THE CODE EDITOR

INTRODUCTION

This chapter explains the operation of Resorcerer's Code Editor, which is an integrated extension to its Hex Editor. The Code Editor lets you disassemble code resources containing Motorola 68000-family machine instructions. The Editor can disassemble all 68040 instructions, including PMMU and FPU co-processor instructions.

This chapter only covers the extensions the Code Editor brings to the Hex Editor, so you should also be familiar with the operation of the Hex Editor, whose operation is documented in the "Hex Editor" chapter later in this volume. Also, this chapter does not attempt to document 68040 assembly language. The disassembler's output is based on Motorola's documentation.

If you are not already familiar with general resource editing, see the "Editing Resources" chapter earlier in the manual.

TOPICS COVERED

- Creating new code resources
- Opening a code resource
- Using the Code Editor
- Making patches

CREATING A NEW CODE RESOURCE

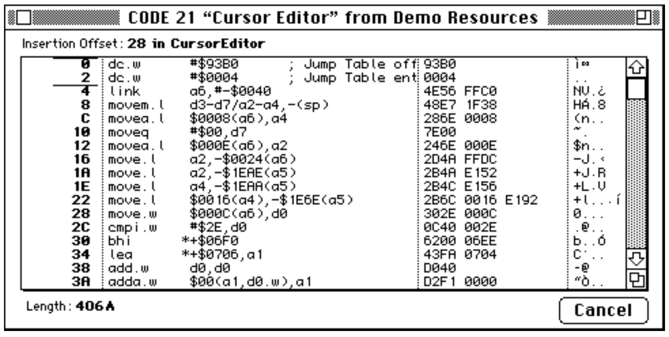
Code resources are typically created by the compiler or assembler in an application development environment. The Code Editor does not contain an assembler, although it can make a few of the most common patches for you.

OPENING A CODE RESOURCE

Standard 68000 application code is compiled and stored in resources of type 'CODE'. There are also a variety of other standalone code resource types commonly used in applications, such as Menu Definition ('MDEF') Procedures, Control Definition ('CDEF') Procedures, and List Definition ('LDEF') Procedures. Resorcerer is shipped with the knowledge of about 50 resource types that contain system or application code. These are all declared as synonyms of 'CODE' resources so that the Code Editor will open them automatically for you (for more on declaring synonyms, see the "Synonym Preferences" section of the "Preferences" chapter).

DISASSEMBLING THE CODE

The Code Editor extends the Hex Editor by adding an extra code viewing area to the Hex Editor's standard offset, hex, and text display areas. When the Editor opens the code resource, the disassembler parses the entire resource into individual machine instructions, each of which ranges in length from 1 to 11 16-bit words. The Hex Editor then inserts a paragraph break at the start of each instruction. Unlike the Hex Editor, the Code Editor does not mark the paragraph boundaries in the offset area, since the code area makes these obvious. The code area then disassembles the starts of each paragraph (line) on the fly as you display different parts of the hex data.


CODE 21 "Cursor Editor" from Demo Resources

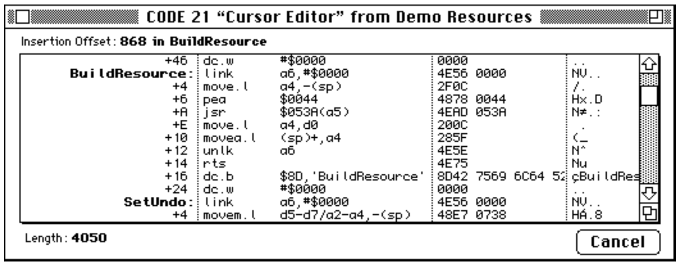
Insertion Offset: **28 in CursorEditor**

```
   0  dc.w      #$93B0       ; Jump Table off 93B0            ì∞
   2  dc.w      #$0004       ; Jump Table ent 0004            ..
   4  link      a6,#-$0040                    4E56 FFC0       NV.¿
   8  movem.l   d3-d7/a2-a4,-(sp)             48E7 1F38       HÁ.8
   C  movea.l   $0008(a6),a4                  286E 0008       (n..
  10  moveq     #$00,d7                       7E00            ~.
  12  movea.l   $000E(a6),a2                  246E 000E       $n..
  16  move.l    a2,-$0024(a6)                 2D4A FFDC       -J.‹
  1A  move.l    a2,-$1EAE(a5)                 2B4A E152       +J.R
  1E  move.l    a4,-$1EAA(a5)                 2B4C E156       +L.V
  22  move.l    $0016(a4),-$1E6E(a5)          2B6C 0016 E192  +l...í
  28  move.w    $000C(a6),d0                  302E 000C       0...
  2C  cmpi.w    #$2E,d0                       0C40 002E       .@..
  30  bhi      *+$06F0                        6200 06EE       b..Ó
  34  lea      *+$0706,a1                     43FA 0704       C˙..
  38  add.w     d0,d0                         D040            –@
  3A  adda.w    $00(a1,d0.w),a1               D2F1 0000       "ò..
```

