The
Puzzle Factory
presents:

ReSource

Intelligent
Interactive
Disassembler
For Amiga Programmers

Glen McDiarmid was born in Queensland, Australia in 1959. He left high school in 11th grade, and worked at various jobs. Never satisfied with any of them, he finally joined the Air Force in 1978, and trained as an instrument fitter. In 1981 Glen bought his first computer, a TRS80 Model 1 with 4K RAM, from a co-worker. He tried using BASIC, but soon bought a book on assembly language. After unsuccessfully trying to get assigned to a job involving computer programming, he left the Air Force when his contract ran out in 1984. Through reading books and experimenting, he taught himself the basics of programming. Since then, all his spare time has been devoted to computing. The Model 1 became a Model 3, then a Model 4P. He wrote "Macasmon", a Z80 debugger/monitor program, and "Time Machine", a utility which to this day defies description. In 1987, he bought an Amiga 1000, learned 68000 assembler, and started writing ReSource in January, 1988. Glen is currently studying at Ipswich College of Technical and Further Education.

The Puzzle Factory is a consortium of programmers and users of small computers. We were troubled by the way most software is sold. The programmer gets crumbs, and you, the purchaser, are seen only as the consumer of a product. We felt there had to be a better way.

We seek out exceptional programs, then sell direct and share proceeds from sales equally with the authors. This means we eliminate expensive distributors, and the author is directly involved with program support and bug fixes. Expect the best software tools and product support at reasonable prices from The Puzzle Factory.

Copyright © 1991 The Puzzle Factory Printed in U.S.A.

The Puzzle Factory, Inc.
presents:

ReSource™ Reference Manual

Intelligent Interactive Disassembler for Amiga® Programmers

Printed in the U.S.A.
Second edition

Copyright

ReSource and associated utility software are copyrighted ©1988-1991 by Glen McDiarmid. All rights Reserved. This manual is copyrighted ©1991 by The Puzzle Factory, Inc. All rights Reserved. Under the copyright laws, this manual or the software may not be copied, in whole or in part, without the written consent of the copyright holder except in the normal use of the software or to make a backup copy.

The Puzzle Factory is the assumed business name of The Puzzle Factory, Inc.

Amiga Workbench Version 1.3
Copyright ©1985-1991 Commodore-Amiga, Inc. All Rights Reserved.

Amiga Includes Version 2.0
Copyright ©1985-1991 Commodore-Amiga, Inc. All Rights Reserved.

Trademarks

ReSource is a Trademark of The Puzzle Factory, Inc.

Macro68 is a trademark of DigiSoft Pty. Ltd.

Amiga, AmigaDOS, Kickstart, Intuition, and Workbench are trademarks of Commodore-Amiga, Inc.

All products mentioned in this manual are trademarks of their respective owners.

Disclaimer

The Puzzle Factory, Inc. provides this program "as-is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the results and performance of the program is assumed by you. Should the program prove defective, you (and not The Puzzle Factory, Inc.) assume the entire cost of all necessary servicing, repair or correction. Further, The Puzzle Factory, Inc. does not warrant, guarantee or make any representation regarding the use of, or the results of the use of, the program in terms of correctness, accuracy, reliability, currentness, or otherwise; and you rely on the program and results solely at your own risk.

The material contained in this documentation is believed to be accurate, but The Puzzle Factory, Inc. reserves the right to update the software and/or documentation at any time without notice.

Distribution

ReSource is available from:

The Puzzle Factory, Inc.
P.O. Box 986
Veneta, OR 97487
U.S.A.
(503) 935-3709

and local distributors and dealers worldwide. Contact us for the name of a dealer near you.

Update & Support Policy

Our update policy is very simple: To be able to obtain updates of the program, or technical support, you must complete and return the warranty registration card.

For updates within a version (revision changes only), send us a letter with your name, address, and registration number along with $10, and we will return the latest revision.

For major updates, if you have returned the registration card, you will be notified by mail about price and ordering information. Otherwise, please contact us for instructions on how to get the latest version.

Technical support is provided free of charge to registered owners. Ttechnical support is available from 10AM to 5PM (PST), Monday through Friday, at (503) 935-3709.

Introduction to ReSource

ReSource is an interactive disassembler for Amiga computers. There are other Amiga disassemblers, both public domain and commercial. The following points characterize ReSource as distinct from other disassemblers:

1. ReSource is fully interactive.
2. ReSource was written for the serious user.
3. ReSource understands the Amiga®.

Because ReSource can get its input from a file, track, or memory, and output all or in part to any of these, it is more a general purpose tool than just a disassembler. It can be used to browse through files, without ever bothering to save anything, just to find out what is inside. It also knows quite a bit about AmigaDOS™ and the Amiga® OS. This built-in intelligence will save you a great deal of time when disassembling files. ReSource is also blindingly fast. It is written entirely in assembly language, and does all text rendering using its own special internal routines. The following are some of ReSource's features:

ReSource is easy to work with. ReSource will amaze you with its ease of use and straightforward user interface. Unlike other disassemblers, ReSource is fully interactive, and allows you to "do it your way". There are plenty of options, and decisions are not forced on you.

ReSource is fast and thorough. ReSource was designed to be used by serious software developers. ReSource is aimed at those that need to disassemble large programs into source files that will reassemble and run correctly. For a disassembler to achieve this, it must be able to produce totally bug-free output. Furthermore, it must be able to produce high-quality output with the minimum of effort from the user. ReSource achieves both of these objectives, albeit at the cost of large memory usage.

ReSource makes the most of your programming skills. Because ReSource understands AmigaDOS™ and the Amiga® OS, you can concentrate on the task at hand, and not expend effort on trying to look up or remember details better left to a computer.

Before you go further

Before you begin using ReSource, please take a few minutes to:

Back up the ReSource disks. After you have backed up your disks, store the originals in a safe place in case you need to revert to them at a later time.

Fill out the enclosed warranty registration card. You are provided with two stickers that contain your unique registration number. Affix one sticker to the warranty registration card and the other to your ReSource master disk. Then mail the warranty registration card. Technical support is free, but only if your warranty card is on file. When you call for technical support, please have your registration number ready in case you are asked for it.

ReSource Documentation

The documentation for ReSource is presented as a tutorial manual and an interactive online help system. Previously, one printed manual contained both a tutorial and all reference material, and last minute changes were documented in a file on the distribution disk. We found that there were so many enhancements made to ReSource that the printed manual became out of date to a large extent, requiring a very large file to describe changes done between the time the manual was printed and when the software was updated.

Putting all reference material in the online help system is better, as it makes it easier to find documentation about any specific function very quickly, as well as having instant references to key words inside the online help system itself. It also makes it possible to keep the reference material up to date with every new release of ReSource.

About this Manual

Your ReSource manual is divided into several sections that follow this introduction:

ReSource - An Overview - Read this first to learn how ReSource works and how it can help you deal with technical problems on the Amiga®.

Getting Started - Basic techniques and information to begin working with ReSource. This section includes information about installing ReSource, contains an overview of the basics

of ReSource, and features a short tutorial to show you how easy it is to begin using ReSource.

Working with ReSource - Instructions for using ReSource, including using the online help facility, starting ReSource, setting your preferences, loading and saving files, and a general tour of the major functions available.

Disassembling techniques - This section explains how labels and symbols are used in ReSource, what kind of documentation facilities are available, and how ReSource helps you to identify different areas of the code.

An In-Depth Tutorial - This section is real immersion training. We disassemble a working program. Along the way, we learn about base-relative addressing, symbol bases, macros, comments, data structures, and more.

Advanced Topics - In this section we cover some of the more complex things that you may want to do with ReSource in an easy question and answer format.

Appendices - Appendix A covers the differences between old and new Motorola M68000 Family syntax. Appendix B contrasts strict and relaxed syntax modes.

Note that although ReSource is capable of outputting assembly source in either old or new syntax, all examples in this manual are given in new syntax. As none of the more complex addressing modes used by the 68020/68030 processors are used in the examples, we feel that even beginners should have no trouble understanding the code as presented.

Table Of Contents

ReSource - An Overview

Overview 1-1
Limitations 1-3
- Memory Usage 1-3
- Saving Executables 1-3
- Raw Tracks 1-4

Getting Started

Before Installing 2-1
- System Requirements 2-1
- Making Backup Copies 2-2
- Update List 2-2

Installation 2-2
- Contents of Disks 2-2
- ReSource Libraries 2-3
- Installing on a Hard Disk 2-3

Basics 2-4
- Basic terms 2-4
- Starting ReSource from a Floppy 2-5
- Starting ReSource from a Hard Disk 2-5
- Working in ReSource 2-6
  - The Titlebar 2-6
  - About Menus 2-6
  - About Keys 2-7
  - About the Mouse 2-8
  - About the Display 2-9
  - Definition of the Current Line 2-10
  - Quitting ReSource 2-11

A Short Tutorial 2-12
- Start ReSource 2-12
- Opening the Load File 2-12
- Disassembling a Program 2-12
  - Using the Built-In Disassembly Function 2-13
  - Creating Symbols 2-15
  - Creating Labels 2-16
  - Following Code Flow 2-20
  - The Finished Example 2-21

Working with ReSource

Online Help 3-1
- Hyper-help 3-1
- Help on Help 3-2

Starting ReSource 3-2
- The Command Line 3-2
- Icon Tooltypes 3-4

Input and Output 3-5
- About Load Files 3-5
- About Data Files 3-8
- Other Input 3-9
- About Output Files 3-10

Display Preferences 3-11

About Searches 3-11

About Macros 3-13
- Suspend learn 3-14
- Conditionals 3-14
- Labels in Macros 3-14
- Strings in Macros 3-15
- String Indirection 3-15
- Commentary Level 3-16
- Wait on Mouse 3-17
- Aborting Macros 3-17
- Demo Macros 3-17

User-defined Symbol Bases 3-18

Utilities 3-18
- ShowMacros 3-19
- ShowKeys 3-19

Disassembling Techniques

About Labels 4-1
- How Labels are used in ReSource 4-1
- Duplicate Labels 4-2
- Local labels 4-2

About Symbols 4-3
- How Symbols are used in ReSource 4-3
- Symbol equates 4-4
- Operand Fields 4-4

Comments 4-4
- End-of-line Comments 4-5
- Full-line Comments 4-5

Code Identification 4-5
- Hiliting 4-5
- Attribute Bits 4-6

About Data Types 4-7

An In-Depth Tutorial 5-1

Advanced Topics 6-1

Credits

Appendices

Bibliography

Index

ReSource - An Overview

Overview

ReSource is an intelligent, interactive disassembler for the Amiga® computer. Written entirely in assembly language, ReSource is fast and powerful. Intelligent, because it understands AmigaDOS™ and the Amiga® OS. This built-in intelligence will save you a great deal of time when disassembling files. Interactive, because ReSource allows you to decide specifically how you want to display code and data, and gives you a chance to change your decisions as new information becomes available.

ReSource can get its input from a file, disk track, or memory, and output all or in part to any of these. If the input is a load file, ReSource will make use of all the reloc32 and symbol information in the file.

ReSource creates symbols and labels for you, either manually or automatically, to take much of the drudgery out of disassembly.

```
MOVEA.L  (4),a6
LEA      (START+$06B6,PC),A1
JSR      (-$0198,A6)
MOVE.L   D0,(START+$2248)
BNE.B    START+$6AE
MOVEA.L  D0,A6
MOVE.L   #$3EE,D2
LEA      (START+$06C2,PC),A0
MOVE.L   A0,D1
JSR      (-$1E,A6)
TST.L    D0
BEQ.W    START+$1044
```

becomes

```
MOVEA.L  (AbsExecBase),a6
LEA      (DosName,PC),A1
JSR      (_LVOOldOpenLibrary,A6)
MOVE.L   D0,(_DosBase)
BNE.B    NoDos
MOVEA.L  D0,A6
MOVE.L   #MODE_NEWFILE,D2
LEA      (ThisCON,PC),A0
MOVE.L   A0,D1
JSR      (_LVOOpen,A6)
TST.L    D0
BEQ.W    NoFile
```


Note that this and other examples of ReSource output are all shown using Motorola's new M68000 Family assembly language syntax. The new syntax was developed specifically for the addressing modes available on the 020/030 CPUs, and output from ReSource in this mode is completely compatible with the Macro68 assembler.

