

ResEdit CODE Editor (@CODE RSSC)

(Ver 2.9)

General Description

The file “CODE editor for ResEdit 2.1” contains a ResEdit 2.1 (or later) RSSC resource that adds a disassembly viewer to ResEdit to handle CODE and code-like resources. This allows you to view code resources as assembly code instead of “raw” hex. When added to ResEdit version 2.1.1 or later, the CODE editor can also operate concurrently with the basic HEXA editor, providing editing and searching capabilities as well. The disassembly code is annotated with the module names extracted from the MacsBug strings in the code. Navigation facilities are provided to allow viewing related pieces of code. Special formatting may be applied to the disassembly for embedded data that cannot normally be distinguished from actual code. There are also facilities to allow you to answer the question “who references this location?”

A typical CODE editor window has the following appearance:

| RSSC “@CODE” ID = 2500 from CODE editor for ResEdit 2.1 | | | | | | |
|---|--------|---------|----------------------------------|--------------------|-----------|---|
| Offset | Addr | Opcode | Operand | Comment | Hex | |
| +0084 | 0034AA | DC.B | \$80+\$0A, 'autoScroll', \$00 | | 8A61 7574 | ↑ |
| +0090 | 0034B6 | DC.W | \$0000 | ; size of literals | 0000 | |
| extendSelection | | | | | | |
| +0000 | 0034B8 | LINK | A6, #0000 | | 4E56 0000 | |
| +0004 | 0034BC | MOVEM.L | D6/D7, -(A7) | | 48E7 0300 | |
| +0008 | 0034C0 | MOVE.L | \$0008(A6), D6 | | 2C2E 0008 | |
| +000C | 0034C4 | MOVE.L | \$0014(A6), D7 | | 2E2E 0014 | |
| +0010 | 0034C8 | CMP.L | D6, D7 | | BE86 | |
| +0012 | 0034CA | BLE.S | extendSelection+\$001E; 000034D6 | | 6F0A | |
| +0014 | 0034CC | MOVEA.L | \$0010(A6), A0 | | 206E 0010 | |
| +0018 | 0034D0 | MOVE.L | \$0018(A6), (A0) | | 20AE 0018 | |
| +001C | 0034D4 | BRA.S | extendSelection+\$0028; 000034E0 | | 600A | |
| +001E | 0034D6 | CMP.L | D6, D7 | | BE86 | |
| +0020 | 0034D8 | BGE.S | extendSelection+\$0028; 000034E0 | | 6C06 | |
| +0022 | 0034DA | MOVEA.L | \$000C(A6), A0 | | 206E 000C | |
| +0026 | 0034DE | MOVE.L | D7, (A0) | | 2087 | |
| +0028 | 0034E0 | MOVEM.L | -\$0008(A6), D6/D7 | | 4CEE 00C0 | |
| +002E | 0034E6 | UNLK | A6 | | 4E5E | |
| +0030 | 0034E8 | RTS | | | 4E75 | |
| +0032 | 0034EA | DC.B | \$80+\$0F, 'extendSelection' | | 8F65 7874 | |
| +0042 | 0034FA | DC.W | \$0000 | ; size of literals | 0000 | |
| clickPosition | | | | | | |
| +0000 | 0034FC | LINK | A6, #FFFFE | | 4E56 FFFE | |
| +0004 | 003500 | MOVEM.L | D5-D7/A3/A4, -(A7) | | 48E7 0718 | |
| +0008 | 003504 | MOVEA.L | \$0014(A6), A3 | | 266E 0014 | |
| +000C | 003508 | MOVE.L | \$000C(A6), -(A7) | | 2F2E 000C | |
| +0010 | 00350C | MOVE.L | \$0008(A6), -(A7) | | 2F2E 0008 | ↓ |
| autoScroll | | | | | | |

The window shows the disassembled lines and the associated hex, resource addresses and module offsets. Module names are shown in bold. A module with no name (i.e., no MacsBug symbol) is shown as AnonN, where N is an integer.

In addition to the basic display the CODE editor provides some additional navigational facilities. The “status panel” in the lower left of the window generally shows the module name containing the first line in the window. *The status panel can be clicked to position to the first line of the window.*

The left margin in the window is reserved for arrows. Whenever the mouse passes over a PC-relative address (or a Jump Table reference to an address in the same resource) the arrow appears showing to where that PC-relative address points. The above example window illustrates this.

The PC-relative addresses can be clicked to “go to” the referenced address. In the CODE editor, a clickable item is signaled by a gray-lined box around it. When a PC-relative address is clicked a “marker” is displayed at the destination. The marker is a pair of diamonds around an address. *The marker (either diamond) can be clicked to “go back” to the referencing instruction.*

Hex selections are made by selecting the hex in the conventional way. A hex selection is used to set the selection in the HEXA editor and *to copy lines to the clipboard or for printing*. For copying and printing, entire lines containing the selection are copied. Copied lines can be pasted into a text editor.

In addition to the PC-relative arrow, there are also a persistent arrow which shows a “references”. This arrow comes up from the “Goodies/Refs to...” menu described later. The arrow will appear whenever a PC-relative arrow is not being displayed. Associated with such references are clickable offset and address “buttons”.

Finally, there are trap “buttons” which allow you to find the next same trap.

All of these are described in detail later.

CODE-Type Resources

If a ResEdit picker does not have a specific editor to handle a resource, it opens @HEXA as the default. This is a basic character/hex editor that allows you to find and change a resource as characters or in hex. The @CODE editor, because of the way it is named, overrides the default editor for resources of type 'CODE'. In addition, ResEdit RMAP resources are supplied to change the default to use the CODE editor for the following code-type resources:

| | | | |
|------|------|------|------|
| ADBS | FKEY | PACK | scod |
| CACH | FMTR | PATC | segg |
| cdec | INIT | PDEF | SERD |
| CDEF | it12 | play | snth |
| cdev | it14 | PTCH | WDEF |
| clck | LDEF | proc | XCMD |
| dcmp | MBDF | ptch | XFCN |
| dh1r | MDEF | RQvr | |
| DRVR | mh1r | RSSC | |

I won't pretend to understand what all these are! If you have other code-type resources you can easily add RMAP's for them. More about this later in the section entitled “Adding New Code-like Resources”.

