Table Of Contents

Credits, Acknowledgements, Copyrights and Your Rights

Credits

MacNosy was programmed by Steve Jasik, aka the Head Nose, with some help from Andy Gottlieb.The original and infamous **MacNosy** documentation was written by Steve Jasik. The current documentation was pulled together by Stan Krute. It includes work done by Steve Jasik, Howard Katz, Stan Krute, David W. Smith, Judd Stiff, Michael Swaine, and Nick Turner. The nice people at SuperMac Technology, in the midst of riding their own tornado, graciously provided LaserWriter facilities and expertise.

When this documentation uses a first–person pronoun -- I, me, my, mine, etc. -- Jasik is speaking.

Acknowledgments

When I started coding MacNosy I was undereducated in the way of the Mac and under capitalized. With the assistance from all of you who bought it, and the below, it has become a living, breathing reality, fulfilling my credo of free exchange of ideas and code in the software marketplace.

Special thanks go to:

Larry Rhyne (publisher of Cat-Mac) who let me use his Lisa until I could afford my own.
Wayne Martin, who capitalized me.
Dennis Brothers for helping spread the word on CompuServe.
The members of the press who spread the word in print (Fred Davis, John Markoff, Steve Bobkor, ...).
John Mitchell and Bill Parkhurst for extensive testing and encouragement.
Cary Wyman (MindWrite) who was customer number Uno

Copyright

Your Rights and Privileges

Your purchase of **MacNosy** gives you a number of rights and privileges.

- You can use the program on any computer you have access to.
- You may obtain updates **within a version.**

Updates and Versions

MacNosy is **updated** to correct bugs and add new features. Free updates are mailed out to registered owners of the Universal Version on an irregular basis (3 to 6 months) during the first year of ownership. After the first year, you will be billed for continuing support, which includes updates for that year.

Scope

Features and examples in this document apply to Nosy 2.9x and later versions.

Help

You deserve a useful, friendly, bug–free program. I need your help to keep moving in that direction. I'm open to all suggestions, so please send me your feedback. I am available for tech support questions via telephone or on the following services:

Internet	macnosy@netcom.com	(**preferred**)
Compuserve	70277,776 [Steve Jasik]	

A Word From Steve

Back in my old CDC days, when we wanted to get a project done, we threw a bunch of programmers in a locked room, shoved food under the door at regular intervals, and didn't let them out until they passed back a suitable design or program. After numerous frustrations, of trying to find the right documentator, Stan appeared on the scene, dove into the project with tremendous energy, and put together the work of art and love you have before you. I hope you give it an evening or so of **active** reading in front of your Mac to become familiar with the many features in Nosy.

Steve Jasik, Menlo Park, Jan 87

A Word From Stan

I get a call from Dan Weston one night. He's tied up, Jasik needs a nerd who likes to write, and can I think about it ?

A month later, I'm parked in the front room of 343 Trenton. There's good food, a great stereo, lotsa Macs, and the mind of the Head Nose to pick. Down from the Siskiyou Mountains, I'm on a mission, chance representative of the Mac hacks, trying to grease the **Nosy** learning curve.

Back in the early days, when 128K Macs shipped out of Cupertino crippled and full of promise, I found myself seeking wisdom. Some things Apple tech support wasn't allowed to tell us. Along came **Nosy**. Complete with the now-classic TTY mode and hyper-condensed instruction sheets. I was young, and desperate. The ROM quivered, strained, started to leak, then gushed big and wide under **Nosy**'s ministrations. Fervent thanks flew south to Steve.

The wheel's turned. Apple's waking up, **Nosy** has windows, and I get to do some payback by coordinating this latest iteration of the **Nosy** documentation. My sincere thanks to Steve, Dan, and all the writers who worked on this project before me. I couldn't have done my part without their patience, work, and love. Besides, it's an honor to be in their company. I hope this documentation helps you dive into Nosy. Hang in there, play like crazy, and keep up the feedback. Steve's one of us, and he's listening hard.

The great programs are yet to come. **Nosy** and **The Debugger** are powerful tools that can help you bring them to life.

Stan Krute, Menlo Park, Jan 87

Documentation Conventions

A Short Glossary

We try to hew closely to standard Apple and Motorola terminologies. We try to minimize jargon and obscure symbolism. What follows is a brief list list of lapses. Be sure to check out the section **Program Conventions** for a list of goodies used in the program itself.

Δ stands for an otherwise invisible space, as in the file name **ΔROM**

the **BCUR** the **Block Currently Under Review** when you're reviewing data blocks

boldface type used for command names, file names, key names, window names, menu names, anything else we think should stand out

Cmd stands for **Command**, Apple's name for the key with the funny cloverleaf design

the **CNR** the **C**urrently **N**osied **R**esource(s) -- what it is you're examining, be it ROM, a set of CODE resources, or another code-containing resource

the **CNF** the **C**urrently **N**osied **F**ile -- where the **CNR** comes from, be it a file or an application running in another Switcher partition.

cnf_id the name of the **CNF**, used to build generalized, kindred names. For example, ***cnf_id*.jrnl** signifies a journal file associated with the **CNF**.

fba **f**irst **b**yte **a**ddress, the beginning location of a piece of code or data

hexit a hexadecimal digit

IM **Inside Macintosh**, the bible

menuName:commandName this is how we refer to a command from a menu : the menu name, followed by a colon, followed by the command name. Example: the **Display:Data Blk**s command

Nosy often used like a verb, as in: Nosy a file, the Nosied resource, Nosying around, a standing Nosation

OS stands for Operating System, as i : OS Trap call

the selection the currently-selected text in the topmost window

TB stands for Toolbox, as in: TB Trap call

WME in the context of a TTY mode command, its **W**indow **M**ode **E**quivalent

An Informal Descriptive Syntax

Some parts of this documentation need a descriptive syntax. We do it pretty informally. Literal characters are printed in a normal typeface, like this. Characters that are stand-ins for other things come in italics, *like this*. Optional elements are surrounded by curly brackets {like this }. Mandatory elements are either naked or surrounded by arrow braces<like this>. When choices are involved, they're separated by vertical lines, like | this. We try to explain everything in context, so you don't have to memorize a bunch of terms.

Program Conventions

Abbreviations

@	at	lbls	labels
att	attribute byte	len	length
blk	block	myst	mystery
c@	code at	OS	Operating System
Cmd	**cloverleaf** key	n_branch	number of branches
cmds	commands	p	procedure
c	common	pt	point
conf, config	configuration	rel	relative
cons	constants	refs	references
ctl	**cloverleaf** key	rsrc	resource
d	data	sel	select
errs	errors	stmts	statements
fba	first byte address	s	System symbols
fmt	format	sys	System
funRslt	function result	syms	symbols
g	global	t	trap
glob	global	TB	Toolbox
indx	index	tbl	table
inst	instruction	tbl_addr	table address
#inst	number of instructions	trp	trap
jmp	jump	wind	window
jmp_addr	jump address		

Names

In the following form descriptions: *number* represents a decimal number
xx represents two letters.

form	indicates	example
glob *number*	a program global variable	glob1
data *number*	a data block	data354
proc *number*	a procedure block	proc21
com_ *number*	a common block	com_74
l*xx*_ *number*	a label local to a procedure	laz_2
v*xx*_ *number*	a local variable referenced off a stack frame	vrs_1
param *number*	a parameter referenced off a stack frame	param2
funRslt	a function result referenced off a stack frame	funRslt

Name Lists

Some TTY mode commands take name lists as parameters. In these lists, individual elements are separated by commas. An inclusive range of elements is indicated with a hyphen. For example: the list **proc1**, **proc7**, **proc9–12** indicates six procedure blocks: 1, 7, 9, 10, 11, and 12.

Register-Based Trap Call Symbols

In the following form descriptions: *reg_id* represents a 68000 register descriptor: A0, D3, etc.
var_id represents a variable name
type_id represents a Macintosh type name

form	indicates	example
reg_id | *var_id* : *type_id*	an input/output (VAR) parameter	A0|IOPB:ParmBlkPtr
reg_id / *var_id* : *type_id*	an input parameter	D0/Count:Integer
reg_id \ *var_id* : *type_id*	an output parameter	D1\Size:Integer
reg_id \ *type_id*	a function result	D0\OSErr

Files On The Disk

There are lots of files on your **Nosy/Debugger** disk. Some of them are required to run **Nosy** (see **Installation Notes)**, others are optional or bonuses. Here's the current list:

Launch Nosy	a dummy document that lets you launch **Nosy** no matter where it's living in an HFS System
Nosy	the application itself
sij_*xxxx*	files of tables and constants used by **Nosy**, with *xxxx* replaced by a descriptive string
-Help-	**Nosy**'s window mode help file
xxxx.aci	additional comment information attached to specific files, *xxxx* indicates the file attached-to
ΔROM	a dummy document that lets you select the Mac ROM to disassemble
ROM/CODE.snt	a record of **Nosy**'s ROM analysis used when you disassemble the Mac ROM
LisaPas_GL	small sample applications you can disassemble as you start to learn the art of Nosying
Amacs	assembly language macros necessary to assemble a file 'sourced out' by Nosy
xxxx.jrnl	various journal files you can use as you start to learn the art of Nosying, some for applications on the **Nosy** disk, some for other common applications and ROM, with the *xxxx* replaced by the journaled file
copyROM128	a small application that copies a 128K ROM and the first $9000 bytes of RAM to a disk file. Used as a simple Nosy example.

File Naming Conventions

MacNosy marks its files with a variety of prefixes and suffixes. Some names[1] are truncated to ten characters, but these markers remain intact. The descriptions below follow this pattern:

marker **w**hat **i**t **s**tands **f**or – some explanation
a few examples

.aci[1] **a**dditional **c**omment **i**nformation – text file that holds comments you've added to a resource's disassembly
examples: MDEF0.aci Nosy.aci Word.aci

.jrnl[1] **j**ou**rn**a**l –** text file - holds a record of TTY mode commands you've chosen during a resource's disassembly
examples: MDEF0.jrnl Nosy.jrnl Word.jrnl

.npl **N**osy **p**attern **l**ibrary – holds patterns Nosy uses to feign intelligence during a disassembly
examples: Mac_C.npl LisaPas.npl

sij_ **s**tephen **i**van **j**asik (it's "sij/" in V2 and "sij_" in V3) – used to hold constant tables Nosy uses
examples: sij/typinfo.c sij/Colors sij/disatbl

.snt **s**aved **n**osy **t**ables – copy of the internal tables Nosy requires to restart a session
examples: System/MDEF_0.snt Nosy/CODE.snt Word/CODE.snt

tf_ **t**emporary **f**ile – used by Nosy for miscellaneous storage needs - should disappear when you leave Nosy. (You may delete these with impunity).
examples: tf_AA tf_AB tf_AC

Installation Notes

Nosy is installed automatically as part of The Debugger installation. It resides in the folder 'Debug/Nosy files'

If you are using System 7 or later, you can make an Alais to it and put it on the desktop or some other convenient place.

About MacNosy

What, Why, And How

MacNosy is an information recovery tool for Macintosh programmers. I wrote it because I needed it. MacNosy takes object code resources, and tries to return a good approximation of what the programmer intended when he wrote the code.

A Disassembler, And More

MacNosy is a disassembler, and more. A disassembler takes object code and robotically transforms it into assembly language. The problem is that most pieces of code contain embedded data as well as legitimate machine instructions. A disassembler stumbles when it hits data, at best indicating that the offending bytes are not instructions. Macintosh programs are particularly complex in this regard, with code being used as data, data being used as code, and lots of funny nooks and crannies.

MacNosy is much more intelligent than a standard disassembler. Instead of blindly stepping through the code, MacNosy attempts to reconstruct the original structure of the program the code implements. With some help from you, MacNosy recognizes data, works around it, and uses it to smarten up the disassembly of the surrounding code. It's particularly bright when it encounters the wide array of specialized Macintosh programming and data structures. MacNosy performs pattern-seeking recursive searches, called treewalks, that allow it to produce an internal model of a program's structure. While MacNosy does need human help at times, it eases things by providing a powerful set of codoscopic tools. In the long run, MacNosy is intended to be a true decompiler, more and more automatic, returning not just assembly code, but a decent reconstruction of high-level source code. Each update moves closer to that goal.

You And Nosy -- Together

You and Nosy work together on a piece of code, in a rhythmic kind of motion. Nosy starts the dance with an initial treewalk through the code. Then the two of you look over the results, and remove a few of the ambiguities. Then you lead Nosy into another treewalk, eliminate some more ambiguities, and keep cycling around until you've found what you're looking for. Each version of MacNosy gets more intelligent and automatic, but there are still areas where the program requires your partnership.

All Kinds Tricks

Working with MacNosy, you can perform all kinds of tricks. The program can be used as

1. ... an online database for programmers, supplying reams of information on the Mac and the 68000. MacNosy can produce accurate and informative listings of the ROM, and any code-based resource living in a disk file. It knows about the special structure of DRVR's and PDEF's. This is its most static role.
2. ... an educational tool for learning 68000 assembly language and the structure of Macintosh code. The MacNosy disk contains a number of examples, including an extensively commented fast Shell sort. But you can study anyone's source code, not just cooked examples. Want to know what Andy Hertzfeld did in Switcher and Servant ? Just MacNosy the suckers.
3. ... a way to defeat copy protection. Unlike programs that apply fixed algorithms to counter specific copy-protection techniques, MacNosy is a programmer's tool that allows you to examine and deal with whatever technique has been used.
4. ... a tool for inspecting and improving your own code. You can use **MacNosy** as a source-level cross-reference map facility to aid in tracking down bugs or making modifications to your code. **MacNosy** is used intensively today in its own development, for successive refinement of code initially generated by MPW Pascal. **MacNosy** can create **.asm** files that (with a bit of masasage) can be fed into assemblers .
5. ... a tool for modifying applications and desk accessories. Sometimes you 'd like to do a little code massage, so you **Nosy** the trouble spots, then jump to a file editor like **Fedit Plus** to macerate the offensive code/data.

Are We Having Fun Yet ?

Effective MacNosy usage requires knowledge of Macintosh programming principles and 68000 assembly language. Read the section **Alternate Sources Of Truth** for a guide to learning materials. Then, properly prepared, you can go through this documentation and work out some simple disassemblies. The more you fiddle with Nosy, the faster you'll reach escape velocity, and that's when the fun really begins.

Introductory Cruise

What Is A Block ?

Okay, here's one of Nosy's big ideas: a program can be viewed as a set of blocks.

A block is a **contiguous set of bytes** identified by a first byte address (fba) and a length. Nosy classifies blocks as **code or data**, and further classifies code blocks as procedures or common. A block reached via a JSR or BSR instruction is a procedure block. A block reached via a jump or branch instruction -- JMP, BRA, Bxx -- is classified as a common block. A block referenced by an LEA or PEA, or not reached by one of the code-classifying instructions, is classified as a data block.

Most programmers aren't idiots. A block of code usually performs a coherent task. Seeing a program's blocks gives you big clues as to what the code's up to.

Nosy's initial tree search produces a set of block classifications. Due to wrinkles and complications, these classifications aren't perfect. As you dissect a program with Nosy, you supply information that lets it refine the block analysis on subsequent tree walks.

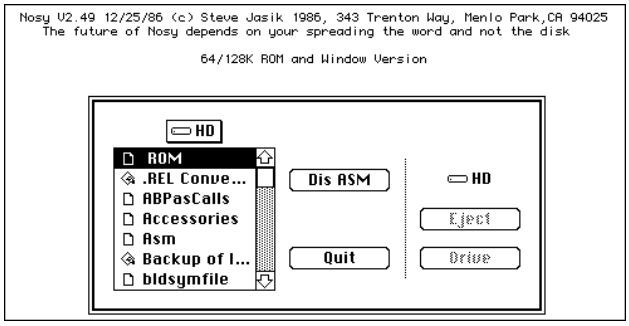
Okay, that's enough of big ideas. More later. Let's move on to an actual code-cracking session experience.

Hands On

Every Nosy disk comes with a small application called CopyROM128. Historically it was there so folks with Mac XLs (Lisas) could disassemble the Macintosh ROM. Since Mac XLs don't have a Mac ROM, Nosy needs a copy of the ROM to work with. CopyROM128 builds such a file. In our introductory cruise through MacNosy, we'll take a look at CopyROM128, seeing what Nosy can tell us about it.

Start by launching MacNosy. You can do this by double-clicking on the icon for Launch Nosy. Launch Nosy is a little kludge that makes it easy to start Nosy under both Mac file systems, HFS and MFS. See **Installation Notes,** Note #2.

You'll see something like this when Nosy starts up :



This standard file dialog lets you choose the file you want to disassemble. Search around until you find CopyRom128, then select it.

Two Dialogs And A Treewalk

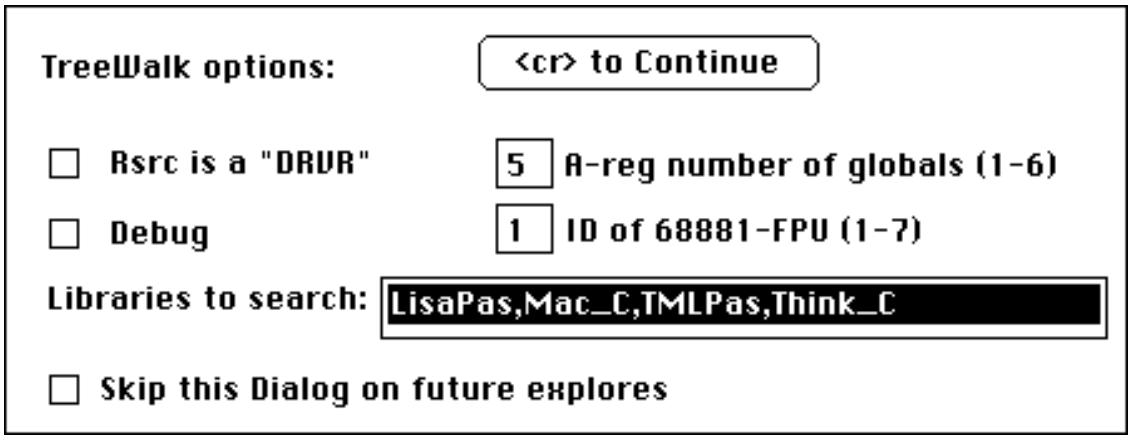
Nosy goes into its TTY (teletype) mode, where it acts like an old–fashioned computer terminal, and you'll see the following interactive dialog. Note that the "y" at the end of the second line is typed by you and not the program, followed by a **Return**.

```
 copyROM128   2K bytes read
list resources [n]  ?y
       1 resource types, data index =   100
Type CODE    att   indx  length
  ID      0   20      0      18
  ID      1   34     1C     3CA
 enter a leading ">" to bypass uppercasing the rsrc type
disassemble resource type [CODE] ?
```

The first line above simply reports the name of the program that you're disassembling and its length. The second line asks if you want to display all the resources Nosy has discovered in the resource fork of the file. The **default is [n] for No**, shown in brackets. If you want the default selection, simply hit the **Return** key. In this case, we want to view the resources, so we've entered a "y" for Yes and pressed **Return**.

Nosy has found only two resources in the file, both of type CODE. Following the "Type" entry are two lines which summarize what Nosy knows about the resources. They have **ID**'s of 0 and 1, respectively, and the three fields following the ID are **att**, **indx**, and **length**: **att** shows the resource attributes of each resource, **indx** shows the offset of the resource in the file's resource fork, and **length** gives the size of the resource, all in hexadecimal bytes.

The last prompt asks what particular resource type you'd like to disassemble - the type 'CODE' is always the default. Press **Return** and the following **TreeWalk options** dialog appears on the screen. You don't have to know about the various options at this point; simply press **Return** or click in the button at the top of the dialog.



Nosy goes into Explore mode, making an initial treewalk through the resource in an effort to determine what's code and what's not. Nosy will post some messages to the screen as it goes; with a short file like this one, these messages will probably fly by too quickly for you to read.

Window Mode, With A Pair Of Them

Once Nosy finishes its initial treewalk , the menu bar will appear on the screen, followed by two windows, –**Notes-** and **-Code Blks-**. These two appear whenever Nosy enters Window mode. The **-Notes-** window posts messages from Nosy. You can also use it as a scratch pad. In this case, it informs you that Nosy's named 10 Library routines. We'll explain this in a moment.

The **-Code Blks-** window shows a named list of all the blocks in the application that Nosy initially determined to be procedures, as discussed earlier. Take a good look at the names.

Note that several of the names appear meaningful, while others have generic names like proc4 and proc14. What's going on?

Whence The Names ?

There are several possible sources for procedure block names. First, one procedure gets the name of the resource you're Nosying, CopyROM128 in this case. That block's first instruction is the entry point to the code. An easy name for Nosy to assign.

Next possible source: some compilers can put procedure names right into the object code after each procedure. That's for the benefit of debuggers. Nosy can recognize and use these names.

Nosy can also reconstruct the names of procedures from a MDS style or MPW **.MAP** file that may have been produced during the development process.

Nosy can also use library files, marked by the suffix **.npl**, to help in its naming activities. If it finds a pattern match between a procedure block and a library entry, it assigns the name of the library routine to the procedure. Take a look at the **TreeWalk options** dialog we roared past earlier. One of the dialog items is "Libraries to search", and Nosy supplied a default entry of "LisaPas,Mac_C,TMLPas,Think_C". This dialog item specifies library files Nosy is to search when it does a tree walk. You can specify other libraries for these searches, and even build your own libraries. (See the section **Building Pattern Libraries**).

Nosy also knows about ROM traps and System globals. It can name procedures based on this knowledge. If you know the ROM and System globals, you'll recognize those names.

After all the above techniques have been applied, some procedure blocks may remain unnamed. Nosy simply names each procedure as it finds it, using the form **proc**#, where # is a number. The reason the numbers aren't sequential is that *all* the code blocks, even those given more meaningful names, receives an implicit number. So **proc14** is really just the 14th procedure block, in order of address, named by Nosy.

Leaf Blocks

Any procedure in the **-Code Blks-** window whose name is preceded by a ">" is a **leaf procedure**. A leaf procedure doesn't call any other procedures. They're at the end of a calling chain. Such procedures are a good place to start analyzing and renaming code blocks.

What's To Be Done Now ?

There are a number of things we might do at this point Let's start by examing some of the data blocks to see if they're really code. Go to the **Reformat** menu and choose "Review." Windows disappear, then new windows pop up. MacNosy has entered its **Review Data mode**.

Reviewing Data

Before we get our feet too wet, let's talk about reviewing data. This is a process that you can do many times during the disassembly of a program. Most Nosied resources will have code blocks that show up as data. Reviewing is a process whereby Nosy shows you all the data blocks that it's "not sure about" and asks your opinion as to whether they're really code or not. You can also reformat and rename data blocks to more closely match their usage.

