXT	ATR	SEMI	DBL	HSS3	HSS2	HSS1	HSS0	Horizontal Smooth Scroll
26	$1A	FG3	FG2	FG1	FG0	BG3	BG2	BG1	BG0	Foregrnd, Backgrnd Color
27	$1B	AI7	AI6	AI5	AI4	AI3	AI2	AI1	AI0	Address Increment/Row
28	$1C	CB15	CB14	CB13	RAM	-	-	-	-	Character Base Address
29	$1D	-	-	-	UL4	UL3	UL2	UL1	UL0	Underline Scan Line

Register # Dec	Hex	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Short Description
30	$1E	WC7	WC6	WC5	WC4	WC3	WC2	WC1	WC0	Word Count
31	$1F	DA7	DA6	DA5	DA4	DA3	DA2	DA1	DA0	Data
32	$20	BA15	BA14	BA13	BA12	BA11	BA10	BA9	BA8	Block Start Address Hi
33	$21	BA3	BA2	BA1	BA0	BA3	BA2	BA1	BA0	Block Start Address Lo
34	$22	DEB7	DEB6	DEB5	DEB4	DEB3	DEB2	DEB1	DEB0	Display Enable Begin
35	$23	DEE7	DEE6	DEE5	DEE4	DEE3	DEE2	DEE1	DEE0	Display Enable End
36	$24	-	-	-	-	DRR3	DRR2	DRR1	DRR0	DRAM Refresh Rate
37	$25	-	HSync	VSync	-	-	-	-	-	Horz/Vert Sync Polarity

Appendix I: 8563 VDC Screen Colors

Color Nibble (in binary) R G B I	Decimal Equivalent	Color Name	BASIC 7.0 Color Number
% 0 0 0 0	0	black	1
% 1 1 1 1	15	white	2
% 1 0 0 0	8	dark red	3
% 0 1 1 1	7	light cyan	4
% 1 0 1 1	11	light purple	5
% 0 1 0 0	4	dark green	6
% 0 0 1 0	2	dark blue	7
% 1 1 0 1	13	light yellow	8
% 1 0 1 0	10	dark purple	9
% 1 1 0 0	12	dark yellow	10
% 1 0 0 1	9	light red	11
% 0 1 1 0	6	dark cyan	12
% 0 0 0 1	1	medium gray	13
% 0 1 0 1	5	light green	14
% 0 0 1 1	3	light blue	15
% 1 1 1 0	14	light gray	16
R G B I			

Note: The four color nibble bits correspond to the four video signals Red, Green, Blue, and Intensity. This is indicated by the letters R, G, B, and I in the chart above.

Appendix J:
8563 VDC Attribute Bytes

Each character position in an 8563 VDC display has an attribute byte. Each bit in the attribute byte controls an aspect of the character displayed at that position:

BIT		ATTRIBUTE IF SET TO 1
7	. . .	alternate character set
6	. . .	reverse video
5	. . .	underline
4	. . .	blinking
3	. . .	foreground color has red component
2	. . .	foreground color has green component
1	. . .	foreground color has blue component
0	. . .	foreground color has intensity component

Appendix K: Poke Codes

Poke code	Set 1	Set 2
0	@	@
1	A	a
2	B	b
3	C	c
4	D	d
5	E	e
6	F	f
7	G	g
8	H	h
9	I	i
10	J	j
11	K	k
12	L	l
13	M	m
14	N	n
15	O	o
16	P	p
17	Q	q
18	R	r
19	S	s
20	T	t

Poke code	Set 1	Set 2
128	@	@
129	A	a
130	B	b
131	C	c
132	D	d
133	E	e
134	F	f
135	G	g
136	H	h
137	I	i
138	J	j
139	K	k
140	L	l
141	M	m
142	N	n
143	O	o
144	P	p
145	Q	q
146	R	r
147	S	s
148	T	t