Resource types CODE and scod are handled fully. The CODE editor knows how to handle Jump

Table segment 0 and segment loader headers for both 16-bit and 32-bit segments. Other code resources are treated as “pure” code where segment 0 has no special meaning.

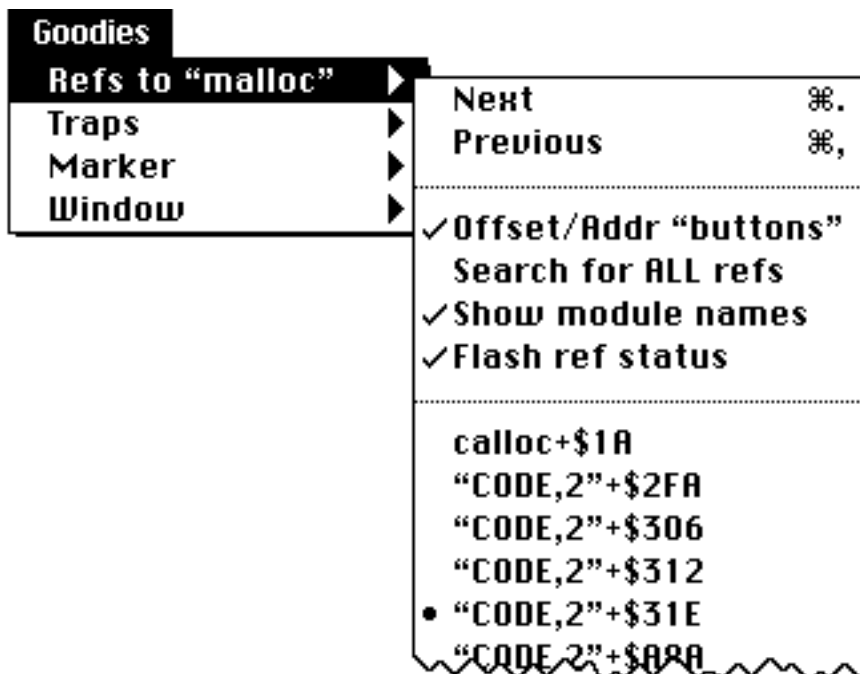
The Menus

When the CODE editor is opened, additional menus are added to the menu bar. The following sections describe those menus, changes to some of the standard ResEdit menus, and how you use the CODE editor. The menus are presented in a logical order to aid in describing the CODE editor and not the order they appear in the menu bar.

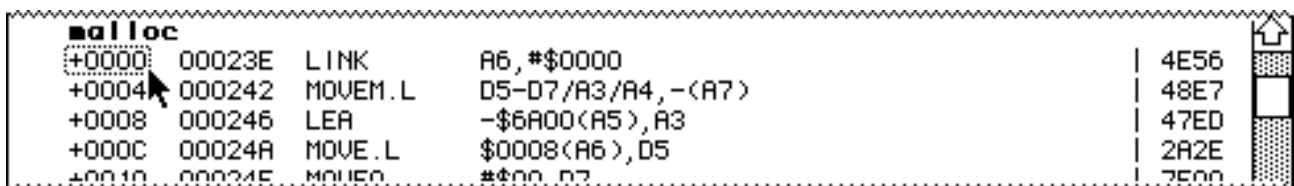


The Goodies menu is a hierarchical menu that provides additional functionality (i.e., “goodies”) for viewing a CODE resource. The following functions are provided:

- The “Refs to...” menu allows you to answer the question “who who references this location?”
- The “Traps” menu allows you to look at the traps in a CODE resource.
- The “Marker” menu allows you to manipulate the “marker”.
- The “Window” menu allows you to set the preferences to remember window size and position.



The “Refs to...” submenu is used to find references to a particular location. You can limit the search to just the currently displayed resource or search the *entire* file for inter-segment references. You specify a location by clicking an offset or address for an instruction shown at the left side of the window. An offset or address is indicated as “clickable” with a gray-lined box around it as the mouse passes over it. For example, for the above menu, the address for “malloc” was clicked when the mouse was pointed at its address as illustrated below.



The “Refs to...” menu allows you to specify some preference items as follows:

- Offset/Addr “buttons” This allows you to disable the “Refs to...” clicking facility. You may not need it and you may find all the flashing boxes disturbing. If this item is checked then the facility is enabled. If unchecked then it is disabled.
- Search for ALL refs Normally the search for references is limited to just the currently displayed resource. If this item is checked *all* resources of the same type in the file are searched. Since this may take a “little time”, a dialog is displayed illustrating the status of what’s going on. Using this dialog you may kill the search at any time. What has been found up to that point will be shown in the menu.

If you hold the OPTION key down when you click an address

you temporarily reverse the meaning of this menu item.

- Show module names You have the choice of displaying the references in the menu as module name plus displacement in the module or just as a resource offset.
- Flash refs status The results of the search are flashed for four seconds in the status panel in the lower left of the window. If any references are found the status panel is flashed with “Goodies menu for refs!” to remind you to look in the Goodies/Refs to... menu to see the references found. If none are found, “No references!” is displayed along with a “beep” . You may turn the status result if you don’t like the flashing with this preference item. The beep can be used to signal the “no references” condition.

The menu illustrated above shows the results of searching for references to “malloc”. In this case, since the “Search for ALL refs” item is not checked, the OPTION key was used to temporarily reverse the setting to search for *all* references in all segments.

Local references are shown as module+\$offset (or just resource address if “Show module names” was not checked). The example menu shows just one (calloc+\$1A). External references from other segments are shown as “CODE,n”+\$offset.

You may select references from the “Refs to...” reference list!

If you select a local reference, a persistent arrow is displayed going from the reference to its referenced address. The window is positioned to the referencing instruction. The arrow looks exactly like the PC-relative arrow. Indeed intra-segment references are usually PC-relative references, so you would see the same arrow if you now placed the mouse over the PC-relative operand. The difference is that the arrow will *not* go away when you move the mouse off the PC-relative operand (it is also a different color if you are using a color monitor).