Getting everything categorized is an iterative process: first you Review the data , and if you make some changes to the blocks you then tell Nosy to do a a treewalk via the Explore command. Once Nosy finishes the treewalk, you can do another Review, and then another treewalk. The process is repeated iteratively until you're satisfied that the data blocks are properly classified, or you want to use another of Nosy's tools.

Back To The Screen

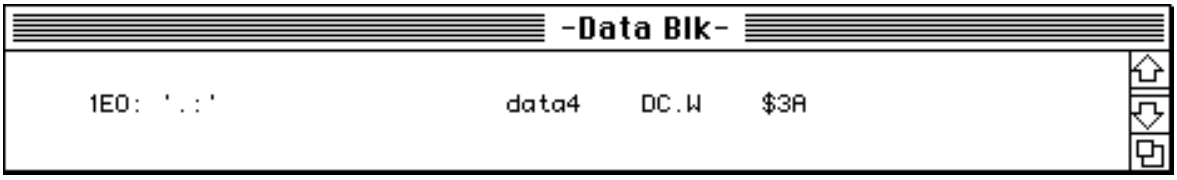
Alright, enough theory. You've selected Review, and four new windows appear on the screen. The topmost window contains a TextEdit box labelled "**Cmd**." A cryptic listing of the commands available in Review mode appears in a window below the TextEdit box and in the figure shown below. The second window, titled **-Data Blk-**, shows the contents of the data block under review. The last window, titled **-Use-**, is partially obscured beneath the **-Data Blk-** window. If there's a reference to the data block in a code block, **-Use-** shows it.

```
<Hex|Dec|Asc|Zero><B|W|L>n_item, Wstr, Str, WZSTR, Zstr, Jumpc, JUMPP, BRA<L|C>
Undo, Code,  Quit, New{Byt} cnt, <NewLast|NUntil>{hhh}, NewWstr, New*, cOmbine
Mname=fmt1,fmt2., =mac_name, <H|D|A><Fld>cnt, ADDR, LADR<C|P>, DRVHD=pfx
```

This Review menu has all kinds of powerful commands. See **The Review Data Blocks Menus** section for a complete rundown. For now, though, we're interested in changing data blocks to code. Look carefully, and you'll find that the second entry in the window's second line reads **Code**. Just what we need.

Format Of The -Data Blk- Window

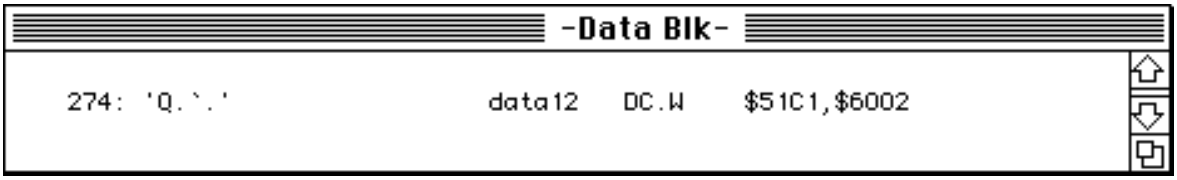
You should be looking at a -Data Blk- window that looks like this :



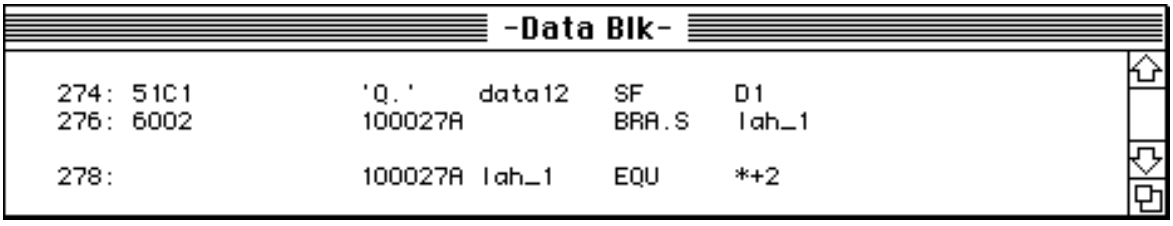
The hex number in the leftmost column is the **fba** of the data block, which is the **segment relative address** of the block. The second column shows the ASCII equivalent of the contents of the data block, which are shown in hex on the right. The "." (period) following the first quote sign indicates a value for which there is no ASCII equivalent (or it could be a real period). The third column holds the name of the block, data4 in this case. Data blocks are numbered consecutively. Since this block came up for Review, Nosy "typed" the prior three data blocks. Finally, Nosy assigns a **DC.W** (Define Constant word) pseudo-op to the data. If you're not familiar with the standard data pseudo-ops, check a 68000 assembly language manual.

Looking For Code

Could this data block really be code in disguise? It doesn't seem likely. Let's continue on. Press **Return** to get to the next questionable (untyped) block. The -Data Blk- window will now contain the following:



This appears a little more promising. Let's take a look at what this data would look like if it were disassembled as code. Type a **C** for Code and see what happens:



This looks promising! The two numbers in column two, 51C1 and 6002, show the hex equivalents of the instructions on the right. The number 100027A represents the target address of the BRA.S instruction. This number is Nosy shorthand and consists of a segment number followed by the six hexit offset within that segment. In this case, there's only one segment in the program. The EQU value indicates that the branch is to a location two bytes beyond the value of the Program Counter when the branch is executed.

The label **lah_1** is a local label. That is, the label is local to the procedure block it occurs in, which is somewhere else. Given the two-byte, PC-relative offset of the label, this procedure is obviously the next block in the program. The **l** stands for "local"; the **ah_1** was assigned by Nosy to uniquely identify the label.

We've been temporarily viewing the block as code. Now we have to tell Nosy to mark it as such for its next treewalk. Nosy has anticipated this, and entered an **i** in the command line. This **i** stands for **Is_proc**, or **Is a procedure**, as indicated in the **Code** menu that appears where the **Review** menu was. To flag the block as a procedure, simply press **Return**.

Note: If you don't want to mark the block as code, make sure that you backspace to delete the **i** and then type an **r** to **Rever**t the block to its original format. The block will appear as it did, and the **Review** menu reappears. You're back to where you were before pressing **c**.

The final block that comes up for **Review** is **data13**. It it garbage that the Lisa WorkShop Linker puts at the end of a code segment. If you type **C** to see it as code, you get the following nonsense :

```
3BE: 8100          '..'    data13   SBCD    D0,D0
3C0: 0008 0000     '....'           ORI.B   #0,A0
3C4● 03BE          '..'             BCLR    D1,
3C6● 0000          '..'             DC.W   0
3C8: 0100          '..'             BTST    D0,D0
```

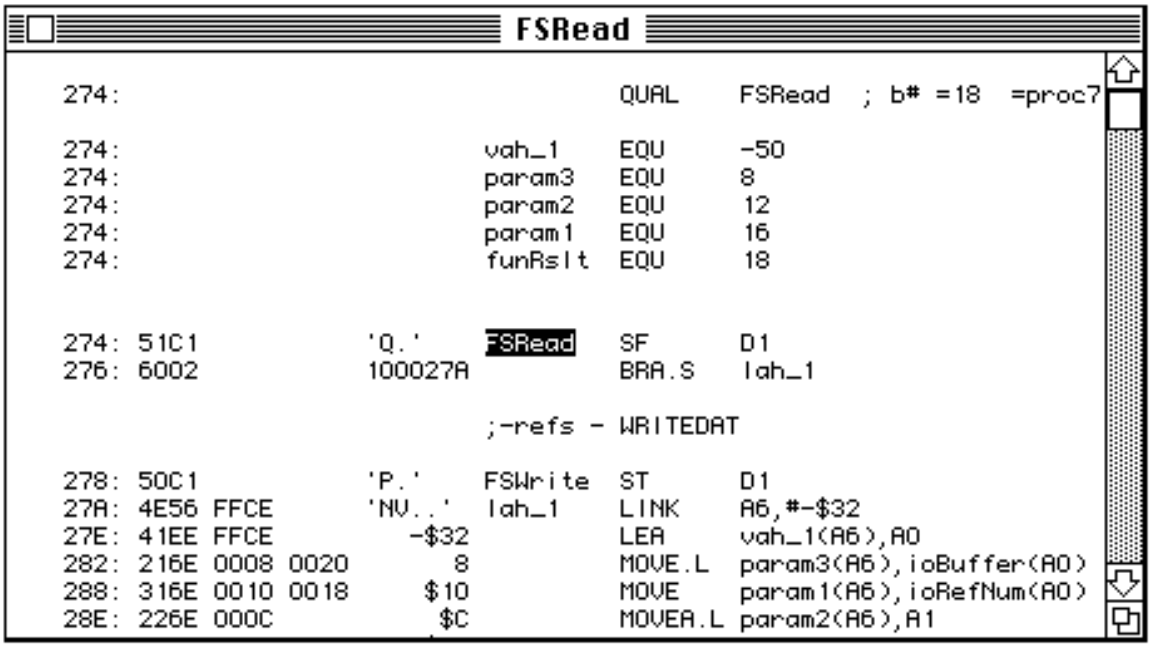
As a sequence of instructions, this makes no sense. Nosy has even marked two of the lines with a bullet to indicate their flakiness. Backspace to get rid of the default **i**, then enter **R** to **Revert** to data. Press an "empty" **Return** to move on to the next block. There aren't any others to examine, so Nosy exits Review mode and returns you to the standard Nosy Window mode desktop. You can always force an exit from Review mode by entering **Q** for **Quit**.

Time For Another Treewalk

At this point, you've marked one data block as code. You'll notice that this new procedure block has shown up in the **-Code Blks-** window as **proc7**, between **FSClose** and **FSWrite**. The next step is to select **Explore** from the **Reformat** menu. That starts another treewalk. When you make the selection, Nosy goes back into TTY mode and the **Treewalk options** dialog reappears. You can check the **Skip this Dialog on future explores** option to do what it says (you could also have done this the first time around). Press **Return** or click in the top button to do the new exploration. Once Nosy has finished exploring, Window mode reappears. Notice that Nosy, in checking through its Lisa Pascal library, has determined that **proc7** matches the library routine **FSRead** and renamed it as such. This is a fairly tame example - sometimes the results of an **Explore** are dramatic, and dozens of new procedures appear in the **-Code Blks-** window.

Looking At A Proc Block

Let's look at the **FSRead** procedure block. Select the **FSRead** entry by double-clicking on it. Go to the **Display** menu and select **Code|Data blk**, or type its keyboard equivalent, **Cmd-D**. The following window appears:



Note that Nosy has highlighted the procedure's label, indicating its entry point. Here's a nice trick : select **Refs to** from the **Display** menu, or type **Cmd-R**. A small reference window will appear, showing the names of all the procedures that call **FSRead**. In this case there aren't any, so Nosy will only display the name of the procedure, its segment number, and offset. Go down a few lines in the block and highlight **FSWrite** and do the same thing. This time, the **Refs to** window will show you that **FSWrite** is called from somewhere in the procedure **WRITEDAT**. If you wanted to, you could highlight the word **WRITEDAT** and do a **Cmd-D** to bring up its listing. Then bring up its references. And so on. Tracking chains of references like this is a good way to get a feel for what a program's up to. There's even a shortcut command, **Call Chain** in the **Display** menu, that shows a complete chain of calls in one window.

The Block Header

The **FSRead** block's header contains a lot of information. Let's take a closer look. The **QUAL** label at the top indicates that all the labels that follow, to the bottom of the listing for the procedure, are meant to be local, except for the main entry points, **FSRead** and **FSWrite**. The top line in the listing also says that this procedure is the 18th block of any type in the program, and the 7th procedure block.

There are five label/offset equates (`VEQU`) in the block header. They make it easier to follow the stack-frame conventions in the code. Labels that begin with a **v** refer to the offsets for local stack frame variables that are set up by a LINK instruction. The LINK instruction at $27A allocates $32 (50) bytes on the stack for local variables and loads the current Stack Pointer into register A6. Labels that begin with **param** refer to offsets for parameters sent to the procedure. **funRslt** refers to the offset for a returned function result. Nosy is intelligent enough to recognize that three parameters, and space for a function result, have been placed on the stack by the calling routine **WRITEDAT**, and is able to recognize the required offset values.

Nosy Aids Trap Comprehension

We can learn some other good Nosy tricks by looking at the code in this window. Scroll down until you see this line :

```
2A2: A002          '..'            _Read    ; (A0|IOPB:ParmBlkPtr):D0\OSErr
```

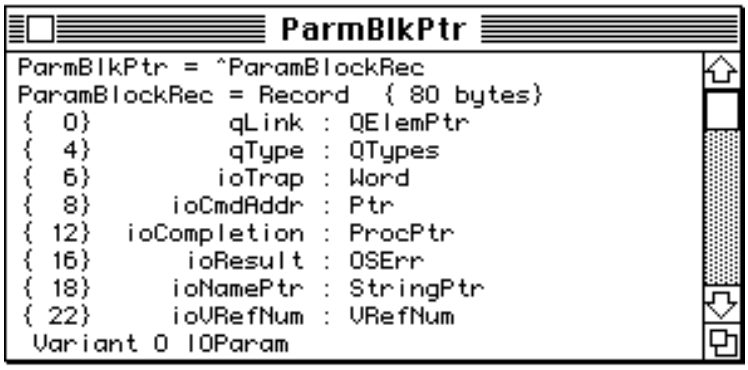
$A002 is the OS trap **Read**. Nosy has helpfully placed the parameters required by the Read call in parentheses at the end of the line in a Pascal-like notation. The vertical bar ("|") separating the register name **A0** from the name of the single parameter, **IOPB**, indicates that register **A0** is used to pass the parameter, and that the parameter is used both for input and output (the equivalent of a Pascal **VAR**). A slash ("/") would be used if the parameter was an input parameter only, and a backslash ("\") would be used if the parameter was for output only. Finally, the **D0\OSErr** entry at the end of the line indicates that the trap is actually a function that returns a value of type **OSErr** in register **D0**.

Help

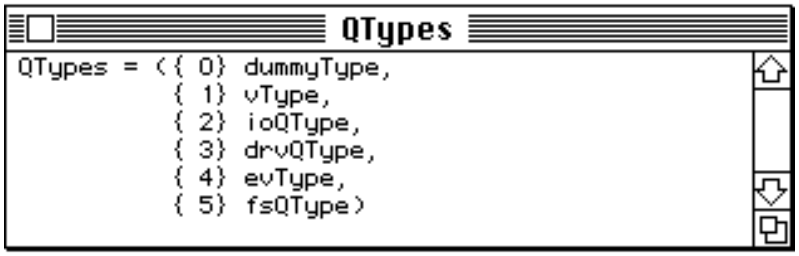
Nosy has a **Help** facility for times when the above notation or anything else slips your mind. Go to the **Apple** menu and select **Help**. A **Help** window appears. It's got a table of contents at the top. To find information on a particular topic, double-click to select its identifier and type **Cmd-F**, for **Find**. If Nosy finds a match, it will scroll the information that you're looking for to the middle of the window . To position the selected line of the info at the top of the window, type **Cmd-T**. To view the information on the parameter-passing conventions just discussed, look under the heading **_Tables**.

Inside Mac Online

Here's another goody : at any time you can get information about the various constants and record structures used by the Macintosh ROM. If you don't feel like dragging out **Inside Macintosh** to see what a **ParmBlkPtr** is, double-click on the word **ParmBlkPtr** in the above line, then type **Cmd-Space**. This is the keyboard equivalent for the **Fields of...** command in the **Tables** menu. What you get is this window:



Stupendous! This window shows *everything* you'd ever want to know about paramBlock records. The numbers in curly braces down the left side show the byte offsets of each of the fields in the record. You can continue to use the **Fields of...**command to bring up further information on the other fields in the record. Hilight the word **QTypes** and do a **Cmd-space**, you'll get the following:



This structure is not a Pascal record, but an enumerated type. The type identifiers are in parentheses, with numbers in curly braces that show the numeric values. The upshot : any time that you find a named Macintosh data structure, you can select it and press **Cmd-space** to get a display of its structure.

And there's more. **Cmd-space** is just one of the commands in the **Tables** menu. Every item in this menu brings up some listing of Mac-related information. For example, the **Sys Syms** (**Cmd-W**) command shows a listing of the Mac's low-memory global variables. These features turn Nosy into an electronic encyclopedia. For now, we'll let you explore the extent of this feature on your own.

Back to TTY Mode

Some useful Nosy tasks can only be done in TTY mode. You can send a complete listing of a disassembly to the screen or a text file. You can also search for various types of program objects, or produce reference maps of how program objects relate to one another. Select **to TTY mode** from the **File** menu or press **Cmd-Q**.

TTY Mode's Main Menu

Once you're in TTY mode, Nosy posts its TTY mode **Main** menu:

```
Review{Untyp|All}{d#}, By_attr, Explore, Display, Allproc{seg#}, Proc_byname, addr
Srch <Mark | Hex hhh | Str chrs | Rsrc_typ chr4 | {Addr_ref | Trap_ref}<name_list> >
 Hardcopy, +Jrnl, Wait, Tbls, List_rsrc{type}, ref_Map, Outdir{file}, saV_snt, Quit
 ?
```

The question mark is the **Main** menu's prompt. If you type anything, it appears next to the prompt. The menu shows a number of possible commands, some of which take parameters. Commands are separated by commas (**,**). Curly braces (**{** and **}**) indicate optional parameters. Vertical bars (|) indicate alternative choices for a parameter. No braces or arrow brackets (< and >) indicate **required** parts of a command. The uppercase letters in a command or parameter identifier are all you type to choose that command/parameter. Parameter identifiers that are all lowercase indicate you have to supply some kind of object ; for example, **hhh** in the **Hex** command indicates a hexadecimal number is required. See the **TTY Mode Command Language** section for more details, and the reference sections that document each of the five TTY mode menus.

Only From Here

Let's look at a listing of our entire program. This is something that can be done only from TTY mode. Type an **A**, for **Allproc.** This tells Nosy to list all blocks As soon as you press **Return**, the disassembled listing will start printing to the screen. **Click the mouse to temporarily stop the listing**; click again to restart it.

If you read the listing as it scrolls by, you'll notice some intelligible English comments. When Nosy first loaded **CopyROM128**, it found a file named **CopyROM128.aci** in **CopyROM128**'s folder, and loaded that file as well. An **.aci** file is a text file with comments that Nosy can put into a disassembly at specified points. See **Commenting With .aci Files** for more info.

TTY Mode's Intermediate Menu

Pressing the space bar during a listing stops it and brings up the TTY **Intermediate** menu :

```
Quit, cr to_eXit, conVert{addr}, Chng P|D|G|C #/newname, ls_proc<data#>, User_stop
Not_proc <name>, Tbls, ref_Map, Outdir{file}, Define_case_jmp{jmp_addr}
 }
```

This menu offers some of the same alternatives as the **Main** TTY menu, plus some extras. Note that the prompt in this **Intermediate** menu is a right-curly brace (**}**). If you pressed **Return**, you'd exit this menu, and the block list would continue. For now, type **Q** to quit the current command (A) and exit from this menu to the **Main** TTY menu.

Sending The Listing To A File

Let's save this listing to a file, for later re-assembly, editing, and/or printing out. You do this with the **Main** menu's **Hardcopy** option. Type **H** and press **Return**. Nosy will ask what format you want to save the file in:

```
assembly list format ("n" for .asm) [y] ?
```

The assembly list format is the one that we've been working with, with the address of each instruction and data statement, and their hex and ASCII equivalents, shown on each line to the left of the actual instruction. The **.asm** format omits this extra information and outputs a file that is suitable for immediate re-assembly. The default here is **y** for saving in **list** format. Press **Return**, and Nosy tells you the name of the file that will be produced. If you selected **.asm** format, for example, Nosy will say

```
Hard copy on copyROM128.asm
```

and redisplay the **?** prompt. If you want to name the file, put a name right after the **H** For example,

```
HmyVolume:myFile
```

creates a file named **myFile** on the volume **myVolume**. The **Hardcopy** option is a **toggle**. The file won't actually be produced until you type **A** (for **Allprocs**) to start the listing. You'll notice that **Hardcopy** now has a plus sign (+) in front of it, indicating that the option is on. When the listing to the screen has finished, type another **H** to close the file.

Reference Maps

When you're doing a complex disassembly, it's nice to see a concise listing of what calls on what in the code. Nosy can give you such a reference listing, and more, with the **Main** TTY menus's **ref_Map** command. Some reference mapping is also available in Window mode, from the **Display** menu.

From the Main TTY menu, type an **M** (the capitalized letter in **ref_Map**). You'll get the **Map** menu:

```
All_refs{G|D|P|C|T|S}, Refs<name_list>, Fmt_by<Proc|Addr|Name|Seg/proc>, cr to eXit
Trace<C|P #>, Seg_map
 >
```

Note that prompt : a right angle bracket (>). Let's look at the first command, **All_refs**. The letters in curly braces following the command are optional arguments. **AG**, for example, produces a reference map of All Globals in the program, and **AD** produces a reference map of All Data blocks . **P**, **C**, **T**, and **S** stand for, respectively, **P**rocedures, **C**ommon, **T**raps, and **S**ystem symbols. **A** by itself lists all of them.

Type **AT** and press **Return** - you'll see the following:

```
         6E  InitGraf      proc4
         FE  InitFonts     copyROM128
        112  InitWindows   copyROM128
        130  InitMenus     copyROM128
        17B  InitDialogs   copyROM128
OS        0  Open          FSOpen
OS        1  Close         FSClose
OS        2  Read          FSWrite
OS        3  Write         FSWrite
OS        8  Create        Create
```

A Valuable Map

This listing shows which procedures trap out on Mac ROM routines. Mac trap usage provides good clues as to what's going on in a piece of code, so this is valuable information. **InitGraf**, for example, is trap $6E and is called by **proc4**. You'll also note, from the absence of **GetNextEvent**, that **copyROM128** has no event loop.

The reference map simply notes the names of the procedures where the traps are located. You can change this formatting with one of the options from the TTY **Map** menu's **Fmt_by** command. For example, **FN** - **F**mt_by **N**ame - causes the next call for a traps reference map to look like this:

```
        6E  InitGraf      proc4+0
        FE  InitFonts     copyROM128+22
       112  InitWindows   copyROM128+20
       130  InitMenus     copyROM128+32
       17B  InitDialogs   copyROM128+30
OS       0  Open          FSOpen+20
OS       1  Close         FSClose+E
OS       2  Read          FSWrite+2A
OS       3  Write         FSWrite+2E
OS       8  Create        Create+18
```

The map now shows the procedure names, plus the offset within each procedure where the trap is located.

File Those Maps Away

Getting a reference map saved is similar to the way we nabbed an assembly-language listing. The **Outdir** (for **Output redirection**) command does the job. Note that while the command only appears on Main menu command list, it is available in all TTY mode command menus (as an invisable menu item).

To turn Output redirection on, type an **O**, followed immediately by the name of the file you want to save the reference map(s) to. This is similar to the **H** command, except that you **must** specify a file name. Call up the particular reference map listing that you want using the **All_refs** command. When you're finished, type an **O** again to close the file.