Virtually all Amiga V1.3 and V2.0 symbol bases are included, over 7000 symbols in all. User-defined symbol bases may be used, allowing the disassembly of even non-standard structures to be automated. Your symbol bases may be edited, too.

Full key rebinding is supported. You may load or save keytables of your choice to be used at any time.

You may load, define, and save macros. Macros support absolute looping, conditional looping, and variable execution speed.

A number of numeric and text processing functions allow you to automate many of the chores that used to make disassembling so slow! ReSource supports code, longword, word, byte and ASCII data types, as well as binary, decimal, hexadecimal, and ASCII numeric bases.

In order to allow you to move freely around in code or data, ReSource includes forward and backward referencing, to allow you to quickly examine referenced code/data, and return to your original position (nested references are supported to 127 levels). Very sophisticated and flexible search functions are provided. You can search the buffer, or a disassembly of the buffer. Search strings may contain wildcards, and a binary search mode is supported. You can search forwards, backwards, or bi-directionally. You may restrict your search to just labels, or just symbols.

There is special support for base-relative data referencing. Any address register may be selected to hold the base, which makes ReSource particularly valuable for work with compilers that use unusual base-registers, such as Modula2.

ReSource can simplify your work in a variety of ways. You can patch files on disk or programs in memory, even while running. You can locate image data in memory, convert it to binary, then save to a file and include in your program: instant gadget images! ReSource has even been used to reconstruct MFM data on corrupted disks. You may also inspect and perhaps alter boot blocks.

Before leaving this section, a brief look at how ReSource achieves its fast screen output is in order. All text that is displayed on the ReSource screen, with the exception of menus and requesters which are rendered by Intuition, uses text rendering routines internal to ReSource. This makes text rendering extremely fast, but also means that the font can't be easily changed. The text rendering routines were originally inspired by Warptext II. Fast, smooth scrolling is made possible by direct hardware access of the blitter.

Limitations

Even though ReSource has exceptional flexibility and power, no program can be all things to all people. We note for you that ReSource has the following limitations.

Memory Usage

When loaded with a file of the minimum size, ReSource uses a total of 280K of memory. Loading a larger file will require additional memory. The following example shows how much memory, above the initial 280K, is required for a small executable:

File: "DiskCopy"	
Size on disk	12K
Size loaded as executable	79K
With labels created (min. necessary for reassembly)	84K
With many symbols created, some custom labels	140K

As the file size increases, above the initial requirements, memory usage increases linearly. In other words, if you loaded a file ten times the size of "DiskCopy", it would require around 10 times as much memory as would "DiskCopy".

In the design of any program, there must be compromises. ReSource was designed such that whenever there was a compromise between anything and memory usage, memory usage always lost. Therefore, ReSource is fast and powerful, but very memory hungry. For several reasons, this was considered not to be a great problem. Memory prices are still dropping, and it is expected that the users who need ReSource the most will have plenty of memory.

Saving Executables

ReSource is generally able to re-create executables, thus saving you the time and effort of going through the disassemble/reassemble/link process. There are a few programs, though, that make references to memory near, but not inside of, one of the file's hunks. In some cases, ReSource is not able to correctly determine which hunk is assocated with the reference, and will not be able to create a functional executable. When this happens, saving to a ".asm" file and adjusting the reference manually will be necessary to recreate a working executable. Note that this is a subtle problem, and you will generally not be notified in the event of an executable that won't run properly. If you think that this may be the case, you should try to arrange for maximum safety of files, etc., when attempting to run the newly created program.

Raw Tracks

ReSource can read only normal disk blocks, and is not able to read MFM or CGR data directly. In order to look at raw data from disk, you must first use a program that is able to read raw data, and pass the address of this program's buffer to ReSource's "Disassemble memory" function. Once this is done, you are able to save to a file, which can be loaded in the normal fashion at any time.

Getting Started

This section is designed to get you started with ReSource. It will guide you through installing ReSource on either a floppy based Amiga®, or a system with a hard drive. It will explain the basics of operating ReSource, and walk you through a short tutorial for an overview of how to work with ReSource. This section is intended to give you a quick understanding of ReSource so you can begin using it right away.

Here is what you will find in this section:

Installation - Before you use ReSource, you should read this section to make sure your ReSource package is complete, to verify you have the proper Amiga® hardware to operate ReSource, and to ensure that ReSource is ready to be used by your computer.

Basics - This contains an overview of the basics of ReSource. You will learn how to start ReSource, how to use its user interface, and how to quit ReSource.

Tutorial - This section contains an easy to follow lesson using an actual file to show how to work with ReSource.

Before Installing

1. Make backup copies of the ReSource disks, and store the originals in a safe place.

2. Fill out your registration card and mail it. Technical support is free for ReSource, but your registration card must be on file, and you may be asked for the ID number when you call.

System Requirements

ReSource is designed to work on any properly configured Amiga® 500, 1000, 2000, 2500, or 3000.

Version 1.3 or higher of the Amiga® OS is required.

One megabyte of RAM is required, and you will need 1 1/2 to 2 megabytes of RAM if you want to work with files larger than 30-40K.

Because ReSource uses functions in the "ARP" library, you must

have the file "arp.library" in your "LIBS:" directory when you run ReSource. This file is supplied with ReSource, and is also available from most BBS's. The version of "arp.library" must be 34 or higher.

If you will be using ReSource under V1.3 of the Amiga® OS, we suggest that you run FastFonts© or Blitzfonts©. Otherwise, scanning through the menus will be very sluggish due to the large number of sub-items.

Making Backup Copies
Your license agreement authorizes you to make backup copies of the ReSource disks. If you have not already backed up your original ReSource disks, do so now.

Update List
If the version of ReSource you have purchased has been changed since the publication of this manual, there will be a file named "History.doc" on your ReSource disk containing incremental documentation. A "ReadMe" file may also be present containing important information about your particular release of ReSource. Please take the time to read these files.

Installation

If your Amiga® has a hard disk drive, you will need to install ReSource on the hard drive. If your Amiga® does not have a hard disk drive, no installation is necessary.

Contents of Disks
Although the distribution of files may change in the future, at the time of this writing, the ReSource release disks contained the following files:

ReSource1:	
:Install	Installation information and script
:ReadMe	Important information about this release
:ReSource	The main disassembler
:ReSource.info	icon for ReSource
:ShowKeys	Utility for viewing keytables
:ShowKeys.doc	Documentation for ShowKeys
:ShowMacros	Utility for viewing and editing macros
:ShowMacros.doc	Documentation for ShowMacros
:libs/arp.library	Library required for ReSource
:libs/ReSourceloader.library	Library required for ReSource
:libs/ReSourcemenus.library	Required for access to menus
:libs/ReSourcesyms.library	Required for access to internal symbol bases
:libs/ReSourcehelp.library	Required for online help facility
:s/RS.keytable	Default keytable
:s/RS.macros	Example macros

ReSource2:	
:Docs/	
History.doc	Optional incremental documentation
Example_Macros.doc	Documentation for example macros
Libraries.doc	Description of ReSource's libraries
:UserSymbols/	
CIA_Regs.symbols	User-symbol base for CIA registers
CIA_Regs.symbols.asm	Source code for above
JFuncs.symbols	Janus Library functions user-symbol base
JFuncs.symbols.asm	Source code for above
JServices.symbols	Janus service numbers user-symbol base
JServices.symbols.asm	Source code for above
UserSymbols.doc	How to create user-defined symbol bases
:Offsets/	All the V2.0 library offsets
:Includes/	All the V2.0 include.i files

ReSource Libraries
You may have noticed that ReSource consists of the main executable, ReSource, and several libraries. The benefits of using libraries can be enormous. There are savings in both load time and memory usage to be had every time you use ReSource. One of ReSource's libraries is specially designed to allow you to edit many of the strings in ReSource's menus, and other internal structures. For systems that are tight on memory, some libraries have been designed to be optional. For a full description of each of ReSource's libraries, see the "Libraries.doc" file on the "ReSource2" disk.

Installing on a Hard Disk
If you have a hard drive on your Amiga®, follow the procedure below to install ReSource. This installation does not affect any of your system files, it only does what is necessary to run ReSource from your hard disk.

WARNING: You will need 2-3 megabytes of disk space on your hard drive for the ReSource program and related utilities and files. For full details on how to install ReSource on a hard disk, please refer to the "Install" doc file, located on the "ReSource1" disk.

Following is a quick guide to a minimum installation:

1. Turn on your Amiga® and your hard drive if they are not already on.

2. If prompted for a "Kickstart" disk, insert your "Kickstart" disk.

3. If prompted for a Workbench™ disk, insert the disk you normally use to start your Amiga®.

4. After the computer completes its start up, place the "ReSource1:" volume into any drive. Assuming that your hard disk device name is "DH0:", and that "DH0:c" is in your execution path, type the following lines:

```
copy ReSource1:ReSource#? DH0:c
copy ReSource1:ShowKeys DH0:c
copy ReSource1:ShowMacros DH0:c
copy ReSource1:libs DH0:libs ALL CLONE
copy ReSource1:s/RS.#? DH0:s CLONE
```

ReSource is now installed on your hard drive, and ready to work for you.

Basics

This section contains essential information about ReSource as well as explanations of basic terms used in this manual. Included are steps for starting ReSource from a floppy or a hard disk, information about ReSource's user interface, and instructions for quitting ReSource.

Basic Terms

The following are some basic terms used throughout this manual. Experienced Amiga® users may wish to skip this section and resume reading on the next page.

Point	To move the mouse so that the mouse pointer is either pointing at or pointing on top of some object.
Press	To hold the mouse button down.
Click	To press and quickly release either the left or right mouse button.
Double click	To click the left mouse button twice quickly.
Drag	To press the left mouse button and move the pointer in some direction while still holding the mouse button down.

Multiple select	In this manual, multiple selection will always be referring to menus. Access the menu with the right mouse button. Instead of releasing the right button over just one menu item, position the mouse pointer over each desired menu item and press the left mouse button to select that item.
shift	Refers to a particular shift key, which will always be indicated, i.e. left shift.
ctrl	Refers to the control key. "ctrl-s" means to hold down the control key, and while holding it down, press the "s" key. Then release the control key.
alt	Refers to either the left or right Alt key. "Alt-z" means to hold down the Alt key, and while holding it down, press the "z" key. Then release the Alt key.
kp	Refers to keys on the numeric keypad, as in "kp1".

Starting ReSource from a Floppy

If you are operating an Amiga® without a hard drive the following instructions will enable you to start ReSource from a floppy disk. If you have a hard drive and have installed ReSource on it, please skip to the next section.

1. Turn on your Amiga®, if it's not already on, and insert the "ReSource" disk when prompted for the Workbench™. The "ReSource" disk is a Workbench™ disk.

2. If you will be starting ReSource from the CLI, at the prompt type:

```
1> run ReSource          ;the word "run" is optional.
```

   If you will be starting ReSource from the Workbench™, you will see several icons, including the "ReSource" program icon. Double click on the "ReSource" program icon.