Poke code	Set 1	Set 2
64	─	─
65	♠	A
66	│	B
67	─	C
68	─	D
69	─	E
70	─	F
71	│	G
72	│	H
73	╮	I
74	╰	J
75	╯	K
76	└	L
77	╲	M
78	╱	N
79	┌	O
80	┐	P
81	●	Q
82	─	R
83	♥	S
84	│	T

Poke code	Set 1	Set 2
192	─	─
193	♠	A
194	│	B
195	─	C
196	─	D
197	─	E
198	─	F
199	│	G
200	│	H
201	╮	I
202	╰	J
203	╯	K
204	└	L
205	╲	M
206	╱	N
207	┌	O
208	┐	P
209	○	Q
210	─	R
211	♥	S
212	│	T

Poke code	Set 1	Set 2	Poke code	Set 1	Set 2	Poke code	Set 1	Set 2	Poke code	Set 1	Set 2
21	U	u	149			85		U	213		
22	V	v	150			86		V	214		
23	W	w	151			87		W	215		
24	X	x	152			88		X	216		
25	Y	y	153			89		Y	217		
26	Z	z	154			90		Z	218		
27	[	[	155			91	+	+	219		
28	£	£	156			92			220		
29	]	]	157			93			221		
30	↑	↑	158			94			222		
31	←	←	159			95			223		
32			160			96			224		
33	!	!	161			97			225		
34	"	"	162			98			226		
35	#	#	163			99			227		
36	$	$	164			100			228		
37	%	%	165			101			229		
38	&	&	166			102			230		
39	'	'	167			103			231		
40	(	(	168			104			232		
41	)	)	169			105			233		
42	*	*	170			106			234		

Poke code	Set 1	Set 2	Poke code	Set 1	Set 2	Poke code	Set 1	Set 2	Poke code	Set 1	Set 2
43	+	+	171			107			235		
44	,	,	172			108			236		
45	-	-	173			109			237		
46	.	.	174			110			238		
47	/	/	175			111			239		
48	0	0	176			112			240		
49	1	1	177			113			241		
50	2	2	178			114			242		
51	3	3	179			115			243		
52	4	4	180			116			244		
53	5	5	181			117			245		
54	6	6	182			118			246		
55	7	7	183			119			247		
56	8	8	184			120			248		
57	9	9	185			121			249		
58	:	:	186			122			250		
59	;	;	187			123			251		
60	<	<	188			124			252		
61	=	=	189			125			253		
62	>	>	190			126			254		
63	?	?	191			127			255		

Appendix L: SID Registers

SID starting address is 54272 ($D400)

Register number Decimal	Register number Hex	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	This register controls:	
0	$00	FR7	FR6	FR5	FR4	FR3	FR2	FR1	FR0	Low byte of frequency	Voice 1
1	$01	FR15	FR14	FR13	FR12	FR11	FR10	FR9	FR8	High byte of frequency	Voice 1
2	$02	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0	Low byte of pulse width	Voice 1
3	$03	—	—	—	—	PW11	PW10	PW9	PW8	High nibble of pulse width	Voice 1
4	$04	Noise	Pulse	Saw-tooth	Trian-gular	Test	Ring mod	Sync	Gate	Gate and wave-form control	Voice 1
5	$05	ATK3	ATK2	ATK1	ATK0	DCY3	DCY2	DCY1	DCY0	Attack/decay	Voice 1
6	$06	SST3	SST2	SST1	SST0	RLS3	RLS2	RLS1	RLS0	Sustain/release	Voice 1
7	$07	FR7	FR6	FR5	FR4	FR3	FR2	FR1	FR0	Low byte of frequency	Voice 2
8	$08	FR15	FR14	FR13	FR12	FR11	FR10	FR9	FR8	High byte of frequency	Voice 2
9	$09	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0	Low byte of pulse width	Voice 2
10	$0A	—	—	—	—	PW11	PW10	PW9	PW8	High nibble of pulse width	Voice 2
11	$0B	Noise	Pulse	Saw-tooth	Trian-gular	Test	Ring mod	Sync	Gate	Gate and wave-form control	Voice 2
12	$0C	ATK3	ATK2	ATK1	ATK0	DCY3	DCY2	DCY1	DCY0	Attack/decay	Voice 2
13	$0D	SST3	SST2	SST1	SST0	RLS3	RLS2	RLS1	RLS0	Sustain/release	Voice 2