Length: **406A**

Note: The Hex/Code Editor is a 32-bit Editor, and requires a 10-byte overhead per instruction (paragraph). Thus, a 32K code resource with an average of 3 bytes/instruction will require about 100K available memory to open, plus the resource data, a possible backup copy, and another 80K or so for the Editor itself. All of which adds up to an easy 250K. To open multiple or much larger code resources, you will likely have to increase Resorcerer's partition size.

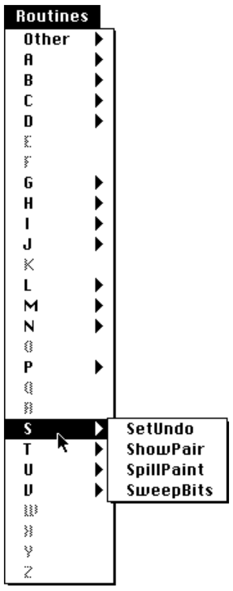
If an instruction appears illegal, the disassembler assumes it is actually inline data, and breaks it up into the appropriate number of word data declarations. It then continues to disassemble at the next instruction.


CODE 21 "Cursor Editor" from Demo Resources

Insertion Offset: **868 in BuildResource**

```
          +46 dc.w      #$0000              0000                ..
BuildResource: link      a6,#$0000           4E56 0000           NV..
           +4 move.l    a4,-(sp)            2F0C                /.
           +6 pea       $0044               4878 0044           Hx.D
           +A jsr       $053A(a5)           4EAD 053A           N≠.:
           +E move.l    a4,d0               200C                 .
          +10 movea.l   (sp)+,a4            285F                (_
          +12 unlk      a6                  4E5E                N^
          +14 rts                           4E75                Nu
          +16 dc.b      $8D,'BuildResource' 8D42 7569 6C64 52   çBuildRes
          +24 dc.w      #$0000              0000                ..
     SetUndo: link      a6,#$0000           4E56 0000           NV..
           +4 movem.l   d5-d7/a2-a4,-(sp)   48E7 0738           HÁ.8
```

Length: **4050**



In addition to breaking the data up into instructions, the Code Editor analyzes the code when it opens it, looking for likely procedure boundaries so that it can parse any MacsBug name strings and extra compiler data that might be appended to routines. The Editor marks the starts of the routines with their names, and the offset area displays them (in red) on the left. If a routine has no name, the disassembler assigns it a default name, "<Anon-xx>", where each "xx" is a unique number.

Sorcery: You can configure the Editor to avoid using brackets ('<' and '>') for unnamed routines. Search for "<anon" in Resorcerer's own string lists and delete the initial '<'.

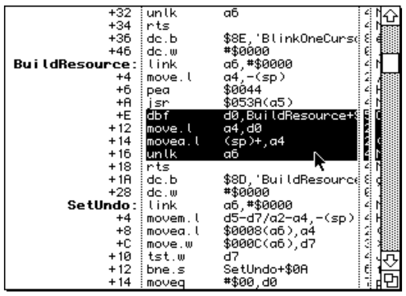
Resorcerer sorts the names of procedures in the code alphabetically, and installs them in the **Routines** menu's 27 submenus. It places any non-alphabeticly named routines in the **Other** submenu. You can navigate (scroll to) any named routine simply by choosing its name from its menu.