Starting ReSource from a Hard Disk

If you have installed ReSource on your hard disk, the following steps will enable you to start the program. These instructions assume that you have successfully installed ReSource on a hard drive using the instructions earlier in this section.

1. Turn on your Amiga® and follow the normal startup procedures you go through to start your hard drive.

2. Once you are running on your hard drive:

   If you will be starting ReSource from the CLI, at the prompt type:

   ```
   1> run ReSource          ;the word "run" is optional.
   ```

   If you will be starting ReSource from the Workbench™, double click on the "C" drawer. Shortly you will see several icons, including the "ReSource" program icon. Double click on the "ReSource" program icon.

Working in ReSource

Once you have started ReSource, you will see a screen with several lines of disassembled code. This is the default ReSource screen. The following are descriptions of the various elements of the user interface:

The Titlebar

The default behavior of the titlebar is to show the name of the program, ReSource, the file position as a percentage and as an absolute offset from the beginning of the file, and the name of the file. In this case, because a file wasn't specified, "MEMORY" is displayed as the file name, and sixteen bytes of ReSource itself are displayed on the screen, as code.

About Menus

Because of the large number of functions in ReSource, it was decided to use one-and two-character menu names, so that more menus could be used. Looking from the left, the first menu is "P", which stands for "PROJECT". The real menu name is directly under the one-or two-character name, in capital letters, to be easy to find. In all future reference to menus, the full menu name will be used.

Throughout this manual, a heading ending with a ":" refers to the lowest hierarchical menu item for that function. In some cases, the heading will end with a "/:" to indicate that there are several sub-items with related functions being discussed, e.g.:

"DISPLAY/Set Counter:"	Refers to a single item.
"DISPLAY/Hiliting/:"	Refers to a group of sub-items.

In both this manual, and the online help, functions will generally be referenced by menu, rather than by key name even though you will probably want to use single keys to access many of the functions. For example, the "Quit" function will be described as "PROJECT/Quit:". The function to restore the current file will be "PROJECT/Restore:". You will find both of these under the "PROJECT" menu. Functions in sub-menu boxes will be described in a similar way, e.g., "SAVE/Tabs/Spaces:".

The menus in ReSource fully support drag-selecting, and multiple selection.

Before we leave the subject of menus, there is one function that can save you a lot of time. "SPECIAL FUNCTIONS/Repeat last command" is normally bound to the spacebar. This function repeats the last command that was initiated from a menu. It does not repeat commands that came from keys. What this can do for you, is that if you will be using a particular function quite a bit around the piece of code on which you are currently working, you can call that function from the menu. Now that function may be repeated just by hitting the spacebar, while you can still use keys to call other functions in between. This will remain true until another menu item is selected.

About Keys

Any function can be bound to any key. All non-qualifier keys can be used, with or without any of the following combinations of qualifier keys:

left shift or right shift
left alt or right alt, treated as one
ctrl, shift-ctrl, alt-ctrl, shift-alt, shift-alt-ctrl
left-Amiga, right-Amiga

However, to make it easier to get started, some key bindings will be present, even without loading a keytable. Note that loading a keytable may override these key bindings:

up arrow	means	scroll up one line
down arrow	means	scroll down line line
shift-up arrow	means	page backward
shift-down arrow	means	page forward
right arrow	means	forward reference
left arrow	means	previous location
rightAmiga Q	means	quit
rightAmiga O	means	open a file

A sample keytable is provided for your use. The key mapping used in the keytable is described in a file named "ShowKeys.doc". You may also use the supplied program "ShowKeys" to read your keytable at any time. For more information, see the above file, as well as the Utilities section.

About the Mouse

ReSource uses some rather sophisticated scrolling techniques to move smoothly through a file. The mouse may be used to set both scrolling direction and speed. Press and hold down the left mouse button while the pointer is near the vertical center of the ReSource screen. Gently move the mouse toward the bottom of the screen, and text will start moving up the screen continuously. If you move the mouse toward the top of the screen, text will move down. The farther you move the mouse pointer vertically from where you started, the greater the scrolling speed.

At the slowest rate, the display is shifted one pixel at a time followed by a short delay. The second slowest rate is the same number of pixels with no delays. The third moves 2 pixels at a time, fourth slowest is 4 pixels at a time, and the fastest rate is 8 pixels at a time. It is possible to select any of these rates by using menus or keys. Mouse scrolling also lets you scroll at rates faster than 8 pixels at a time. By moving the mouse vertically further away from the point where the left button was pressed, up to several hundred lines of text will be skipped between display refreshes.

Several functions can be used during mouse scrolling; one example of this is automatic label creation. To do this, hold down the left-Amiga key while scrolling. Note that under V2.0 of the OS, the mouse button must be pressed before the left-Amiga key. This gives close control over the disassembly process; you examine the code as it scrolls up and ReSource creates the necessary labels.

If you are unsure which data type, code, ASCII, bytes, words, or longwords, should be used for any part of a file, you can have the display lock on one of these display types temporarily. While holding the left mouse button, even while scrolling, press and hold down the control key. Everything will be shown as code, regardless of how it is normally shown. Press the left-shift key, and everything will be shown as bytes. Similarly, the left-alt key will give you ASCII, left-shift and left-alt together will give you words, and all three, left-alt, left-shift, and ctrl, will display everything as longwords.

If you wish to change the data type permanently, you would normally use one of the "DISPLAY/Set data type/:" functions, but while using the mouse and the above qualifier keys, it is also possible to set the data type by pressing the right mouse button. Be sure that the location where you want the new data type to start is at the top of the screen.

When the left mouse button is held down, as well as the Ctl key, symbols and comments are temporarily disabled, as well as optimization. This allows you to quickly see the value of any symbols present in code.

While using the mouse to scroll, if the left Amiga key is held down, labels are created using forward references. If the left mouse button is pressed while the pointer is within 3 pixels of the right screen border, ReSource will assume that you are holding the left-Amiga key down, until you let the left mouse button go. This makes it easy to create labels single-handed. You don't have to keep the pointer on the right while scrolling, only its position when you actually press the LMB is important. If you press the left-Amiga key while in this mode, and then release it, ReSource will again test for the mouse position.

If ReSource's window is selected (activated) when the mouse pointer is on the far left of the screen, a screen update will *not* be performed. This may be necessary after saving to a ".asm" file, when the source profile information is to be saved to a file.

About the Display

ReSource has many similarities to a text editor. Its buffer contains data, which is interpreted using a complex set of rules, and a display of information is produced. In a text editor, you modify the contents of the buffer, and the display is immediately refreshed, reflecting the change in the buffer. In ReSource, the buffer contents are generally not changed. Instead, you give instructions to ReSource to change the set of rules used, in displaying the contents of the buffer. For example, here are 10 different ways of displaying the same data:

```
MOVEA.L  D0,A5
DC.W     $2A40
DC.W     10816
DC.W     %0010101001000000
DC.W     '*@'
DC.B     '*@'
DC.B     '*','@'
DC.B     $2A,$40
DC.B     42,64
DC.B     %00101010,%01000000
```

Any of the above lines could be inserted into an assembly language source file, and yield exactly the same output—two bytes, the first one being 42, and the second one being 64. While it often makes no difference to an assembler how the data is displayed, it does make a lot of difference if you wish to make any sense of what you are disassembling. If something was originally assembled as code, it will mean nothing if displayed as ASCII, or a series of bytes, words or longwords. And just as importantly, if something was originally assembled as ASCII:

```
doslibrary.MSG      DC.B   'dos.library',0
```

...it will mean very little to you if displayed as bytes:

```
lbB000232           DC.B   $64,$6F,$73,$2E,$6C,$69
                    DC.B   $62,$72,$61,$72,$79,0
```

When you tell ReSource that something should be displayed as code, or bytes, or perhaps ASCII, you are "setting the data type". Data types supported by ReSource include:

1. CODE
2. BYTES
3. WORDS
4. LONGWORDS
5. ASCII
6. UNKNOWN

As your disassembling skills increase, you will learn to recognize where something is displayed using the wrong data type. Also see About Data Types, page 4-7.

"Data type" is only one of many attributes of a position in ReSource's buffer that determine just what will be displayed.

Definition of the Current Line
Many functions in ReSource operate only on the current line. Because there is no real "cursor", it is necessary to define the "current line":

The "current line" is the top line, as you see it on ReSource's screen.

If there is a section statement (e.g., SECTION BSS) to be displayed

at the cursor address, this will be shown before any other text, and will force the other text to be shown on the next line down. If this is not the first section statement in the program, there will be an extra blank line, which will be shown before the section statement. After the section statement, and before the rest of the line, one or more "full-line comments" can be attached. Immediately after the cursor line, any "hidden labels" will be shown. A hidden label may also have a full-line comment attached, which will be shown immediately before it, but after the cursor line. Note that in this example:

```
      SECTION prog000000,CODE  ;Section statement
; This is a full line comment!
; This is another full line comment!
      MOVE.L  D1-D4/A2-A6,-(SP);Line of code
```

all of the above text is defined as one line. Whenever you scroll up or down a line, all parts of the "current line" will scroll together. To put it simply, the first "real" line of code or data in the window is the current line.

Quitting ReSource
Select "PROJECT/Quit" from the menu. You will be returned to the CLI or Workbench™.

A Short Tutorial

This section contains a short lesson to give you a feel for working with ReSource. The exercise is organized into a sequence of steps to follow during a single disassembly session. These steps will describe what functions to invoke, as well as what you will see on the screen. To take full advantage of the tutorial and learn about ReSource's features, you should execute each step completely.

All steps that you should perform will be indicated by a small pointing hand.

Start ReSource

Start the ReSource program. Depending on your Amiga's hardware configuration use either the instructions on starting from a floppy or starting from a hard drive (Pages 2-5 and 2-6).

Opening the Load File

Open the load file named "Tutorial1" provided with ReSource for this tutorial.

To open the load file:

1. Select "Open load file" from the "PROJECT" menu.
   ReSource will display a file requester.

2. Click once on "Examples".
   The file requester will show the contents of the Examples directory. In the list you will see the name "Tutorial1".

3. Double click on "Tutorial1".
   The file will be loaded into ReSource and displayed on the screen. The titlebar will display:

   ReSource 0% 000000 Examples/Tuturial1

Disassembling a Program

After "Tutorial1" has been loaded into ReSource, but before we do anything else, it looks like this:

```
                SECTION Tutorial1000000,CODE
ProgStart
                MOVEM.L D0/A0,-(SP)
                MOVEA.L (4),A6
                LEA     (START+$30,PC),A1
                JSR     (-$228,A6)
                MOVE.L  D0,(lbL000054)
                BNE.B   START+$24
                MOVE.L  #$38007,D7
                JSR     (-$6C,A6)
                BRA.B   START+$2A

                JSR     (lbC00003C)
                ADDQ.W  #8,SP
                MOVEQ   #0,D0
                RTS

                BCC.B   ????
                ????
                BGE.B   ????
                BHI.B   *+$74
                BSR.B   *+$74
                ????
lbC00003C       TST.L   (lbL000054)
                BEQ.B   START+$52
                MOVEA.L (lbL000054),A1
                MOVEA.L (4),A6
                JSR     (-$19E,A6)
                RTS

lbL000054       DC.L    0
                END
```

It doesn't look too much like code yet, does it? Nothing has been scanned, so ReSource doesn't know what is code or data.

If a line of code makes a reference to somewhere within the program that doesn't have a label yet, it will be shown as some offset from the label "START", which is imagined to be attached to the first byte of the program. Also, ReSource displays question marks when the data type is code and there is no legal instruction at that location.

Using the Built-In Disassembly Function

To disassemble a program is to interpret what went into the program to make it work. It can be a very complex and trying task, but with practice one can become quite adept at it. There are many different

and separate things to do when disassembling. One of the most important is to separate the code from the data. There is only one part of a load file that is guaranteed to be code, and that is at the very start.