Register number Decimal	Hex	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	This register controls:	
14	$0E	FR7	FR6	FR5	FR4	FR3	FR2	FR1	FR0	Low byte of frequency	Voice 3
15	$0F	FR15	FR14	FR13	FR12	FR11	FR10	FR9	FR8	High byte of frequency	
16	$10	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0	Low byte of pulse width	
17	$11	–	–	–	–	PW11	PW10	PW9	PW8	High nibble of pulse width	
18	$12	Noise	Pulse	Saw-tooth	Trian-gular	Test	Ring mod	Sync	Gate	Gate and waveform control	
19	$13	ATK3	ATK2	ATK1	ATK0	DCY3	DCY2	DCY1	DCY0	Attack/decay	
20	$14	SST3	SST2	SST1	SST0	RLS3	RLS2	RSL1	RLS0	Sustain/release	
21	$15	–	–	–	–	–	CFR2	CFR1	CFR0	Low 3 bits of cutoff/ center frequency	Filter/volume
22	$16	CFR10	CFR9	CFR8	CFR7	CFR6	CFR5	CFR4	CFR3	High 8 bits of cutoff/ center frequency	
23	$17	RES3	RES2	RES1	RES0	Filter external	Filter V3	Filter V2	Filter V1	Resonance/ filter	
24	$18	V3 silent	High pass	Band pass	Low pass	Volume 3	Volume 2	Volume 1	Volume 0	Filter mode/ volume	
25	$19	GPX 7	GPX 6	GPX 5	GPX 4	GPX 3	GPX 2	GPX 1	GPX 0	Game paddle X	Other
26	$20	GPY 7	GPY 6	GPY 5	GPY 4	GPY 3	GPY 2	GPY 1	GPY 0	Game paddle Y	
27	$21	V30 7	V30 6	V30 5	V30 4	V30 3	V30 2	V30 1	V30 0	Voice 3 oscillator	
28	$22	V3E 7	V3E 6	V3E 5	V3E 4	V3E 3	V3E 2	V3E 1	V3E 0	Voice 3 envelope	

Appendix M: SID Note Values

Octave	Note name	Frequency in hertz	SID freq. setting	High byte of SID freq. set.	Low byte of SID freq. set.
0	C	16.4	269	1	13
0	C#	17.3	284	1	28
0	D	18.4	302	1	46
0	D#	19.4	318	1	62
0	E	20.6	338	1	82
0	F	21.8	358	1	102
0	F#	23.1	379	1	123
0	G	24.5	402	1	146
0	G#	26.0	427	1	171
0	A	27.5	451	1	195
0	A#	29.1	477	1	221
0	B	30.9	507	1	251
1	C	32.7	536	2	24
1	C#	34.6	568	2	56
1	D	36.7	602	2	90
1	D#	38.9	638	2	126
1	E	41.2	676	2	164
1	F	43.7	717	2	205
1	F#	46.2	758	2	246
1	G	49.0	804	3	36
1	G#	51.9	851	3	83
1	A	55.0	902	3	134
1	A#	58.3	956	3	188
1	B	61.7	1012	3	244
2	C	65.4	1073	4	49
2	C#	69.3	1137	4	113
2	D	73.4	1204	4	180
2	D#	77.8	1276	4	252
2	E	82.4	1352	5	72
2	F	87.3	1432	5	152
2	F#	92.5	1517	5	237
2	G	98.0	1608	6	72
2	G#	103.8	1703	6	167
2	A	110.0	1804	7	12
2	A#	116.5	1911	7	119
2	B	123.5	2026	7	234
3	C	130.8	2146	8	98
3	C#	138.6	2274	8	226
3	D	146.8	2408	9	104
3	D#	155.6	2553	9	249
3	E	164.8	2703	10	143
3	F	174.6	2864	11	48
3	F#	185.0	3035	11	219
3	G	196.0	3215	12	143
3	G#	207.7	3407	13	79
3	A	220.0	3609	14	25
3	A#	233.1	3824	14	240
3	B	246.9	4050	15	210

