# BCPL Menu Package Description

Keith Knox

July 5, 1978

Filed on: [Maxc1]<AltoDocs>Menu.press

## 1. Introduction

An interactive program is one which communicates with the user, and whose output depends on responses it receives from the user. Two possible forms that these responses can take are 1) a dialog -- an exchange of questions and answers or 2) a menu -- a visual display of choices on the screen which are selected with the mouse. One generally opts for the dialog method because the graphical method of using menus is so cumbersome to implement. When a menu is used, it must first be designed on paper (or in your head) and then translated into code to create it on the screen. As for inputting information from the mouse, individual programmers may have developed special procedures for their own use, but even so, they are probably not easily transferable from one application to the next.

The menu package, described in this report, is an attempt to simplify both the design and the implementation of menus for use in BCPL software. This package is only a first cut at this goal and it is expected to undergo many changes in the future as it matures. In its present form, a programmer can use the Alto screen and the mouse to interactively create a menu, and then use the software package to implement it in his own program.

### 1.1. Creating and Using a Menu

To illustrate the method of creating and using a menu, first consider a very simple program. Let's say that someone wishes to write a program that will do only one function -- invert the screen. To implement this he would like to present the user with a menu containing two boxes, one labeled INVERT and one labeled QUIT. When the user selects the INVERT box, the screen should turn from white to black (or visa versa), and when he selects the QUIT box, the program should exit.

A program using this menu can be easily implemented with the Menu package. First the menu is designed with the menu editor, MenuEdit.RUN. This program lets one create and manipulate rectangular boxes on the display screen using the mouse and a few simple keyboard commands. The description of how this is done is given in section 3. Then MenuEdit writes out two BCPL files which contain a few manifest constants and some tables describing the layout of the menu. These files are compiled and loaded with the main program along with the rest of the Menu package, which consists of three .BR files. The initialization procedures then use this code to create and display the menu. These points are illustrated in the implementation in the sample program given below.

## 2. Sample Program

The sample program, "Invert.bcpl", is shown below in section 2.1. It simply inverts the display screen when the "Invert Screen" box is selected. A menu was created with MenuEdit and recorded onto the two source files "InvertNames.d" and "InvertTables.bcpl" also shown below. The "names" file contains the name definitions of the two boxes and is included in the program with a "get" statement. The "tables" file, on the other hand, is compiled separately and loaded along with the main program. It contains a procedure called *MenuInitHelp*() which is called by *CreateMenuDisplayStream*(). The procedure *MenuInitHelp*() returns a pointer to the DATA structure which contains the menu pointer, a string list pointer and the dcb for the menu. This structure is stored in the external static *MenuData*. The definitions file "MenuDefs.d" contains the external statements for the procedures that are used from the package along with a few structure definitions.

To initialize the menu, the programmer first allocates some space for the menu bitmap. The amount of core required can be found from *MenuSize*(). Next a display stream is formed and returned by *CreateMenuDisplayStream*(). This completes the initialization of the menu. The OS routine *ShowDisplayStream*() can be used to display, move or remove the menu. In the sample program, the menu is displayed below the system display stream, although selections may be made on the menu no matter where it is finally located on the screen. If the menu is removed from the screen and its bitmap area is re-used by some other part of the program, a second call to *CreateMenuDisplayStream*() will restore the bitmap for the menu.

Selections on the menu are determined by repeated calls to *ScanMenu*(). The selection of a box is made by pointing to the box with the cursor and pressing and releasing a mouse key. If the cursor is moved out of the box before the key is released then the box is not selected. The process of selection leaves the sense of the box flipped. In the sample program, it is inverted again with a call to *DeSelect*(). The value that is returned by *ScanMenu*() when a box is selected is the value defined in the manifest "names" file that is included in the program. The names used below, "invert" and "quit", were identified with the appropriate boxes when the menu was first created in MenuEdit. When a box is selected, the proper action is taken simply by switching on the returned value into one of the defined "names".