USING THE CODE EDITOR

As an extension of the Hex Editor, most of the features of the Hex Editor are also available in the Code Editor. You can select bytes in either the hex or text data areas, enter data, search for and replace hex strings, etc. The rest of this chapter explains the features the Code Editor adds to the Hex Editor.

VIEWING OPTIONS

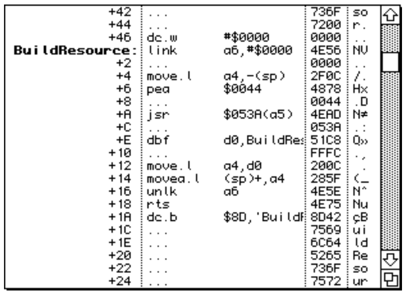
To adjust the widths of the individual display areas, click on a vertical divider line and drag it right or left. If you have long MacsBug names, you may want to slide the first divider (the one between the offset area and the code area) to the right. In the illustrations here, we've done this as well as dragged the second and third divider lines all the way to the right.



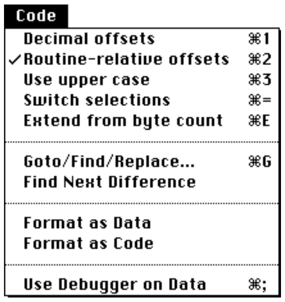
```
             +32  unlk      a6
             +34  rts
             +36  dc.b      $8E,'BlinkOneCurs
             +46  dc.w      #$0000
BuildResource:  link      a6,#$0000
              +4  move.l    a4,-(sp)
              +6  pea       $0044
              +A  jsr       $053A(a5)
              +E  dbf       d0,BuildResource+$
             +12  move.l    a4,d0
             +14  movea.l   (sp)+,a4
             +16  unlk      a6
             +18  rts
             +1A  dc.b      $8D,'BuildResource
             +28  dc.w      #$0000
      SetUndo:  link      a6,#$0000
              +4  movem.l   d5-d7/a2-a4,-(sp)
              +8  movea.l   $0008(a6),a4
              +C  move.w    $000C(a6),d7
             +10  tst.w     d7
             +12  bne.s     SetUndo+$0A
             +14  moveq     #$00,d0
```

The window's Grow and Zoom Boxes enlarge all display areas proportionally.

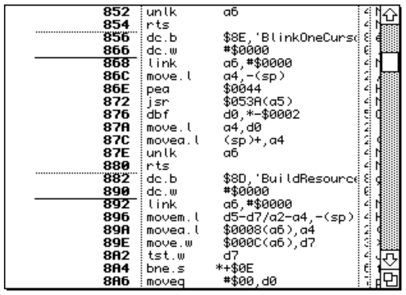
When working on small displays, there may be times when you want to ensure that you can always see all the hex data. You can do this by double-clicking on either divider line next to the hex data. This turns on the hex word wrap feature. If the hex data of an instruction wraps, subsequent lines in the code area show continuation dots until the next instruction starts.



```
             +42  ...                      736F  so
             +44  ...                      7200  r.
             +46  dc.w      #$0000         0000  ..
BuildResource:  link      a6,#$0000      4E56  NV
              +2  ...                      0000  ..
              +4  move.l    a4,-(sp)       2F0C  /.
              +6  pea       $0044          4878  Hx
              +8  ...                      0044  .D
              +A  jsr       $053A(a5)      4EAD  N≠
              +C  ...                      053A  .:
              +E  dbf       d0,BuildRes    51C8  Q»
             +10  ...                      FFFC  .,
             +12  move.l    a4,d0          200C   .
             +14  movea.l   (sp)+,a4       285F  (_
             +16  unlk      a6             4E5E  N^
             +18  rts                      4E75  Nu
             +1A  dc.b      $8D,'BuildR    8D42  çB
             +1C  ...                      7569  ui
             +1E  ...                      6C64  ld
             +20  ...                      5265  Re
             +22  ...                      736F  so
             +24  ...                      7572  ur
```