PC-relative arrows always have precedence over the “refs to...” arrow. So as you move the mouse around, the “refs to...” arrow will flash off and then back on as you pass the mouse over PC-relative address operands or comments.

You can “kill” the arrow by clicking in a nonactive (i.e., not clickable and not in the hex) area of the window. You can always get it back by reselecting the “Refs to...” menu reference item .

If you want to see one of the external references in the “Refs to...” list, you can go back and select the resource from ResEdit’s CODE resource picker window (the window with the list of CODE resources you used to get to the CODE editor in the first place). You must then position the new window to the offset shown the menu item. That’s the *long* way!

The *short* way is to just select the external reference item from the “Refs to...” list. The CODE editor will perform the picker functions, i.e., load the referencing segment and position the window to the reference for you. Either way, this will only happen if the partition size for ResEdit is large enough. In other words, this operation is memory limited (you probably should increase the ResEdit partition size anyway).

The following illustrates selecting the “CODE,2”+\$31E reference from the above example menu.

| CODE “STDCLIB” ID = 6 from xxxxxx | | | | | | |
|-----------------------------------|--------|---------|--------------------------|------------|------|--|
| Offset | Addr | Opcode | Operand | Comment | Hex | |
| malloc | | | | | | |
| →+0000 | 00023E | LINK | A6, #0000 | | 4E56 | |
| +0004 | 000242 | MOVEM.L | D5-D7/A3/A4, -(A7) | | 48E7 | |
| +0008 | 000246 | LEA | -\$6A00(A5), A3 | | 47ED | |
| +000C | 00024A | MOVE.L | \$0008(A6), D5 | | 2A2E | |
| +0010 | 00024E | MOVEQ | #00, D7 | | 7E00 | |
| +0012 | 000250 | CMPI.L | #00800000, D5 | | 0C85 | |
| +0018 | 000256 | BLS.S | malloc+\$0020 | ; 0000025E | 6306 | |
| malloc | | | | | | |
| CODE “Init” ID = 2 from xxxxxx | | | | | | |
| Offset | Addr | Opcode | Operand | Comment | Hex | |
| +0034 | 000312 | JSR | “CODE, 6”+\$023E | | 4EAD | |
| +0038 | 000316 | MOVE.L | D0, -\$04C2(A5) | | 2B40 | |
| +003C | 00031A | PEA | \$1389 | | 4878 | |
| ←+0040 | 00031E | JSR | “CODE, 6”+\$023E | | 4EAD | |
| +0044 | 000322 | MOVE.L | D0, -\$04AA(A5) | | 2B40 | |
| +0048 | 000326 | MOVE.L | -\$04B2(A5), -\$04A6(A5) | | 2B6D | |
| +004E | 00032C | TST.L | -\$04AA(A5) | | 4AAD | |
| initMacro | | | | | | |

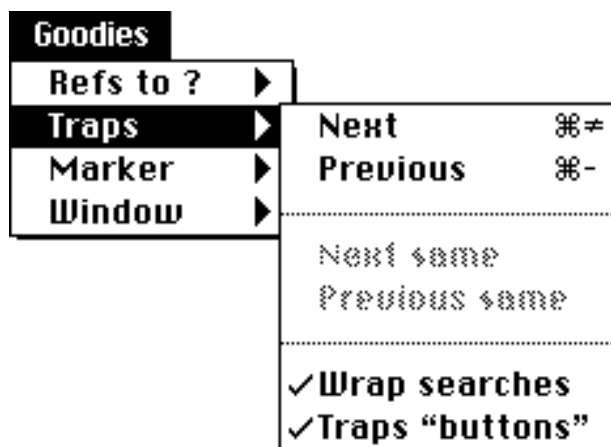
After the item is selected “CODE,2” is loaded as if selected by ResEdit’s CODE picker and the window positioned to offset \$31E. A horizontal reference (i.e., persistent) arrow is displayed pointing left out of the window at the offset and the current selection made an insert point at the offset in the hex (selections are describe in a later section). At the same time the referenced address (in this example, “malloc”) gets a horizontal reference arrow pointing “in”, i.e., pointing right coming into the window.

In order to remember which item you selected from the “Refs to...” list, a bullet (“•”) is placed next to the item. You can select the items in any order and the bullet will be placed next to the most recent item. Before you select any items there is no bullet.

The “Next” and “Previous” items allow you to move up and down the list sequentially (the list wraps and beeps when a wrap occurs). The command keys were chosen as “,” and “.” which are the unshifted “<” and “>” keys.

Caution! If you have a lot of external references from a lot of segments you can “fill up” ResEdit very quickly by moving up and down the list!

Have said all these neat things about the reference facility, users with “slow” machines are cautioned that this facility can be *very* time consuming. The CODE editor only remembers *one* reference at a time. It must search on each new reference.

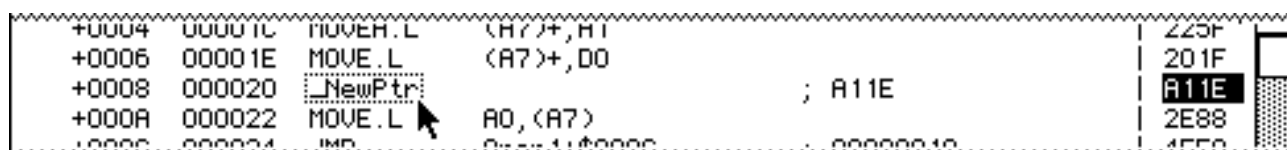


Sometimes when analyzing a code resource you can get an idea of what is going on by the kinds of traps being performed. The Goodies/Traps submenu aids in this analysis by allowing you to find the next (COMMAND-OPTION “=”) or previous (COMMAND-OPTION “-”) trap relative to the current selection (selections are discussed in a later section). When the next trap is found it is made the current selection. Searches will wrap around from bottom to top (next) or top to bottom (previous) if the “Wrap searches” item is checked in the Traps menu.