More On The Two Ways Out

Outdir and **Hardcopy** both write text to files. **O** sends everything, and even persists across **Explores**.

H writes blocks only. Both are toggles, though **O** doesn't show its state as positively as does **H**. One final point: you can use the **File:Open** command to view any of these listings from Window mode.

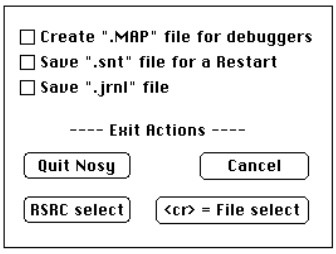
Moving On

We're going to change applications to look at some different code. If you want to return to **Window** mode to play for a while, enter a **D** (for **Display**) from the Main TTY menu. Come back by **Cmd-Q** when you're through.

More Reviewing

Reviewing data blocks is a key skill. So far we've used the **Code** and **IsProc** features to change data blocks to code. Let's look at some more block-reformatting options. Ambition's singing its song, and we're going to Nosy one of the System file's PACK resources.

If you're not in TTY mode, get there. Then select **Q** for **Quit** from the **Main** menu. Up pops the **Exit Actions** dialog, which lets you save some Nosy-created files before changing applications :



Two File-Saving Options

Nosy can save two types of files via this dialog's check boxes

The first option stores a file that copies the internal tables Nosy produced during a disassembly. An **.snt** file gets the same name as the application you've been disassembling, followed by the suffix **.snt**. The next time you invoke Nosy for a particular application, the program looks **in the applications's folder** for a matching **.snt** file. If it finds it, it loads it. This restores the state of Nosy's tables to what they were when you exited the previous session, and you can continue where you left off.

The second option stores a journal file., which gets the application's name with a **.jrnl** suffix attached. Journaling is one of Nosy's advanced features. A Nosy journal maintains a record of every command that Nosy executes when it's running. Journals can be played back, putting Nosy on automatic pilot for a variety of educational purposes. Journaling allows you to pass around your disassemblies in a convenient (and legal) format Several examples of Nosy journal files are on your disk. See the section **Journaling** for more detail.

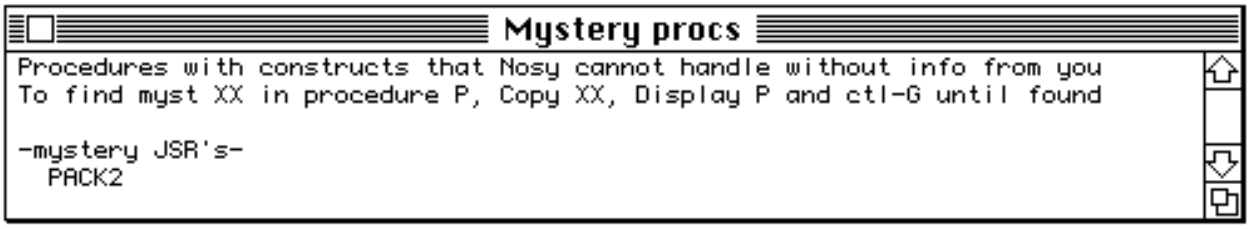
Four Buttons, And One We'll Click

We're not going to select any of these file options now. The dialog also sports four button controls. You can leave Nosy with the **Quit** button. You can get back to Window mode by clicking the **Cancel** button. The **RSRC select** button lets you disassemble another resource from the currently Nosied resource's file. The fourth option, **<cr> = File select**, takes you back to Nosy's **File Select** dialog.

Click this button with the mouse or press **Return**. Then find and open the **System** file. When the resource type is requested, enter **pack**. Lowercase letters are converted to uppercase by default, which is generally what you want. If a resource type name includes lowercase letters, precede the type name with a >.

Speaking Of Mysteries

Nosy will show you a list of all the **PACK** resources in the **System** File. Let's take a look at **PACK 2**, the Disk Initialization package. Enter a **2** to identify this package and press **Return**. Nosy will do an initial treewalk. This time when Nosy's desktop appears on the screen, the top window will be one that we didn't see the last time around. This **Mystery procs** window looks as mysterious as it sounds.



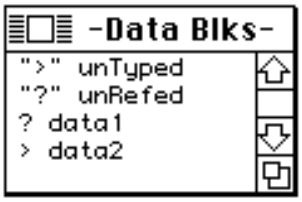
Mystery procs refer to procedures that contain JMP or JSR calls, and a few other constructions, that Nosy can't quite resolve. These items are really only mysterious to Nosy, and aren't a matter for much concern. To see this, do what is suggested at the top of the window: select the word JSR in the window and **Copy** it to the Clipboard. Then highlight the entry PACK2 and do a **Cmd-D** to bring up the assembly-listing window for that block. Finally, do a **Cmd-G**. **Cmd-G** triggers the **Edit**:**Grab Clip And Find** command. This gem grabs the Clipboard and searches the current window for a matching entry. The Select-Copy, Select-Display Block, Grab Clip And Find sequence is a common Nosy activity.

In this case, **Cmd-G** brings up and highlights a **JSR (A1)** command. This indirection mystifies Nosy; it can't figure out where the jump is going. Future Nosies may have enough intelligence (through advanced data flow analysis) to figure out a situation like this. In the meantime, don't worry about it. By the way, if you do another **Cmd-G**, you'll notice that the selection doesn't change. There's only the one such construction in the block, and Nosy is wrapping its text search around to bring up the same item again. If there were more JSR entries, each **Cmd-G** would move on to the next.

Each time you return to Windows mode after doing an **Explore,** the **Mystery procs** window will come up if any "mysteries" remain. In most cases you can ignore the window or close it. It can always be recalled with the **Display:Mystery procs** comand.

Selecting A Block For Review

Enough for that - let's look at a few more reviewing techniques. One handy Nosy feature lets us **Review** a specific data block, and not start at the top of the list each time we enter **Review** mode. To select a particular data block for Review, pull up the **-Data Blks-** window from the **Display** menu and highlight the name of the particular data block that you want to **Review**.



Notice the two-line legend at the top of the window. Both of these data blocks are candidates for **Review**. If you hold down the shift key before calling for a **Review**, by the way, *all* data blocks are candidates, not just the ones that Nosy found questionable bcause they're untyped or unreferenced. And if you hilight a data block that's normally not reviewable, the shift key will force a **Review** for the particular block. In this case, hilight the entry for **data2** and select **Review**. The data listing will appear in a **-Data Blk-** window.

Recognizing Code

Click in the window to make it active, then pull down the grow box so you can see more of it. If you look through the data block with a keen eye, you'll probably notice a number of **$4E75**'s. These are the hex equivalent of an **RTS** instruction and are a good sign that you're looking at code. They also have a convenient ASCII (or is it Yiddish?) equivalent, **Nu**, which you might find easier to spot. In any case, do a **C** for Code and look at the assembly listing that results. What was one large data block is now revealed to contain a number of procedures. Press **Return** to do the default **IsProc** that marks the block as code. There aren't any other untyped data blocks (we skipped the first data block, which wasn't code in any case--look at it), so we'll return to Nosy's desktop. Pull down the **Reformat** menu and do an **Explore**. When you return from the treewalk, you'll notice that **-Code Blks-** window now shows 20 procedures and 10 common blocks - a lot of code has been uncovered!

Nosying Nosy, or How do you hold your Teacup ?

In the world of programmmers, opinions are like religions. You can suffer permanent bodily harm by subscribing to the wrong one. When it comes to sorting, some programmers belong to the Heapsort religion, and others, like myself, think that Shell sorts are the greatest thing since sliced bread. They are simple to program, and very fast for small sets of elements (< 10000).

Nosy contains 2 very fast shell sorts that can sort 1500 elements in less than .3 seconds. You can obtain a commented listing of the sorts, by disassembling Nosy, and looking at the routine **SHL_SORT**. The file **Shl_sorts** contains source for the Sort in Pascal and C.

A Sudden Departure

Well, we could go on forever, but room's tight. We hope you got the big idea: you and Nosy work together in an iterative way, deciphering blocks, naming names, gathering information, looking at maps, tracking reference chains, drawing conclusions, walking trees, etc., repeating the processes until the code you seek is comprehended. As with any problem-solving activity, each scan over the situation yields greater resolution.

How to continue learning about Nosy ? The rest of this documentation contains some specific Nosying techniques, with examples, and a slew of reference information. In addition, the disk contains journal files you can play back and learn from. If you've got a modem, take advantage of Delphi and the Nosy SIG. You'll find fellow Nosiers and lots of downloadable files. Try looking at all kinds of code, get familiar with the big features, and explore Nosy's half-hidden corners. The program is a deep and rich tool. Use it well.

The Art Of Disassembly

The True Art

Macintosh programs aren't simple. Neither is disassembling/decompiling them with **MacNosy**. The true art lies in your ability to enter a roundabout frame of mind.

Some Strategies (Theoretical)

Let us explain. Nosying a piece of code is a fuzzy, open-ended problem-solving situation. Such problems are slippery, and usually immune to a completely frontal assault. But that very slipperiness points to a number of useful strategies. Among them are these:

When stumped, move on. Other areas may yield information more easily, and that information may help if and when you come back to the stumpage. Don't bang your head against walls incessantly, as there may be a door just down the way.

Gather data with an open mind. You never know what will trigger a crystallization of analysis. Let your human powers of pattern and metaphor work their background-processing magic.

Zoom in and out. Alternate looking at details with grand overviews.

Cycle around. Look at something, go somewhere else, come back for another look.

Repeat the following :

gather data, focus, unfocus, hypothesize, test, move on, come back

until :

the light comes on (trust us, it almost always does)

Be playful.

Be deliberate.

Percolate.

Some Strategies (Practical)

You'll find more detail on the following operations in the **Techniques** and **Reference** sections of this documentation. There are two general reasons for disassembling a program. The first is to locate the copy protection, and the second is to obtain general information about how a program works. When locating the copy protection one disassembles as much of the code as necessary to locate and understand it.

When disassembling for general information the following is a more complete strategy:

If the Mystery procs window comes up after a Treewalk, then check out the mystery JMP (Ai)'s and the mystery CASE's. If you discover any that you can resolve, do an Explore afterward, as it will probably find a number of new procs.

Look at the various maps.

Look for noteworthy traps and System globals.

Review data blocks, looking for procedures, strings, jump tables, structured data.

Follow data block reviews with treewalks.

Rename identifiers to indicate usage and meaning, (start with the Leaf procs).

Add record structure information with **/W** comments

Add other comments liberally.

Follow chains of reference.

Pre-Nosy File Preparation

You may run into memory problems if you're **Nosy**ing a large file. One work–around (other than upgrading to more Mac memory) is to prepare a smaller file for Nosying. Go into **ResEdit**, and create a new file. Then copy the resource(s) you want to **Nosy** into that new file. For example, you might be having problems analyzing the System file's **PACK 1** resource. So go into **ResEdit** and copy **PACK 1** into a new file that holds it and nothing else. You can even pull this trick with selected **CODE** resources, so long as you include **CODE 0** (the intersegment jump table).

Window And TTY Modes

History

The current version of MacNosy has two major modes, Window and TTY. The program starts up in TTY mode, then moves over to Window mode after the initial treewalk. Back in the misty fog days of Nosy Version 1, TTY mode was all you got. Version 2 brought Window mode, to the delight of all. Most of the TTY mode commands have been brought over to Window mode, but a few goodies still require you to travel to TTYville. See below. This will probably change in the future.

Transfers

From Window mode, **Cmd-Y** (keyboard equivalent for **File:to TTY mode**) takes you to TTY mode's **Main** menu. From TTY mode, get to the **Main** menu, then enter **D** (for **D**isplay) to get to Window mode.

Window Mode

The Mac-like MacNosy mode. It's the easiest to use. You can do most of your Nosying in it.

TTY Mode

The mainframe-like MacNosy mode. It's tougher to use, but does some things that Window mode can't do yet : list an entire resource file or range of blocks to a file,search for a variety of objects in the resource file, send all console activity to a file, et al. Unless you abhor Window mode's Mac interface, we recommend you use TTY mode just for these specialties.

Starting Up

From The Finder

Just double–click on **Launch Nosy**. **Nosy** starts up, posts some information about itself (its **Version # and creation date**), puts up an opening GetFile dialog box so you can select a file (or ROM) to disassemble. You can also choose to **Quit** Nosy from that dialog. If you select a file, it becomes the Currently Nosied File (**CNF**), and you proceed to **The Opening Questions**. See below.

From Inside Nosy

Get to the **Exit Actions** dialog. From there, if you don't **Cancel** or **Quit Nosy**, **File Select** takes you to the opening GetFile dialog box (see above). Or choose **RSRC select** to grab a different resource from the **CNF**. That choice sends you right to the **The Opening Questions**.

The Opening Questions Mode

You get here after a **CNF** has been chosen. Depending on the files on the disk, and your answers to particular questions, various messages and questions are presented. There are four ultimate exits from this mode : return to Window mode, the **TreeWalk options** dialog, the opening GetFile dialog, or journal playback mode. Here's a summary of the questions. For ease of reference, we've arbitrarily labelled each question as **Q*n***. For each one, default responses are in square brackets (**[** and **]**). You can press **Return** to accept the default.

Q1 read saved Nosy tables (bypass treewalk) [y] ?

Asked if there's a **.snt** file associated with the **CNF** in the same folder. **y** tells **Nosy** to use the saved tables, and takes you directly to Window mode. **n** ignores the file, and takes you to another question.

Q2 list resources [n] ?

To see a list of all a file's resources, answer **y**. Otherwise, answer **n,** which takes you to **Q3**.

Q3 disassemble resource type [code] ?

This question wants you to specify a resource type to be disassembled. The default is **CODE**. If the chosen type is **CODE**, or a valid type with just one resource, it gets read in, and you go on to the **TreeWalk options** dialog. Other valid type names lead to **Q5**. Invalid type names lead to **Q4**. Entries here get filtered into uppercase characters, unless prefixed with a >.
Hint: if you get here and want to escape, just enter a garbage resource type name, like **zz**.

Q4 try again or quit file [y] ?

Asked if you enter an invalid resource type name in response to **Q3**. **y** means you want to try again, bringing back **Q2**. **n** tells **Nosy** to bag it, and brings up the opening GetFile dialog.

Q5 enter ID of resource to process, cr to exit ?

Asked after you've chosen a resource type (**Q3**) that has more than one resource present. The resources will just have been listed, so you can see their ID numbers and names. Entering a valid ID number tells **Nosy** which resource to work on. An invalid ID brings **Q5** right back at you. A **Return** brings up **Q4**.

Q6 read names from .MAP file [y] ?

Asked if there's a .**MAP** file associated with the **CNF**. **y** tells **Nosy** to read the map names and use them, **n** says not to. Either way, you proceed to the **TreeWalk options** dialog.

Q7 commands from the *CNF_id*.jrnl [y] ?

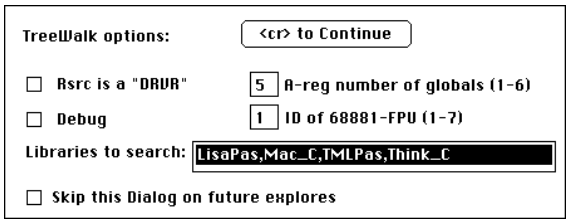
Asked if there's a journal file whose name matches the **CNF**'s (***CNF_id***). Answer **y** if you want the journaling commands to be played back. **Nosy** will move on to **Q8**. Answer **n** to ignore the journal. **Nosy** will move on to **Q2**.

Q8 list Review data during jrnl file read [n] ?

Asked if you chose to play back a journal. Answer **y** to watch the data blocks as block review commands are played back, or default to **n** if you don't. Either way, Nosy moves on to play back the journal file.

Treewalk Options

The first time you **Explore** a resource, and on subsequent **Explores** if you don't check the **Skip this Dialog...** item, the **Treewalk Options** dialog comes up.



Rsrc is a "DRVR" **Nosy** recognizes **DRVR** resources, so this item comes up correctly set. You'll have to set it for resources that follow driver format (see **IM** for details) but try to hide the fact.

Debug reserved for use by The Head Nose. Caveat emptor for the rest of us.

A-reg number of globals (1-6) Mac programs generally index global variables off register **A5**. But there are exceptions. If you notice, after the first treewalk, lots of addressing done with constants off of another **A*n*** (constructions like **-234(A4)**), then do another **Explore** and set this thing to ***n***.

ID of 68881-FPU (1-7) Some compilers produce code variants for the 68881 math coprocessor chips. This item comes pre–set to 1, the proper Apple/Motorola default.

Libraries to search Defaults to the full set of **Nosy** library files. Adjust as you see fit. See **Building Pattern Libraries** if you want to add new library files.

Skip this Dialog on future explores You'll usually want to set this item. Saves time on future treewalks.

Shutting Down

From Window Mode

1 If you have some notes to save, send them to a text file via the **File** menu's **Save**, **Save As…**, **Append to *CNF_id*.txt**, and/or **Append to…** command(s).
2 If you want to extract any comments, do so, and append to them to the **.aci** file.
3 If you've put any comments in the **.aci** file, open it up, and see if it looks like it'll do what you want. See **Commenting Via .aci Files**. Save any changes you make.
3 Invoke the **File:quit to TTY mode** command to get to TTY mode's **Main** menu. Continue below.

From TTY Mode

1 Get to the **Main** menu
2 Enter **Q** to **Quit**. . The Hardcopy and OutDir files are closed automatically. That'll get you to the **Exit Actions** dialog. See Page 16 for a picture.
3 Decide if you want to save **.snt**, and/or **.jrnl** files, then check the appropriate dialog item(s).
4 To leave the program, click the **Quit Nosy** button.
5 To get back to the program, click the **Cancel** button. **Nosy**'ll go back to **Window** mode.
6 To select a new resource to **Nosy**, click **RSRC select**. **Nosy**'ll go to its **Opening Questions** mode.
7 To select a new file to **Nosy** from, press **Return** or click the **<cr>=File Select** button. **Nosy**'ll present its opening GetFile dialog box, and continue a standard startup from there. See **Starting Up**.

Recording Your Work

There are several ways to record the fruits of a Nosation. You can print things out -- see **Printing**. You can record text files -- see **Text File Output**. You can record comments added to a disassembly -- see **Commenting Via .aci Files**. You can save a record of the commands you use during a session -- see **Journaling**. You can save Nosy's internal tables in a **.snt** file, so it'll be able to start the next session with the **CNR** right where it left off -- see **Shutting Down**. You can save a record of procedure names and addresses via a **.MAP** file -- also covered in **Shutting Down**.

Printing

The File menu in window mode of **Nosy** has the standard **Page Setup** command and a command to print the contents of the frontmost window with a header.

Text File Output

One of **MacNosy**'s great virtues is the ease with which you can save your work as standard Macintosh text files. These files can then be edited by Nosy or by any of the standard Mac editors.

From Window mode, you can save the contents of whole windows as text files. This is done with commands from the **File** menu. You can save selected text, also with **File** menu commands. See **The File Menu**. In addition, **.aci** comment files are saved as text. See **Commenting Via .aci Files**

From TTY mode's **Main** menu, **Hardcopy** saves Nosy output as text. **Outdir** does that too, and also saves input to Nosy, thus letting you keep a complete record of a TTY session. **Outdir** can also be invoked from the **Map** and the **Table & Debugging** menus. See the appropriate TTY mode menu reference sections. **Hardcopy** is particularly noteworthy for the following: when used with the **Allproc** command, it lets you send a whole program off to disk for later re_assembly. See **Assembling A Nosied File**.

Finally, **.jrnl** and **.aci** files are saved as text. See **Journaling** and **Shutting Down**.

Commenting Via .aci Files

What It's All About

.aci stands for **a**dditional **c**omment **i**nformation. **.aci** files store comments that are attached to a Nosy disassembly. When you re-Nosy a resource, Nosy can merge those comments back into the disassembly. And **.aci** comments aren't limited to passing intelligence along to you, the reader of the code; they can also pass it to Nosy, via a special class of **slash** comments. Between **.aci** and **.snt** files, you can pass complete disassemblies along to others.

How To Do It

0 You can only add comments from Window mode, so get there
1 Open up a procedure window
2 Add comments to lines, in **add 'em** format (see below)
3 Do an **Edit:Select All** command (**cmd-A**) to select the contents of the procedure window
4 Do a **Misc:Extract Comments** command (**cmd-E**)
5 A new window pops up, titled **-comments-**. It contains the procedure window's comments in **extracted** format (see below)
6 Do an **Edit:Select All** (**cmd-A**) to select the contents of the **–comments–** window
7 Do a **Misc:Append to .aci** to add the selection to ***cnf_id*.aci**

The Add 'Em Format

Use this format when you add comments to a line. In **add 'em** format, each comment begins with a semi–colon, followed by the text of the comment:

; *commenText*

The *commenText* can be any string of Mac characters, or a slash command.

There are four flavors of slash command. Each one begins with a slash (/). Here are the formats:

The **with** slash command looks like this :

/wΔ*type_id* {=A*n*} (remember, Δ signifies a space)

where *type_id* is a Mac structured type name, and *n* is an A register number (0 - 6).

The **endWith** slash command looks like this :

/eΔ*type_id*

where *type_id* is a Mac structured type name, and the **e** can be upper– or lowercase.

The **cascade** slash command looks like this :

/c {D|A}

The **link frame** slash command looks like this :

/L where the **L** must be uppercase.

Here are some examples of comment lines in **add 'em** format:

a comment for humans:	; some clarifying, semi–poetic insight
a **with** slash command:	; /w GrafPort =A4
an **endWith** slash command:	; /E GrafPort
a **cascade** slash command:	; /cD
a **link** slash command	; /L

The Extract Format

It's how comments look when they're extracted. Use it when editing extracted comments or the contents of an **.aci** file. Note that the *commenText* element is the same as in **add 'em** format.

There are three flavors of extracted comments. Each one starts with *srel_addr*, a segment–relative address. Here are the three, along with a little explanation of each.

A **procedure start** comment looks like this:

srel_addr = *seg_num* / {*proc_id*}

…where *seg_num* is a segment number, and *proc_id* is a proc name. It marks the start of a proc block's comment group, and is automatically inserted by Nosy when you **Extract** comments

An **appended** comment looks like this:

srel_addr Δ *commenText*

It's a comment **appended to** the line at *srel_addr*. Don't forget the space character (Δ).

An **inserted** comment looks like this:

srel_addr i *commenText*

It's a comment **inserted after** the line at *srel_addr*.

Here are some examples of comment lines in **extract** format:

a comment group start marker:	35F=2/HappyStuff
an inserted comment:	42Aisome clarifying, semi–poetic insight
an appended comment:	612 /w GrafPort =A4
an appended comment:	71E /E GrafPort
an inserted (blank line) comment:	726i

The Slash Commands

The **with** slash command tells MacNosy that **An** is the base register for references to a data structure of type *type_id*. The **=An** is optional when the instruction on the line implies the address register; for example: LEA window,A4. **with** comments may be or follow inserted comment lines.