Note Values

Octave	Note name	Frequency in hertz	SID freq. setting	High byte of SID freq. set.	Low byte of SID freq. set.
4	C	261.6	4291	16	195
4	C#	277.2	4547	17	195
4	D	293.7	4818	18	210
4	D#	311.1	5103	19	239
4	E	329.6	5407	21	31
4	F	349.2	5728	22	96
4	F#	370.0	6070	23	182
4	G	392.0	6431	25	31
4	G#	415.3	6813	26	157
4	A	440.0	7218	28	50
4	A#	466.2	7648	29	224
4	B	493.9	8102	31	166
5	C	523.3	8584	33	136
5	C#	554.4	9095	35	135
5	D	587.3	9634	37	162
5	D#	622.3	10208	39	224
5	E	659.3	10815	42	63
5	F	698.5	11458	44	194
5	F#	740.0	12139	47	107
5	G	784.0	12861	50	61
5	G#	830.6	13625	53	57
5	A	880.0	14436	56	100
5	A#	932.3	15294	59	190
5	B	987.8	16204	63	76
6	C	1046.5	17167	67	15
6	C#	1108.7	18188	71	12
6	D	1174.7	19270	75	70
6	D#	1244.5	20415	79	191
6	E	1318.5	21629	84	125
6	F	1396.9	22915	89	131
6	F#	1480.0	24278	94	214
6	G	1568.0	25722	100	122
6	G#	1661.2	27251	106	115
6	A	1760.0	28872	112	200
6	A#	1864.7	30589	119	125
6	B	1975.5	32407	126	151
7	C	2093.0	34334	134	30
7	C#	2217.5	36377	142	25
7	D	2349.3	38539	150	139
7	D#	2489.0	40831	159	127
7	E	2637.0	43258	168	250
7	F	2793.8	45831	179	7
7	F#	2960.0	48557	189	173
7	G	3136.0	51444	200	244
7	G#	3322.4	54502	212	230
7	A	3520.0	57743	225	143
7	A#	3729.3	61177	238	249
7	B	3951.1	64815	253	47

Appendix N: ANDing And ORing

ANDing and ORing are logical operations your Commodore 128 uses to play with bits and check on the truth of complex expressions. I'll try to give you a brief glimpse of how they work.

First, a few conventions:

—When the computer tries to decide whether a number is true or false, any nonzero number is considered true.

—When the computer looks over a comparison, and decides that the comparison is true, it assigns it the value −1. A false comparison is assigned the value 0.

Here's a brief program that illustrates these two conventions at work:

```
10 IF 8 THEN PRINT "8 IS TRUE"
20 IF 0 THEN PRINT "0 IS TRUE":
   GOTO 40
30 PRINT "0 IS FALSE"
40 PRINT (9 = 8)
50 PRINT (9 = 9)
```

Running the program will give these results:

```
8 IS TRUE
0 IS FALSE
0
-1
```

The Commodore 128 performs ANDing and ORing on numbers in the range −32768 to +32767. The numbers first have any fractional parts dropped, and then they're converted into 16-bit binary format. Here are some examples:

ORIGINAL VALUE	FRACTION DROPPED	16-BIT BINARY
−1	−1	1111 1111 1111 1111
254.75	254	0000 0000 1111 1110
513	513	0000 0010 0000 0001
0	0	0000 0000 0000 0000
15.4	15	0000 0000 0000 1111

Note that I have inserted spaces into the 16-bit binary values just to make them easier for humans to read.