### 2.1. Invert.bcpl

```
// Invert.bcpl -- Inverts the screen, uses output of MenuEdit.run
// bldr Invert InvertTables Menu MenuBox MenuBoxUtils

get "InvertNames.d"
get "MenuDefs.d"

external
    [
    GetFixed
    ShowDisplayStream
    ]

let main() be
    [
    // test of the menu package
    let length=MenuSize()
    let buffer=GetFixed(length)
    let stream=CreateMenuDisplayStream(buffer,length)
    ShowDisplayStream(stream)

    // loop over menu
    let menu=MenuData>>DATA.menu
```

```
        [
        let selection=ScanMenu(menu)
        switchon selection into
            [
            case invert:    Invert() ; endcase
            case quit:      finish
            ]
        DeSelect(menu!selection)
        ] repeat

    ]



and Invert() be
    [
    let dcb=@#420
    while dcb do
        [
        dcb>>DCB.background=not dcb>>DCB.background
        dcb=@dcb
        ]
    ]
```

**2.2. InvertNames.d**

```
// InvertNames.d  --  Manifest names for menu windows.


manifest
    [
    invert=1
    quit=2
    ]
```

**2.3. InvertTables.bcpl**

```
// InvertTables.bcpl  --  Tables for setting up menu windows.


external MenuInitHelp


let MenuInitHelp() = valof
    [
    // set up menu table
    let menu=table
        [
        2
        0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0
        ]
    menu!1=table [ 0;#4;#114;#164;#336;#227 ]
    menu!2=table [ 0;#4;#456;#164;#700;#227 ]

    // set up stringlist table
```

```
let stringlist=table
      [
      2
      0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0
      ]
stringlist!1="Invert Screen"
stringlist!2="Quit"


// set up menuDCB table
let menuDCB=table
      [
      0
      0;#0;0;58
      0;#2032;0;18
      0;#0;0;328
      ]
test (menuDCB&1) eq 1 ifso menuDCB=menuDCB+1
      ifnot for n=0 to 11 do menuDCB!n=menuDCB!(n+1)
for n=0 to 1 do menuDCB!(4*n)=menuDCB+4*(n+1)

// now finish up
let temp=table [ 0;0;0 ]
temp!0=menu
temp!1=stringlist
temp!2=menuDCB
resultis temp
]
```

### 3. MenuEdit.run

This program is just a first cut at a menu editor. As a result, MenuEdit itself does not use a menu. It does, however, allow one to interactively design a menu on the screen.

There are no command line switches. After starting the program the version number is written on the system stream and the user is informed to press the red key. Continuing by pressing the red key leads to a completely blank screen where menu boxes can be defined.

### 3.1. Commands

There are two ways of entering commands, by the keyboard and by the mouse. These commands allow one to define rectangular boxes on the screen and then move them, change their shape, outline them, insert strings etc.. The description of the final menu is then be written out as two BCPL files containing the proper definitions in the form of BCPL tables and manifest statements. This description can later be read back in and editted.

### 3.2. Mouse Commands

A box is created by pressing the left (red) mouse key, moving the cursor and releasing the key. The two diagonal corners of a rectangle are defined by the starting and ending points and this area is marked with a one bit wide outline. If the middle key is now pressed, the cursor moves to the lower right corner of the nearest box and the box follows the cursor, stopping when the key is released. The size of a box can be changed by pressing the right key. The cursor then moves to the lower right corner of the nearest box. The upper left

corner of the box remains fixed while the lower right corner follows the cursor stopping again when the key is released. Because of space allocations made, there is an upper limit of 256 boxes imposed on any menu created by MenuEdit.

If the left mouse key is pressed when the cursor is inside an existing box, then instead of creating a new box, the existing box can be "selected". Releasing the key inside the box selects that box. Selected boxes are marked by flipping the sense within the box. All "selected" boxes can then be acted upon by the keyboard commands discussed in the next section. Re-selecting an already selected box de-selectes it.

### 3.3. Keyboard Commands

The keyboard commands that are available are given below. Except for *R*ead and *Q*uit, none of them are active unless there is at least one box created on the screen.

For some keyboard commands, input from the keyboard may be required. In most cases, the menu display is removed and the system display appears with instructions or questions. When entering a value from the keyboard, terminate the entry with a *CR*. Typing a *CR* alone gives the default value which is shown enclosed in square brackets.