The **endWith** slash command terminates a **/with** . Nosy does this automatically when the **with**ed **A** register is redefined or the proc ends. You have to do it when a called proc redefines the register.

The **cascade** slash command begins or ends a cascade subtract and test sequence. Such sequences consist of **SUBI** or **SUBQ** and test (**B*xx***) instructions. They're sometimes used to implement simple CASE situations. For example, the **PACK 3** resource in a recent System file had a cascade sequence between locations 130 and 15C.

The **link** slash command declares that **A6** has been set by some other procedure, and is being used in this procedure as a local stack frame. The **L** must be capitalized.

Some Commenting Do's And Don'ts

Do…

… open your **.aci** file to check it before leaving Window mode. As the comment extraction algorithms are simple, funny stuff can creep in. You can edit it out,and save the fixed file.

… follow **/w** with a space in the **with** slash command

Don't…

… type over blank lines

... try to comment lines that Nosy has already placed some annotation on

At Any Time …

... you can open the **CNR**'s **.aci** file, examine it, make any changes, close it, then check out its effect by: closing any commented procedure block windows, invoking the **Misc:save .snt, reRead .aci** command, then opening any commented procedure block windows. The comments should show up, along with any slash command effects.

A Short Illustrated Example

We'll use **prSpool** for an example. Open it up for Nosying. Then open up a window on the **%OUTCHCO** procedure. If you scan down the window a ways, you'll find this section of code :

```
73C: 487A 017E      10008BC          PEA     data24      ; len= 206
740: A86F           '.o'             _OpenPort ; (port:GrafPtr)
742: 45FA 0178      10008BC          LEA     data24,A2   ; len= 206
746: 357C 0004 0044 '5|...D'         MOVE    #4,68(A2)
74C: 357C 0009 004A '5|...J'         MOVE    #9,74(A2)
```

insertion point

Hmm. Looks like **A2** gets pointed to a **GrafPort**. Let's add a little reminder. Place the insertion point at the end of the line at address **742**, as shown above. Then press **Return**, and type in the slash command **;/w GrafPort**. Here's a picture of what things look like now:

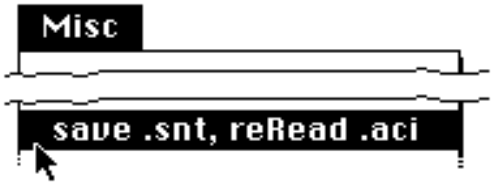
```
   73C: 487A 017E      10008BC          PEA     data24      ; len= 206
   740: A86F           '.o'             _OpenPort ; (port:GrafPtr)
   742: 45FA 0178      10008BC          LEA     data24,A2   ; len= 206
;/w GrafPort
   746: 357C 0004 0044 '5|...D'         MOVE    #4,68(A2)
   74C: 357C 0009 004A '5|...J'         MOVE    #9,74(A2)
```

Okay. Do a **Cmd-A** to select all the contents of the **%OUTCHCO** window, then **Cmd-E** to extract comments. This window appears :

Do a **Cmd–A** to select the contents of the **–comments–** window. Then do a **Misc:Append to .aci** command :

The comments have gone to the file **prSpool.aci**, to test it out. Close the **%OUTCHCO** window

Then tell Nosy to re-read the **prSpool.aci** file :



Finally, re–open the **%OUTCHCO** window. Notice how the comment's been merged into the disassembly, and the new GrafPort field information in lines **746** and **74C**:

```
73C: 487A 017E      10008BC          PEA     data24      ; len= 206
740: A86F           '.o'             _OpenPort ; (port:GrafPtr)
742: 45FA 0178      10008BC          LEA     data24,A2
                                ;    /w GrafPort
746: 357C 0004 0044 '5|...D'         MOVE    #4,txFont(A2)
74C: 357C 0009 004A '5|...J'         MOVE    #9,txSize(A2)
```

And, if you cruise down the procedure, you'll find that other constants indexed off **A2** have also gone symbolic. Thanks, Nosy.

Journaling

What It Is

Journaling creates a text file record of a Nosy work session. You can edit journal files, and play them back. Journaling is a nice way to pass on Nosy information and techniques.

Turning Journaling On And Off

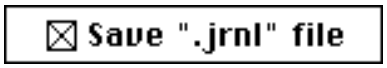
Journaling is controlled with a toggle switch. It defaults to **on** when Nosy starts up.

In TTY mode, you toggle journaling with the **Main** menu's **Jrnl** command (see **The Main Menu**). If journaling is on, there'll be a plus sign (**+**) in front of **Jrnl** in the menu.

In Window mode, you toggle journaling with the **Misc:Journal** command. (see **The Misc Menu**). If journaling is on, there'll be a check mark in front of **Journal** in the menu.

Saving A Journal File

While you're Nosying, and journaling's on, Nosy records to a scratch file named ***cnf_id*.scr**. If you want to save the journal file, get to the **Exit Actions** dialog (see **Shutting Down**). Then check this item:



Once that's done, the scratch file will be saved as ***cnf_id*.jrnl** when you leave the **Exit Actions** dialog.

Adding Comments And Pauses While Journaling From TTY Mode

Any line with a leading blank will be treated as a comment and written to the journal file.

For pauses, use the **Main:Wait** command, which does nothing until the journal file's played back. At that time, it'll cause Nosy to wait for the user to press the **Spacebar**.

Editing A Journal File

Journal files aretext files. You can edit them in Window mode or with any text editor.

Playing A Journal File

The journal file must be in the same folder as the **CNF**, and be named ***cnf_id*.jrnl**. Nosy will ask you if you want it to be used (see **Starting Up**). If you answer **y**, then Nosy will ask if you want to see any data that gets reviewed during the playback. Usually you'll answer with the default, **n**. Then Nosy will start the playback.

You can pause/restart the playback by clicking the mouse. You can abandon the playback or skip a line by pressing the spacebar. When you do so, this question appears: **Abandon *cnf_id*.jrnl or skip line [n] ?** Press **Return** (for the default **n)** if you want to skip a line and restart the playback; enter **y** if you want to abandon the playback.

Format Of Lines In A Journal File

A journal is a collection of one or more lines. You may place comment lines at the beginning of the file. They have a semi-colon in column 1. They are terminated by the title line.

A title line consists of the file's name, followed by a vertical bar. Here's an example: abc.jrnl|

An entry line begins with an optional user response, followed by a vertical bar, followed by a descriptor of Nosy's prompt or state at the time of the response. If there's no user response on an entry line, that indicates the user pressed the **Return** key. Here are some examples:

```
R|main
|sd 1000000
```

Bring 'em Back Alive - an Actual Example

Here are some excerpts from the journal file **Fedit Plus.jrnl**, which is on your Nosy disk. The excerpts are on the left, and we've added some elucidation *in italics* on the right.

```
;  Jrnl file to Disassemble Boot Blks from the BOOT rsrc in Fedit Plus.   a comment line
;                                                                         a (blank) comment line
Fedit Plus.jrnl|                                                        the journal's title line
boot|disassemble resource type [CODE]                                   specifying a type to disassemble
5|enter reg                                                             a default from the treewalk dialog
R|main                                                                  Review command from Main menu
```

n1|sd 1000000	*a block–splitting New command*
ab2|sd 1000000	*format as 2 ASCII bytes*
|sd 1000000	*press Return to move to next block*
mfstr=db1,ab15|sd 100000A	*set Macro format for 15 char P–strings*
q|sd 100000A	*Quit the Review menu*
E|main	*go on another treewalk (Explore)*

Reviewing Data Blocks

Essentials

Reviewing and reformatting data blocks is an essential Nosational task. The most important part of the process is finding code blocks. Nosy relies on your human intelligence to recognize code blocks it's unable to see. Then it can do another treewalk and refine its model of the program's structure. The review process is a pattern recognition skill with a low–slope learning curve. And perfection -- nailing every block correctly -- isn't required, since Nosying is an iterative, fuzzy–edged task. So you can start out and do it successfully, albeit slowly, then pick up speed through practice, until you find yourself wailing through block reviews.

Some Simple Beginner's Strategies

First: work out of Window mode. In Window mode, most of the other Nosy commands are available during a block review, and they can be used to gather block–cracking information.

Second: Start by using the **Code** command on a block. Then see how the results look. If you get a bunch of illegal instructions (marked with a bullet •), or the code's a series of nonsensical instructions (things like repeated ORI's), or the **–Use–** window uses the block in an obviously "data–like" way, then **Revert_to_data**. Otherwise (the code looks okay), do an **Is_proc**.

Going a Bit Further

A refinement: if only the first few lines of code look wacky, try a **New** command to split the block into two. **Revert** the first piece back to data, then look at the next block (second piece from the split) as code. If it's good looking, do an **Is_proc** and you're set. Otherwise, revert it to data and **cOmbine** it with the previous block to get back where you started.

Another idea :If there's nothing in the **–Use–** window (the block's not referenced), select the block's name and do a **Shift–Cmd-D** to display it with the preceding code block. This may give you some usage clues.

Some Common Patterns :

After a while, you'll notice patterns of hexadecimal words that indicate a likely code block. One pattern is the **$4Exx** series. This pattern fits several common instruction object codes: LINK, UNLK, JSR, JMP, RTS, RTR, RTE, RESET, TRAP, NOP, STOP, and others. Another pattern is the **$6xxx** series. This pattern fits a number of branching instruction object codes: BRA, BSR, Bcc. If the current block contains words fitting both these patterns, the likelihood of code–ness gets even higher. Another thing about blocks that could be code: the words will usually all be ≥ $1000.

Some Examples

You saw some examples in the **Introductory Cruise**. Here are some more from disassembling Nosy.

Here's an easy one to get you going. Notice anything interesting about this block?

```
48: '................'   data2    DC.W    1,$203,$405,$607,$809,$A0B,$C0D,$E0F
58: '................'              DC.W    $1011,$1213,$1415,$1617,$1819,$1A1B,$1C1D,$1E1F
68: ' !"#$%&'()*+,-./'              DC.W    $2021,$2223,$2425,$2627,$2829,$2A2B,$2C2D,$2E2F
78: '0123456789:;<=>?'              DC.W    $3031,$3233,$3435,$3637,$3839,$3A3B,$3C3D,$3E3F
88: '@ABCDEFGHIJKLMNO'              DC.W    $4041,$4243,$4445,$4647,$4849,$4A4B,$4C4D,$4E4F
```

Yup, a nice orderly table of data. The reference to the block confirms its use as data:

```
lac_1    MOVE.B  (A0)+,D0
         MOVE.B  (A1)+,D2
         CMP.B   data2(D0.W),D2
         BNE.S   lac_2
         DBRA    D1,lac_1
         MOVE.B  #1,4(A7)
lac_2    RTS
```

We can reformat the block to make it more readable with the command **HB8**, and rename it with the

Misc:name cHange command to something vaguely mnemonic :

```
48: '........'          smarTable  DC.B    0,1,2,3,4,5,6,7
50: '........'                     DC.B    8,9,$A,$B,$C,$D,$E,$F
58: '........'                     DC.B    $10,$11,$12,$13,$14,$15,$16,$17
60: '........'                     DC.B    $18,$19,$1A,$1B,$1C,$1D,$1E,$1F
```

Example #2

This next block smells like code:

```
26C: ' _ .C..Dt...BA..'   data5   DC.W    $205F,$201F,$43ED,$DD44,$7404,$D511,$4241,$1211
27C: '......SASBn.N.'             DC.W    $1380,$1000,$E088,$5341,$5342,$6EF4,$4ED0
```

Note the **$4Exx** and **$6xxx** words. Also, each word is ≥ $1000. Using the **Code** command, we get this:

```
26C: 205F           ' _'    data5    POP.L   A0
26E: 201F           ' .'             POP.L   D0
270: 43ED DD44       -$22BC          LEA     glob170(A5),A1
274: 7404           't.'             MOVEQ   #4,D2
276: D511           '..'             ADD.B   D2,(A1)
278: 4241           'BA'             CLR     D1
27A: 1211           '..'             MOVE.B  (A1),D1
```

This code seems reasonable, so we'll do an **Is_proc**.

Example #3

This example also has those telltale **$4Exx** and **$6xxx** words, and each word's ≥ $1000. Looks like code.

```
532: 'N.e.p.Nu'             data6    DC.W    $4EBA,$6590,$70FF,$4E75
```

The **-Use-** window shows how a pointer to this block is stored as a variable. See the snapshot on the left below. After we do a **Code** command, the block looks like the snapshot on the right below. Very reasonable code. If you haven't noticed by now, **$4E75**, the object code for **RTS**, is an especially persuasive code indicator.

```
MOVE    #$FFFF,72(A4)
LEA     data6,A0     ; len= 8
MOVE.L  A0,42(A4)
LEA     proc32,A0
MOVE.L  A0,38(A4)
```

```
532: 4EBA 6590      1006AC4 data6    JSR     P_AUTOSC
536: 70FF           'p.'             MOVEQ   #-1,D0
538: 4E75           'Nu'             RTS
```

Example #4

It's a good idea to look at the ASCII version of a block's information, over on the left–hand side of the Nosy listing. Look at this, for example:

```
221A: '.non-digit in nu'  data65  DC.W  $136E,$6F6E,$2D64,$6967,$6974,$2069,$6E20,$6E75
222A: 'mber,.......'              DC.W  $6D62,$6572,$2C00,$FC18,0,0
```

String time, eh? So we use the **NW** command, which splits a block into two pieces, the first piece being a word–aligned Pascal–type string. Here's the happy result:

```
221A: 136E 6F6E 2D64 6967    data65   STR     'non-digit in number'
```

The Amacs Macros

What, When, How

Amacs is a text file that holds definitions for the macros MacNosy puts into a disassembly. There's a set of definitions for MDS, and one for MPW. That covers the majority of Mac assemblers.

Amacs is needed to assemble a file produced by Nosy. You'll have to remove the set of definitions you don't need. Add an **include** statement to the beginning of the file, if necessary.

Studying **Amacs** will help you understand the effects of Nosy's block and jump table formatting commands. Refer to the MDS and MPW documentation for information on the macro syntax.

Brief Descriptions Of The Amacs Macros

Here's a quick reference guide to Nosy's macros. For more detail, list out and ponder the Amacs file.

STR	Generates a Pascal–style string
WSTR	Generates a word-aligned Pascal-style string
ZSTR	Generates a C-style string
WZSTR	Generates a word-aligned C-style string
DZS.B	Generates a block of zeroes (bytes)
DZS	Generates a block of zeroes (words)
DZS.L	Generates a block of zeroes (long words)
DNAME	Generates a debugger symbol (8 characters of ASCII)
DRVHD	Generates four driver header flag/data words
DrvAdd	Generates a driver routine offset
FLD	Lets you see data as bit fields
CALL	Generates a call that with its return address in a register - (ROM init)
QUAL	Creates locality for a section's variables - only works with MPW
JBIAS	Sets jump table bias factor
ADDR	Generates a word address
ADDRL	Generates a long word address
JUMP	Generates a jump table offset (routineFBA - tableFBA)
VEQU	Define a equate with a record structure
VEND	Terminate a list of VEQU's
CJMP	Jump table entry for Think C™
POP	Pop a word off the stack
POP.L	Pop a long word off the stack
POP.B	Pop a byte off the stack
PUSH	Push a word onto the stack
PUSH.L	Push a long word onto the stack
PUSH.B	Push a byte onto the stack

Linking Jumps To Tables

Jump Tables

A Mac program waits around for an event. When it gets one, it reacts according to one or more qualities of the event. It's common for a Mac program to use jump tables to implement its reactions. Jump tables, whether written in assembly language or produced by a compiler transforming a case statement, are an efficient way to transfer control in such situations.

Sometimes the jump table entries are short pieces of code, some kind of branch or jump instruction. In these situations, the calling code picks a spot in the table, jumps right into the table itself, then moves from there to a particular code location.

The more common case is a jump table that consists of offsets to a number of code locations. The calling code grabs a particular offset, adds it to a base address, then jumps to the resultant target address. For the remainder of this section, we'll be discussing these offset jump tables. They're the kind Nosy knows how to deal with.

Why Nosy Has Difficulties With Some Jump Tables

Nosy does a remarkably good job deciphering jump tables. But sometimes it needs your help.

Why? Well, it can miss any piece of code and associated data due to limitations in its treewalk intelligence. And someone can come up with a new compiler (Nosy handles all the current ones) that builds jump tables whose structure is so unorthodox that Nosy can't figure out what's up. Or an assembly language programmer might do things in a convuluted way (see the **.MPP** example below for one of these). Or an instance may be a flavor of jump table other than offset.

Sometimes Nosy finds an offset jump table, but can't determine its length. There's usually no problem if one or more of the offsets is positive. In those cases, Nosy just assumes that the table runs up to the first piece of jumped–to code. But, if all the offsets are negative, Nosy may have trouble figuring the table's length.

A Typically Clean Jump Table And Code

Just for reference, here's a model for an offset jump table and associated calling code:

```
            MOVE.W    Selector,D0              ; fetch table entry selector (0, 1, 2, ...)
            ADD.W     D0,D0                    ; word–sized index = 2 * selector
            MOVE.W    JumpTable(D0),D0         ; offset to code location
            JMP       JumpTable(D0)            ; add that offset to table base and jump

JumpTable   DC.W      FirstRoutine-JumpTable
            DC.W      SecondRoutine-JumpTable
            DC.W      ThirdRoutine-JumpTable
            ad nauseum
```

Unfortunately, many programs aren't this clean.

How To Find Jump Tables Nosy's Having Trouble With

(1) Look in the **Mystery procs** window for mystery JMP's. Then track them down. If the code and table look like an offset jump table, you can go ahead and link the JMP to its table.

(2) Look in the **-Case Jumps–** window for anomalies in the jump tables that Nosy's aware of. This window lists the following information for each jump table:

jmp_addr	the address of the jump instruction
tbl_addr	the **fba** of the table itself
bias	an adjustment factor Nosy uses to figure jump addresses (see below),
n_branch	the number of branches (entries) in the table
info	some debugging information for Jasik, along with a JUMPx indicator of the nature of the jump targets (see below)

Anomalies show up in the **tbl_addr** and **n_branch** fields. If the values there are 0 or negative, the jump's worth examining. Double click on the address (in the **jmp_addr** field) and do a **Cmd–D** to get there fast.

(3) Sometimes you'll discover a jump table during a block review (see **The Review Menu** and **Reviewing Blocks**). If you're a good Nosite, you always follow such fruitful reviews with an **Explore**. Always check the **–Case Jumps–** window after the treewalk.

What's This Bias Thing?

The bias parameter is an adjustment factor Nosy uses to figure jump table destinations. There's a little formula that reveals all: For an entry in an offset jump table, add the base of the jump table to the offset, then subtract the bias. The result is the entry's target address. In other words,

BaseOfJumpTable + Offset - Bias = Target Address

In the model calling code and jump table above, the bias would be zero. But this isn't the case with a lot of real–world code. Bias helps Nosy deal with these deviants. Think about it.

You can print out the **Amacs** macros file for more details on the actual implementation of jump table biasing. And look below for an example of figuring it out.

Linking Jumps To Tables: An Information–Packed Example

Let's mini–cruise through an example. It's taken from a Nosation of the System file's **.MPP** driver.

After the initial treewalk, this shows up in the **–Mystery procs–** window:

Let's look at the mystery in **Contrl**: double–click on JMP, **Cmd–C**, double–click on **Contrl**, **Cmd–D**, **Cmd–G**. We're there, and this is what we see in the **Contrl** procedure's window:

```
4A: 47FA 0010       100005C           LEA     data4,A3     ; len= 16
4E: D6C3            '..'              ADDA.W  D3,A3
50: D6C3            '..'              ADDA.W  D3,A3
52: 3653            '6S'              MOVEA   (A3),A3
54: 487A FFAA       1000000           PEA     .MPP         ; len= 8
58: D7DF            '..'              ADDA.L  (A7)+,A3
5A: 4ED3            'N.'              JMP     (A3)
                            ;-refs - Contrl
5C: '.....l.<.,.....'       data4     DC.W    $82,$280,$26C,$23C,$12C,$10E,$C4,$8C

6C: 222A 0084       '"*..'  lab_2     MOVE.L  132(A2),D1
```

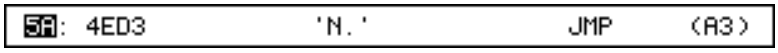
Aha! Hidden in this mess is an offset–type jump table, along with its calling code. The table's at **data4**. D3 holds an index into the table as we enter the code fragment. A3 starts out by getting the address of the jump table, and from there moves in relentlessly on a target address for jumping to. The selector gets doubled, then added to the base of the table so we can grab a table entry . That entry is an offset, and it gets added to the address of the driver's first byte (called **.MPP** here) to form a target address in A3. Finally, the indirect jump at $5A sends the flow of control to that target address. A bit convoluted, but still following the basic calling pattern.

Let's do a check on that reasoning. If we're right, the first table entry means there's a block of undiscovered code living at $82. Let's peek:

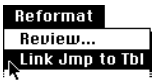
```
82:   data5     DC.W    $242A,$84,$6A2C,$2642,$4ED3,$4240,$4A2A,$80
92:             DC.W    $6B20,$598F,$2F3C,$4E42,$5043,$3F3C,1,$A9A0
```

It's a data block, all right. And see all those $4Exx's and $6xxx's ? Code tracks. Let's go link.

Double–click on the jump instruction's address:



Then select the **Reformat** menu's **Link Jmp to Tbl** command:



The first link dialog box appears. It wants us to type in the jump table's block's name. So we do it:

Click the **Continue** button, and the second link dialog box appears. It wants a few more items of information: **Table format**, **# of Jumps**, and **Table Bias**.

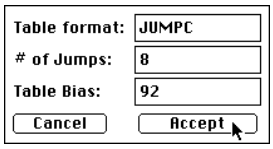
There are three possible **table format**s. They differ in how the targets of the jumps will be labeled. JUMPP tells Nosy the jump targets should be labelled as procedure blocks. JUMPC tells Nosy that they're common blocks. And JUMPL says they should get local labels. Which one to use? JUMPP and JUMPC will **break a large procedure up**, and JUMPL keeps it intact. If the disassembly of a procedure is overflowing a window, then JUMPP is advised. In this example, the **number of jumps** is easy to figure. That's because we can clearly see the beginning and end of the table (see snapshot above). The jump table runs from $5C through $6B. That's 16 bytes worth of word–length entries, or 8 jumps.

Now the weirdo, **bias**. Remember our little mechanical formula?

BaseOfJumpTable + Offset - Bias = Target Address

Let's apply it. Examining the calling code again, then factoring in our earlier deductions and the discovery of that probable code at a target address ($82) equal to its offset, and finally sprinkling in a little 6th–grade algebra, we tentatively conclude that the bias is equal to BaseOfJumpTable. If we're wrong, we can always try another value. By the way, this dialog item wants decimal notation. BaseOfJumpTable = $5C = +92.