When two numbers are ANDed together, they're first put into this chopped-off 16-bit binary format. Then corresponding bits are ANDed together according to the following arbitrary rules:

```
  0       0       1       1
AND 0   AND 1   AND 0   AND 1
-----   -----   -----   -----
  0       0       0       1
```

The result is then converted back to decimal form. Here are some examples of ANDing:

```
                       -1  decimal
                 AND    0  decimal
    -------------------------------
     1111  1111  1111  1111  binary
AND  0000  0000  0000  0000  binary
    -------------------------------
     0000  0000  0000  0000  binary
    -------------------------------
                        0  decimal

                      255  decimal
                 AND   15  decimal
    -------------------------------
     0000  0000  1111  1111  binary
AND  0000  0000  0000  1111  binary
    -------------------------------
     0000  0000  0000  1111  binary
                       15  decimal
```

In graphics and sound programming on the Commodore 128, ANDing is often used to turn certain bits in a register off. For example, if you wanted to turn off bits 4, 5, 6, and 7 in a register, you'd AND the register value with the number 15. Take a look at the last example to see why this is so.

When two numbers are ORed together, they're first put into the familiar chopped-off 16-bit binary format. Then corresponding bits are ORed together according to the following arbitrary rules:

(sound familiar?)

0	0	1	1
OR 0	OR 1	OR 0	OR 1
0	1	1	1

The result is then converted back to decimal form. Here are some examples of ORing:

				-1	decimal
			OR	0	decimal
	1111	1111	1111	1111	binary
OR	0000	0000	0000	0000	binary
	1111	1111	1111	1111	binary
				-1	decimal

				537	decimal
			OR	131	decimal
	0000	0010	0001	1001	binary
OR	0000	0000	1000	0011	binary
	0000	0010	1001	1011	binary
				67	decimal

In graphics and sound programming on the Commodore 128, ORing is often used to turn certain bits in a register on. For example, if you wanted to turn on bits 0, 1, and 7 in a register, you'd OR the register value with the number 131. Take a look at the last example to see why this is so.

So much for a brief look at ANDing and ORing. They're really quite remarkable functions. In fact, your Commodore computer spends most of its time, at its deepest subconscious levels, ANDing and ORing away several million times each second.

Appendix O: Merlin-128 Pseudo-Ops

Pseudo-ops are fake assembly language instructions that let you control an assembler and the code it produces. Each assembler has its own pseudo-ops. The Merlin-128 assembler has a particularly rich set. Here's a list of the few I've used in this book's source code, along with brief explanations, examples, and usage tips:

DCI— Tells the assembler to put a string of C-ASCII character codes into the program, with the hi bit (bit 7) of the last character set to 1. Setting this bit makes it easy for routines to know when they've come to the end of a string. The Commodore machines use this format to store the text of BASIC commands. And that's how it's used in this book's programs, to add commands to BASIC.

DDB— Tells the assembler to put an actual word-sized value into the code stream, hi-byte first. For example, DDB 310 tells the assembler to put the values 1 and 54 into the code, in that order. (Think about it.) Not used too often, since the standard 6502 word-ordering is lo-byte first. But it comes in handy where there's an assembly language data interface with BASIC's variables, parts of which are stored this way.

DFB— Tells the assembler to put an actual byte-sized value into the code stream. For example, DFB %10101111 tells the assembler to put the binary value %10101111 into the code. And DFB 150 tells it to insert the decimal value 150. Used to set up constants.

DS— Tells the assembler to reserve a number of bytes of storage. For example, DS 1 tells it to reserve one byte of space, and DS 4 tells it to reserve four bytes. Used to set up variables.

HEX— Tells the assembler to put an actual byte-sized hexadecimal value into the code stream. Unlike other assembler commands, you don't need a $ to indicate hexness. For example, HEX 25 tells the assembler to put the hexadecimal value $25 into the code. Used to set up constants that are best expressed in base 16.