One very useful function of ReSource is its ability to scan lines of code, looking for references to other parts of the program being disassembled. When a reference is made, it creates a label at that address, which will be used in all future references to that part of the program. What is more important, ReSource can determine what type of data is being referenced, and be correct (almost) every time. If you intend to reassemble the source, you should still verify it before saving, as mistakes will occasionally occur. Before ReSource makes a decision about what type of data is being referenced, it examines many different pieces of information that are available, including the actual instruction that made the reference. There are a few ways of getting ReSource to scan lines of code. The simplest way is by using the built-in disassembly function.

☞ Select the "PROJECT/Disassemble:" function from the menu now. The screen will now look like this.

```
                SECTION Tutorial1000000,CODE
ProgStart
                MOVEM.L D0/A0,-(SP)
                MOVEA.L (4),A6
                LEA     (doslibrary.MSG,PC),A1
                JSR     (-$228,A6)
                MOVE.L  D0,(lbL000054)
                BNE.B   lbC000024
                MOVE.L  #$00038007,D7
                JSR     (-$6C,A6)
                BRA.B   lbC00002A

lbC000024       JSR     (lbC00003C)
lbC00002A       ADDQ.W  #8,SP
                MOVEQ   #0,D0
                RTS

doslibrary.MSG  DC.B    'dos.library',0

lbC00003C       TST.L   (lbL000054)
                BEQ.B   lbC000052
                MOVEA.L (lbL000054),A1
                MOVEA.L (4),A6
                JSR     (-$19E,A6)
lbC000052       RTS

lbL000054       DC.L    0
                END
```

What has happened is that ReSource has "scanned" all code in the program and determined which bytes were code and which were data, and the appropriate data types set. Labels have been assigned to locations that were referenced from within the program. Note that ReSource uses the offset into the file as the last 6 characters of "shop" labels, with the first 3 characters being "lbC" for code, "lbL" for longwords, "lbW" for words, "lbB" for bytes and "lbA" for ASCII. Additionally, if ReSource finds some ASCII data that will make a legal assembly language label, it will use as much of the text as it can, appended with ".MSG".

At this point, if you output to a ".asm" file, it will be good enough to reassemble. However, most people will want to examine the file, and make it more readable, especially if the program needs to be modified. An excellent place to start is by finding where any libraries are being opened, finding where the library base is being stored, and giving meaningful names to these data storage locations.

Creating Symbols

The astute among you will have noticed that because the A6 register was loaded from location 4, it will point to the Exec library base. Therefore, the fourth line of this block of code is a call to somewhere in the Exec library.

☞ Scroll so that the line that reads "JSR (-$228,A6)" becomes the top line on the screen, and select "SYMBOLS 1/Libraries/Exec:" from the menu. Our block of code will now look like this:

```
                SECTION Tutorial1000000,CODE
ProgStart
                MOVEM.L D0/A0,-(SP)
                MOVEA.L (4),A6
                LEA     (doslibrary.MSG,PC),A1
                JSR     (_LVOOpenLibrary,A6)
```

```
                MOVE.L  D0,(lbL000054)
                BNE.B   lbC000024
                MOVE.L  #$00038007,D7
                JSR     (-$6C,A6)
                BRA.B   lbC00002A

lbC000024       JSR     (lbC00003C)
lbC00002A       ADDQ.W  #8,SP
                MOVEQ   #0,D0
                RTS

doslibrary.MSG  DC.B    'dos.library',0

lbC00003C       TST.L   (lbL000054)
                BEQ.B   lbC000052
                MOVEA.L (lbL000054),A1
                MOVEA.L (4),A67
                JSR     (-$19E,A6)
lbC000052       RTS

lbL000054       DC.L    0
                END
```

You have just created a symbol, from one of ReSource's internal symbol bases.

Creating Labels

Checking the documentation for the "OpenLibrary" Exec call tells us that on return, the D0 register will contain the library base for the library just opened, if successful. Because we know that it was the DOS library that was opened, the label "lbL000054" referenced on the fifth line could now be called "_DOSBase", instead of "lbL000054". To make it so, do the following:

☞ Scroll so that the line that reads "MOVE.L D0,(lbL000054)" becomes the top line on the screen, and select "CURSOR/Absolute/Forward reference:". This will normally be bound to the right cursor key. The line with the label "lbL000054" will now be at the top of your sceeen.

☞ Select "LABELS/Create single/Label" from the menu, and when the requester comes up, type in, "_DOSBase". Select "CURSOR/Absolute/Previous location" (this is normally bound

to the left cursor key) and notice how ReSource remembered our previous location. Our block of code will now look like this:

```
                SECTION Tutorial1000000,CODE
ProgStart
                MOVEM.L D0/A0,-(SP)
                MOVEA.L (4),A6
                LEA     (doslibrary.MSG,PC),A1
                JSR     (_LVOOpenLibrary,A6)
                MOVE.L  D0,(_DOSBase)
                BNE.B   lbC000024
                MOVE.L  #$00038007,D7
                JSR     (-$6C,A6)
                BRA.B   lbC00002A

lbC000024       JSR     (lbC00003C)
lbC00002A       ADDQ.W  #8,SP
                MOVEQ   #0,D0
                RTS

doslibrary.MSG  DC.B    'dos.library',0

lbC00003C       TST.L   (_DOSBase)
                BEQ.B   lbC000052
                MOVEA.L (_DOSBase),A1
                MOVEA.L (4),A6
                JSR     (-$19E,A6)
lbC000052       RTS

_DOSBase        DC.L    0
                END
```

Notice that every place in the program that formerly said "lbL000054" has been changed to "_DOSBase", not just the label you changed. This happens automatically, whenever you replace any label. The process helps itself; starting is hardest, the rest come fairly easily.

In general, where a label is used, if you have any idea of what is happening at all, you should replace it with a label that will remind you of what the code is doing. It doesn't matter what name you use, as long as it is a legal label for the assembler that you plan to use; if

necessary you can replace it with something better later on. Anything nominally mnemonic will be superior to a "shop" label while you're trying to understand the code.

Getting back to our program, we see that if the D0 register did contain a non-zero value, it means that "dos.library" did open successfully, and the following conditional branch will be taken. The destination of this branch is labelled "lbC000024". A better name for this might be "OpenedOK".

☞ Scroll so that the label "lbC000024" is on the top line of the display, and select "LABELS/Create single/Label". When asked, type in "OpenedOK". Now we see that, as before, the change we made has been echoed to other parts of the program:

```
                SECTION Tutorial1000000,CODE
ProgStart
                MOVEM.L D0/A0,-(SP)
                MOVEA.L (4),A6
                LEA     (doslibrary.MSG,PC),A1
                JSR     (_LVOOpenLibrary,A6)
                MOVE.L  D0,(_DOSBase)
                BNE.B   OpenedOK
                MOVE.L  #$00038007,D7
                JSR     (-$6C,A6)
                BRA.B   lbC00002A

OpenedOK        JSR     (lbC00003C)
lbC00002A       ADDQ.W  #8,SP
                MOVEQ   #0,D0
                RTS

doslibrary.MSG  DC.B    'dos.library',0

lbC00003C       TST.L   (_DOSBase)
                BEQ.B   lbC000052
                MOVEA.L (_DOSBase),A1
                MOVEA.L (4),A6
                JSR     (-$19E,A6)
lbC000052       RTS

_DOSBase        DC.L    0
                END
```


If the call to "OpenLibrary" was unsuccessful, the branch on the sixth line would NOT be taken. In this case, the D7 register is loaded with the value "$38007", and a call is made to the subroutine at offset -$6C from where the A6 register is currently pointing. We do know that the A6 register was pointing to the Exec library base before, and we also know that none of the code executed so far has destroyed the value in A6, so we can safely assume that A6 still points to Exec library base.

☞ Scroll so that the line "JSR (-$6C,A6)" is on the top line of the display, and select "SYMBOLS 1/Libraries/Exec" again. Our block will look like this:

```
                SECTION Tutorial1000000,CODE
ProgStart
                MOVEM.L D0/A0,-(SP)
                MOVEA.L (4),A6
                LEA     (doslibrary.MSG,PC),A1
                JSR     (_LVOOpenLibrary,A6)
                MOVE.L  D0,(_DOSBase)
                BNE.B   OpenedOK
                MOVE.L  #$00038007,D7
                JSR     (_LVOAlert,A6)
                BRA.B   lbC00002A

OpenedOK        JSR     (lbC00003C)
lbC00002A       ADDQ.W  #8,SP
                MOVEQ   #0,D0
                RTS

doslibrary.MSG  DC.B    'dos.library',0

lbC00003C       TST.L   (_DOSBase)
                BEQ.B   lbC000052
                MOVEA.L (_DOSBase),A1
                MOVEA.L (4),A6
                JSR     (-$19E,A6)
lbC000052       RTS

_DOSBase        DC.L    0
                END
```

other local labels, and to the initial global label in a piece of code. ReSource defines a local label as either a decimal number directly followed by a dollar sign, "$", or as a period, "." followed by any normal label characters.

The benefit derived from local labels is that not all the labels in a program need be unique. As you are disassembling a program, it is possible to treat each routine like a "black box" where only the global label is accessible to the rest of the program. This technique works fairly well with "C" programs, which tend to treat routines as objects anyway, but be careful with programs originally written in assembly language where routines may have entry points almost anywhere.

About Symbols

While labels generally mark locations, symbols are generally used to stand for values. This makes your program easier to read, because you can see at a glance what is being done, rather that trying to remember what that number means. This is particularly true when disassembling Amiga® programs because almost all system functions in the Amiga® OS involve structures, and they are accessed by offsets from the structure base. ReSource can help you immensely here because it can supply all the structure offsets for you. In this way, you can concentrate on the job, instead of remembering details.

How Symbols are used in ReSource

Unlike labels, ReSource does not automatically generate symbols for you. You will need to recognize which symbol base the code references, and tell ReSource to use that symbol base when replacing a number with a symbol. For example, if you happened to know that at a particular point in some code, the A6 register was pointing to the Exec library base, you would interpret the line:

```
JSR    (-$0228,A6)
```

as

```
JSR    (_LVOOpenLibrary,A6)
```

To make ReSource do this, you would scroll so that this line became the current line, and then select "Exec library" from the SYMBOLS menu. There are hundreds of other symbol bases that are available, including libraries, devices, structures, parameters, and error code names.

If you need to define a symbol that ReSource doesn't know about, there are two other options. You can define that symbol immediately by using the "Create symbol" function. If you have an entire symbol base that's undefined, say, the base for a custom library, you may define your own symbol base and use it in exactly the same manner as the built-in symbol bases. See the section on User-defined symbol bases for more information on this subject.

Symbol Equates

When you have completed disassembling your program, and wish to save it as assembly source, ReSource gives you several options on what to do with all the symbol definitions you have made. If your assembler needs the symbol values, you would want to select "SAVE/Symbol table/EQUate:". If this piece of code will be part of a larger project that will later be linked, you might want to select "SAVE/Symbol table/XREF:". On the other hand, if you don't want ReSource to bother you with this information, select "SAVE/Symbol table/ None:".

Operand Fields

The 68000/68010 processors can have instructions with either a source operand, destination operand, or both. The 68020/68030 processors can have instructions that contain up to 4 references. Because of this, it is sometimes necessary to inform ReSource which field you are referring to when creating or editing symbols. This is done by selecting one of the "Set field" functions, and then selecting your symbol base. This only has to be done if there is more than one operand field. A single operand instruction, or a two operand instruction where one operand is a register, does not need the field set. This is true even if the register is the source operand. For instructions with multiple fields, the selection must be made for each instruction for which you access a symbol base; selecting the third field for a symbol assignment will not carry over to the next line for another assignment, the selection is always reset to the first field.