Both the next and previous trap menu items are disabled if there are no traps in the code resource. Just by looking at the submenu you can tell whether there are any traps in the resource.

The most recent trap found using Next or Previous is remembered. By using the “Next same” or “Previous same” the next or previous instance of the same trap can be searched for. The “same” in the menu item is replaced with the trap instruction word that was remembered. Note that “same” here means the same trap ignoring any special bits in the trap instruction (e.g., immed, async, sys, clear).

The CODE editor provides an alternate way to search for the next or previous same trap. That is by treating the traps instructions and comments in the disassembly display as active “buttons” in much the same way offsets and addresses are treated as described earlier for the “Refs to...” submenu. When the mouse is placed over a trap opcode or its trap number comment, a gray-lined box will be placed around it as shown below.

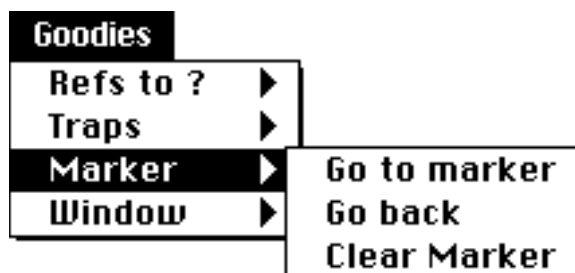


Clicking the box will have the same effect as “Next same”. *If the OPTION key is pushed while clicking, a “Previous same” is tried.*

You may or may not find all the flashing of boxes around things disturbing. If you are not interested in the traps you might prefer to not have them as potentially active buttons. Thus this functionality is controlled by the “Traps buttons” menu item in the Traps menu.

The “Wrap searches” controls all trap searches. If wrap is off, the CODE editor will beep if there is no next or previous trap. If wrapping is on, the code editor will beep if the search wraps.

Both the wrap and trap button settings are saved as part of the CODE editor's preference items.



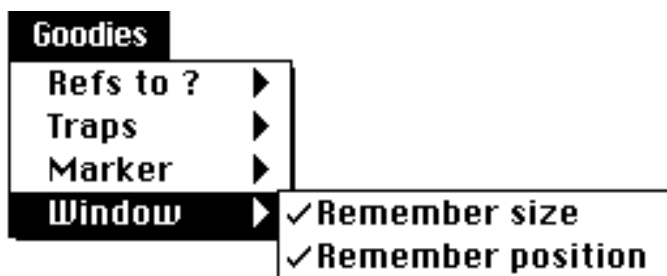
The Marker menu is alternate way from the mouse of manipulating the “marker”. The marker is the small pair of diamonds shown in the window around a resource address. The marker results from clicking a PC-relative operand or comment. An address or comment is indicated as active by showing a gray-lined box around the address or comment pointed at by the mouse, and an arrow, in the left margin, extending from the line pointed to by mouse, to the line containing the address referenced. Off-window arrows (and references) extend to the top or bottom of the window in the appropriate direction. The target module offset and resource address are also enclosed in rectangles. See the window pictured earlier for an example.

You may treat the gray-lined box as a button. Clicking it will let you “go to” the specified destination address. The destination address is indicated when you “get there” by the marker.

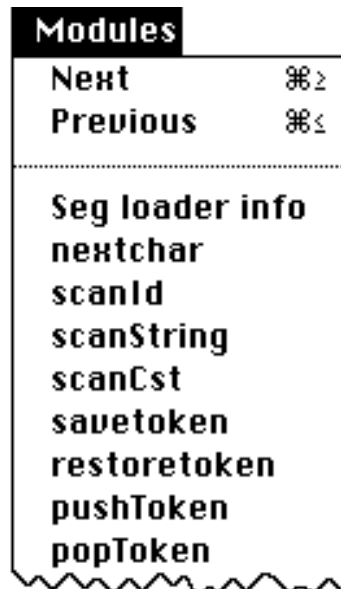
The marker itself is also a button. By clicking either one, you can go back to (display in the window) the line you most recently clicked that got you there. In this case the process is reversed and original “go-to” line gets the marker. At this point you could click the markers to “ping-pong” back and forth!

The Marker menu duplicates the mouse operations. They would generally be used when the marker has been scrolled out of the window. “Go to marker” moves the display to show the marker. “Go back” is the same as clicking the marker. “Clear marker” is special, in that it makes the marker “invisible”. Clicking it (if you can find it) will make it visible again.

Note, you can use the marker facility in conjunction with the references arrow describe earlier to “ping-pong” back and forth on a local reference arrow.



The window menu sets the CODE editor preferences to remember the size and/or position of the current window. If both are checked, then opening another CODE editor window will place that window in exactly the same place and make it the same size as the previous window. By toggling these menu items you can independently control these actions. If you intend to open more than one window then it is usually preferable to remember size but not the position.



The modules menu lists the names of all the modules found in the resource. By selecting a module name in the Modules menu, you change the window display to show that module. By selecting Next (COMMAND-OPTION “>”) or Previous (COMMAND-OPTION “<”) you can position the display to the next or previous module relative to the module containing the top line in the window (the one displayed in the status panel).

The names are determined by the MacsBug symbols usually placed at the end of each module by the language compilers (e.g., MPW C and Pascal). If the CODE editor detects what it thinks is the end of a module, but cannot find a MacsBug symbol, the name “Anon*N*” is used (which stands for “anonymous”), where *N* is a unique integer number.

For segments of type CODE or scod, some additional special names are placed in the Modules menu. They have the following meaning:

- Seg loader info The segment header “module” at the start of CODE or scod segment.
- A5-rel Reloc Info “Module” containing the 32-bit A5-relocation information.
- Seg-rel Reloc Info “Module” containing the 32-bit segment relative relocation information.