ORG— Tells the assembler where the next instruction should be compiled to run at. For example, ORG $1300 tells the assembler the next instruction should be compiled to run at memory location $1300. It's usually used once, at the beginning of a program.

PAG— Tells the assembler to output a formfeed command. I use this command so each part of a multi-file program starts printing on a fresh page. Doesn't affect the code at all. You won't see PAG commands in the listing; Merlin doesn't print their lines. You will see them if you purchase the program disks.

PUT— Tells the assembler to grab another source code file and use it to continue the current assembly. This pseudo-op lets you assemble programs whose source code is too large to fit into the computer in one chunk. For example, PUT "GRAFIX 80 2.S" pulls in a source code file named Grafix 80 2.S. If your assembler doesn't have this facility, you can still put together large programs, but you'll have to do a lot of grubwork.

TTL— Tells the assembler to use a particular string as a title at the top of each page of a listing. For example, TTL "Grafix 80 2.S" will put the title Grafix 80 2.S at the top of each subsequent page of a listing. This pseudo-op is particularly useful if you want to change a listing's title partway through the listing.

Appendix P: Last Minute Program Adjustments

As this book went to press, I found some adjustments that should be made to the programs. They're minor in the sense that the programs work just fine without them. And it's too late to go back and reprint the listings. But my aesthetic sense is a vicious thing, and won't let me relax into an escape from truth.

Adjustment 1

Line 1990 of the program G80 Test Suite should have its comment adjusted, from

```
REM   FIVE TESTS (2 VARIANTS EACH)
```

to

```
REM   SIX TESTS (2 VARIANTS EACH)
```

Adjustment 2

Similarly, Line 1970 of the program G40 Test Suite should have its comment adjusted, from

```
REM   FIVE TESTS (2 VARIANTS EACH)
```

to

```
REM   SIX TESTS (2 VARIANTS EACH)
```

Adjustment 3

Line 212 of Grafix 80 4.S should be changed from

```
:Tstl     LDX #FETokFlg     ;our commands start with FE
```

to

```
:Tstl     CPX #FETokFlg     ;our commands start with FE
```

Remarkably, the program works with the LDX. Can you figure out why? That stroke of good/bad luck is why I snoozed thru this typo.

Adjustment 4

Line 196 of Grafix 80 4.S should have its comment adjusted, from

```
; if sum is <15, it's two part
```

to

```
; if sum is <16, it's two part
```

Adjustment 5

Here's another strangey. Grafix 80 5.S has a routine that clears the 80-column graphics screen via a series of block writes. It was one of my first 80-column graphics routines, and it works, but the block write procedure is slightly incorrect. Beats me why it works.

Anyways, when I wrote Section 3.2.34 and got to thinking seriously about block writes, I realized I'd screwed up this old routine. Here's how to fix the offending code. Lines 66 thru 70 of Grafix 80 5.S currently read

```
66   LDX  #DataReg         ;store 0 as the data to
67   JSR  VDCRegPoke       ;. . . be written
68
69   LDX  #BytCntReg       ;tell the chip to store
70   JSR  VDCRegPoke       ;. . . 256 copies of it
```

The correct version replaces line 68 with seven new lines. I also change the last two lines' comments. The corrected code looks like this (The new lines' numbers are in boldface):

```
66   LDX  #DataReg         ;store 0 as the data to
67   JSR  VDCRegPoke       ;. . . be written
68                         ;. . . and write one byte
69   LDX #24               ;this reg controls block
70   JSR VDCRegPeek        ;. . . stuff via bit 7
71   AND #01111111         ;clear bit 7 for block write
72   JSR VDCRegPoke        ;store set reg
73
74   LDA  #255             ;write 255 more bytes
75   LDX  #BytCntReg       ;. . . for a total of 256
76   JSR  VDCRegPoke       ;write those bytes
```

This fix adds twelve bytes to the Grafix 80 object code.

Index

Index

C

D

E

T

U

V

W

ISBN 0-8306-0930-X