Comments

When writing any code, documentation is important because it reminds you of what the code was doing when it was written. Later, when you need to make modifications or bug fixes, it can be difficult to figure out what a fiendishly clever piece of code is trying to accomplish. Nowhere is this more relevant then when trying to disassemble code, especially if you did not write it originally. ReSource offers two ways to add documentation to your disassemblies; End-of-line and full-line comments.

End-of-line Comments
These consist of a string of less than 240 characters, which will become a comment to be attached to the end of the cursor line. Only one end-of-line comment is allowed per line; if you create one on a line that has already got one, it will replace the old comment. End-of-line comments may be freely edited.

Full-line Comments
These consist of a string of less than 240 characters, which will become a comment inserted just ahead of the current line. Multiple full-line comments are allowed, and successive comments will be displayed after previously defined full-line comments. Normally, full-line comments will be displayed starting with a semicolon, ";". However, if you start the string with a semicolon, the logic is reversed; the comment is added without a leading semicolon, and it becomes an extra line of code/data when reassembled. Thus, you can insert extra lines of code/data, even before saving the source code to a text file. By creating a label that starts with a semicolon for a given line, when that code is assembled, that line of code/data is effectively treated as a comment only, therefore you can both insert and effectively remove code/data using ReSource, during the disassembly process.

Please note that both "Create full-line comment" and "Edit full-line comment" will place the newly created or edited comment after any previous full-line comments on the current line. When deleting full-line comments, the first comment on the current line will be the one deleted. A related function lets you rotate a group of full-line comments to get a specific comment to the top, where it can be either edited or removed.

Code Identification

When you're disassembling a complex program, one of the most difficult jobs is to identify the type of code or data that exists in a particular area. ReSource can help you do this more quickly and easily by sharing some of its internal information about the file being disassembled with you

Hiliting
ReSource stores and maintains various pieces of information about each byte in the file being disassembled. Much of this information is used internally when displaying each line, determining data types, etc. The hiliting functions give the user access to some of this information.

For example, when each label is created, ReSource records whether it created the label name, or the user created it. If the label name was created by ReSource itself, the label is referred to as a "shop label". Whenever ReSource displays a label, it can easily determine whether the label is "shop" or "custom". If "DISPLAY/Hiliting/Shop labels" is currently active, whenever ReSource displays a shop label, it will display it with light underlining. Other labels will be displayed using the normal font. Thus, the user can tell at a glance whether a label was created by ReSource itself, or if it is a "custom" label.

One of the most useful hiliting functions is "Reloc32". This is most useful when disassembling executable programs. A "Reloc32" refers to a group of four bytes that contains a pointer to a memory location. The actual memory location being pointed to is calculated by adding an offset to the base address of one of the hunks in the file. Certain assumptions can generally be made about how reloc32s can be displayed. Any line containing a reloc32 pointer must either be code, or longwords, as there is no other sensible alternative.

To allow several differing types of hiliting simultaneously, ReSource uses several different types of hiliting:

Each character is lightly underlined
Each character is fully underlined
The background and foreground colors are reversed.

Apart from telling you where certain things are, hiliting can be used by the beginner to tell what various things are. For example, if you don't know how to recognize a BSS hunk from a DATA hunk, you could tell ReSource to hilite "BSS hunks" exclusively, and then simply scan various programs looking for hilited areas. Once you find a hilited area, you know that it is definitely part of a BSS hunk. As you become more familiar with how hiliting is used in ReSource, you will be able to hilite many different attributes simultaneously, and get the full benefit from each.

Attribute Bits
ReSource uses a table of attribute bits to tell how things should be displayed. The attributes table is approximately four times the size of the current file. It is used internally as a large bitmap, to store information about each and every byte in the current file. Because it is four times the file size, 32 bits are available for each byte in the

current file. This is also the reason that ReSource is so fast, and why it has such a prodigious appetite for memory.

For each byte of the file, some of the things that are marked are:

This byte is the start of a line of code/data
This byte is part of a Code, Data or BSS hunk
This is the first byte of a reloc32 area
There is a label attached to this byte
There is a symbol attached to this byte
This byte was parsed previously (symbol scan)
The data type for this byte hasn't been set yet
The data type for this byte is bytes
The data type for this byte is words
The data type for this byte is longwords
The data type for this byte is ASCII.

You can see this information directly, if you like, by using the "DISPLAY/Titlebar info/Attributes:" function. This will cause the last field of the title bar to display information about the attributes of the current byte, on the current line. This information is not required for general program usage, and was originally included for debugging purposes, and left in to satisfy those who want to know how the attribute table in ReSource works. For more information about attribute bits, use the online help facility.

About Data Types

Earlier we mentioned that you could use several functions to set the data type of a particular byte in the file you are working on, and ReSource would use that information in deciding how to display that line.

Be aware that by setting the data type at a particular offset in a file, you are not only telling ReSource how that line and lines below it should be displayed, you are also telling it where it should start showing that data type, also.

For example, if something is shown as ASCII, and you change the data type permanently at that place in the file to some other data type, i.e. code, permanently changing the data type back to ASCII does not necessarily make it exactly what it was before. This is particularly important when you set the data type to ASCII. Often, many bytes of ASCII are displayed together on one line. By setting the data type,

you are forcing ReSource to start a new line at that point, regardless of what data type is displayed above. If you do set a data type somewhere, and decide that it wasn't a good idea, use the "LABELS/Remove single/All:" or "DISPLAY/Set data type/Unknown:" function.

An In-Depth Tutorial

Now that you know something about the way ReSource works, and how to disassemble a program, we are going to provide an opportunity for you to practice your skills. What follows is a real, working program that, together, we will disassemble and turn back into real source code. This tutorial will be much more fun if you can refrain from running the program before you disassemble it. Otherwise, the suspense is ruined, and it may seem more like an exercise than a new learning experience.

Unlike the first tutorial, we will not show the entire file at every example, but only a small piece, representative of what we are currently working on Also unlike the previous lesson, we will refer to functions by their keys as well as their menu names. All such usage will be based on the default keytable provided with ReSource. So if you have modified it already, load the original as soon as you start the program. We will also refer to lines by both labels and offsets, so keep your eye on the titlebar.

Note that when asked to create or edit symbols, you can undo mistakes by using the "Labels/Remove single/Symbol:" function.

☞ To begin, start the ReSource program. Then open the load file provided with ReSource for this tutorial, "Tutorial2". You should see something like this:

```
                SECTION Tutorial2000000,CODE
ProgStart
                LEA     (lbL000320,PC),A5
                MOVEA.L A5,A0
                MOVEQ   #5,D1
                MOVEQ   #0,D0
                MOVE.L  D0,(A0)+
                DBRA    D1,START+$0A
```

Notice that the first line of code loads something into register A5. Also notice that whatever A5 is pointing to is immediately being cleared. If you use the right cursor key, you will notice that A5 is pointing at a line that looks like:

```
lbL000320    DX.L    8
```

The "DX" stands for "Define eXtra storage". It is a special type of "DS" that takes no disk space, but may have to be cleared by the program, as it is not cleared by the system as BSS sections are.

Using the left cursor key to return to our previous location, you might also notice that several lines of code are referencing things off A5. Taken together, all these clues give us a very good idea that A5 is being used as a base register. Base registers are used both to make programs smaller, and to make the code run faster. Because it is not possible to write to pc relative locations with 680x0 cpus, we must either use absolute addressing, or use base relative addressing, effectively reducing the size of the program by 6 bytes each time we do. Since A5 is our base register, we need to tell ReSource about it.

☞ Pull down the "*" (SPECIAL FUNCTIONS) menu and look at the line where it says "SPECIAL FUNCTIONS/Convert (xx,A4) EAs/:". Now select the function "SPECIAL FUNCTIONS/Specify base register/A5:" and notice that the line you looked at before has changed to A5. Now we will inform ReSource of the base location by selecting "SPECIAL FUNCTIONS/Convert (xx,A5) EA's/This operand".

Notice that what was previously:

```
MOVE.L  D0,(0,A5)
```

has been changed to

```
MOVE.L  D0,(lbL000320-DT,A5)
```

and ReSource has created a label named "DT" for us. Note that because A5 is pointing at "lbL000320", that (0,A5) and (lbL000320-DT,A5) are equivalent. The principal reason for expressing the relationship in this way is that if you decided to modify this program, and either removed or added lines to the DX area, the simple offset would no longer be valid.

Now that our relative base is firmly established, we can proceed to disassemble the program.

☞ Select "PROJECT/Disassemble:".

That was quick, but a whole lot of things were going on. ReSource scanned the code and assigned data types. Notice that we now have labels, and there is even some readable text in the program.

☞ Scroll down to the line at offset $10. We know that location 4 is AbsExecBase, so let's name it that. Use the "V" key to select "Create Symbol" and type "AbsExecBase" into the requester.

The line at offset $14 is an offset of A6, which is pointing to Exec library, so this offset must be in the Exec library base.

☞ Use the "E" key to select "SYMBOLS 1/Exec library".

Note that A2 is now pointing to our process base.

☞ Make the line at offset $18 current. Then use the "T" key to select "SYMBOLS 2/Task control struct".

Here the program is testing the pointer to the CLI structure. If it is zero, this task was launched from the Workbench™. Therefore, if we take the branch, we were launched from the CLI.

☞ Make the line at offset $1C current. Use the right cursor key to take the branch. Then use the "kp9" key and enter "FromCLI" into the requester. Finally, return with the left cursor key.

The next 5 lines are executed if we were launched from Workbench™. They get the Workbench message and save it until we exit the program.

☞ Use the "T" and "E" keys to create symbols for the 4 lines beginning at offset $1E. If you make a mistake, just hit the proper key. When you get down to offset $2E, use right cursor to go to the destination, and use "kp9" to name this "_WBenchMsg". Use left cursor to return.

We are now down to the label "FromCLI" at offset $32. You can probably surmise that we are about to open some libraries.

☞ Make the line at offset $38 current, and use "E" to name this call. Because we just opened Graphics library, go to label "lbL000324", where the pointer will be stored, and rename this "_GfxBase". Then scroll down to offset $60 and do the same things, except rename the label at "lbL000328" "_IntuitionBase". Then scroll back to offset $44.

The program is moving _GfxBase into A6, and then making A0 point to something, at which time a call is made to the graphics library. This is the sort of situation that makes disassembling interesting.

☞ Move to offset $4A, and by using the "g" key to select "SYMBOLS 2/Graphics library", the symbol "_LVOOpenFont" will appear.

What is A0 pointing to? The AutoDocs™ tell us that when calling "OpenFont", A0 should point to a TextAttr structure. They also tell us that if the call is successful, a pointer to a font will be returned in D0. So let's go to lbL000186 and make it look like a TextAttr structure.

☞ The way to do this is to make the line where the label "lbL000186" is the current line, and then tell ReSource this is a longword with left-Amiga L. Scroll to the next line and make it into a word with left-Amiga W. Finally, make the last 2 lines bytes by using left-Amiga B. We can name it "MyTextAttr". It should look like this when we are done:

```
MyTextAttr   DL      topazfont.MSG
             DW      8
             DB      0
             DB      1
```

☞ At offset $4E, press the right cursor key to go to offset $32C, and create the label "_FontPtr" (use keypad "9" key again).

Go back to offset $68, and notice that if the intuition library doesn't open, we take a branch. If you look back over the code we have done so far, you will see that if anything doesn't open, we go to the same location.

☞ Go to label lbC0000E2 and rename it "Cleanup".