Okay, let's fill in that dialog and click the **Accept** button:



We're just about done. Final step: invoke an **Explore** so Nosy can consolidate this new information. Do it.

Okay. We've just linked a jump to a table. Let's check out the results. Open up **Contrl** and and cruise down to the jump table. Look:

```
5C:                              JBIAS  92
5C: 0082       $82 data4         JUMP    com_1
5E: 0280      $280               JUMP    com_11
60: 026C      $26C               JUMP    com_10
62: 023C      $23C               JUMP    com_7
64: 012C      $12C               JUMP    com_6
66: 010E      $10E               JUMP    com_4
68: 00C4       $C4               JUMP    com_3
6A: 008C       $8C               JUMP    com_2
```

So far, so good. Let's peek at a couple of the code targets:

```
8C: 4240         com_2    CLR     D0
8E: 4A2A 0080             TST.B   128(A2)
92: 6B20                  BMI.S   com_3
94: 598F                  SUBQ.L  #4,A7
```

```
10E: 70A5     com_3    MOVEQ   #-91,D0
110: 4A01              TST.B   D1
112: 670A              BEQ.S   com_4
114: 610A              BSR.S   proc5
```

Looks like code to me. Nosy, take a bow.

Final Note : Fine Tuning

You may not get all the linkage parameters right the first time. Not to worry. Examine the results, do some fresh thinking, go though another **Link Jmp to Tbl** dialog pair, **Explore**, and get a bit closer. Nosy likes iteration.

Dealing With Mysteries

What Going On ?

When Nosy finishes a treewalk, there's a good chance it'll show a **Mystery procs** window. It contains the names of code blocks containing constructs Nosy needs your help with.

There are several types of mysteries JMP (Ai)'s, JSR (Ai)'s, CASE's, TRAP's, RESET's, and Op Codes. In our experience, more mysteries are benign than malignant, but they're all worth an examination.

Example 1: Examining A JSR (Ai) Mystery

Here's an example to give you a feel for the process. The **Mystery procs** window shows this entry :

```
-mystery JSR's-
  com_1
```

Select **JSR**, **Cmd-C** to copy it, select **com_1**, **Cmd-D** to display the block, then **Cmd-G** to find any **JSR**s. You come upon this :

```
192: 2078 0810          $810            MOVEA.L jScrnSize,A0
196: 4E90             'N.'              JSR     (A0)
```

Hmmm. The proc's jumping to a procedure via one of the low-memory System vectors. This mystery's benign.

Example 2: Examining A JMP (Ai) Mystery

Here's a more complex example, this time involving a JMP (Ai) type mystery. The **Mystery procs** window shows this entry :

```
-mystery JMP's-
  PrStlDialog
```

Select **JMP**, **Cmd-C** to copy it, select **PrStlDialong**, **Cmd-D** to display the block, then **Cmd-G** to find any **JMP**s. We come upon this :

```
1780: 4ED1              'N.'              JMP     (A1)
```

Hmmm, what's in A1 ? Looking back up the procedure, we find this :

```
1776: 224B              '"K'    lcu_2     MOVEA.L A3,A1
```

So, A1 gets stuffed from A3. What's in A3 ? Looking back up the procedure, we find this :

```
1746: 265F              '&_'              POP.L   A3
```

Gotcha! The procedure's return address gets popped off the stack and put into A3. So the mystery jump was just a procedure return. The mystery's solved, and it's another benign one.

Renaming

Renaming labels is an easy way to make a piece of code more comprehensible. As you figure out what a block is up to, give it a mnemonic name that'll help you understand routines that reference that block.

We suggest you do your renaming from Window mode, where it's very simple. Just select a name, then **Cmd-H** to bring up the name change dialog box. Press **Return** when you've got a nice new name entered, or click the **Cancel** button to abort the change.

A good place to start name changing is with leaf procedures. Since these don't call anyone, you can do a quick analysis and come up with a meaningful name. Then you can work your way back up the calling chain, via the **Display:Call Chain** command, and rename the higher–level procedures.

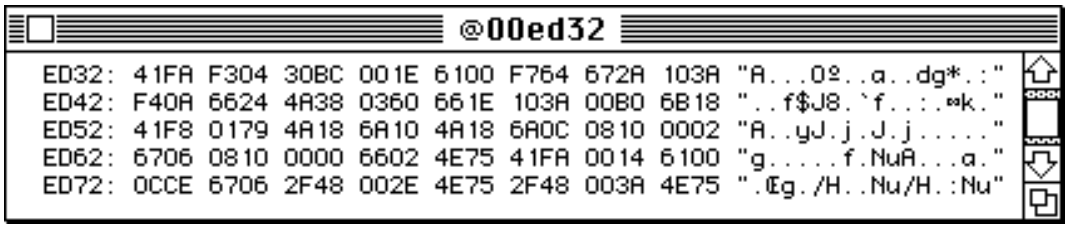
If you want to save your name changes, be sure to check the **Save ".snt" file for a Restart** item in the **Exit Actions** dialog.

Viewing Mac Memory

You can always get a window on a particular area of Mac memory while you're Nosying. You can view Mac memory as code, raw data, or described data. By described data, we mean that there's information given about a variable's type and value. It's done by selecting an identifier, then pressing a Command key combination. Here's how you can view ...

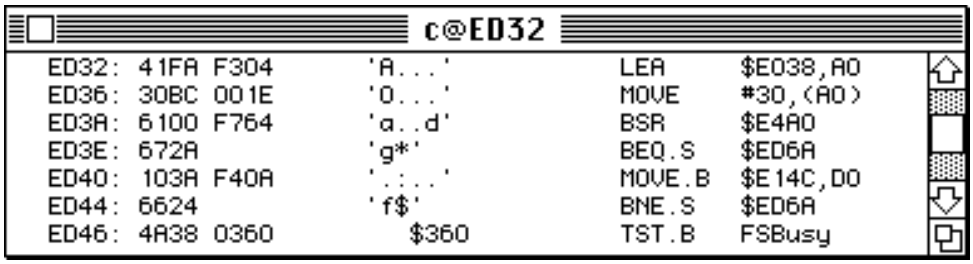
...an area of Mac memory as raw hex and ASCII data:

Identifier: an address, optionally prefixed with **@**
Keys To Press: **Cmd-Space** (the **Table:Fields of** command)
Example : Select **@00ed32** and press **Cmd-Space**. This window appears:



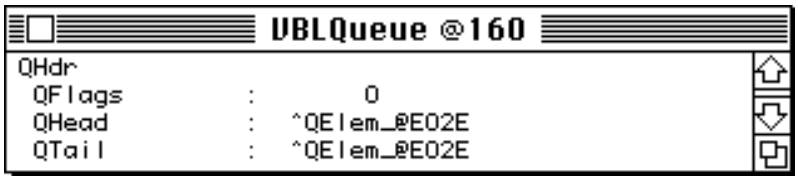
...an area of Mac memory as disassembled code :

Identifier: an address prefixed with **=** or **c@**
Keys To Press: **Cmd-Space** (the **Table:Fields of** command)
Example : Select **c@ED32** and press **Cmd-Space**. This window appears:



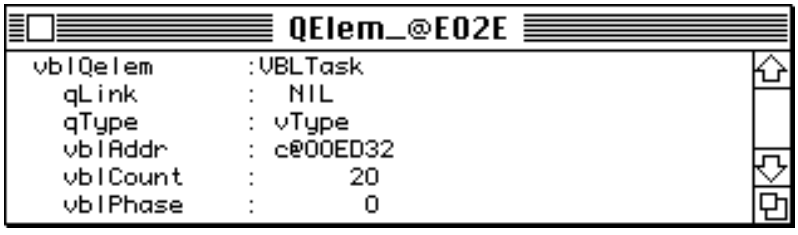
...an area of Mac memory as described data (system symbol method):

Identifier: a system symbol
Keys To Press: **Cmd-W** (the **Table:Sys Syms** command)
Example : Select **VBLQueue** and press **Cmd-W**. This window appears:



...an area of Mac memory as described data (record name method):

Identifier: an address prefixed by a record name, an underscore, and an **@**
Keys To Press: **Cmd-Space** (the **Table:Fields of** command)
Example : Select **QElem_@E02E** and press **Cmd-Space**. This window appears:



File Modification

Comment: This section is really old. It was written before ResEdit existed.
At the present, Fedit doesn't exist.
It you want to modify a CODE segment, etc then use ResEdit or Resourcerer and don't bother with this page.

I. Find the byte you want to change
II. Select the byte's address.
III. Select the **Misc:Addr to File Pos** command
IV. A file position window appears. It tells you this about your byte: its segmented address relative to the start of the Nosied resource, its position relative to the start of its file, its containing block (sector) relative to the start of its file, and its position relative to the start of its block. The block number is decimal, all the others are hexadecimal.
V. Go into a disk editor (**Fedit Plus** works well), move to the indicated file position, modify the byte(s) in question, and write the changed sector back to disk.
   A. One caveat : some file editors have numbering conventions different from Nosy's. For example, whereas Nosy gives block (sector) numbers in decimal, Fedit gives them in hexadecimal.
      1. In Fedit's case, it doesn't really matter, since it gives positions relative to the start of a file in hex, just like Nosy.
VI. Example :
   A. You see this piece of hard-wired code :

```
2C06: 91FC 0000 0400 '......'          SUBA.L  #$400,A0
```

   B. You want to change it so $600 is subtracted.
   C. Highlight the address **2C06** and select **Misc:Addr to File Pos**.
   D. A file position window appears, and gives this information :

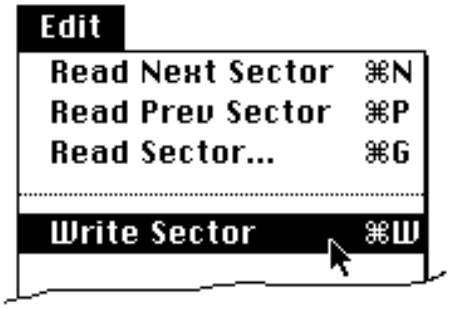
```
2/2C06  File pos: $2D54   blk# 22  offset $154
```

   E. Quit **Nosy** and go to **Fedit Plus**. Open the file you were Nosying, and move along until the sector and file position numbers, on the top and left sides of the Fedit file window respectively, tell you you've arrived. Remember, Fedit uses hex for sector numbers, so don't be surpirsed when you see a sector number of 16 (hex for decimal 22) at the top of the view that shows file position $2D54..
   F. Look down the Fedit window. There's what you're seeking :

```
002D50: 4000 A02D 91FC 0000 0400 2008 90B8 0114
```

   G. Change that $400 to $600, and then record the change :

```
002D50: 4000 A02D 91FC 0000 0600 2008 90B8 0114
```



Warning: If RAM caching is on, you should do Eject a Floppy or such, to ensure that the patch "takes".

Looking At ROM

File Preparation

The Nosy Files folder should contain: **ΔROM**, **ROM/CODE.snt**, **ROM.aci.** You should customize the .snt file to your configuration by doing an Explore, and saving the .snt file. See **Installation Notes** for more details.

The Act Itself

Crank up Nosy. When the opening GetFile dialog appears, just select **ΔROM**. It'll be at the top of the box. In just a few moments, you'll be in Window mode, the ROM spread before you like so many cracked gems. Then just Nosy away like normal.

A Little Example Of What You Can Find

When Stan started work on his Teleport DA, Apple was loathe to share details of the interface between **Launch,** an application's closing **RTS**, and **ExitToShell**. So it's his great pleasure to bring you this example. Press **Cmd-D**, enter Launch **Return** in the dialog box to bring up a listing. Do the same with **ExitToShell**. You'll see lovely stuff. Here are a few excerpts:

```
        ;-          $A9F3   Chain
404C0C: Chain      CLR      LaunchFlag
404C10:            BRA.S     haha

        ;-          $A9F2   Launch
404C12: Launch     ST       LaunchFlag
404C16:  haha      MOVE     4(A0),CurPageOption
404C1C:            MOVEA.L  (A0)+,A0
404C1E:            LEA      CurApName,A1
404C22:            MOVEQ    #32,D0
404C24:            _BlockMove
404C26:            MOVE     CurApRefNum,D0
```

```
              ;-          $A9F4   ExitToShell
  1EA4:    ExitToShell     JMP.L    gitBak
```

```
              ;end of Launch
404CFC:      MOVE.L   AppParmHandle,(A0)+
404D00:      CLR      (A0)+
404D02:      MOVEA.L  A7,A0
404D04:      SUBA.L   DefltStack,A0
404D08:      _SetApplLimit
404D0A:      MOVEQ    #0,D7
404D0C:      JSR      (A3) ;take off to appl
            ;-refs - ExitToShell
404D0E: gitBak  PUSH     CurApRefNum ;we back
404D12:         _CloseResFile
404D14:         CLR      CurApRefNum
404D18:         LEA      LoaderPBlock,A0
404D1C:         MOVEA.L  A0,A1
404D1E:         PEA      FinderName
404D22:         POP.L    (A1)+
404D24:         CLR.L    (A1)+
404D26:         BRA      Launch
```

Looking At Copy Protection

The Key-Disk Method

Some Mac copy-protected software uses a key-disk method. You can start the program from a copy, but at some point you're asked to insert a key, or master disk. The firm that duplicates this key–disk puts one or more files containing bad tracks on the disk. The normal Mac utilities can't copy the bad tracks. A user can copy the program, but not the file(s) with the bad track(s).

The program checks for the existence of the magic file(s). If something is amiss, it refuses to run or mysteriously blows up or pulls some other tiresome trick.

The Encryption Method

Many games are protected by elaborate schemes that involve encrypting part of the program's code. They use non-standard resource ID's for the CODE resources, put garbage in CODE resource 0 (which usually holds the segment jump table), etc. These programs are difficult to crack without first removing the encrypted parts.

Nosy Versus Key-Disks

Here's an algorithm to get you started :

1 Make a backup of the program with one of the nibble copiers. **Copy II Mac** is useful.
2 Run the copy to get a sense of the nasty things the protectors have taught the program
3 Start up **Nosy** and tell it to examine the offensive program.
4 After the first exploratory treewalk, go to TTY mode and try out the **SM** (**S**earch for disk data/address

Markers) command to find blocks that may be fooling with the Sony driver

5 If **SM** finds nothing of interest, stop. You'll have to try something else. See below.

6 **SM** found something. Good. Pop over to Window mode. Use the **Display:Refs to** and **Display:Call chain** commands to find the chain of routines that reference the suspicious block(s).

7 Use the **Display:Code|Data blk** command to list these routines. By inspecting them you should be able to decide which routine(s) to patch to bypass the copy protection algorithm.

8 During these inspections you'll probably find a number of short glue routines. If Nosy hasn't named them intelligently, use the **Misc:name cHange** command to give them the names of their trap call. That'll aid your comprehension when you go back and relist the calling procedure.

This method works with many copy-protected programs. That's because their bad-track files don't contain information necessary for program execution.

Some Other Key-Disk Techniques

Here's a quicker way to find the bad-track checking code : search for references to the ROM tag byte (which lives at location **$400009**). This byte is used to decide what kind of a machine the program is running on. Or look for **Open** trap calls, or the Lisa disk address (**$FCC01E**). Having found one of these indicators, use the techniques outlined in steps 6-8 above.

Another approach : check for references to system symbols with the **Display:Sys syms map** command. Check any references to the symbols: SonyVars, PollStack, PollProc, DskVerify, VIA and IWM.

Still coming up empty-handed ? Look for silly string messages in the code or its allied resources. Things like "program disk must be in internal drive" or "please insert master disk". Inspect procedures that use such messages, looking for decision points. Then try changing these points to unconditional **BRA**nches (machine codes **60xx**) or **N**o**OP**s (machine code **$4E71**).

Nosy Versus Encrypted CODE Resources

We haven't done much of this, but the general idea is to write a “demon” involkable by an F-key that is able to write out the CODE resources of a running program.The idea here is that when the program's running it has to decrypt the encrypted parts.

What Can We Learn From History ?

MacPascal used to have some tenacious copy protection. One had to plug a bit more code, since there were three magic files that it checked, along with a watchdog routine to make sure that you hadn't patched the file checking routines.

A Word About Microsoft Products

They use a UCSD-style P-code interpreter. So all the interesting program code is P-encoded. I haven't had time to decipher or write a disassembler for this stuff.

A Simple Example

Here's an example that shows how to find the standard values for the address/data markers. It does this by studying the 64K ROM from TTY mode. By the way, this example was recorded with the **Main:Outdir** command (see **The Main Menu**):

```
?SM
searching for disk data/addr markers ($D5 xx AD/96)  (example uses the 64K ROM)

match at addr =   402070
                      ;-refs - proc164
 402070: '...'            data29   DC.B    $D5,$AA,$96     ;default values
 402073: '...'                     DC.B    $DE,$AA,$FF
continue [y] ?n

?Map
Trace<C|P #>
>p164
        proc        called by
  402096 proc164      proc115
  40195C proc115
```

Format of a "_GL" APPL file

"_GL" files are fed into Nosy to create ".npl" Libraries. Nosy uses the "npl" Library to do name substitution. An example of a main procedure in a "_GL" file is:

```
MPW_GL      MAIN EXPORT         ; declare Main entry point ( MPW )

      MACRO
      XX    &N
      PEA   #'&N;'              ; push the string 'name;'
      IMPORT      &N
      JSR   &N                  ; call name
      ENDM

      xx    _RTInit             ; force external references to names
      xx    getenv              ; of Library procedures you want to
      xx    onexit              ; be in the ".npl" file
      xx    exit
      xx    _exit
      xx    _RTExit
;     ... and so on,
      RTS                       ; terminate the procedure
      END
```

The MPW shell commands to create the file "MPW_GL" are:

```
asm -case obj MPW_GL.a -o "{ObjNosy}MPW_GL.a.o"  ## C is Case Sensitive
LINK -t "APPL"   -o "{ObjNosy}MPW_GL"
 -sg Main=%A5Init,INTENV,SADEV,SACONSOL ∂  ## force everything into 1 Segment !!
"{ObjNosy}MPW_GL.a.o"  "A:MPW:Libraries:Runtime.o" "A:MPW:Libraries:Interface.o"
```

One could then use Nosy to create the ".npl" file by disassembling MPW_GL. At the end of the treewalk, Nosy will ask if you want to create a ".npl" file.

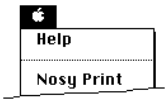
Some Notes

".npl" files were created with the idea of automatically naming the glue routines that were called by the Lisa Pascal compiler to execute the OS trap calls which have a register basaed calling sequence. They also have the property that they don't call (JSR/BSR) any other procedures. With this in mind, I designed a simple pattern match algorithm. Unfortunately, this scheme doesn't work when the Library routine we want to match up with contains calls or references to globals, as many of the C library routines do, and I haven't had time to generalize it.

When it is generalized, it can be turned into a powerful tool to identify the theft of code.

The Apple Menu

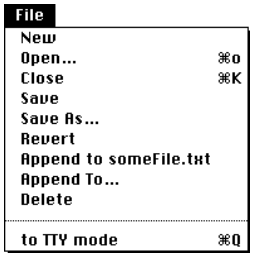
The **Apple Menu** contains the **Help** command and any desk accessories that are installed in **MacNosy** or the **System** file.



The **Help** command is easy to use. Select it, and a window named **–Help–** opens up. You can browse thru this **Help** window via the scroll bar. Or you can take advantage of MacNosy's **Edit** menu commands, as follows: Select one of the topics at the top of the window by double-clicking. Press **Cmd-F** to **Find** the next instance of the selected topic. Press **Cmd-T** to move the help text so the start of the topic's entry is at the **T**op of the help window.

The File Menu

The **File Menu** contains a standard set of Mac file menu commands, and the additon of **Revert**, **Append**, and **Delete** commands. This discussion focuses on the differences from the standard stuff.



The **New** command creates a new Macintosh text editing window. The window comes up with a name of the form **?-*number***. The first such window will be named **?-1**, the second will be **?-2**, and so on. If saved, the saved file's type is **TEXT**, so it can be worked on by any Mac word processor or editor.

The **Open...** command brings up a standard **Open File** dialog and lets you open any Macintosh TEXT file. Pressing **Cmd-O** also triggers this command. If you highlight a name in the topmost window, pressing **Shift-Cmd-O** tells **MacNosy** to try to open a file with that name on the currently MacNosied file's volume.

The **Close** command closes the topmost window. If the window's contents have changed since opening, **MacNosy** gives the standard **Save changes before closing ?** dialog box. Pressing **Cmd-K** also triggers this command. If you want to clean up the MacNosy desktop, just hold down **Cmd-K** until all the windows get gone.

The **Save** command works as you'd expect, as does **Save As...**.

The **Revert** command lets you return a changed window's associated file back to its last saved condition.

The **Append to someFile.txt** command adds the current selection to the named file. The file name is formed by adding the suffix **.txt** to the name of the currently MacNosied file.

The **Append To...** command works similarly, adding the current selection to a file chosen by you from a standard file selection dialog.

The **Delete** command closes the topmost window and deletes the file associated with it. Not to fear, there's an **Are you sure?** dialog before the act is consumated.

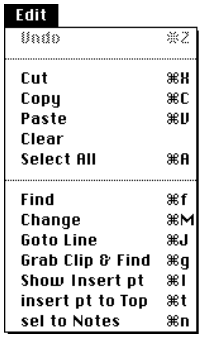
The **Page Setup** is the standard Mac command.

The **Print Window** command prints the contents of the frontmost window with headers.

The **to TTY mode** (**Cmd-Y**) command takes you from **Nosy**'s window mode to TTY mode
To quit **MacNosy**, press **Cmd-Q** to get to **Exit Action** dialog, then click the **Quit Nosy** button.

The Edit Menu

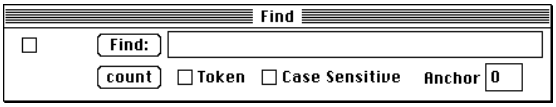
The **Edit Menu** contains the standard Mac edit menu commands, plus some additions. As with our **File Menu** discussion, we'll pinpoint the differences from the standard menu and its commands.



The **Undo** command is unimplemented in **Nosy**.. It's there for any DA's that may want to use it.

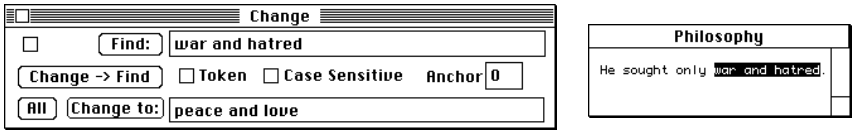
Cut, **Copy**, **Paste**, **Clear**, and **Select All** work in the standard ways.