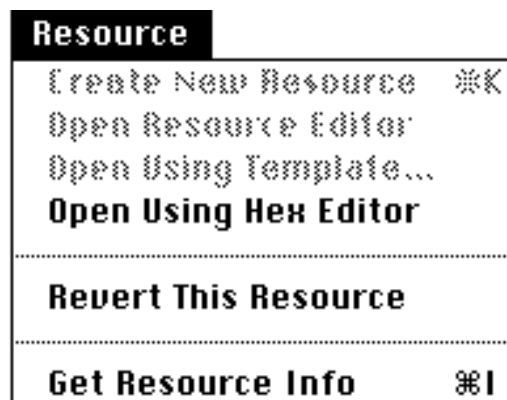
- Unknown data This usually results when data is encountered at the end of a code segment.
- No name Special name used internally. If you see this the CODE editor is “confused”!

The last two Modules menu entries are special cases. “Unknown data” usually results from C data declaration initializations. The more generic “Unknown data” is used because the CODE editor is not really sure! All it knows is that what it has detected as the last module of the code segment is NOT at the end of that segment. Stuff follows it to the end of the resource. That stuff is the “Unknown data”.

The “No name” case is the name the CODE editor gives a module before it knows where the module ends. If the resource is sufficiently confusing (it's not clear how) you could see this name. You probably shouldn't!

It should be pointed out here that the CODE editor is most reliable in detecting module boundaries when MacsBug symbols *and all the rules that go with them* are present in the code segment. The CODE editor looks for an RTS, RTD, or JMP (A0) as the end-of-module instruction. Following that there should be a MacsBug symbol and a constant data area size word (possibly with the value 0) immediately following the MacsBug symbol. That size tells the CODE editor how much to skip to the start of the next module and to more appropriately display the constants.

One final point about modules; as illustrated in the window pictured earlier, the status panel to the left of the horizontal scroll bar shows the module name of the module containing the top line in the window. As the window is scrolled, the module name in the panel will change appropriately. The panel may be clicked like a button. It has the same effect as selecting that module name from the Modules menu. This is convenient for quickly moving to the top of a module.



The Resource menu is the standard menu supplied by ResEdit. Note that the “Open Using Hex Edit” item is enabled. If you are using ResEdit 2.1.1 or later, then by selecting that item, you can open the standard @HEXA character/hex editor window *at the same time* the CODE editor window is open! The two editors can operate on the same resources concurrently. “Concurrently” in this case means that both operate on the same resource, but only one at a time -- whichever one is active.

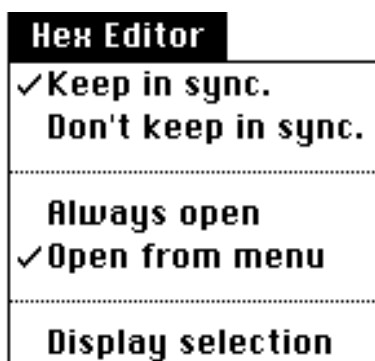
If you use an earlier version of ResEdit then attempting to use the HEXA editor will result in a

dialog and the “Open Using Hex Edit” item will be disabled.

Only one of the windows, CODE editor or HEXA editor is active at any one time. You click the window as usual to activate it. The two editors are distinct, but can communicate with each other. The CODE editor can tell what the HEXA editor's selection is and vice versa. The selection of the inactive editor is NOT updated until you make it active. On the other hand, if you revert the resource, the inactive editor's display is updated immediately.

No more will be said at this point regarding editing of the resource. It is discussed in more detail later. For now, keep in mind that you *can* edit the resource in the HEXA editor. The changes will be known to the CODE editor when you reactivate it. Indeed all the facilities (and warts) in the HEXA editor are available. So finds to change the selection will be reflected in the CODE editor when it is reactivated.

Note, the Hex Editor menu, described next, allows you to specify some preference items on how the CODE editor will interact with the HEXA editor.



The Hex Editor menu tells the CODE editor some preference items on how to interact with the HEXA editor. Except for the last item, these are indeed preferences, and are remembered (along with the CODE editor's window position and size) in the standard ResEdit Preferences file.

In using the HEXA editor with the CODE editor, there are a number of alternative ways you can interact with it. You can open the HEXA editor “manually” using the Resource menu's “Open Using Hex Editor” item described above. Alternatively, you may elect to have the CODE editor open the HEXA editor for you at the time you open the CODE editor. The choice is made with the “Always open” and “Open from menu” Hex Editor menu items.

“Always open” will do the “auto-open” of the HEXA editor at the time the CODE editor is opened. If the HEXA editor is not currently open, it will be opened just as if the “Open Using Hex Edit” was selected from the Resource menu. If it is already opened, it will be made the active window. Note, selecting “Open Using Hex Edit” will also make the HEXA editor active if it is already up. This functionality is the same as using ResEdit's Window menu.

If you choose “Open from menu”, then the HEXA editor must be manually opened using the “Open Using Hex Edit” Resource menu item.

Once the HEXA editor is opened, you have two choices on how to handle the synchronization of

selections between the two editors. A “selection” in the CODE editor refers to a range of the hex bytes displayed at the right of each disassembly line. This is illustrated in the window pictured earlier. In that picture the selection is the inverted bytes at the right.

By choosing “Keep in sync.”, the CODE editor selection can be made to synchronize with the selection in the HEXA editor and vice versa. In that case whenever you switch from one editor to the other, the current selection will be made that of the other editor. The CODE editor will scroll the selection into view when reactivated whenever the selection has been changed.

Alternatively, you can choose “Don't keep in sync.”. In that case the CODE editor's selection is completely independent of the HEXA editor's selection.

Finally, you may have a selection in the CODE editor that has been scrolled off the screen. By choosing “Display selection” you can cause the window to scroll to it. Unlike the other items, this makes no sense to remember as a preference item.

| Edit | |
|---------------------------------------|-----------|
| Undo formatting | ⌘Z |
| Cut | ⌘X |
| Copy | ⌘C |
| Paste | ⌘V |
| Clear | |
| Duplicate | ⌘D |
| Select All | ⌘A |
| Select Changed | |
| Format as Code | |
| Format as Bytes (1/line) | ⌘1 |
| Format as Bytes (up to 8/line) | ⌘B |
| Format as Words | ⌘2 |
| Format as Longs | ⌘4 |
| Format as End of line (CR) | ⌘N |
| Remove all added formatting | |

The CODE editor knows the format of CODE and scod resources and it knows how to detect what it thinks is the end of a module. And that's about it! It does not know about special headers. Nor does it know about embedded data. These will “confuse” the disassembly display. They are treated like code and will usually produce some “valid” MC68XXX instruction according to the bits seen. Even CODE segments generated by most compilers produce areas of embedded data. The classic example is a word offset table for C switch and Pascal CASE statements. The Edit menu tries to address this problem by allowing you to “manually” format sections of the

disassembly display.