*Keyboard Commands:*

*Q* -- quit, asks for confirmation with a *CR*.

*R* -- read in a menu description from two BCPL source files, asks for confirmation with a *CR*. The first is a definitions file containing manifest constants defining the names of each box. The second file is the source which will be used by the Menu package to set up the menu in the user's program. MenuEdit reads these files and displays the menu and waits for new commands. The file names must end in "Names.d" and "Tables.bcpl" in order to be read in. Therefore, only the file name kernel need be entered. For example, to read in "MenuNames.d" and "MenuTables.bcpl", enter the name "Menu".

*W* -- write the description of the existing menu into two source files.

<*TAB*> -- a tab does a refresh of the screen.

*E* -- selects all boxes.

<*DEL*> -- hitting the delete key removes all selections and resets the default case of making the strings visible (instead of the names, see the <*ctrl*>*N* command).

<*ctrl*>*A* -- select all inactive boxes and de-select all active ones. The purpose of this command is to allow identification of inactive boxes. See below for definition of activity.

<*ctrl*>*N* -- display the box names instead of the strings for each box.

In the following commands FIRST select the box or boxes, THEN type the command.

*A* -- switch the definition of the selected boxes between active and inactive. An inactive box will not be able to be selected when the menu is implemented in a program. Identifying which boxes are inactive is accomplished by use of the <*ctrl*>*A* command (see above).

*B* -- switch the definition of the selected boxes between normal and bold display of text.

*C* -- change the size of the selected boxes. First select the boxes to be changed then enter the new width and height from the keyboard.

*D* -- delete the selected boxes.

*F* -- fill the selected boxes with a color. The colors allowed are *W*hite, *B*lack, *G*rey and *U*nfilled. The default case is *U*nfilled when a box is first created. At the present time, this option is not recommended for use, but is included for completeness.

*G* -- set the gap between the selected boxes either vertically or horizontally. The first box selected is left in its original position and the rest are positioned to the right or below. Enter the amount of spacing from the keyboard, then type a *V* or an *H* to line them up vertically or horizontally.

*I* -- insert strings into the selected boxes. Only one string may be included in each box.

*L* -- line up the selected boxes either vertically or horizontally. The first box selected is left in its original position and the rest are lined up to the right or below. Type a *V* or an *H* to line them up vertically or horizontally.

*M* -- move all the selected boxes as a group. The selected boxes are temporarily replaced with the surrounding box which can them be moved with the middle key of the mouse. No further commands can be implemented until the middle key of the mouse is pressed and then released.

*N* -- name the selected boxes. The name is not visible on the menu. It is written in the definitions file which should be included in the user's program. See the sample program for the use of the names. See also the Format of a Menu in section 4.1.

*O* -- outline the selected boxes. The width of the outline in bits is entered from the keyboard. The color of the outline can also be specified. The choices are *W*hite, *B*lack, *G*rey and *F*lipped.

*S* -- makes a group of boxes all the same size. All the boxes are made the same size as the first box to be selected.

*T* -- position the text in the selected boxes in positions other than the default case of the center of each box. For each selected box in turn, the beginning of the text will be written in the cursor and the new text position is indicated by moving the cursor and pressing and releasing the left mouse key. The default position can be restored by pressing and releasing any other mouse key instead.

## 4. Software Package Procedures

There are four basic .BR files which make up the Menu software package. These are Menu.br, MenuBox.br, MenuBoxUtils.br. and MenuKeyboard.br. There is also one definitions file, MenuDefs.d. Menu contains the high level menu routines which the programmer will use most often. MenuBox contains lower level routines which may be used by the programmer, depending on the complexity of the program. MenuBoxUtils contains assembly language routines which are used by MenuBox but are not expected to be used by a programmer. MenuKeyboard is an auxiliary package which contains a keyboard reading procedure for entering data from the keyboard into a box on the screen.

### 4.1. Format of a Menu

A *menu* is a pointer to a structure. The first word of this structure is the length, i.e. number of items on the menu. The following words point to *boxes*.