The **Find** command (**Cmd-F**) lets you find arbitrary strings. All searches wrap around a window's text. A search can look for token (whole word) or partial matches. It can be case sensitive or insensitive. And it can look for anchored matches, which begin before a certain column position in a line, or ones that occur in any position. If there's a current selection, **Nosy** uses it to go on a default search, which is : token-seeking, case sensitive, and not anchored. If there's no current selection, just that blinkin' insertion point, or the **Find** dialog window's hiding somewhere on the screen, the **Find** dialog window comes topmost, and you can set the various search parameters.



The **Find** dialog's **count** button lets you count the number of matches. Finally, holding down the **Shift** key restricts the search to a line's label field: column 1 for **.asm** format windows, column 32 for **list** format windows.

The **Change** command lets you change strings. The keyboard equivalent is **Cmd-M**. A dialog box appears, with options similar to the **Find** dialog and other Mac **Change** dialogs. Here's a snapshot :



The **Goto Line** command lets you hop to any line in the topmost window. Keyboard equivalent is **Cmd-J**. An obvious little dialog box appears -- fill it in and off you go :

The **Grab Clip & Find** command (**Cmd-G**) uses the contents of the Clipboard as the search string for a **Find** in the front window. Caveat: the contents of the Clipboard must be a name no more than 20 characters long. Example: looking at **proc8**, you want to see how another procedure references it. Double-click on **proc8**, **Cmd-C** to copy it to the Clipboard, double-click on the name of a procedure that references **proc8** from the **refs** list at the begining of **proc8**, **Cmd-D** to bring up a display of the caller, and **Cmd-G** to find the reference. Sounds complex, but this is a standard Nosy sequence, fast and useful. Holding the **Shift** key down when invoking **Grab Clip & Find** does a label-field search, as detailed above for the **Find** command.

The **Show Insert pt** command scrolls the window so the text insertion point is at the window's vertical center Also invoked by **Cmd-I**

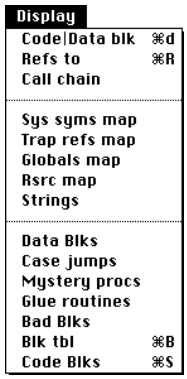
The **Insert pt to Top** command scrolls the window so the text insertion point is at the window's top line. A local Home command. Keyboard equivalent is **Cmd-T**. Holding down the **Shift** key when invoking **Insert pt to Top** moves the insertion point to the beginning of the text prior to the scroll, and thus

functions as a global Home command.

The **sel to Notes** command copies the current selection to the **-Notes-** window. Nice way to click quick snapshots of your Nosying actions. Holding down the shift key will cause the window you copied from to be closed.

The Display Menu

The **Display Menu** contains commands to display all kinds of information about the program being Nosied. This is one of the first places to go when you begin Nosying a piece of code. For most of this menu's commands, selecting one that already has an open window will bring that window to the front.

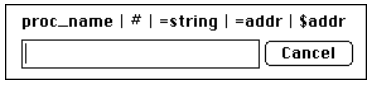


The **Code|Data blk** command displays a block and, if requested, the procedures it calls. **Cmd-D** is the keyboard equivalent. In the discussion that follows, remember that **selecting** and **selected** mean that you've highlighted a string of characters in the topmost window.

If a valid block name is selected, invoking **Code|Data blk** tells Nosy to bring up a display of that block.

If a valid data block name is selected, holding down the **Shift** key while invoking **Code|Data blk** tells Nosy to additionally bring up a display of any procedure block immediately preceding the selected data block.

If there's no selection, a dialog box appears so you can specify a block :



This dialog has several entry options:

You can enter the name of a procedure, common, or data block. For example, entering **com_5** tells **Nosy** to bring up the block of that name.

You can enter the number of a procedure. For example, entering **21** tells **Nosy** to bring up the 21st procedure block.

You can enter a string, prefixed with an equals sign (=), and **Nosy** will bring up a window containing any blocks that contain that string in their name. For example, if you enter **=ate**, and you've got blocks named **Create**, **Ateboy**, and **Material**, you'd get a window containing those three routines (and any others whose names contained **ate**). The search is case–insensitive.

You can enter an address, prefixed with an equals sign (=). That tells **Nosy** to bring up the block containing code or data living at that address. For example, if you enter **=16E**, and **data9** runs from address **$123** thru to **$22F**, **Nosy** will bring up **data9**.

If you're Nosying the Mac ROM, you can enter an address, prefixed with a dollar sign (**$**). That tells **Nosy** to display a procedure whose address is stored in the address you enter. For example, if you enter $124, and $124 contains the value $401234, Nosy will print out the ROM procedure that contains that address. Among other things, this can be used to decipher sequences of the form

PUSH sysGlob
RTS

The **Refs to** command (**Cmd–R**) displays a reference map for the selected name.

The **Call chain** command displays the procs that call a selected proc, and the procs that call them, etc., until all the references are tracked down. Think of it as a multi-level **Refs to** command for procedures.

The **Sys syms map** command displays a map of Macintosh System symbols that are referenced by the **CNR**, along with the parts of the **CNR** that reference them.

The **Trap refs map** command displays a map of Macintosh traps that are referenced by the **CNR**, along with the

parts of the **CNR** that reference them.

The **Globals map** command displays a map of the **CNR**'s global symbols, their values, and the parts of the **CNR** that reference them.

The **Rsrc map** command displays a map of the **CNR**'s resources.

The **Strings** command displays a map of all the data blocks that are formatted as ASCII strings (ABn, Str, ZSTR, etc), along with the parts of the **CNR** that reference them.

The **Data Blks** command shows the data block names. Untyped blocks -- ones that Nosy hasn't yet assigned a data type beyond a simple **DC.W** -- are marked with a > prefix. Blocks that aren't referenced by any other part of the **CNR** are marked with a **?** prefix.

The **Case jumps** command displays information about the case statements.

The **Mystery procs** command displays information about code constructs that **Nosy** needs help to understand.

The **Glue routines** command displays the names of short procedure block that contain 1 trap call and no JSR's. Procedure blocks sporting these features are often used to convert betwen the stack-based calling sequences of most compilers and the register-based calling sequences of certain Mac trap routines.

The **Bad Blks** command displays information about blocks that Nosy thought were code, but contained illegal instructions, so Nosy changed them to data. Of what use is this map ? Well, some programs contain areas of encrypted code. This command reveals many of the encrypted blocks. Rather than trying to decrypt them, you can take a look at the same blocks when the program is running under control of "Thre Debugger". Professor Jasik calls this the "catch them with their pants down" method.

The **Blk tbl** command displays the block table, which contains the following information for each and every block in the CNR : name, segment number, address of the block's first byte, length of the block.

The **Code Blks** command shows the code block names. Procs that don't call other procs (leaves of the call graph) that haven't been given a name beyond the generic **proc***number* are flagged with a >.

The Reformat Menu

The **Reformat Menu** contains commands to change the format of blocks. It's the central clearing house for intelligent block conversation between you and **Nosy**.

The **Review...** command lets you review data blocks. When you invoke it, **Nosy** starts by closing all windows other than **-Notes-**. Then it brings up four new windows : **Cmd:**, a command entry window; **–Cmds–**, which displays the **Review** menu; **–Data Blk–** , which displays the block currently under review (**BCUR**); and **–Use–**, which focuses on another block's reference to the **BCUR** (if any exist)**.** To **review all** blocks (typed and untyped), hold down the **Shift** key when selecting the command. To start with a **particular** data block, select its name. **If the block is typed**, you must hold down the shift key, otherwise Nosy will start at the first untyped block that is physically after the block you selected.
While you're reviewing, keyboard input is directed to the **Cmd:** window. You enter commands indicated in the **–Cmds**– window. The next section of this documentation, **The Review Menu**, covers the various reviewing commands.

The **Link Jmp to Tbl** command defines the connection between a mystery jump and a data block. Here's how you invoke it : select the address of the mystery **JMP**, and then choose this command. You'll get a couple of dialog boxes to deal with. Their use is detailed in the section **Linking Jumps To Tables**.

The **Code to Data** command changes a code block to a data block. Select the name of a procedure or **isPrc** block, then choose this command. The blocks's name won't be changed until after an **Explore** (treewalk).

The **Is Proc** command changes a data block to a code block. It's a faster alternative to **Review...**. Try it out on the System file's PTCH resource The command has a keyboard equivalent, **Cmd-D**. To use it, select the name of a data block, then choose this command. The name of the data block will be changed to **isPrc_***number* until the next **Explore**, and you wont't be able to **Cmd-D Display** it until then.

The **JSR is JMP** command tells **Nosy** about situations where a JSR command is used as a JMP. Huh ? Well,

sometimes a JSR is followed immediately by some data. JSR is used so the code at the destination can pop the JSR's "return" address off the stack, and use it to locate the data. So the destination is not really a procedure block, but a common block. And the data following the JSR is not code, but data. Got that ? Good. To tell Nosy about such situations, select the name of the JSR's destination procedure and choose **JSR is JMP** .

The **Explore** command tells **Nosy** to rebuild its internal tables by going off on another treewalk exploration of the **CNR**. Invoke it after you've changed any blocks.

The Review Menu

The **Review** menu comes up from Window mode when you choose the **Reformat:Review...** command, and from TTY mode when you choose the **Main:Review** command. It means you're in block review mode, where you get to change the format of blocks. A couple of notes : we refer to the **b**lock **c**urrently **u**nder **r**eview as the **BCUR**. And many of the formatting words aren't standard 68000 pseudo-ops, but **Nosy** macros. Check the **Amacs Macros** section for more detail on them.

```
<Hex|Dec|Asc|Zero><B|W|L>n_item, Wstr, Str, WZSTR, Zstr, Jumpc, JUMPP, BRA<L|C>
Undo, Code,  Quit, New{Byt} cnt, <NewLast|NUntil>{hhh}, NewWstr, New*, cOmbine
Mname=fmt1,fmt2., =mac_name, <H|D|A><Fld>cnt, ADDR, LADR<C|P>, DRVHD=pfx
```

<**H**ex><**B**|**W**|**L**> n_item Formats the **BCUR** as **H**ex **B**ytes, **W**ords, or **L**ong words with **n_item** items per line. Example: **HL2** would produce lines like this: DC.L $13F0F052,$40EA1BC8

<**D**ec><**B**|**W**|**L**> n_item Formats the **BCUR** as **D**ecimal **B**ytes, **W**ords, or **L**ong words with **n_item** items per line. Example: **DW2** would produce lines like this: DC.W 5104,21234

<**A**sc><**B**|**W**|**L**> n_item Formats the **BCUR** as **A**SCII **B**ytes, **W**ords, or **L**ong words with **n_item** items per line. Example: **AB4** would produce lines like this : DC.B 'Walt'

<**Z**ero><**B**|**W**|**L**> n_item Formats the **BCUR** as a block of zeroes, organized as **B**ytes, **W**ords, or **L**ong words with **n_item** items per line. Example: **ZB4** would produce lines like this : DZS.B 4.

Wstr Formats the **BCUR** as a word–aligned Pascal–type string.

Str Formats the **BCUR** as a Pascal string.

WZSTR Formats the **BCUR** as a word–aligned zero-terminated string.

Zstr Formats the **BCUR** as a zero-terminated string.

Jumpc Formats the **BCUR** as a set of jump table entries relative to the beginning of the block. The spots jumped to are marked as entry points to common blocks. So the lines take the form **JUMP** ***com_id***, where ***com_id*** is the name of the common block. When you **Q**uit review mode, use **Reformat:Link Jmp to Tbl** to connect the table up to its **JMP** command. See **Linking Jumps To Tables** for more information.

JUMPP Formats the **BCUR** as a set of jump table entries relative to the beginning of the block. The spots jumped to are marked as entry points to procedure blocks. So the lines take the form **JUMP** ***proc_id***, where ***proc_id*** is the name of the procedure block. When you **Q**uit review mode, use **Reformat:Link Jmp to Tbl** to connect the table up to its **JMP** command. See **Linking Jumps To Tables** for more information.

BRA<**L**|**C**> **BRAL** formats the **BCUR** as code. If it finds branching instructions -- **$6*hhh*** -- their destination is given a label of the form **lxx_n,** where **xx** are any two letters, and **n** is a decimal number. **BRAC** does the same thing, but marks branch destinations as com blocks, and gives them a label of the form **com_n .**

Undo **U** changes the **BCUR** back to its original size and format.

Code **C** tentatively changes the **BCUR** to code, then brings up the **Code** menu for further action.

Quit **Q** exits review mode.

New{**B**yt} cnt This command is used to split a block into two blocks. The first block gets the first **cnt** words (or bytes) of the **BCUR**, the second block gets the rest. For example, **NB3** puts three bytes into the first block. **N4** puts four words into the first block. You get an error message if you try to create blocks bigger than **BCUR**.

New**L**ast hhh **NL** followed by a hexadecimal address **hhh** tells **Nosy** to split the **BCUR** into two blocks so

that **hhh** is the address of the last word of the first block. For example, **NL234A** will start the second block at segment–relative byte $234C.

New**U**ntil {hhh} **NU** followed by a hexadecimal pattern **hhh** tells **Nosy** to split the **BCUR** into two blocks so that the second block begins at the word following an occurrence of **hhh**. **NU** alone tells Nosy to search for an RTS, RTD, RTE, RTR or JMP (A0) or JMP (A1) instruction ($4E7h with h = 3,4,5, or 7, $4EDi, i = 0 or 1). Either way, the first block is tentatively changed to Code, and the **Code** menu appears.

New**W**str **NW** splits the **BCUR** into two blocks with the first block typed as a word–aligned Pascal-string.

New***** Tells **Nosy** to split the **BCUR** into two pieces. **N*** looks in the first four bytes (first long word) of the **BCUR** to find a byte count (ignoring the upper 6 bits) which it then uses as the length for the first piece.

c**O**mbine If the previous block is a data block, **O** tells Nosy to combine it with the **BCUR** to form one block.

Mname=fmt1,fmt2,.. This command is used to impose a template or record structure over a block. **Name** is the macro or format name. **fmt1,fmt2** are format descriptiors. The format descriptors take two forms. One is based on the formatting commands available in this **Review** menu. For example, **DB2** describes a component that's two decimal bytes. **Str** defines a component that's a Pascal–type string. And so on. The second format component descriptor form is used with tables. Its syntax is:

jumpc|addr = <*label_prefix*> + <*suffix*>

jumpc is used for tables that are collections of offsets to code blocks. **addr** is used for tables that are collections of addresses. ***label_prefix*** is an identifier that'll be the first part of the table's labels. ***suffix*** can be **P** followed by a decimal number ***n***, indicating that the value of the ***n***th parameter in the macro definition will be used as the second part of a table entry's label. Or ***suffix*** can just be a decimal number ***n***. In that case, a table entry's ordinal position in the table gets added to ***n*** to form the second part of its label.

For example: a STRG rsrc I found in the Mac editor can be formatted by **Midstr=db1,db1,Zstr** or **Midstr=db1,Str,zb1**. A jump table I saw in Macsbug can be done as **Mcmdtbl=ab2,jumpc=cmd_+p1**. This would produce table entry labels like **cmd_qr**, **cmd_dx**, etc. Another jump table in Macsbug can be formatted as **Mopnib=jumpc=opn_+0**. This would produce entry labels like **opn_1**, **opn_2**, **opn_3**, etc.

=mac_name Format the **BCUR** according to the previously defined format macro **mac_name**.

<**H**|**D**><**F**ld>cnt The **F**ield command lets you deal with blocks that include bit–fields. If the **BCUR** begins with a bit field, use this command to define it. **H** signals hexadecimal, **D** decimal, **A** ASCII, and **cnt** is the decimal size of the field. For example, to define a 26-bit field displayed as hexadecimal, enter **HF26**. User beware: this command can be confusing.

ADDR Format the **BCUR**'s as a table of word–length addresses of procedure blocks. It gives you the equivalent to a set of **DC.W proc*xx*** statements (where xx is some decimal number).

LADR<**C**|**P**> This command tells Nosy that the **BCUR**'s a table of long-word addresses : **LADRC** if they're addresses of common blocks, **LADRP** if they're addresses of procedures. For example, if you want the equivalent to a set of **DC.L com_*xx*** (where ***xx*** is some decimal number) statements, use **LADRC**. Likewise, use **LADRP** for the equivalent to a set of **DC.L proc*xx*** statements.

DRVHD=pfx This command tells Nosy to treat the **BCUR** as a nine–word–long driver header: four words of binary info followed by a five–word jump table to the driver's routines. It prefixes the jump table labels with a **pfx** you supply. For example, **DRVHD=laser** would give you the labels laserOpen, laserControl, laserStatus, etc. See the Serial driver journal files on Delphi for more notes.

The Code Menu

The **Code** menu comes up from Window or TTY mode when you choose a **Review** menu command that tentatively turns the **BCUR** (**B**lock **C**urrently **U**nder **R**eview) into code. Among other things, the **Code** menu lets you formalize the change or undo it

```
Undo, Is_proc, Quit, Revert_to_data, New... cr for next
```

The **U**ndo command (enter **U**) changes the **BCUR** back to its original size and format. Nosy then takes you back to the **Review** menu, looking at the same block in its **U**ndone form.

The **I**s_proc command (enter **I**) tells Nosy to formalize the **BCUR**'s change to code. The change will survive the next treewalk. Nosy then takes you back to the **Review** menu and on to the next block. If there's no next block to review, Nosy leaves the **Review** menu and goes back to where **Review** was invoked from.

The **Q**uit command (enter **Q**) undoes the **BCUR**'s change to code, then takes you back to where the **Review** menu was invoked from.

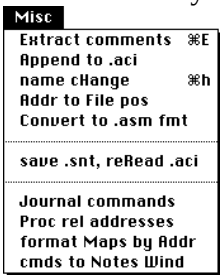
The **R**evert_to_data command (enter **R**) changes the **BCUR** back to its original form, but it retains its current size. Nosy then takes you back to the **Review** menu, looking at the same block in its **R**everted form.

The **N**ew… command lets you split the **BCUR** into pieces. It works just like the **Review** menu's **New** commands. Look at **The Review Menu** for the multitude of details.

The **cr for next** command (press **Return**) tells Nosy to temporarily keep the **BCUR** as code. The change **won't** survive the next treewalk. Nosy then takes you back to the **Review** menu and on to the next block. If there's no next block to review, Nosy leaves the **Review** menu and goes back to where **Review** was invoked from.

The Misc Menu

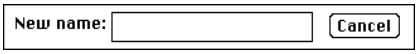
The **Misc** menu is a catch-all for commands that don't fit in any other category.



The **Extract comments** command (**Cmd–E**) extracts comments from the selection. The selection is assumed to be the display of a procedure. The comments appear in a window (called **-comments-**, surprisingly) for your inspection. You can add them to the **CNR**'s **.aci** file by selecting them and invoking the **Append to .aci** command. See **Commenting Via .aci Files** for more detail.

The **Append to .aci** command appends the selection to the end of the **CNR**'s **.aci** file. See **Commenting Via .aci Files** for more details.

The **name cHange** command lets you rename a procedure, data, global or common symbol. **Cmd–H** is the keyboard equivalent. Select the old name, and up pops a dialog box:



Press **Return** when you've entered an agreeable new name, or click **Cancel** to abort the change. If you complete the change, all windows on the desktop get updated. **Cmd-Shift-H** is a temporary kludge to change names local to a procedure. See **Renaming** for more details.

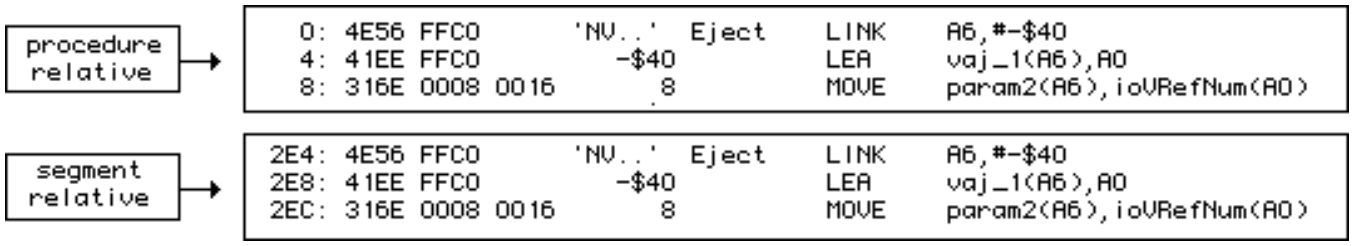
The **Addr to File pos** changes the selected **CNR** segment–relative address to a file–relative disk address You get a file–relative offset (in hex), a block (sector) number (in decimal), and a block–relative offset (in hex). Then you can use a disk editor to patch the resource. See **File Modification** for an example.

The **Convert to .asm fmt** changes the topmost window's listing by removing the first 31 columns of each line. It also appends **.asm** to the window's title.

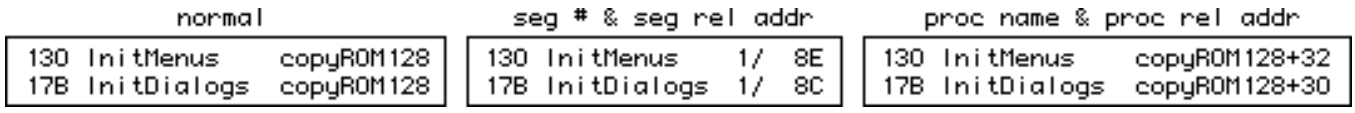
The **save .snt, reRead .aci** command tells Nosy to save its internal tables to disk on the ***cnf_id*.snt** file, and re–read the **CNR**'s **.aci** file (if it exists).

The **Journal** command toggles the recording of commands on the journal file. A check mark in front of **Journal** indicates journaling is **on**. See **Journaling** for more information.

The **Proc rel addresses** command tells Nosy to toggle the way it displays a listing's first–column addresses. Normally they're segment relative. This command lets you make them procedure–relative. A check mark in front of this command indicates the procedure–relative option is **on**.



The **format Maps by addr** command tells Nosy to toggle the format of **Display** menu maps. Normally, references just give a procedure name. This command lets you see the references as a segment number, followed by a slash, followed by a segment–relative address. When it's on, you'll see a check mark before the menu item. And, if both this option and **Proc rel addresses** are **on**, the references are formatted as a procedure name, followed by a plus sign, followed by a procedure–relative offset. Here are some pictures:



The **cmds to Notes wind** command isn't implemented yet. When it is, it'll send a record of your commands to the **Notes** window.

The **Extract Map Names** command acts on the selected part of the front window (assumed to be a MPW map file) to extract the names of the procedures and convert them to Nosy name change commands. This command is only used to convert the ROM map files supplied with MPW 2.x to a format where the names can be added to the ROM disassembly, and hence saved in the ROM .snt file.

The Tables Menu

The **Tables** menu gives you an on-line, interactive version of **Inside Macintosh**.

The **Record names** command brings up a window that lists Macintosh data type names that Nosy knows about. Select a name, then invoke the **Fields of** command to see its structure.