The “Format as...” items in the Edit menu take the current selection and reformat it according to the item selected. There are certain rules and restrictions associated with this reformatting as follows:

(1). What can be formatted...

Formatting may only be applied to code or an entire “Unknown data” module. It cannot be applied to a Jump Table or to a CODE module's segment header, or anything from the MacsBug symbol to the module that follows it (i.e., the stuff at the end of a module). The intent is to allow you to reformat only code (for those who care, the implementation is much more complex without these restriction).

(2). Which “Format as...” items can be applied...

“Format as...” items which cannot be applied are disabled (grayed out). However, items that are not disabled are NOT guaranteed to be allowed either! What an enabled item indicates is that the formatting *may* be possible for that item according to the rules we will discuss shortly. Until the actual formatting is attempted, the CODE editor does not know the selection is legal according to rule (1). Reformatting requires full reanalysis of the resource. That is when rule (1) is applied. The cost for this is unacceptable for a large resource just to enable and disable menu items.

The “Format as...” items are enabled as a function of alignment and what is selected. Thus “Format as Code” and “Format as Words” are allowed for selections which start on a word boundary and contain an even number of bytes. “Format as Longs” must start on a word boundary and contain a multiple of four bytes. “Format as Bytes...” has no such restrictions, and will always be enabled unless the current selection is just an insert point. For an insert point, only the “Format as End of line (CR)” (disabled in the above picture) will be enabled. The formats for all these items is discussed next.

(3). How the disassembly is reformatted by the “Format as...” items...

Format as Code This is generally the original format for the selection. You would use this to reverse the effects of another formatting item applied to the selection. You could also use this to reformat a “Unknown data” module as code. According to rule (1) above, it's all or nothing for an “Unknown data” module. But you can fool the CODE editor by first formatting the module as code. It will cease to be “Unknown data” (it will become an unnamed module -- “AnonN”). After that you can use the other items as appropriate.

Format as Bytes (1/line) Each byte in the selection is reformatted as “DC.B \$XX”.

Format as Bytes (up to 8/line) This is similar to “Format as Bytes (1/line)”, but with 8 bytes per disassembly line instead of 1. If there are less than 8 bytes in the selection, or it is not a multiple of 8, the only or last line will show the remaining bytes.

Format as Words Each pair of bytes in the selection is reformatted as “DC.W \$XXXX”.

Format as Longs Each four bytes in the selection is reformatted as “DC.L \$XXXXXXXX”.

Format as End of line (CR) This is a special formatting case that is allowed only when the selection is an insert point. Functionally it is equivalent to selecting all the hex bytes on the line to the left of the insertion point and doing a “Format as Bytes (up to 8/line)”. This places those bytes on a separate line. The effect is to place a Return at the insertion point. A shorthand for the menu item (other than the COMMAND-r key) is to *simply press the Return key*. The “(CR)” in the menu item is a reminder that you can do this. You can easily split up a sequence of bytes this way to reformat them on separate lines.

(4). What happens to the bytes on the line before and after the selection....

Formatting is applied to what was originally considered as “legal” code by the CODE editor. A selection that doesn't completely select all the bytes for an instruction is effectively producing an invalid instruction on the unselected bytes of that instruction line. This can happen at the start and/or end of the selection. In these cases the bytes before and after the selection could be formatted as “Format as Bytes (up to 8/line)”.

Up to three reformats can happen as a result of a single reformat operation. The stuff on the line before the selection. The selection itself. And the stuff on the line following the end of the selection. Of course, if the selection starts on an instruction boundary there is nothing to its left. Similarly, if the selection ends with the last byte of an instruction there is nothing to the right. But if the selection ends before that last instruction byte, the stuff to the right is only reformatted if the selection ends on an odd byte boundary. If it ends on an word boundary it is NOT reformatted.

The reasoning behind the treatment of the end of the selection is to allow you to separate data from “real” code that follows it. The code is always assumed to start on word boundaries. The data in front of an instruction could have been disassembled so that the real instruction got “sucked up” by a data word in front that was originally treated as an instruction. By reformatting the data, the actual instruction becomes visible. This is where the Return comes in handy to “Format as End of line (CR)”. Put the insert point at the end of the data and just hit Return!

Those are the rules. You really don't have to memorize them. Just try a reformat and see what happens. If it's wrong, you can always “Undo...”.

If you decide you no longer want to see the reformatting you can choose the “Remove all added formatting” menu item. All the reformatting information accumulated up to that point is thrown away, and the display reverts to its original format. Note, that all the reformatting information is also thrown away when you close the window. It is *not* saved!

Use this reformatting sparingly! For one thing, you will not be able to save it as I just said. For another, each reformat is treated as pseudo edit operation, and as such, initiates a *full reanalysis* of

the resource. Modules and their boundaries may have changed. Further, the formatting information must be scanned for each line of the disassembly. The more formatting you do, the more scanning that must be applied. All of this can be a time consuming processes on large modules when using lower performance machines! I do not recommend you use the CODE editor on such machines. As with using references described earlier, consider yourself warned!

Copying to the Clipboard

Note that the “Copy” item is enabled in the Edit menu shown above. If you want a copy of the selection you can use the “Copy” menu item. *All* the lines, from the line containing the start of the selection, to the line containing the end of the selection are *copied as TEXT to the Clipboard*. A module name in front of a selection is copied when the first byte of a module is selected. Once in the Clipboard, you can paste the lines into another text editor to manipulate them.