Notice that at offset $6A the program moves _IntuitionBase into A6 and, after pointing A0 to something, makes a call to intuition library. This is the same situation we had before where we needed to find out what A0 was pointing to. The best way to resolve these cases is to gather all the evidence you can, and then make an informed choice.

☞ Go to offset $70 and use the "I" key to name this call to intuition library.

If the call is to OpenWindow, A0 must be pointing to a NewWindow structure. Let's go to lbL00018E and check it out.

☞ Go to lbL00018E and rename it "MyNewWindow". Get out the include.i files and look in "intuition/intuition.i" for the NewWindow structure. We are going to clean up MyNewWindow so that it's more readable.

☞ Make the label MyNewWindow the current line, and use the left-Amiga W key to tell ReSource that these lines are words. Scroll down 4 lines to offset $196 and use left-Amiga B to set the data type to bytes. Scroll down 2 lines to offset $198 and set the data type to longwords. Scroll down 7 lines to offset $1B4 and set the data type to words. Scroll down 4 lines to offset $1BC and again set the data type to words. Why we are setting this line to words again will be clear in a moment.

☞ While still at offset $1BC, select "SYMBOLS 2/S/Screen flags:". This should create the symbol "WBENCHSCREEN". Scroll up to offset $198 and select "SYMBOLS 2/H-I/IDCMP classes:". Scroll down 1 line to offset $19C and select "SYMBOLS 2/U-Z/ Window flags:".

☞ Type a shiftCtl-upArrow (shift + control + cursor up) to select "CURSOR/Relative/Previous label:", which will take us to the label MyNewWindow. Once we're there, we can do some magic. Select "DISPLAY 2/Mult constants override/Set:" from the menu. Notice that we have all four words on the same line now. Use the shift-~ (shift + ~) key to set these values to decimal. Scroll down 1 line to offset $196, and hit the spacebar to call the function, "Repeat last command". Notice that even though we used another function from a key, the spacebar is still bound to the last menu function we selected.

☞ While still at offset $196, type a "V" and enter "-1" for nw_DetailPen. Type shift-rightArrow (shift + right cursor) to move forward 1 byte, and set nw_BlockPen to -1 also.

☞ Move down to offset $1AC and hit the spacebar. Do the same at offset $1B4. Notice that this last "Multiple constants" command

put 4 words on the same line, but left WBENCHSCREEN alone. This is why we used an extra set data type to word before. "Multiple constants" stops when it reaches either a different data type, or a byte that already has its data type set. Lastly, we can add a blank line before and after the NewWindow structure by using full-line-comments. Remember, if a full-line-comment has a leading semicolon, it will not use the semicolon. So if our comment consists of only a semicolon, we will have a blank line. Go to offset $1BE and type a "kp-", then enter a ";" in the requester. Type a shiftCtl-upArrow to get back to MyNewWindow, and put a blank line there as well.

We have just completed our NewWindow structure. Looks pretty good, doesn't it? Let's take this opportunity to save the work we've done so far, and then we'll tackle something else.

☞ Select "PROJECT/Save .RS/Save:" from the menu. When the requester comes up, if you don't like the default name, type in a path and filename. It's usually a good idea to use the ".rs" suffix so you can tell that this is a ReSource data file.

Now that your work is saved, take another look at the NewWindow structure. Notice that the nw_FirstGadget field is non-zero. This means that this window has at least one gadget attached.

☞ Go to the nw_FirstGadget field of MyNewWindow (at offset $1A0) and use the right cursor key to go to that location. While we're here, we'll rename this from "lbL0001BE" to "First_Gadget". This may not be the final name of this structure, but as we stated previously, it makes the name much easier to remember.

We are now going to construct a macro that will turn disassembled gadgets into code that looks like a gadget. First_Gadget should be on the current line for this exercise.

☞ Select "MACROS 1/Create:" and the first empty slot in the menu. When the requester asks for a name, enter "Gadget". Select all the functions from the following list. Take your time, and if you make a mistake, end the macro and start over. The text preceded by semicolons is for your benefit, not ReSource's, and shouldn't be entered.

Here is the function list for the "Gadget" macro.

Function Name	Default Key	Gadget Field
DISPLAY/Set data type/Longs	;left-Amiga L	gg_NextGadget
CURSOR/Relative/Next line	;downArrow	
DISPLAY/Set data type/Words	;left-Amiga W	gg_LeftEdge
CURSOR/Relative/Next line	;downArrow	gg_TopEdge
CURSOR/Relative/Next line	;downArrow	gg_Width
CURSOR/Relative/Next line	;downArrow	gg_Height
CURSOR/Relative/Next line	;downArrow	
DISPLAY/Set data type/Words	;left-Amiga W	gg_Flags
SYMBOLS 2/E-G/Gadget flags		
CURSOR/Relative/Next line	;downArrow	gg_Activation
SYMBOLS 2/E-G/Gadget activation		
CURSOR/Relative/Next line	;downArrow	gg_GadgetType
SYMBOLS 2/E-G/Gadget types		
CURSOR/Relative/Next line	;downArrow	
DISPLAY/Set data type/Longs	;left-Amiga L	gg_GadgetRender
CURSOR/Relative/Next line	;downArrow	gg_SelectRender
CURSOR/Relative/Next line	;downArrow	gg_GadgetText
CURSOR/Relative/Next line	;downArrow	gg_MutualExclude
CURSOR/Relative/Next line	;downArrow	gg_SpecialInfo
CURSOR/Relative/Next line	;downArrow	
DISPLAY/Set data type/Words	;left-Amiga W	gg_GadgetID
CURSOR/Relative/Next line	;downArrow	
DISPLAY/Set data type/Longs	;left-Amiga L	gg_UserData
CURSOR/Relative/Next line	;downArrow	
LABELS/Edit single/Full-line comment	;kp-	Enter ";"
CURSOR/Relative/Previous line	;upArrow	Do this 14 times
DISPLAY 2/Mult constants override/Set		
DISPLAY/Set Numeric base/Decimal	;shift-~	
CURSOR/Relative/Previous line	;upArrow	Should be at First_Gadget now
CURSOR/Absolute/Forward reference	;rightArrow	This takes us to gg_NextGadget

Press rightAmiga-A to end the macro.

Now we can use this macro to process any other gadgets (there are 2 more), and you will always have it to use on future gadgets. Note that the last line takes us to the next gadget in the linked list—if there are no more gadgets this field will be zero and the macro will fail at that point.

☞ Without changing the location, which should be lbL0001FE, select "MACROS/Execute/Gadget:" twice to process both gadgets.

We are now positioned at the third and last gadget, but we don't know much about it. One of the best ways to find out about an Amiga® program is by looking at its menus and gadgets. And the best way to find out about a gadget is to look at its text, if it has some. This gadget has a pointer in its gg_GadgetText field, so we know there is an IntuiText structure. We will clean up this structure, and find the text.

☞ Go to the gadget's gg_GadgetText field at offset $26C and then forward reference to the IntuiText structure at offset $2B6. Using the skills that you have just learned, clean up and rename the structure so it looks like this:

```
False_ITxt    DB      1,0,0,0     ;Was lbB0002B6
              DW      -4,20
              DL      MyTextAttr
              DL      False.MSG
              DL      0
```

Notice that we incorporated the text in the name of the IntuiText structure. It cannot be over emphasized that there is a world of difference between "lbB0002B6" and "False_ITxt". Even though "False" has no significance to us yet, "lbB0002B6" is an anonymous string of characters, and looks much like all other shop labels to our eyes. But "False_ITxt" is distinctive, and we will gradually build a mental model of what this code is doing. Even then, we can change the name if it no longer suits the code.

☞ Go back to lbL000252 and rename the gadget "False_Gadget".

Notice that the gadget has a pointer to something in its gg_GadgetRender field too. Because the GADGIMAGE flag isn't present, we know that this must be a Border structure.

☞ Forward reference to this structure at offset $27E. Referring to your ROM Kernal Manual, clean it up so that it looks like this:

```
Border1       DW      0,0
              DB      2,0,0,3
              DL      Coords1
              DL      Border2

Coords1       DW      0,14,0,0,30,0
```



```
Border2       DW      0,0
              DB      1,0,0,3
              DL      Coords2
              DL      0

Coords2       DW      30,1,30,14,0,14
```

☞ Left arrow back to the gadget at offset $1FE. Notice that this gadget has no pointer to IntuiText, but does have pointers to both gg_GadgetRender and gg_SelectRender. Also note that both the GADGIMAGE and GADGHIMAGE flags are present. This means that we have Image structures. Follow the gg_GadgetRender and gg_SelectRender pointers, and clean up the Image structures so that they look like this:

```
Normal_Image DW       0,0,31,15,1 ;Was lbL00022A
             DL       lbL000340   ;ig_ImageData
             DB       1,1
             DL       0

Select_Image DW       0,0,31,15,1 ;Was lbL00023E
             DL       lbL00037C   ;ig_ImageData
             DB       1,1
             DL       0
```

Remember that image structures point to image data, so we will also rename those, and clean them up too. Before we go, note the width of the images so you will know what data type to use. Remembering that image data must be a multiple of words, 31 bits would fit into either 2 words or 1 longword. It's fully your choice, but longwords will have fewer "%"s to get in your way than words. In this case, we will use longwords.

☞ Go to the first image data at offset $340. Name this "Normal_Data". Set the data type to longwords. Then, use shiftCtl-downArrow to select "CURSOR/Relative/Next label:" to go to the second image data at offset $37C. Insert a blank line here, using kp-and ";", and then name the second image, "Select_Data". Set the data type here to longwords, too. Use shiftCtl-upArrow to get back to offset $340.

☞ We are now going to build another macro, but this one is simple; only 2 instructions. First, Select "MACROS 1/Create:" and the

first empty slot in the menu. When the requester asks for a name, enter anything you like. Then select "DISPLAY/Set Numeric base/ Binary:", CURSOR/Relative/Next line:" and "MACROS 1/ Previous macro label/#1:". The macro is done, so press rightAmiga-A to end the macro. Execute this macro now.

☞ Now move the cursor back to the label "Normal_Data". But this time, do it by selecting "CURSOR/Absolute/Specify label", and entering "Normal_Data".

What happened here is that the macro converted every line to binary. Take a moment and look at the neat images you just made.

☞ Left arrow back to the gadget at offset $1FE and rename it "Image_Gadget". Then go to the gadget at offset $1BE. Fix up it's IntuiText structure and rename that to "True_ITxt". Then rename the gadget to "True_Gadget". Finally, go back until you hit the reference to MyNewWindow at offset $6C.

We can now forward reference the location after the OpenWindow library call, and rename it "_WindowPtr". Since the line at offset $7A puts a pointer to the window structure into A0, then the number in the line at offset $7C must be an offset into that structure.

☞ Select "SYMBOLS 2/U-Z/Window:" for that line. The program now puts the pointer to the Window's RastPort into A1, and stores that for future use. Make offset $80 current, forward reference, and rename this "_RastPort". While we're at it, fill in the graphics library call at offset $8C, by using the "G" key.

The next line gets the pointer to the window into A2, so the number in the line at offset $94 must be an offset into that structure.

☞ Select "Window" again to have ReSource fix this up. The line at offset $98 puts AbsExecBase into A6, so it looks like we're getting ready to call Exec library. Use the "E" key on the line at offset $9C.

☞ This looks suspiciously like it could be our main GetMsg loop, so go back to the previous label and rename it "GetMessage".

After we call GetMsg, we test D0 to see if we actually got one. If so, we branch to label lbC0000AE. Since we come here if we get a

message, let's call this "GotMessage". Moving to offset $A4, we see that if there is no message, we get the pointer to the UserPort from the Window again and then call another Exec library function.

☞ Use "SYMBOLS 2/U-Z/Window:" for the Window offset and the "E" key for the call.