The **Fields of** command has a number of uses. Its keyboard equivalent is **Cmd-Space**.

selection	**display**
Typename	fields of the record Typename
Typename@address	Structured display of the values of Typename
number, @number	Hex/ASCII display of memory at number
=nnnn, c@nnnn	Unstructured Disassembly starting at nnnn

The **OS Traps** command opens a window that shows Operating System trap information. Each trap entry shows a hex trap number and a Pascal-like function/procedure definition. See the **Register-Based Trap Call Symbols** paragraph of **Program Conventions** for a description of the format used.

The **TB Traps** command opens a window that shows Toolbox trap information, formatted just as is the aforementioned OS trap information.

The **Sys Syms** command displays **IM** System symbols, mostly low memory globals. The keyboard equivalent is **Cmd-W**. If the **Shift** key is held down, current values are given. The information includes : a hex address, a symbolic name, an **IM** type name, a current value (if called Shifted), and a comment. If a valid symbolic name is selected, then a window comes up with just that System symbol's information. Otherwise, all globals between **$100** and **$C00** are displayed.

The **Sys Errs** command opens a window that displays the various **IM** error codes (in decimal), their symbolic names, and pithy comments on their meaning.

The **Constants** command opens a window that contains **IM** constants displayed as equate definitions and/or enumerated type names.

The **Ascii** command opens a window that contains decimal, ascii and hex equivalents.

The **Calculate** command (**Cmd-[**) calculates the value of the selected expression or opens a dialog box that you can type an expression into. A complete description of the expression syntax can be found in “The Debugger” manual.

The **Convert Hex to Decimal** command (**Cmd-minus**) converts the value of a selected number to decimal, and displays it as a decimal number, an unsigned decimal integer (low 16 bits), its ASCII equivalent, and the name of the System global (if any) associated with the value.

NOTE: To obtain the Hexidecimal value of a number , use the calculate command. Remember that all numbers are hexidecimal unless preceeded by a “#” sign. for example: “#256” -> $100

The **add/ZAP Type Defs** command adds a set of Pascal format type declarations to the set of types known to Nosy. If the shift key is held down then **all** user defined types are deleted.
To view the user defined types, hold down the shift key and execute the **Record names** command.

Command Key Summary

Remember, this is just a summary. Check out a particular menu's reference section for more detail on each command. Menus and commands are indicated in square brackets, like this : [Edit:Cut]

Cmd-period.....Abort command that is in the progress of generating a window

Cmd-A.....Select all the contents of the topmost window [Edit:Select All]

Cmd-B.....Show the blocks table [Display:Blk tbl]

Cmd-C.....Copy the selection to the clipboard[Edit:Copy]

Cmd-D.....Display a block [Display:Code|Data Blk]

Shift-Cmd-D .....Display a data block with the preceding code block [Display:Code|Data Blk]

Cmd-E.....Extract comments from the current selection [Misc:Extract comments]

Cmd-F.....Find a string [Edit:Find]

Shift-Cmd-F.....Find a string, restrict search to label field [Edit:Find]

Cmd-G.....Find the string that's on the Clipboard [Edit:Grab Clip & Find]

Shift-Cmd-G .....Find string that's on the Clipboard, restrict search to label field [Edit:Grab Clip & Find]

Cmd-H.....Change the selected name [Misc:name cHange]

Cmd-I.....Move a window's contents so the insertion point is visable [Edit:Show insert pt]

Cmd-J.....Go to a particular line in a window [Edit:Goto Line]

Cmd-K.....Close the file associated with the topmost window [File:Close]

Cmd-M.....Change one string to another [Edit:Change]

Cmd-N.....Copy current selection to the **-Notes-** window [Edit:sel to Notes]

Shift-Cmd-N .....Copy current selection to the **-Notes-** window and close its window [Edit:sel to Notes]

Cmd-O.....Open a file via the standard selection dialog [File:Open...]

Shift-Cmd-O .....Open the file whose name is selected in topmost window [File:Open...]

Cmd-P.....Mark the block whose name is selected as a procedure block [Reformat:Is Proc]

Cmd-Q.....Exit to TTY mode [File:to TTY mode]

Cmd-R.....Show the references to the block whose name is selected [Display:Refs to]

Cmd-Space.....If the selection is the name of a record or enumerated type, show the type's structure. If the selection is an address, show memory as hex and ASCII data. If the selection is an address prefixed by = or c@, show memory as disassembled code. If the selection is typename_@addr, show memory as structured data [Tables:Fields of]

Cmd-T.....Move the topmost window's contents so the insertion point is at the top[Edit:insert pt to Top]

Shift-Cmd-T.....Move the topmost window's text insertion point to the first character, then move things so the insertion point is at the top of the window (home window) [Edit:insert pt to Top]

Cmd-V.....Paste the Clipboard into the topmost window [Edit:Paste]

Cmd-W.....Explore [Edit:Paste]

Cmd-X.....Cut the selection to the Clipboard [Reformat:Explore]

Cmd-Z.....Undo [Edit:Undo]

Cmd-minus.....convert Hexidecmal number to Decimal [Tables:Convert Hex to Dec]

Cmd-[.....Calculate [Tables:Calculate]

Cmd-].....If the selection's the name of a low–memory global (system symbol), bring up a window showing its type and value, else bring up a table of system symbols. [Tables:Sys Syms]

Shift-Cmd-].....If selection, then Cmd-], else bring up a table of system symbols and their current values. [Tables:Sys Syms]

Command Language

Two More Acronyms

The MacNosy TTY mode commands form a simple language. For convenience and levity we'll call that language **MTCL** (standing for MacNosy TTY Command Language, and pronounced "MighTy CLear").

The MacNosy TTY mode menus use a particular notation to describe the syntax of **MTCL** commands. We'll refer to that notation as **NotJas** (standing for Notation of Jasik, and pronounced "Not Jazz").

Selectors, Arguments, And The Structure Of A Command

In **MTCL**, a command consists of a command selector followed by zero or more arguments. Arguments may have one or more elements, and an argument element is chosen by a selector or by specific data. **NotJas** indicates command selectors and selectored arguments with capital letters embedded in descriptive strings.

Command Entry

Type it in, then press **Return**.

Mandatory And Optional Arguments

A **MTCL** argument can be mandatory or optional. **NotJas** indicates optional arguments by enlosing them in curly braces (**{** and **}**). Mandatory arguments show up enclosed by angle brackets (< and >) or naked.

Alternatives

When an argument has several options, a vertical bar (|) separates the alternatives.

Non-Selectored Arguments

Some arguments are indicated via command selectors, as noted above. Other arguments are indicated with specific data, numbers or strings. **NotJas** indicates non-selectored arguments by using all lowercase letters in the argument's descriptive string. The individual command descriptions in each TTY menu's reference section describe the required data.

Spaces

MTCL doesn't like spaces. Do not type in spaces. A command with a leading space will be treated as a comment in the **Main** TTY menu, and ignored in other menus.

Defaults

Some commands default to particular argument values. The individual command descriptions in each TTY menu's reference setion describe any defaults.

Consistency

Nosy is not perfectly consistent in its use of **NotJas**. Remember, as in all computer applications, reality is what happens. Again, refer to the individual TTY menu reference sections for command clarification.

A Few Examples

Here are some **MTCL**/**NotJas** examples from the TTY menus:

The **NotJas** Notation : **Review{Untyp|All}{d#}**
Some Explanation : **R** invokes the **Review** command, which lets you review data blocks. **Review** takes two optional arguments. The first tells Nosy what kinds of data blocks to review, the second gives a data block number to start the search at. **RU** tells Nosy to review untyped blocks. **RA** tells Nosy to review all blocks. The **d#** non-selectored argument wants a data block number. **R** defaults the first argument to **U**, the second parameter to 1. Thus, **RA21** starts a review of all data blocks starting at the 21st data block., and a simple **R** command starts a review of all untyped data block, starting with the first.

The **NotJas** Notation : **Not_proc <name>**
Some Explanation : **N** invokes the **Not_proc** command. **Not_proc** tells Nosy to change a block's classification from procedure to data. It requires a non-selectored parameter, **name**. As you may have guessed, the **name** parameter wants a procedure block's name. **Nproc1**, for example, tells Nosy to mark **proc1** as a data block..

The **NotJas** Notation : **Explore**
Some Explanation : **E** invokes the **Explore** command, sending Nosy off on a treewalk. No parameters for this baby.

The Main Menu

The **Main** menu is your central TTY mode command post. Full of powerful commands, it's also the doorway to the other TTY menus. It contains two powerful commands -- **Allproc** and **Srch** -- that still have no Window mode equivalents. And it's the only way to get to the **Exit Actions** dialog (see **Shutting Down**).

```
Review{Untyp|All}{d#}, By_attr, Explore, Display, Allproc{seg#}, Proc_byname, addr
Srch <Mark | Hex hhh | Str chrs | Rsrc_typ chr4 | {Addr_ref | Trap_ref}<name_list> >
 Hardcopy, +Jrnl, Wait, Tbls, List_rsrc{type}, ref_Map, Outdir{file}, saV_snt, Quit
```

Review{**U**ntyp|**A**ll}{d#} This command lets you review data blocks. **R** alone or **RU** reviews questionable blocks (untyped and/or unreferenced), starting with the first. **RA** reviews all blocks, also starting with the first. If you supply **R**, **RU**, or **RA** with the optional data block number, the review starts with that block. As you review, a block comes up, followed by the **Review** menu. See **The Review Menu** for the details of its commands. **Introductory Cruise** and **Reviewing Data Blocks** contain information on the art and science of block reviewing.

By_attr This command, invoked by entering **B**, lists parts of the **CNR** that Nosy's having trouble understanding. They're the same things that get listed in the **Window** mode **Mystery procs** window. You're better off tracking them down in that mode.

Explore This command, invoked by entering **E**, tells Nosy to go on another treewalk exploration of the **CNR**. This is how Nosy metabolizes any information you've supplied since the last treewalk.

Display This command, invoked by entering **D**, takes you to Window mode

Allproc{seg#|**B**blk#|**B**n-**B**m} This command lets you list one or more blocks of the **CNR**. **A** alone lists all blocks in all segments of the **CNR**. **A** followed by a number lists blocks in the segment with that number. For example, **A2** lists all the blocks in segment 2 of the **CNR**. **AB** followed by a number lists all blocks, starting with a particular block. For example, **AB14** lists all blocks, starting with block 14. You can also list a range of blocks. For example, **AB12-B24** lists the twelfth through twenty–fourth blocks. If you don't know a block's number, check the listing that's obtained with the **Table & Debugging** menu's **Blocks** command.

There's no equivalent to this command in Window mode. When you combine it with **Hardcopy**, you can get a complete listing of the disassembled **CNR** onto a disk file.

Proc_byname This command lists a procedure and (optionally) the procedures it references, in order of reference. Invoke it by entering **P**. A prompt for procedure selection then appears:

proc name | # | =str | =addr |$addr

This works just like Window mode's **Display:Code|Data blk** command's dialog box -- see **The Display Menu** for details. After you select a procedure, you'll be asked whether you want to see the chain of referenced procedures.

addr This command tells Nosy to list several hundred bytes of the **CNR** as code, starting at the given address. The address should be 8 hexits long, with the segment number in the first two hexits. For example, entering **01000234** will start the listing at byte **234** of segment **1**. Entering **0** (**zero**) will cause the entire file to be listed as code. The listings produced ignore the results of Nosy's treewalk explorations. Note: Nosy scans the **CNR**, building a table of local labels, before starting the listing. So it takes some time before a long **CNR**'s listing appears.

Srch <**M**ark | **H**ex hhh | **S**tr chrs | **R**src_typ chr4 | {**A**ddr_ref | **T**rap_ref}<name_list> > This is another TTY mode command that (currently) has no Window mode equivalent. It lets you search the **CNR** for various types of objects.

Search **M**ark **SM** tells Nosy to search the data blocks for occurrences of disk data/address prologue (**D5 xx AD**) or epilogue (**DE AA FF**) markers. There's a good chance that the code referencing these markers is doing funny disk stuff. Thus, for many programs, the **SM** command lets you track down copy protection code. See **Looking At Copy Protection** for more information and an example.

Search **H**ex hhhh - **SH**, followed by an even number of hex digits, tells Nosy to search for an occurrence of those hexits in the **CNR**. For example, **SH4e0103** searches for occurrences of **$4E0103**.

Search **S**tr chrs - **SS**, followed by a string, tells Nosy to search the **CNR** for occurrences of those characters. The search is case–sensitive. For example, **SSApple** lets you find copyright notices in the Macintosh ROM.

Search **R**src_typ chr4 - **SR**, followed by a four–character resource type name, tells Nosy to search for occurrences of that 4–character string. For example: **SRcdef** tells it to look for references to **CDEF**.

Search **A**ddr_ref name_list - **SA**, followed by a list of names, tells Nosy to search code blocks for references to the addresses represented by the names. For example: **sap2,g4,d5** tells Nosy to look for references to **proc2**, **glob4** and **data5**.

Search **T**rap_ref name_list - **ST**, followed by a list of names, tells Nosy to search for occurrences of the given trap names. For example, **stopen–launch** tell Nosy to look for traps in the range from **Open** to **Launch**.

Hardcopy {volname:|**?**:}{fname} This command lets you send Nosy output to a file. The options are:

You can choose from two listing formats: **assembly list** format, which is how Nosy normally lists things, or **asm** format, which can be sent to an assembler. Nosy prompts you for a choice; the default is **assembly list**.

If you enter **H** alone, and **Hardcopy** isn't turned on yet, the file name for the hardcopy output defaults to ***cnf_id*.list** or ***cnf_id*.asm**, depending on the listing format you've selected.

If you enter **H** alone, and **Hardcopy**'s been turned on, Nosy will close the hardcopy file and turn **Hardcopy** off.

If you enter **H?:**, **H?:** followed by a file name, or **H** followed by a volume name and colon (no file name), Nosy will bring up a standard PutFile dialog box. This lets you set a file name, folder (under HFS), and volume for the hardcopy file.

If you enter **H** followed by a file name, Nosy will send hardcopy to that file on the default volume.

If you enter **H** followed by a volume name, a colon, and a file name, Nosy will send hardcopy to that file on that volume.

Our advice: if you don't want a default name, enter **H?:** and work with the PutFile dialog box.

A **+** before **Hardcopy** in the menu tells you it's **on**.

Hardcopy doesn't save output from every command. It does work with **Allproc**, **Proc_byname**, **By_attr**, and **Srch**. If you want to capture everything, use **Outdir**. See below.

Jrnl **J** toggles the journaling flag. Nosy starts up with journaling turned on. The **on** state is indicated by a **+**in front of **Jrnl** in the menu. See **Journaling** for much more detail.

Wait This command doesn't do anything when entered from the console. But when it's encountered in a journal file, **W** causes Nosy to wait for the user to press the **Spacebar**.

Tbls Entering **T** brings up Nosy's **Table & Debugging** menu.

List_rsrc{type} **L** all by itself tells Nosy to summarize the **CNR**'s resource map. For each resource type, you get its type name, then information about the particular resources of that type. This information includes the resource's ID number, its attribute byte, an index indicating its relative position, and its length.

L followed by a resource type name tells Nosy to list info for all resources of that type. You get the same information that **L** alone would give, plus a hexadecimal dump of the actual resource data. And you can add a resource ID number after the type name, which limits the information to that particular resource. To bypass uppercasing of the type name, prefix it with a **>**.

Some examples: **L** maps all of the **CNR**'s resources. **Lcode** maps all **CODE** resources. **Lcode0** maps the **CODE 0** resource. **LPack4** maps the **PACK 4** resource. **L>DoBe** maps all **DoBe** resources.

ref_**M**ap The **M** command takes you to the **Map** menu, where you can get info on the **CNR**'s symbols and chains of reference. See **The Map Menu** for more details.

Outdir{file} **O** followed by a file name redirects all console activity to that file. **O** all by its lonesome stops the redirection and shuts the file. It differs from **Hardcopy** mainly in that **all** console activity is saved to disk.

sa**V**_snt This command tells Nosy to save the current state of its internal tables as a **.snt** file (if they've changed since the last such save) file, and to reread the **CNR**'s **.aci** file, if it exists.

Quit This is your door out of Nosy. Entering **Q** takes you to the **Exit Actions** dialog.

The Review And Code Menus

When you select TTY mode's **Main:Review** command, **Nosy** takes you to a **Review** blocks menu. This menu's the same as the one you get with Window mode's **Reformat:Review…** command. And the **Code** menu you get to with the **Review:Code** command is also the same as Window mode's **Code** menu. So just check out the prior Window mode reference sections **The Review Menu** and **The Code Menu**.

The Intermediate Menu

This menu is encountered when you press the **Spacebar** during some TTY listing process.

```
Quit, cr to_eXit, conVert{addr}, Chng P|D|G|C #/newname, Is_proc<data#>, User_stop
Not_proc <name>, Tbls, ref_Map, Outdir{file}, Define_case_jmp{jmp_addr}
```

Quit Entering a **Q** aborts the current command and returns Nosy to the menu the process was invoked from..

cr to_e**X**it Pressing **Return** or entering an **X** gets you out of the **Intermediate** menu, and restarts the listing process the **Spacebar** press interrupted.

con**V**ert{addr}This command converts a segment–relative address into a file–relative address. **V** alone uses the last listed address. **V** followed by a segment–relative address uses that address. The file position is indicated with a file–relative offset (in hex), a file–relative block number (in decimal), and a block–relative offset (in hex).

Chng **P**|**D**|**G**|**C** #/newname This command lets you change a symbol's name. **P** stnads for procedure block, **D** for data block, **G** for global, and **C** for common block. For example, **CP2/myname** changes the second procedure block's name to **myname**.

Is_proc<data#> This command tells Nosy that a data block should be regarded as a procedure during the next treewalk. For example, **i8** indicates that.the eighth data block should be regarded as a procedure block.

User_stop When inserted in a journal file by Nosy, this command will cause a listing to stop at that spot. Otherwise it has no effect. Useful for educational purposes.

Not_proc <name> This command tells Nosy that a procedure block, indicated by its name, should be regarded as a procedure during the next treewalk. For example, **NinitVars** tells Nosy that the procedure block named **initVars** should be regarded as a data block.

Tbls This command, invoked by entering **T**, brings up the **Table & Debugging** menu.

ref_**M**ap This command, invoked by entering **M**, brings up the **Map** menu.

Outdir{file} This command lets you send all console activity to a file. See **The Main Menu** for more detail.

Define_case_jmp{jmp_addr} Fill in information about a jump table that Nosy can't figure out. A series of questions appears. You get to specify the location of the jump instruction, the location of the jump table, the structure of the jump table, the number of entries in the jump table, and the table bias. See **Linking Jumps To Tables** for more detail. Frankly, it's easier to just do your jump table work from Window mode.

The Map Menu

The **Map** menu lets you get lists of symbols in the disassembly, and references to them. During a listing, clicking the mouse button lets you pause/restart. Pressing the **Spacebar** brings up the **Intermediate** menu -- see the previous section.

```
All_refs{G|D|P|C|T|S}, Refs<name_list>, Fmt_by<Proc|Addr|Name|Seg/proc>, cr to eXit
Trace<C|P #>, Seg_map
```

All_refs{**G**|**D**|**P**|**C**|**T**|**S**} This command lets you list all the references to a set of symbols. **A** alone lists everything, **AG** lists references to globals, **AD** lists references to data blocks, **AP** lists references to procedure blocks, **AC** lists references to common blocks, **AT** lists references to trap names, and **AS** lists references to System symbols.

Refs<name_list> This command lets you see references to one or more symbols. Symbols of interest are specified via a list of names. Abbreviations are accepted for elements in this list: **g** for global, **d** for data, **p** for proc, **c** for common, **t** for trap. Follow these abbreviations with a decimal number. For example, **Rc1,p1-p5,t224** will list the references to **com_1**, **proc1** to **proc5**, and **Trap $E0** (**OfSetRgn**)

Fmt_by<**P**roc|**A**ddr|**N**ame|**S**eg/proc> This commmand lets you set up the format of references. References can be noted by procedure block name (**FP**), by segment number and segment–relative address (**FA**), by procedure name and procedure–relative offset (**FN**), or by segment number and procedure block name.

cr to e**X**it Pressing **Return** gets you out of here, back to where you invoked the **Map** menu from.

Trace<**C**|**P** #> This command lists a calling chain of procedures that reference a given coomon or procedure block. **TC2** would do it for the second common block, **TP12** would do it for the twelfth procedure block, and so on. Conceptually similar to a static stack crawl. Use it in conjunction with the **Main:SM** command to hunt down copy protection -- see the example in **Looking At Copy Protection**.

Seg_map This command, invoked by entering **S**, produces a map of the block names and references to them, organized by segment. Each line of the map takes the form

seg_number] *block_id fba blen ref_list*

where *seg_number* is a segment number, *block_id* is the block's nmae, *fba* is the segment–relative address of the first byte in the block, *blen* is the block length, and *ref_list* is a list of references to the blocks. Each reference takes the form

{*seg_number /* } *block_id*

thus pinpointing the block containing the reference. The *seg_number* is optional here, present only if it differs from the referenced block's segment.

The Seg map is useful for tracking and deleting unnecessary jump table entries caused by poor placement of procedures, etc.

The Table & Debugging Menu

This menu lets you look at various tables and memory areas. It can be reached from the TTY mode **Main** and **Intermediate** menus. Its prompt character is a right parentheses (**)**).

```
Blks{S#|addr}, Syms{wf}, Tbl<GLDCPV>, Heap, Names, Mem adr/#, Case, Rec  cr = eXit
```

Blks{**S**#|addr} The **B**locks command triggers a listing of the blocks table. **B** lists all blocks. **BS**#lists all blocks in segment **#**. **B*addr*** lists all blocks at addresses greater than ***addr***. **addr** may be an address or a six-hexit address prefixed by a 2-hexit segment number. Examples : **BS2** lists all blocks in code segment 2. **B06003EF7** lists all blocks in segment 6 at addresses greater than $3EF7. **B3EF7** lists all blocks in segment 1 located at addresses greater than $3EF7.

Syms{wf} The **S**ymbols command lists the names and values of System symbols and error messages. Similar to the Window mode **Tables:Sys Syms** and **Tables:Sys Errs** commands. Enter **S** to list the symbols to the screen. Enter **S*wf***, where ***wf*** can be any character, to write the tables to disk files whose names are determined by Nosy. That option was used to debug MacNosy.

Tbl<**GLDCPV**> The **T**ables command produces a table that shows the values held in a particular Nosy symbol table. **TG** does it for **g**lobal variables, **TL** does it for **l**ocal procedure labels, **TD** does it for **d**ata blocks, **TC** for **c**ommon blocks, **TP** for **p**rocedure blocks, and **TV** for local stack frame **v**ariables. By the way, vertical bars (|) are missing from this command's **NotJas** notation because of a lack of space.

Heap Enter **H** to lists the current state of Nosy's internal tables.

Names Enter **N** to list Nosy's internal name table.

Case Enter **C** to list jump table information. Jump tables implement **case** statements.