A *box* is pointer to a structure which describes a rectangular region on the display screen. This structure is 6 words long and is described in the file MenuDefs.d. The first word is the dcb (if any) that covers the designated area of the screen. The second word contains several flags which tell if (and how) the box is outlined, colored, inactive, selected and bold. The next two words are the x-y coordinates of the upper left corner (the *origin*) of the box. The last two words are the x-y coordinates of the lower right corner (the *corner*) of the box. These four words also contain the description of where the text is to be positioned within the box.

If the dcb entry is zero then the x-y coordinates are absolute coordinates on the screen, i.e. they vary between 0-605 and 0-807. If the dcb entry is non-zero, i.e. there is a dcb covering the area defined by the box, then the x-y coordinates are defined relative to the dcb. This is an important point. This means that as a dcb is moved on the screen, the box definition moves with it. Therefore, a routine like *CursorInside* (see MenuBox) can always tell if the cursor is inside a particular box, no matter where the dcb is moved to, as long as it stays on the dcb chain. Another important point is that a dcb relative description of a box is restricted to one dcb. It may not cross dcb boundaries. Of course, an absolute coordinate description (i.e. dcb entry = 0) is independent of dcb boundaries. Any routine which manipulates bitmaps can deal only with a box with a dcb relative description.

Another item which has the same format as a menu is the list of the strings that go in the boxes. Again the first word of the structure is the length and is equal to the length of the menu. The n-th element of the string list is a pointer to the string that gets written in the corresponding box on the menu. There may not be a string for each box, however, and if not the corresponding element in the array is zero.

This array is initialized by the set up routines called by the programmer and returned in a static called *MenuData*. The static points to a DATA structure. The first word of this structure points to the menu, the second word to the string list and the third word to the dcb for the menu. See the file Menudefs.d.

The names of the boxes are used as follows. A routine called *ScanMenu*() from Menu returns a value which is the position in the menu of the box that was selected. A name which is assigned to a box, is equal to its position in the menu, so that the programmer does not have to try to figure out which number goes with which box that he sees on the screen. These names are written on a definitions file as manifest constants which can be included in the user's main program in a "get" statement. In this way, a programmer can define a meaningful name to each box which can indicate the function that was selected. No names are assigned to boxes unless an explicit indication is made within MenuEdit.run by use of the *N* command.

### 4.2. MenuDefs.d

This file contains external statements and structures which the programmer will find useful and should be included in a "get" in the main program. It includes the BOX, MENU and DATA structures described in the previous section.

### 4.3. Menu.br

*MenuSize*()

If the menu has not yet been initialized, MenuSize calls the routine *MenuInitHelp*(). This routine is part of the BCPL code generated by MenuEdit.RUN and is used to initialize the menu descriptions. It returns a pointer to the DATA structure which is then stored in the external static *MenuData* (see section 4.1.). *MenuSize* returns the number of words that are needed to display this menu. This is the minimum number of words required. It is the programer's responsibility to allocate the proper amount of storage.

*CreateMenuDisplayStream*(buffer, length)

If the menu has not yet been initialized, *CreateMenuDisplayStream* also will call the routine *MenuInitHelp*() to initialize the menu. *CreateMenuDisplayStream* returns a pointer to a two word display stream that can be used by the OS routine *ShowDisplayStream* to display the menu bitmap. *Buffer* is a pointer to a block of storage that can be used to create the bitmap for the menu. If the length of the buffer is less than is required then *CreateMenuDisplayStream* returns false. A complete dcb chain including blank dcb's to skip the gap areas is created but the stream returned has the top and bottom blank dcb's stripped off. Strings are written on the screen by the procedure called *WriteBox*(), (see section 4.4 ). The font used is obtained by a call to *GetFont*(dsp).

*ScanMenu*(menu, loopOverMenu [true], returnKey [false], sweep [false])

*ScanMenu* continuously loops over the menu and if a box is selected then it flips the sense of the box and returns its position in the menu. Only active boxes are scanned, inactive boxes are ignored, unless the external static *EverthingActive* is non-zero. If *loopOverMenu* is false, it makes one pass over the menu returning false if nothing was selected. This allows the programer to check other conditions such as the keyboard stream at the same time. If *returnKey* is non-zero then the mouse key that was used to select the box is returned in the left byte. The value of the mouse key is given by (not @#177030) & 7. The parameter *sweep* is put in as the last argument of select (see MenuBoxUtils). If *sweep* is false then when the cursor is moved out of a box while the key is still depressed, the box is not selected. If *sweep* is true then the box is selected.