The **Decimal Offsets** command in the **Code** menu tells the disassembler and the offset display area to produce base-10 numerical output. In this mode, the code area displays 3-digit numbers for bytes, 5-digit numbers for words, and 10-digit numbers for longs.



```
852  unlk      a6
854  rts
856  dc.b      $8E,'BlinkOneCurs
866  dc.w      #$0000
868  link      a6,#$0000
86C  move.l    a4,-(sp)
86E  pea       $0044
872  jsr       $053A(a5)
876  dbf       d0,*-$0002
87A  move.l    a4,d0
87C  movea.l   (sp)+,a4
87E  unlk      a6
880  rts
882  dc.b      $8D,'BuildResource
890  dc.w      #$0000
892  link      a6,#$0000
896  movem.l   d5-d7/a2-a4,-(sp)
89A  movea.l   $0008(a6),a4
89E  move.w    $000C(a6),d7
8A2  tst.w     d7
8A4  bne.s    *+$0E
8A6  moveq     #$00,d0
```

The **Line-break relative offsets** menu item in the Hex Editor is renamed **Routine-relative offsets** in the Code Editor. With this option normally on, the disassembler generates numerical offsets that are relative to the start of each routine. With this option turned off, disassembled offsets are with respect to the start of the data, although the offset area still marks the starts of routines with a horizontal line above the starting offset. In addition, lines mark the starts of the actual MacsBug name data at the ends of routines. Regardless of this option's setting, the Editor displays the current insertion offset or selection bounds just above the data display area in resource-relative offsets.

It is sometimes more useful to view instruction formats in binary than hex (for instance, Motorola's documentation is in binary). You can use Resorcerer's **Value Converter** (in the **Edit** menu) to view and edit any selected 8-, 16-, or 32-bit hex value in binary or a variety of other common formats (for more information on this, see the "Value Converter" chapter later in the manual).

MAKING SELECTIONS

To select one instruction, click on its line in the code area, or double click within its hex or text data.

To select a sequence of instructions, click on the first one, and drag the mouse, or Shift-click to extend the selection. For the purposes of cutting and pasting, you can also click just to the left of any instruction to place a code insertion caret at the start of the instruction without selecting it.

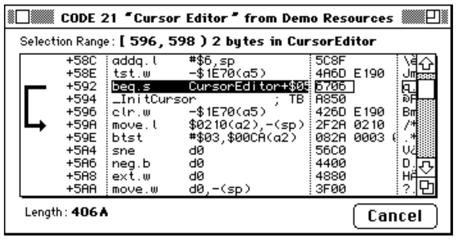
To select an entire subroutine, double-click on any instruction in it.

Note: You cannot select portions of an instruction's text disassembly, nor can you enter assembly instructions.

Above the upper left corner of the scrollable area, the Editor displays information about the current selection. If the selection is empty, then it shows you the position, as an offset from the beginning of the resource, of the blinking insertion caret, as well as what routine it is in. If the selection range is non-empty, you can see the range's endpoints (again, as offsets from the start of the resource data) followed by the range length and the routine the selection starts in.

To the right of the data is a scroll bar that lets you position your view to any word offset in the data. As you drag the scroll bar's thumb, the Code Editor immediately updates the offset, and routine containing that offset, to give you a better idea of where you're going to scroll to.

To change the active selection highlight from the code area to the hex area or to the character area, use **Switch Selection** in the **Code** menu.



```
Selection Range: [ 596, 598 ) 2 bytes in CursorEditor
  +58C  addq.l     #$6,sp                5C8F       \
  +58E  tst.w      -$1E70(a5)            4A6D E190  Jm
  +592  beq.s      CursorEditor+$05      6706       g.
  +594  _InitCursor           ; TB       A850       
  +596  clr.w      -$1E70(a5)            426D E190  Bm
  +59A  move.l     $0210(a2),-(sp)       2F2A 0210  /*
  +59E  btst       #$03,$00CA(a2)        082A 0003  .*
  +5A4  sne        d0                    56C0       V
  +5A6  neg.b      d0                    4400       D.
  +5A8  ext.w      d0                    4880       H
  +5AA  move.w     d0,-(sp)              3F00       ?.
Length: 406A
```

If the first instruction of the selection contains a PC-relative branch offset, the Editor displays a (red) arrow on the left side of the offset area, which extends either forward or backward (depending on the sign of the offset) to the destination instruction. You can instantly scroll to one end or the other of the arrow by clicking anywhere on it. The arrow adheres to the branch instruction until you select another branch.