Rec Enter **R** to list Macintosh record structure information.

cr = e**X**it Press **Return** or enter **X** to exit this menu and return to where it was called from.

Some Technical Poop

First, A Word from Our Sponsor: CODE Resources

The format of CODE resources is defined in the Segment Loader Chapter in **IM**. Nosy expects CODE resources to be in that format, and handles it specially. Be aware that some developers of copy protection schemes subvert things by using non-standard formats for these resources. In such cases you may have to write a special–purpose program to massage the CODE resources, prior to disassembling them.

Now, on the format of CODE resources: When you choose to disassemble the CODE resources, Nosy assumes that there is a resource with ID =0 (that's the inter-segment jump table) and that the rest of the CODE resources have the format:

id_len:Longint; seg_index:integer; n_entries:integer; followed by the code for the segment.

When you select a resource of type other than CODE, Nosy will disassemble only one resource ID at a time. You may use these facts and ResEdit, etc to create files with resources that Nosy can disassemble.

Programmers, Compilers, MacNosy, And You

Programmers take human concepts and transform them into computer language concepts and/or actual processor instructions. Compilers take computer language concepts and transform them into other computer language concepts and/or actual processor instructions. MacNosy takes actual processor instructions and transforms them into computer language concepts. You take these computer language concepts from MacNosy, refine them, then try to figure out what the original programmer was up to. An interesting ride up and down the levels of abstraction.

Nosy's job in this chain isn't trivial. It makes use of a number of concepts from compiler theory. This section discusses some of them, along with a few details of Nosy's operation. We try to keep things simple. The discussion is also brief. For further information, read (and reread) Aho, Sethi, and Ullman's book **Compilers : Principles, Techniques, And Tools** (hereinafter referred to as **CPTT**).

Flow Of Control

Von Neumannesque computers do one thing, then another, then another,…. The sequence of execution -- which thing gets done after which other thing -- is also know as the **flow of control**. Depending on the level of abstraction, the thing that gets done may be a machine instruction, a language statement, a subroutine, a program, or a suite of programs.

Basic Blocks

"A **basic block** is a set of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end." --Section 9.4 of **CPTT** In terms of control flow, a basic block is a self-contained unit. Entry to the block is at the first statement, and exit from the block is at the last statement. No instruction in the block branches, except possibly the last. All this makes it safe to think of a block's collection of statements as a single unit.

Consider this procedure fragment written in GRoovy Universal Example Language (GRUEL):

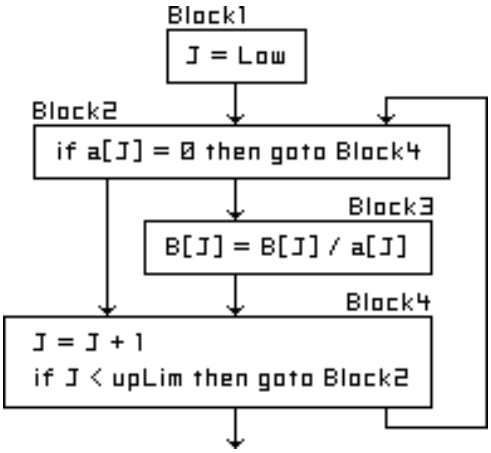
```
(1)    J = Low
(2)    if a[J] = 0 then goto (4)
(3)    B[J] = B[J] / a[J]
(4)    J = J + 1
(5)    if J < upLim then goto (2)
```

By the above definition, we have four basic blocks here: statements (1), (2), (3), and (4) and (5). Think about it. Got it? Good. If not, refer to Section 9.4, Algorithm 9.1 of **CPTT** for a nice block–partitioning algorithm.

Flow Graphs

We can draw a picture that helps us see control flow relationships among a set of basic blocks. We put each block in a box, then draw arrowed lines between boxes that represent the flow of control. The boxes are called nodes, the lines are called directed edges, and the result is a flow graph.

Here's a flow graph for the program above:



A few notes are in order: First, because of the way our definition restricts control flow in and out of a basic block, we're able to replace line–oriented jump destinations with block identifiers.Second: we can use any simple closed figure for the nodes: circles, rectangles, triangles, etc. Fourth: the directed edges can be straight or curved lines. Third: flow is assumed to enter the graph at the topmost block.

Control Flow Analysis

Control flow analysis concerns itself with understanding the topology of control flow in a piece of code. In compiler design, we can use its insights to organize and optimize the generated code. In MacNosy, we use it to discover basic blocks hidden in the spaghetti pile of machine code and data that make up the **CNR**.

Program Call Graphs

The process of collecting flow of control information is fairly simple for MacNosy. Analyzing it to find simple topologogical structures (loops, subroutines, et al) is slightly more complicated, but still pretty straightforward. When we do this on an inter–procedural level, the resulting graph is called the **program call graph**.

Building a complete program call graph in the presence of procedures that accept parameters as arguments is a bit more difficult. This is easier for compilers, since very few of them do inter-procedural optimization. For Nosy, it represents a problem, since Nosy is initially ignorant of the type and number of a procedure's arguments. To date, my solution has been to have you, the Nosier, supply certain information manually with the **Review data** and **IsProc** commands. As time goes on, Nosy will get smarter, and be able to get by without your help.

Data Flow Analysis

One of the techniques Nosy uses to get smarter about program call graphs is called **data flow analysis**. It's the process of collecting information about the way variables are used in a program. Data flow tables summarize this information.

Both control and data flow information are necessary in order to do effective global optimization or de–compilation. The first step is to gather a procedure's control flow information, and build data flow tables for its basic blocks. Then the control flow information is used to solve the data flow equations globally. An optimizing compiler can then answer a number of questions, including:

- Which data entities are not defined in a basic block? The resulting set of expressions that use those items are candidates for various optimizations, including code motion out of the block.
- If a data entity is assigned to a register inside a basic block, will it have to be stored in memory prior to exiting the block? In other words, does it get referenced later on? If so, it's called live on exit.

The Use/Def And Live/Dead Information

A compiler gathers **use/def** information to answer the first question above. For each data entity in a basic block, the use/def information determines whether it's: used in the block, defined in the block, and/or used before being defined.

From the use/def information and results from the control flow analysis, the compiler can answer the second question above for all a program's data entities. This is called the **live/dead** information.

Aliases

For real–world computer languages, the problem of gathering use/def information is complicated by the **aliasing** of names. This aliasing comes about through the presence of array references and pointer variables. You're probably familiar with the problem, but haven't given it a printable name.

Here's a simple example of aliasing, where **p** is a pointer to an integer variable **i** :

```
(1)   p = address(i)        ; p gets pointed to i
(2)   x = A[i]              ; x gets the ith element of A[]
(3)   p^ = 2;               ; i (the thing pointed to by p) gets set to 2
(4)   y = A[i]              ; y gets the ith (2nd) element of A[]
```

Should an optimizing compiler replace line 4's **A[i]** with **x** ? No, because line 4's **A[i]** may or may not be the same as line 2's **A[i]**, since line 3 has assigned a value to **i** that may've changed that variable.

Aliasing makes code optimization much tougher, thus inhibiting its usage. A language like C, with its weakly typed pointer variables, presents a real challenge to compiler designers trying to generate hot code.

Back To Nosy, And Another Type Of Data Flow Problem

Now back to Nosy. Another data flow problem involves determining a variable's type based on its usage. Nosy must solve this in order to automatically recognize procedural parameters. It can then propagate the information back to the calling procedure(s), and thus be able to recognize more code blocks.

Solving this data flow problem means Nosy has to dynamically simulate the 68000 registers and stack as it disassembles instructions. That'll let it determine the usage of variables (and thus their types) in a basic block. Then, if it keeps enough control flow information around, it can propagate the type information out of the procedure and back to the caller.

I've been working on this problem for a while now, but diversions keep appearing. Soon, soon …

Block Hunting - The Simplicity Of Direct Jumps

Entry points to procedures are signalled by a reference to that address from somewhere else within the program. In terms of the 68000, the address of a called procedure has to follow one of the variants of a Jump or Jump Subroutine instruction. If Nosy finds an instruction of the form JSR aaaa, for example, it knows that the word lying at address aaaa is the start of a code procedure and it flags it as such. Similarly, address references following BSR, JMP, BRA, and Bcc instructions also indicate entry points into code.

(Actually, it's not *quite* that simple. Nosy does a fair bit of control-flow analysis to try and refine its understanding of program structure.)

Addresses that get jumped to via a JSR or BSR instruction are labelled as procedures, or **proc**s. Addresses that get jumped to via JMP, BRA, or BCC instructions are generally labelled as common, or **com** blocks. And, until Nosy gets smarter, everything else gets set as one or another type of data block.

Block Hunting - The Complexity Of Indirect Jumps

There's a complication to the above scheme: procedures are often called by a variant of the JSR (Ai) instruction, where the jump takes place indiriectly via an address register. This happens a lot in the Mac world, where programs often require that procedures be passed as parameters.

There's also a second flavor of jumping indirection. Compilers usually set up a *local stack frame* for called procedures, via the LNK and UNLK instructions. Such procedures typically end with a JMP (Ai) instruction, but this is just a jump back to the procedure's caller.

For Nosy, the problem with the first flavor of indirect jumps is that the address is determined and loaded into the address register at *runtime*, not when the program is compiled. Consequently, Nosy has no way of determining what that address is, and the bytes at that address may not get flagged as belonging to a code block. This problem requires advanced flow analysis, not currently present in Nosy. So Nosy might flag the block as data, when it's really code. Another case where your help (via the **Review** menu) is required.

Case Statements And Jump Tables

Case statements, which are usually implemented by jump tables, are another complication in Nosy's life. The problem lies in determining the length of the jump table. When a jump table takes the form of offsets to code blocks, and has at least one positive entry, Nosy can use that entry to determine the length of the table. It does this by assuming that code for the cases immediately follows the jump table. But, if you've got a table of offsets, and all the entries are negative, Nosy can't be sure where the table ends. When you look in the **–Case Jumps–** window, you'll be able to recognize these cases by nonsensical information in the **n_branch** or **tbl_addr** columns. See **Linking Jumps To Tables** for more detail.

What Happens During A Treewalk

To close up this short technical discussion, here's a brief outline of the steps Nosy takes during a treewalk :

1 Build tables of referenced symbols

globals signalled by *displacement* (A5)

procedure blocks	signalled by JSR/BSR
common blocks	signalled by JMP
data blocks	what's left

2 Collect reference information (for globals, procs, com_'s, data's, trap calls & System symbols)
3 Identify case jump blocks
4 Assign data block lengths
5 Sort by block table by addresses
6 Auto format data blocks
7 Search libraries and name routines

Alternate Sources Of Truth

It's hard to get too much information when you're a Mac programmer. Here are some of my favorite sources :

Networking

Respected and/or disreputable members of the Mac programming world, as well as folks at Apple who don't hang up the phone on me. Foster your own connections, using modems, mail, conventions, pilgrimage, coincidence.

APDA

Now run by Apple, the Apple Programmer's and Developers Association (**APDA**) is a one-stop shop for programming tools from Apple and third-party vendors. Join it. Phone: (800) 282-APDA

Apple Computer, Inc.

See if you can become an Apple Partner. You get all kinza goodies : tech notes, support via electronic mail, humorous mailings, marketing nostrums, and a touch of credibility. Phone: (408) 996-1010

Magazines

MacTech - The one and only magazine for serious Mac programmers. A good selection of articles on the finer points of the Mac interface, done up in a variety of programming languages. Good gossip, interesting ads for the major programming tools. The max in Mac hacks. Phone: (310) 575-4343

MacWeek - This is an excellent weekly Mac Magazine slanted at Business and workstation users.

MacUser - General interest Mac magazine with good columnists and a strong technical flavor. Subscription Phone: (800) MAC-USER

MacWorld - General interest Mac magazine with good graphics and content that's becoming substantive. Subscription Phone: (800) 524-3200

Byte - The grandaddy flagship of the computer revolution. Professionally put together, with mostly–excellent technical articles interspersed between pounds of ads. Build a strong bookshelf. Subscription Phone: (800) 258-5485

Also: the June, 1986 issue of IEEE Spectrum has a great article on copy protection, "How Disks Are Padlocked"

Weeklies/Tabloids

InfoWorld - Slanted towards a target audience of DP managers. It's a bit less lively since Dvorak left, but still a must read for news of the PC industry. Subscription Phone: (215) 768-0388

EE Times - I prefer this to EW (Electronic Weekly), as the content is a bit more technical. The publisher, Frank Burge, is a refugee from Regis McKenna, and for some strange reason they do a good job of covering Apple. Qualified/imaginative readers can obtain a free subscription. Phone: (516) 356-4600

Computer Currents, Microtimes - These two cover PCs and Macs in the Bay area. Both make interesting reading, with regular appearances by nationally–known columnists. Computer Currents Phone: (415) 547-6800 Microtimes Phone: (415) 652-3810

User Groups

These "false religions" are a welcome relief from the solitary act of attempting to communicate with one's computer. They give you a chance to obtain the latest shareware, gossip, info about the latest products, etc. The largest national groups are BMUG (Berkeley MUG) and BCS (Boston Computer Society). BMUG puts out a large (200 page+) "newsletter" semi-annually. If you're in the Bay area, BMUG holds its general meeting every Thursday at 6 PM in PSL 100 on the Berkeley campus. BMUG Phone: (510) ???

Classic Jasik On Bookstores And Books

A quick trip to the local technical bookstore can be an overwhelming experience. Entering, I feel like the ladies with the license plate frames "So Many Men, So Little Time". My normal reaction is to browse for 15

minutes, keep my wallet in my pocket, and try not to bring home any more dust collectors. I live in a dangerous area, 5 minutes from Stanford University, 15 minutes from Silicon Valley. In addition to the Stanford Bookstore and the local B. Dalton's, the area is blessed with Stacey's, a pre-eminent technical bookstore, and Computer Literacy, which has an exhausting selection.

Here are some guidelines for book buying : First, follow intelligent recommendations (book review, friend, me, etc). Next, consider books from the old line publishing houses : Addison-Wesley, Scott Foresman, Wiley, Prentice-Hall, Van Nostrand, Springer-Verlag, et al. These companies have stronger editorial staffs, attract the best authors, etc. Two exceptions to this rule are Microsoft Press and Hayden's Apple Press, both of which have produced a number of excellent books. Stacey's : (415) 326-0681
Computer Literacy : (408) 730-9955

Books - Mac Specific

Inside Macintosh (Addison-Wesley) - The Bible, you can't live without it. Get **all** available volumes.

How to Write Macintosh Software, by Scott Knaster (Hayden-Apple) - The former head of Apple tech support tells us like it is. Mandatory reading.

Macintosh Revealed, Vols I & II by Steven Chernicoff (Hayden-Apple) - More readable than Inside Mac, with good graphics, but doesn't hit all the nitty–gritty details.

Books - 68000 Specific

Programming the 68000, by Rosenzweig & Harrison (Addison-Wesley) - A good intro to 68000 assembly language and the Macintosh. Has a nice spreadsheet programming example.

68000 Assembly Language Programming, by Kane, Leventhal & Hawkins,2nd Edition (Osborne/McGraw Hill) - A very good introduction to the 68000 and assembly language programming.

M68000 16/32-Bit Microprocessor Programmer's Reference Manual, 4th Edition, by Motorola (Prentice-Hall)

MC68020 32-Bit Microprocessor User Manual, 2nd Edition, by Motorola (Prentice–Hall) - **Required**.

Also #1: The theme of the Sept, 1986 issue of Byte is the 68000 family. Mike Morton's article contains an interesting catalog of tricks and bad excuses for architectural asymmetries in the 68000 family.

Also #2 - Most of the above books were discussed on pages 9-12 of the May, 1986 issue of MacTutor.

Books - Computer Science

The Art Of Computer Programming, volumes I-III, by Donald Knuth (Adison–Wesley). The densely–packed source for a large percentage of all other computer books and articles. Get these, then write Don a note and tell him to finish up the series already.

Algorithms, by Sedgewick (Addison-Wesley) - A good general reference. An alternative to Knuth's books.

The Design and Analysis of Computer Algorithms, by Aho, Hopcroft & Ullman (Addison-Wesley) - A good graduate level computer science book.

Compilers - Principles, Techniques & Tools, by Aho, Sethi & Ullman (Addison-Wesley) - An excellent introduction to compilers and associated processors. University level again.

Fundamentals of Interactive Computer Graphics, by Foley & van Dam (Addison-Wesley) - This or the book by Newell should be part of everyone's library.

Applied Concepts in Microcomputer Graphics, by Bruce Artwick (Prentice Hall) - The programmer of "Flight Simulator" tells it like it is. Contains a good discussion of hardware.

C - A Reference Manual, by Harbison & Steele (Prentice Hall) - The current bible for C programmers. It supercedes Kernigan & Ritche's white book. Thorough, rigorous, clear.

Object Oriented Programming, by Brad Cox (Addison-Wesley) - A clear introduction to the whys of object oriented programming.

Object Oriented Programming for the Macintosh, by Kurt Schmucker (Hayden-Apple) - The title says it.

The Psychology Of Computer Programming, by Gerald Weinberg (Van Nostrand Reinhold) - A late '60s classic. Looks at computer programming as a human activity. Read this book.

Interactive Programming Environments, edited by Barstow, Shrobe, and Sandewall (McGraw-Hill) - Selected papers from the last 20 years of industrial-academic thought about programming spaces. Some of the history that led to the Mac.

Programming Pearls, by Jon Bentley - Jon's been writing a popular column of the same name for a few years in Communications of the ACM. A wide-ranging collection of intelligent techniques.

The C Companion, by Allen Holub (Prentice–Hall) - Background material for, and advanced treatment of, a number of vital C programming topics. Excellent intermediate text by the popular Dr. Dobbs' C

columnist.

Inside Jasik

On Copy Protection

It's like a dog pissing on a tree. It leaves a mark. Later on the dog'll come back and use its nose to find the tree. It doesn't matter what the mark smells like. The techniques a copy protector uses don't matter. Laser holes, spiraling tracks, all that crap -- it doesn't matter. When the program runs, the software will be living in memory, and it'll talk to the disk. That's the Achilles heel. The software has to get into the computer, and it comes from the disk. We know that. Just like we know the dog uses its nose.

On Programming

I started programming computers in 1966. At that time, computers were expensive, and programmers relatively cheap. Today the economics are reversed. Given this, and a very competitive micro market, it is important that we find ways to maximize our productivity. While there is a certain macho thing about writing tight assembly code, the loss in productivity far outweighs the improved efficiency of the resulting code. Another disadvantage of assembly code is that is harder to modify. These factors combine to make it a primitive tool that should only be used when absolutely necessary.

On C Versus Pascal and Programming Environments

The C language has many features which make it attractive to programmers. One is able to obtain fairly efficient code with primitive compilers, and its operators nicely match the 2–address computers which are in vogue today. On the negative side, the weak typing in the language make it difficult for compiler writers to produce optimizing compilers for it. I find the style of most C programs a bit on the terse side for my taste. Pascal, which started as a teaching (small) language is a bit easier to optimize, and in many cases a bit easier to read. Neither language is ideal.

The real problems in programming are not what language you use, but the programming environment which surrounds it. This includes the editor, compiler, linker, and runtime support (debugger, etc). For those of us building programs of any size, the lack of real tools in the form of a database manager to help us manage the world (name space) we have created with our program becomes apparent when we try to modify it. Think C and Pascal are a nice start in the direction of integrated environments.

On Assemblers

On a multi-register machine like the 680x0 series the primary uses of assemblers are to generate tables that cannot be clearly or conveniently generated by compilers, hardware drivers that are time critical, and other applications that require interface to the machine at a low level. I say multi-register machine, because it is difficult to write a set of macros that will do code generation, and keep track of the register usage. This function requires the intelligence of a compiler.

The MPW assembler is vastly more powerful then the MDS assembler. Unfortunately it is not as powerful as it could be. In particular, the macro facility is harder to use then it should be, lacks the ability of macros to define other macros, and is saddled with some silly limits because of the designed decision to write an "IBM" style assembler with arrays of symbols.

MDS Versus MPW Versus Other Systems

MPW supercedes MDS. The only reason to use MDS as a development system is that you don't have a hard disk and a meg or more of memory. While MPW is head and shoulders above MDS, it is still a far cry from the ease of use in the Lightspeed products.

How Nosy Came To Be

My curiosity about what scan conversion algorithm Atkinson was using to draw circles, and the Mac ROM in general, were the origional reasons for starting what became MacNosy. The program is written in MPW Pascal (neé Lisa Pascal) with some critical parts in assembly language. The source text for it , "The Debugger" and some auxiliary programs take over a meg of disk storage. There used to be a version that ran under the Lisa workshop, but that was put to rest when I switched over to MPW in July 86. The program, or output from it was first shown at the Hacker's Conference in Nov 84.

On Management

Managers are usually a pain in the butt to us programmers. I mean like they make us account for our time so that we may justify our salary, and theirs, which is spent accumulating statistics about us (I'm going to have a bumper sticker made up that says "Real Hackers do it for love not money"). I had one period where I was on a tight schedule, and had a mild case of coder's cramp. My manager at that time was not very understanding of my predicament, and came around on a daily basis with a rectal thermometer that was calibrated in lines of code per day. It was a rough period for me! There I was trying to put a proper ending on the great American Fortran optimizer for CDC, and all **he** cared about was lines of code! The only interesting things I learned from him were the acronyms WAG and SWAG (Scientific Wild Ass Guess). After burning out on managing programmers he tried to sell swimming pools for a living in Fresno.

Planning is a royal pain, but management likes it. It gives them the feeling that they think that they know what you are doing. When CDC started, one of the VP's approached Seymour Cray and asked if he might submit 1 and 5 year plans. He reminded Seymour that everyone else had, etc. The next day the VP found two binders on his desk. Seymour's 5 year plan was one sentence: "to build the world's fastest computers". His one year plan was also short and sweet. It said,"accomplish 1/5 of the 5 year plan"!

On Memory Bandwidth (comments that were true in 1984)

I find it a bit of a joke that most of today's personal computers "give away" about 50 percent of their memory bandwidth to the video display. The problem will disappear in tomorrow's computers as the price of video RAM comes down. In 1968, when I was writing the operator's console display driver for the 6600, I got involved in a battle royal over 1 percent of the memory bandwidth of the 6600. The full bandwidth of the 6600 was 600 million bits per second. The display driver ran in a small computer with 6K bytes of memory called a PPU (Peripheral Processing Unit) and had no display memory of its own, so it had to generate the displays and repaint the screens at least 30 times a second to avoid flicker. The users (I sided with them) wanted an expandable display driver that they could easily customize. One of the more popular displays was the job log message display, which, if put in the processor's memory, would not have enough space to display an adequate number of lines. I and "the Bear" (who wrote the previous version, and was a proponent of placing the display in the processor's memory) battled it out at 90 Db in a formal design meeting. While "the Bear" weighed a bit more than I did, I carried the day, and eventually my version saw the light of day.