Having arrived at GotMessage, we move the message pointer into A1 and reference offsets into whatever A1 is pointing at on 2 lines. Since this message came from the UserPort of a Window, it must be an IntuiMsg.

☞ Select "SYMBOLS 2/H-I/IntuiMessage:" for the lines at offsets $B0 and $B4. Then use the "E" key to fix the call to Exec library on the next line.

The next line compares D2 with some value. If we look back 3 lines we will find that im_Class was moved into D2.

☞ Select "SYMBOLS/I-H/IDCMP classes:" to change this into a symbol. Notice that we are making the same type of comparisons at offsets $CA and $DA. Move to those lines and just hit the spacebar for each one.

Notice that if the im_Class in D2 is equal to GADGETUP, we do a BSR to some routine at lbC000146. That might give us a clue to what the program is about, so let's go take a look at it. First, it clears D0, then it gets the address of something into A0. Let's take a look at that. Referencing to "ascii.MSG" we find 4 spaces.

☞ As "Space4.txt" is much more informative, we'll rename it to that. Use leftArrow to return to lbC000146.

It appears that this subroutine contains 3 calls to graphics library. The best thing to do at this point is to name those, so we can see what is going on.

☞ Use the "G" key to name the 3 calls to graphics library.

This subroutine is setting the color of the front pen, moving to a particular spot in the RastPort, and writing the 4 spaces. So let's call our subroutine at lbC000146, "Print4Spaces".

☞Rename lbC000146 to "Print4Spaces".

That looks better, but the trouble is that we still don't know what that label, lbC00014C is for. Let's find out right now. ReSource has a number of different ways of looking for things. We could search for the label, but searches are relatively slow. Even though this is a tiny program and it would not make any difference, we will use something else. ReSource has two functions that are frequently used together. The first function finds the absolute first backward reference to a label, i.e. something else that's referencing that label. The second function finds all the other references, one at a time. These functions are extremely fast.

☞ Make the label, lbC00014C, the current line and type Alt-s to select "CURSOR/Absolute/Backwards reference:". Note where it landed. Then type Ctl-s to select "CURSOR/Relative/Next backwards reference:". Type Ctl-s repeatedly, until you are back at the original label, making note of the references as you go.

As it turns out, there were only two other references, and they were both close by. It looks like lbC00014C is the common code for all 3 routines. They enter with a pen number in D0, and a pointer to some text in A0. So let's call this subroutine, "PrintText".

☞ Rename lbC00014C to "PrintText". Use leftArrow to return to offset $C4. Note that you will have to use it several times because the previous functions push the target offsets onto the cursor stack, so that you can easily find them later.

Moving down to label lbC0000CA, we see that if the im_Class in D2 is equal to GADGETUP we take something out of what A2 is pointing at and put it into A0. Remembering that at offset $B4, the program put im_IAddress into A2, and this is a GADGETUP class message, A2 must point to a gadget.

☞ Select SYMBOLS 2/E-G/Gadget:" for the line at offset $D2.

Because the next line does an indirect JSR off this data, these must be pointers to routines for the gadgets. The only way we can trace these is go back to the gadgets themselves, and see what is in the gg_UserData fields. Because we used the "Backwards reference" functions previously, let's use a search function this time. Since we

created the names of the gadgets, we know that all 3 of them have the string "_Gadget" in common.

☞ Type F7 to select "CURSOR/Normal search/Set search pattern:", and type "_Gadget" into the requester. If anything else was already there, type a right-Amiga X to clear it. Now type F9 to select ""CURSOR/Normal search/Find next occurrence:". The first time you will come to the NewWindow structure, so type F9 again to find the first gadget. The gg_UserData field is at the very end of this gadget. Scroll down there and use rightArrow to get to its reference.

This routine calls PrintText with pen #3, and text that says, "True". Since this function address came from True_Gadget, we could call this, "PrintTrue".

☞ Rename lbC000136 to "PrintTrue". LeftArrow back, and type F9 to find the next gadget. Scroll down to its gg_UserData and use rightArrow again to get to its reference.

Since this routine calls PrintText with pen #2, and text that says, "Ouch", we will call this routine, "PrintOuch".

☞ Rename lbC00013E to "PrintOuch". LeftArrow back, and type F9 to find the final gadget. Scroll down to its gg_UserData and use rightArrow again to get to its reference.

Here is something different. This doesn't look at all like the other gadget's action routines. As usual, the first thing to do is to resolve the symbol offsets so that we can find out what is going on.

☞ Use "SYMBOLS 2/U-Z/Window:" and the "I" key to resolve these symbols.

It looks as if the False_Gadget beeps the screen when it's poked.

☞ Rename lbC000174 to "Beep". LeftArrow back to the gadget. Use "PROJECT/Save .RS:" to save your work.

That pretty well wraps up the tutorial. You now know everything you need to know to finish tidying up the program. The original source code is provided so that you can compare it to this disassembly, if you like.

Advanced Topics

After reading the preceeding five sections of this manual, you should have a pretty good idea of how ReSource works, and how to use many of its features. In this section, we will discuss some ideas and techniques that can help you to utilize the full power of ReSource.

➥ How can I easily measure an offset from a base point in a file?

Many times you would like to be able to measure offsets from a specific base point in some code, perhaps in order to locate structure offsets, or maybe library vector offsets.

This can easily be accomplished by using the "DISPLAY/Set counter:" function. Position the base point at the current line and select this function. From now on, the offsets shown in the titlebar will be relative to your selected base, rather than from the start of the file. In order to again show offsets relative to the beginning of the file, select ""DISPLAY/Reset counter:".

➥ How do I create a symbol from the contents of the accumulator?

By using string indirection, instead of actually supplying a string literal to a string requester, the string is copied from the accumulator. Say the accumulator contains a string, and you wish to use that string to create a symbol. Select "LABELS/Create single/Symbol:", and when asked for the symbol name, type the escape sequence "\e\e" (escape twice), and select the "OKAY" gadget, or press return. The symbol in the accumulator replaces the current symbol. The same can be done for the buffers A-M. For instance, typing "\ec" (escape followed by "C") would use the string in buffer C.

➥ How do I get to the start of the next hunk in binary image mode?

In the example below, the first hunk is a code hunk and contains $CA longwords of code (padded to a longword boundary). In order to find the beginning of the next hunk, place the line after this, the actual first line of code in this example, on the cursor line, and select "DISPLAY/Set counter:". The titlebar will now display zero as the offset.

```
            DC.L        HUNK_CODE  ;Hunk type
            DC.L        $CA        ;Size of code
;Start of code
            LEA         (START+$034A,PC),A5
            MOVEA.L     A5,A0
            MOVEQ       #$1D,D1
            moveq       #0,D0
            ...
            DC.L        HUNK_END   ;End of this hunk
```

If you scroll ahead in this file $CA*4, or $328 bytes, ahead in the file, you will come to the the following line:

```
            DC.L        HUNK_RELOC32
            DC.L        8          ;# of offsets
            DC.L        1          ;Hunk number
            DC.L        $238       ;Offsets
            DC.L        $234
            DC.L        $230
            ...
            DC.L        HUNK_END   ;End of this hunk
```

There are many different hunk types, and they have different formats. Consult your AmigaDOS™ manual for more information.

➥ When ReSource disassembled my program, it missed a word offset jump table. How can I reconstruct this?

There are different reasons why ReSource may occasionally miss properly disassembling an offset jump table. However, it is really very easy to convert to code by using some of the built-in tools in ReSource. Imagine that we have a piece of code like the following:

```
            MOVE.L      D7,D0
            SUBQ.L      #1,D0
            BLT.W       lbC00AB6A
            CMPI.L      #7,D0
            BGE.W       lbC00AB6A
            ADD.W       D0,D0
            MOVE.W      (lbW00A8A6,PC,D0.W),D0
            JMP         (lbW00A8A8,PC,D0.W)
```



```
lbW00A8A6   DC.W        $C
lbW00A8A8   DC.W        $A0
            DC.W        $C2
            DC.W        $DC
            DC.W        $154
            DC.W        $1D4
            DC.W        $1EE
```

The last two lines of code tell us immediately that we are getting a word offset from the table at "lbW00A8A6", and that the jump location is "lbW00A8A8". In order to turn the word offsets into a symbolic offset to the routines that we will call, and at the same time create labels for those routines, we need to do two things.

First, tell ReSource where the offset base is by scrolling so that "lbW00A8A8" is the current line and selecting "SPECIAL FUNCTIONS/Convert specific EA's/Set base #1:". Then we can scroll up one line to the start of the table, at "lbW00A8A6" and select "SPECIAL FUNCTIONS/Cvert W/base 1:" to actually do the conversion. The later function must be done for each offset in the table. When we're done, it will look like this:

```
lbW00A8A6   DC.W        lbC00A8B4-lbW00A8A8
lbW00A8A8   DC.W        lbC00A948-lbW00A8A8
            DC.W        lbC00A96A-lbW00A8A8
            DC.W        lbC00A984-lbW00A8A8
            DC.W        lbC00A9FC-lbW00A8A8
            DC.W        lbC00AA7C-lbW00A8A8
            DC.W        lbC00AA96-lbW00A8A8
```

➥ How do I load and disassemble a boot block?

To start ReSource from the CLI and have it immediately load a boot block, enter:

```
1> ReSource *DF0: 0 0 2
```

This will load the boot sectors, a total of 1024 bytes, from DF0:. Similarly, to load the same data when ReSource is already running, select "PROJECT/Read tracks:" and enter "DF0: 0 0 2" into the requester.

Once the boot code is entered, how you disassemble it depends on what the code looks like. A standard boot block might look like:

```
                DC.B        'DOS',0    ;OFS
                DC.L        $C0200F19  ;Checksum
                DC.L        880        ;# of blocks

BootCode        LEA         (DosName,PC),A1
                JSR         (_LVOFindResident,A6)
                TST.L       D0
                BEQ.B       NoDos
                MOVEA.L     D0,A0
                MOVEA.L     (RT_INIT,A0),A0
                MOVEQ       #0,D0
Done            RTS

NoDos           MOVEQ       #-1,D0
                BRA.B       Done

DosName         DC.B        'dos.library',0
```

On the other hand, custom boot code, or a virus might be very complex. Some of the other topics in this section may offer some clues. The best advice is that becoming very good at disassembling code is much like other pursuits; it takes a lot of work, practice, and experience.

➥ I've got a piece of code that is encrypted, and decrypts itself as it's run. How do I decrypt the code in ReSource so that I can have a look at it?

An example of this might be a virus boot block. ReSource can help you in this regard by providing you with source that may then be reassembled with symbol hunks. Disassemble the first part of the code that does the decryption as code, and treat the encrypted code as data by setting its data type to longwords.

After the code is reassembled with symbols, run it through your favorite debugger, and pay careful attention to where the decrypted code is being stored. After you have successfully regenerated the encrypted code, either give ReSource the address of the buffer and use the "Disassemble memory" function, or save the buffer and load it into ReSource as a binary file.

➥ How do I disassemble code that only executes at a specific address?

You may want to use ReSource to disassemble code from a ROM or another 680x0 based machine. In doing so, you deprive ReSource of a great deal of information about the code to be disassembled. For one thing, there is no reloc32 information present. Disassembling such code can be done, but it is time-consuming and tricky work.

Because ReSource cannot use internal information supplied by the loader to make decisions based on data type, it is best not to use the automatic "Disassemble" function. Doing so would likely create a lot of undesirable labels in the middle of instructions as ReSource runs into data or ASCII, and tries to disassemble it. Proceed slowly and carefully, making reasonable sure that an area is code before you attempt to disassemble it and/or create labels.