Copying to the Clipboard is limited to about half the memory available to the CODE editor at the time you do the operation. This is because the “Undo copy” operation is supported. If there is not enough memory for the full copy, you will get an alert. Paste what was copied into whatever you are pasting it into. You can then see what was missed and copy the rest from there.

Printing

The “Print...” command in the File menu is supported. Printing is much like copying. That is *all* the lines, from the line containing the start of the selection, to the line containing the end of the selection are printed. A module name in front of a selection is printed when the first byte of a module is selected.

When the selection is just an insert point or the entire resource (use “Select All” or COMMAND-a from the Edit menu) then the entire resource will be printed unless limited to specific pages in the standard print dialog.

Note, the page range in the print dialog is the *only* situation in which the page range in the print dialog is used. When a byte range is selected to just print the indicated lines, the page range is ignored. Corresponding to this, the “Print...” menu item in the File menu becomes “Print selection...” when the selection is not an insert point or the entire file.

Inter-segment (Jump Table) References

If the disassembly shows operands of the form “CODE,n”+\$XXXX then that is a reference to another code segment (resource) that is accessed through the Jump Table (code segment 0). This is the same format used to display inter-segment references in the “Goodies/Refs to...” submenu discussed earlier. The CODE editor resolves the Jump Table reference for you and shows you the destination code resource (segment n) and offset (\$XXXX). In a program with a lot of segments you may have to look at these resources with the CODE editor. You can go back to ResEdit's CODE picker and select the resource and open it with another instance of the CODE editor. After the resource is loaded you have to scroll to the specified offset. If you do this often enough it can be a real pain!

This is the similar problem as using “Goodies/Refs to...” inter-segment reference items. However, the situation here is the opposite of “Goodies/Refs to...”. There you have the “destination” and want to know who “calls” it. Here you have the “caller” and want to see the destination. But in either case you are selecting a “CODE,n”+\$XXXX address.

In this case there is no menu item list. Instead the CODE editor treats inter-segment operand references just like other active “buttons” (i.e., like offset/addresses, traps, and PC-relative operands/comments). They are drawn with a gray-lined box as the mouse passes over them. This is illustrated as follows:

| | | | | |
|-------|--------|--------|------------------|------|
| +0018 | 00001E | SUBQ.L | *\$1,00 | 3380 |
| +001C | 000020 | MOVE.L | D0,-\$0DC6(A5) | 2B40 |
| +0020 | 000024 | JSR | "CODE,26"+\$231C | 4EAD |
| +0024 | 000028 | MOVEQ | #\$09,D1 | 7209 |
| +0026 | 00002A | CMP.L | D0,D1 | 8780 |

Clicking one of these buttons causes the specified resource (segment) to be loaded (memory permitting) just like the case when a “Goodies/Refs to...” inter-segment reference item is selected. The window is positioned to the “called” address. The resulting display is the same as illustrated for the “malloc” example earlier. Horizontal arrows are drawn just as in the reference case. Indeed they are treated just like references so the arrow becomes persistent and requires a click in a non-active part of the window to clear it.

Editing, Selecting, and Positioning

No editing can be performed in the CODE editor other than the reformatting described above. “Editor” has been used rather loosely up to this point since it is the term, like “picker”, that ResEdit uses for such things. More precisely, the CODE editor is actually a viewer. The actual editing is done only from the HEXA editor. Refer to the appropriate ResEdit documentation for further details on how to use the HEXA editor.

What the CODE editor provides to the HEXA editor is basically another way of viewing the bytes. As such, only the hex bytes in the CODE editor may be selected in much the same manner as done in the HEXA editor and the HEXA editor can use that selection.

In the CODE editor, selections are made in the usual way. The basic arrow key selection operations, in combination with command and shift keys, are supported. OPTION-arrow keys are used only for scrolling. The extended keyboard home, end, page up, and page down keys are also supported. None of these keys are supported in the HEXA editor!

As discussed earlier, the CODE editor does support the Copy command from the Edit menu to copy entire lines from the window to the Clipboard. All the lines from the start to the end of the hex byte selection are copied to the Clipboard.

When editing is done in the HEXA editor, the changes are made in the CODE editor window when that window is reactivated. At this point the CODE editor must completely reanalyze the resource to see what's happened. As stated earlier for the reformatting Edit menu operations and finding

references, this can be a time consuming processes on large modules when using lower performance machines! Consider yourself warned...again!

Installing the CODE Editor Into ResEdit

The CODE editor is only compatible with ResEdit versions 2.1 or later. The protocols for ResEdit editors earlier than 2.1 are different and will NOT work with this editor. The CODE editor is supplied in the file “CODE editor for ResEdit 2.1”.

If you install the CODE editor into version 2.1.1 or later, you will be able to open the HEXA editor from the CODE editor to search or make changes to the resource. You will not be able to open the HEXA editor from the CODE editor in version 2.1.

“CODE editor for ResEdit 2.1” contains the CODE editor and all its associated resources. There are two ways of installing these resources into ResEdit.

The first method involves adding the CODE editor resources to ResEdit itself. Using ResEdit, copy all the resources in “CODE editor for ResEdit 2.1” *except* the ‘ilrP’ resource into ANOTHER copy of ResEdit. The ‘ilrP’ resource should be copied to the “ResEdit Preferences” file in the System Folder. However, if you don’t copy it, the first use of the CODE editor will create one there with its default settings. ResEdit is a program with many large resources so window updates may be relatively slow. Have patience!

The second, easier, and *recommended* method has the benefit of *not* modifying ResEdit. Instead you use ResEdit to copy the CODE editor resources (all of them) into the “ResEdit Preferences” file in the System Folder. You can even do this on the actual preference file while ResEdit is running. You do not need to copy it.

In System 7 “ResEdit Preferences” will be in the Preferences folder in the System Folder. Using this technique means that as newer versions of ResEdit are released they will immediately “get” the CODE editor with no additional work.