REFORMATTING

There may be parts of the data that look to the disassembler like code, but which are in fact in-line non-executable data. You can override the disassembler's initial decision as to whether a line of data is an instruction or data using the **Format as Code** and **Format as Data** commands in the **Code** menu.

Sorcery: Many standalone code resources (for instance, those compiled with THINK C) begin with an unconditional branch instruction around a data header area. When the disassembler sees the first instruction as an absolute branch, it declares the following data as just data instead of disassembling it. It begins disassembling again when it reaches the destination of the branch.

CUTTING, COPYING, AND PASTING

When the active selection highlight is in the text area, copying or cutting places only the selected bytes in the clipboard. When the active selection highlight is in the hex area, the hex digits of the data area are placed in the clipboard.

When the active selection is in the code area, the Editor puts both the instruction data and their text disassembly into the clipboard. You can cut and paste series of instructions this way, although in general this is very dangerous and you should not attempt it unless you *really* know what you are doing. In particular, you cannot easily change the length of any section of code over which a branch or jump statement extends.

SETTING THE TEXT STYLE

The **Set Text Style…** command in the **Resource** menu lets you view and edit your code data using any installed font on the Mac. This is useful for larger displays, where a larger fixed width font is easier on the eyes; or for resources with international string data in them. Both the hex and the character areas of the display draw single characters at a time, so if you choose a variable width font, the spacing may be uneven.

If you click on **Save & OK**, the type style you choose is recorded in Resorcerer's preferences file, so that every time you use the Code Editor it displays with your favorite font. If you click on **OK**, then the change is only temporary.

USING DEBUGGER ON DATA

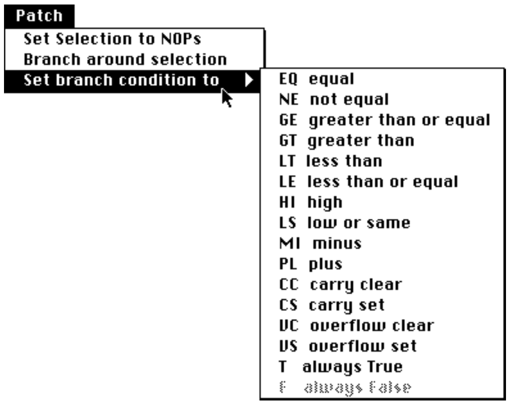
If you have a debugger installed in your Mac, such as MacsBug, you can use it to disassemble the code data, in case Resorcerer's Code Editor is doing something different.

Note: Your low-level debugger's disassembler generally assumes that the code it is disassembling is part of the currently running application, in this case, Resorcerer. Consequently, it may provide incorrect disassembly, particularly with regards to jump table references.

MAKING PATCHES

One of the most common uses of a Code Editor is to perform surgery on an existing application you've built in order to fix (or bypass) a bug that is causing grief to your user. In Resorcerer's own case, we have been able to fix about a third of the reported bugs in earlier versions of Resorcerer by providing simple patches for specific 'CODE' resources.

The Code Editor's **Patch** menu assists you in making the three most common length-preserving patches used to fix certain types of bugs. In the first two cases, you should select the instructions that you want to avoid executing, and either change them all to NOP instructions, which do nothing, or change the first few words of the selection into an absolute branch to the first instruction after the selection.



In the third type of patch, you must first select an instruction that uses the condition code bits syntax. These are the B*cc*, DB*cc*, S*cc*, or TRAP*cc* instructions, where the *cc* can be any of the condition abbreviations listed in the above menu. Just choose the condition you want the selected instruction to use from the heirarchical submenu of conditions.

A typical example of how this is useful is when you have discovered in your source code that the condition computed by an `if` statement should have been >= rather than >. You need first to use your development system debugger or disassembler to help you find the instruction(s) in the segment ('CODE' resource) that correspond to the condition evaluation, and change the related branch instruction.

Note: The B*cc* instruction, where the condition *cc* is T (Always True), is generally disassembled as BRA (unconditional branch).