Credits

This manual was written by Jeff Lavin and Glen McDiarmid.

Much thanks to Doug Sears and Grace Lavin for ideas, proofreading, and general support.

Layout and typesetting done by Jeff Lavin on an Amiga 3000, using Professional Page, and an NEC SilentWriter LC 890.

Printing done by Print Worx of Eugene, Oregon.

Appendix A: Old and New Syntax

First there was the 68000. While a very nice chip to program, and utilizing a rich instruction set and many different addressing modes, it was limited to two operand fields. There was only a possible source and destination.

After a time the 68010 was released, but this didn't change anything; there was still only a possible source and destination.

More time passed, and the 68020 was released, and after it came the 68030. Now we have a whole new ball game. Both of these chips are capable of addressing up to four operand fields.

The problem here is that the original Motorola syntax didn't have room for the new addressing modes. Several assemblers valiantly tried to squeeze the new capabilities into the old syntax, but the results were so cumbersome and difficult that the effort was short lived.

In order to resolve this important issue, Motorola developed the M68000 Family new syntax. The new standard not only solved the problem, but is actually more robust, and helps to eliminate common typing mistakes. While it is beyond the scope of this account to cover the new 020/030 addressing modes, we will compare how the 68000 addressing modes are written in the old and new syntax. In all the examples below, the old syntax is show first, followed immediately by the equivalent new syntax:

Register Direct Addressing

Data Register Direct

```
MOVE.W    D7,D0
MOVE.W    D7,D0
```

Address Register Direct

```
MOVEA.W   D7,A0
MOVEA.W   D7,A0
```

Absolute Data Addressing

Absolute Short

```
MOVEA.L   AbsExecBase,A6
MOVEA.L   (AbsExecBase).W,A6
```

Absolute Long

```
MOVEA.L   #_custom,A0
MOVEA.L   #(_custom),A0
```


Program Counter Relative Addressing

Relative with Offset

```
LEA       There(PC),A0
LEA       (There,PC),A0
```

Relative with Index and Offset

```
JMP       $10(Table,PC)
JMP       ($10,Table,PC)
```

Register Indirect Addressing

Register Indirect

```
MOVE.L    (A0),D0
MOVE.L    (A0),D0
```

Postincrement Register Indirect

```
MOVE.B    (A0)+,(A1)+
MOVE.B    (A0)+,(A1)+
```

Predecrement Register Indirect

```
MOVE.B    -(A1),-(A0)
MOVE.B    -(A1),-(A0)
```

Register Indirect with Offset

```
MOVE.B    MP_SIGBIT(A0),D0
MOVE.B    (MP_SIGBIT,A0),D0
```

Indexed Register Indirect with Offset

```
MOVEA.L   4(A1,D3.W),D5
MOVEA.L   (4,A1,D3.W),D5
```

Immediate Data Addressing

Immediate

```
MOVE.L    #$BABEF00D,D2
MOVE.L    #$BABEF00D,D2
```

Quick Immediate

```
MOVEQ     #'A',D1
MOVEQ     #'A',D1
```

Implied Addressing

Implied Register

```
ORI       #1,CCR
ORI       #1,CCR
```

Appendix B: Strict and Relaxed Mnemonics

Standard Motorola mnemonics include some instructions that logically belong to a larger group of instructions. For example, the "MOVE" group of instructions include "MOVE", and "MOVEA". Most assemblers will accept the generic instruction name in your source code. For example:

```
MOVE.L    D1,A2
```

is accepted, but the strict mnemonic form would be:

```
MOVEA.L   D1,A2
```

Similarly:

```
OR.B      #$20,(A1)
```

is more correctly written as:

```
ORI.B     #$20,(A1)
```

Most assemblers give you the choice of which form to use. You may use the correct mnemonics, or you can use the "relaxed" forms of the mnemonics. Either way, your assembled program will generally assemble to the same code. If you are used to using "relaxed" mnemonics, you may wish to direct ReSource to use relaxed mnemonics when disassembling code.

The Macro68 assembler is capable of assembling faster in strict mode, and will generally pick up more types of errors in strict mode.

If you are still unsure of how "strict" and "relaxed" mnemonics differ, try switching between these two modes in ReSource, examining how the disassembled code differs each time you change.

Bibliography

The following books are recomended to those wishing to learn 680x0 assembly language, as well as those who desire to write programs for Amiga® computers.

Anyone wishing to program in 680x0 assembly language needs good reference material. The following book is written by the folks who make the M68000 family processors, and may be regarded as the final word on the subject. It covers the complete family including the 68040, 68882, and 68851.

TITLE: M68000 FAMILY PROGRAMMER'S REFERENCE MANUAL
ORDER #: M68000PM/AD
PUBLISHER: MOTOROLA INC.

Motorola literature can be obtained by mail or phone from:

MOTOROLA SEMICONDUCTOR PRODUCTS INC.
Literature Distribution Center
P.O. Box 20912
Phoenix, AZ 85036-0924
(602) 994-6561

The following books may be considered a good starting point for the beginning 68000 assembly language programmer. Although they do not have the depth of the Motorola manual, beginners may find them easier to assimilate. The first book takes a structured approach to teaching, starting with the basics, while the second book is more advanced. Both books offer many good examples.

TITLE: 68000, 68010, 68020 PRIMER
AUTHOR: Stan Kelly-Bootle / Bob Fowler
PUBLISHER: Howard W. Sams & Co., Inc. / The Waite Group
ORDER #: 22405
LIBRARY OF CONGRESS CARD NUMBER: 85-61636
INTERNATIONAL STANDARD BOOK NUMBER: 067-22405-4

TITLE: THE 68000: PRINCIPLES AND PROGRAMMING
AUTHOR: Leo J. Scanlon
PUBLISHER: Howard W. Sams & Co., Inc.
ORDER #: 21853
LIBRARY OF CONGRESS CARD NUMBER: 81-51553
INTERNATIONAL STANDARD BOOK NUMBER: 0-672-21853-4

The following books, although written with the C programmer in mind, offer clear concise explanations and examples for programming the Amiga®. The first book starts with basics, and does a fair amount of hand-holding. If you are new to the Amiga®, this book is for you. The set of books by Mortimore cover the entire Amiga® OS (as of V1.3) including all libraries and devices. There are in-depth discussions on selected topics as well.

TITLE: PROGRAMMER'S GUIDE TO THE AMIGA
AUTHOR: Robert A. Peck
PUBLISHER: SYBEX
LIBRARY OF CONGRESS CARD NUMBER: 86-63749
INTERNATIONAL STANDARD BOOK NUMBER: 0089588-310-4

TITLE: AMIGA PROGRAMMER'S HANDBOOK, VOL. I AND II
AUTHOR: Eugene P. Mortimore
PUBLISHER: SYBEX
LIBRARY OF CONGRESS CARD NUMBER: 86-62805
INTERNATIONAL STANDARD BOOK NUMBER: 0-89588-367-8

Of course the definitive source of information on the routines and data structures in the Amiga's operating system is the RKM and related books.

TITLE: Amiga ROM Kernel Reference Manual:
Includes and Autodocs, Third Edition
AUTHOR: Commodore-Amiga, Inc.
PUBLISHER: Addison-Wesley Publishing Co.
ORDER #: RKMIA20

TITLE: The AmigaDOS Manual, Third Edition
AUTHOR: Commodore-Amiga, Inc.
PUBLISHER: Bantam Books
ORDER #: DOS20

TITLE: Amiga User Interface Style Guide
AUTHOR: Commodore-Amiga, Inc.
PUBLISHER: Addison-Wesley Publishing Co.
ORDER #: RKMSG01

Index

aborting
macro conditionals 3-14
macros 3-15, 3-17
ARP
file requester 3-7
required library 2-2
wildcards 3-12
assembler, compatible 2-1
attribute bits 2-10, 4-6, 4-7
base-relative addressing 1-2, 5-2
BSS hunks See hunks
CHIP hunks See hunks
CODE hunks See hunks
code scan 2-14
command line arguments 3-2, 3-3
config See preferences
creating labels 2-16
creating symbols 2-15
current line 2-10
cursor
defined 2-10, 2-11
forward reference 2-16, 2-21
in data files 3-8
in macros 3-14, 3-17
locate comments 4-5
location stack 2-20, 2-21
previous location 2-16, 2-21
DATA hunks See hunks
data types
available types 1-2
defined 2-10
display locking 2-8
effect on display 4-7
illegal instruction 2-13
removing 4-8
set by code scan 2-15, 5-2
set from symbols 3-6
setting data length 5-5, 5-7
setting with mouse 2,9
use of attribute bits 4-7
with image-data 5-9
with multiple-constants 5-6

display
- default 2-6
- defined 2-9, 2-10
- fast 1-3
- interlace 3-3, 3-4
- stopping refreshes 2-9
- scrolling 2-8
- titlebar 2-6

disassembly
- code scan 2-14
- function 2-13, 2-14
- importance of data types 2-9, 2-10
- of a program 2-12, 5-1
- of disk tracks 1-2, 1,4, 3,3, 3,9
- of memory 3-9
- of viruses 3-3
- techniques 2-12, 4-1, 5-1
- using macros 3-13
- using the mouse 2-8

disk tracks 1-2, 1,4, 3,3, 3,9
DT 1-2, 5-2
escape sequences 3-12
file requester 3-7

files
- .asm 2-15, 3-5, 3-8, 3-10
- .rs 3-2, 3-5, 3-8
- binary 3-7
- load 3-5, 4-1
- overlay 3-6

hunks
- chip 3-6
- hiliting 4-6
- in binary files 3-7
- in load files 3-5, 4-1
- overlay 3-6
- reloc32 3-5, 3-6, 4-1
- saving executables Introduction
- symbol 3-5, 3-6

installation
- on floppies 2-2
- on a hard drive 2-3, 2-4

key-binding 2-7, 3-19

labels
- automatic creation 2-8, 2-9
- config options 3-11
- creating 2-16
- defined 4-1
- duplicate 4-2
- from code scan 2-15
- from reloc32 3-6
- from symbol hunks 3-5
- hidden labels 2-11
- hiliting 4-6
- in macros 3-14
- local labels 3-6, 4-2
- memory usage 1-3
- O'lay binary image 3-7
- overview 1-1
- removing 4-8
- shop labels 4-1
- string indirection 3-15

libraries
- ARP required 2-2
- other 2-2, 2-3

limitations 1-3
memory usage 1-3

menus
- help for 3-1
- library 2-3
- multiple selection 2-5
- names 2-6
- setting from preferences 3-11
- speeding up 2-2

numeric bases
- example 5-7, 5-10
- overview 1-2

online help 3-1, 3-2
overlay hunks 3-6

preferences
- saving 3-11
- symbol bases 3-18

priority, set 3-4

reloc32
- in load files 3-5, 3-6, 4-1
- hiliting 4-6

running from the CLI 2-5, 2-6
running from Workbench 2-5, 2-6
screen See display
searching
overview 1-2, 3-11
escape sequences 3-12
in macros 3-14
removing labels 3-11
simple example 5-13
with patterns 3-12
showkeys 3-19
showmacros 3-19
speeding up menus 2-2
symbols
config options 3-11
creating 2-15
defined 4-3
disabling display 2-8
memory usage 1-3
O'lay binary image 3-7
operand fields 4-4
overview 1-1
removing 5-1
standard symbol bases 1-2
user-defined symbol bases 3-18
syntax
new Introduction, Appendix A
old Introduction, Appendix A
raison d'etre 2-1, Appendix A
compatibility 2-1
titlebar 2-6
tooltypes 3-4
undo
removing labels 4-8
removing symbols 5-1
user-defined symbol bases 3-18
utilities
showkeys 3-19
showmacros 3-19
zap 3-10