Note that since the resources supplied for CODE editor version 2.2 and later include a preference resource, ‘ilrP’, your current preferences for older versions of the CODE editor will be lost. But by supplying the resource, or allowing the CODE editor to create it, newer versions of the CODE editor (?) will always be compatible with its preferences. Preferences are discussed further in the next section.

If you do have older versions of the CODE editor, then the original installation instructions were to place the CODE editor resources into ResEdit itself. Although you wouldn’t get “hurt” by installing newer CODE editors over the older, or even switching to using the preferences folder, ***it is highly recommended you start with a “clean” copy of ResEdit 2.1.1.***

It is also recommended you ***increase the partition size for ResEdit.*** With the new inter-segment capabilities of version 2.2 it is very easy to “choke” ResEdit by loading in lots of resources at the same time.

The CODE Editor Preferences

The “ResEdit Preferences” in the System Folder contains all the preferences for ResEdit including the ones for the CODE editor. The CODE editor preferences are contained in the ilrP,25000 resource in the “ResEdit Preferences” file. The following information is saved in the CODE editor preferences:

- HEXA editor “sync” settings from the “Hex Editor” menu.
- HEXA “auto-open” setting from the “Hex Editor” menu.
- Trap wrap search status from the “Goodies/Traps” submenu.
- Traps “buttons” setting from the “Goodies/Traps” submenu.
- Offset/address “buttons” setting from the “Goodies/Refs to...” submenu.
- References display (address or with module name) from the “Goodies/Refs to...” submenu.
- Flash status from “Goodies/Refs to...” submenu.
- Search for all references from the “Goodies/Refs to...” submenu.
- The window size and position.

The CODE editor provides a TMPL for viewing these preferences directly (?). These preference items in “ResEdit Preferences” are updated whenever any of the mode values change. They are also written out whenever a CODE editor is deactivated. They are reloaded (except for window size and position) whenever a CODE editor is reactivated. The implication here is that, if there are multiple CODE editors windows, changing a preference item will reflect in another CODE editor when it is reactivated. The down side of this is that new CODE editors position their windows directly over the old ones.

If anything should go wrong with the CODE editor preferences (?) you can just delete them from the “ResEdit Preferences” file. All that will happen is that the CODE editor will recreate its ilrP,25000 resource with default settings.

Adding New CODE-like Resources

It's very easy to make the CODE editor the default editor for new code-like resources. ResEdit supports a special RMAP resource to tell ResEdit which editor to use for a particular resource. It maps that resource on to the designated editor. The RMAP's are added to ResEdit as you would add any other resource. There are already examples in ResEdit to use as a guide. ResEdit provides a TMPL to make editing easy. The ones for the CODE editor are all numbered from 25000.

Possible Future Enhancements

This section lists some possible future enhancements for the CODE editor. Some of these are more basic than others. The reason that some of these haven't been done is TIME. There isn't any! This editor was started as a simple one-week project to be done over Apple's Christmas break. The original intent was a simple disassembly viewer. Like most code projects, things got out of hand! But enough is enough. I am making no promises that any of these enhancements will be done. Or if they're done, when. So here they are (in no particular order):

1. More intelligent reanalysis. As discussed in the sections on formatting and editing, if you make

any changes to the resource in the HEXA editor, or do any reformatting, the entire resource is reanalyzed. This can be slow for a larger resource on slower machines. It works. It's just slow. One thing that could be done is to only analyze from the point of the change or reformat. This still could be slow when editing the “top” of a resource.

You may question what this “reanalysis” is all about. The way the CODE editor handles the disassembly and the Modules menu is to essentially pre-paginate the entire display. It knows where lines begin as a function of resource offsets down to a resolution of about 50 lines. It does this by actually rehearsing the entire display. It would be great if we could keep this display. But the space requirements would be potentially enormous. Also, I did not want to use some sort of auxiliary disk file. That too would potentially require a lot of disk space. So the only space required is that for data structures and a buffer containing only the lines shown in the window. As you scroll around, the new lines in the window are being generated “on the fly”.

When you edit or reformat the resource, the entire “repagination” must be redone. It's relatively fast. But not that fast. Any improvement here would make editing more acceptable on slower machines. Fortunately, most code resources are not that big. On a Mac IIfx none of this seems to be a problem! But I don't recommend editing this way on a Mac SE or Classic!

2. A template mechanism to describe the layout of special code resources. Some code resources have special formats. For example, cdev's have a header block. Even a CODE resource has a segment loader header. The CODE editor knows about these. It's built-in. But it would be nice if there were some way to generalize this concept to be able to tell the CODE editor about special formats for other code-like resources like DA's. This would reduce, but not eliminate, the need for the formatting menu options.
3. Built-in editor as part of the CODE editor. All the selection mechanisms are in place in support of the HEXA editor. The reason the HEXA editor is used at all is, given the time constraints, there was too much “bang for the buck” to pass it up. There it was. Off the shelf, so to speak. Very little had to be done to it. Indeed, very little could be done to it since it is the default editor used by the rest of ResEdit.
4. Related to 3, find and position functions for the Goodies menu.^{1†}

Putting in a built-in editor is not as simple as it might appear (and no, I don't use TextEdit). It is not clear what the paradigm should be. Currently the editing is modal. You switch to the HEXA editor. Make the changes. And then switch back to the CODE editor where the display is updated. If there is a built-in editor, when is the display updated? What happens if you edit more bytes than is shown for the line you are editing? You get the idea!

That's all the enhancements I can think of right now. I'm sure when you start using this thing you will think of a lot more. Then again, maybe not.

Oh, one more point. **No, I am not going to put an assembler in!** So don't even ask! Although, now that I think of it, it would solve the editing paradigm problem (more-or-less)!

¹ † The menu was named “Goodies” and made hierarchical to allow future functionality. The menu bar could not be made wider with new items due to the limitations of 9” screens and Human Interface Guidelines. Hence the only direction left was to go “down”!

Author

Ira L. Ruben
Principal Engineer
Apple Computer, Inc.
20525 Mariani Ave., MS: 37-A
Cupertino, Ca. 95014

Internet: ira@apple.com
AppleLink: Ruben1
Compuserve: 76666,1032