*DeSelect*(box)

This routine inverts the sense of the indicated *box* and sets the "selected" bit in its structure to false.

*ShowMenu*()

This routine removes whatever is on the screen and shows the complete menu exactly as it is seen in MenuEdit. *CreateMenuDisplayStream* must be run before *ShowMenu* is called. This procedure is provided as an alternative to using *ShowDisplayStream* to display the stream returned in creating the menu.

### 4.4. MenuBox.br

*CreateBox*(Xo, Yo, Xc, Yc, inputZone [sysZone])

The first four arguments are, in order, the x-y coordinates of the upper left corner and the x-y coordinates of the lower right corner of the box expressed in absolute coordinates on the screen. Then *CreateBox* allocates a block 6 words long from *inputZone* (or *sysZone* if *inputZone* is zero or absent) and converts the coordinates relative to a dcb presently on the

screen if possible. The pointer to the block is returned. If no space was available then it returns zero.

*CursorInside*(box, XCursor [0], YCursor [0])

Returns true if the cursor is inside the *box* and false if not. The (0,0) point on the cursor is used unless *XCursor* and *YCursor* are specified. If the box is defined with relative coordinates then *CursorInside* returns false if the dcb is not on the dcb chain.

*OutlineBox*(box, bits [box>>BOX.bits], outline [box>>BOX.outline])

Returns true if the *box* was successfully outlined. It returns false if the *box* is not defined dcb relative. The second argument is the width of the outline in bits. The third argument indicates how the outline is to be done. *Outline*=0 means outline by flipping memory, *outline*=1,2 or 3 means outline by replacing with black (1's), grey and white (0's), respectively. A box can be outlined with a line of zero bits, i.e. no outline. If either of the last two arguments are omitted, then their values are read from the box structure.

*FillBox*(box, background [box>>BOX.background], skip [1])

Fills the *box* WITHIN the present outline and returns true if the box was successfully changed. *Background* has the same meaning as in *OutlineBox*, i.e. *background*=0 means flip the sense of the box, and *background*=1,2 or 3 means fill with black (1's), grey and white (0's), respectively. If the second argument is omitted, then its value is read from the box structure. The last argument is the number of bits to *skip* between the outline and where the box interior is changed. It defaults to one bit.

*NearestBox*(menu)

Returns the number of the box whose lower right corner is geometrically closest to the cursor. Returns false if it is a zero length menu.

*FindDCB*(box, dcb [#420])

Returns the number of scan lines before the dcb containing the *box*. It returns 0 if the box is defined with absolute coordinates and it returns -1 if the box is defined dcb relative but the dcb is not on the dcb chain. The default chain is the display chain, i.e. @#420 if *dcb* is not specified.

*ConvertToRelative*(box, dcb [@#420])

This routine searches the dcb chain (starting at #420), and if possible converts a box defined with absolute coordinates to dcb relative. If the box was already dcb relative then it returns without changing it. If *dcb* is present and non-zero, then it searches the dcb chain starting with *dcb*, instead of the display dcb chain beginning at #420.

*WriteBox*(box, string, bold [box>>BOX.bold], font [sysFont], xmode [box>>BOX.xmode], xoffset [box>>BOX.xoffset], ymode [box>>BOX.ymode], yoffset [box>>BOX.yoffset])

This procedure writes one string, locating it at an arbitrary position within the box. If the *string* is longer than the length of the box (minus twice the width of the outline) then only part of the *string* is written. The writing is done by ORing and the box is assumed to have been erased beforehand. If *bold* is non-zero, then the string is written in bold. If *font* is omitted or zero then it becomes *GetFont*(dsp). If the last four arguments are included, then they define the position where the string will be written within the box. The *mode* is centered if zero and started from top or left if non-zero. The *offset* describes how many bits to offset from the starting position. The x and y dimensions are treated independently. If the

parameters *bold*, *xmode*, *xoffset*, *ymode*, *yoffset* are omitted, then they are read from the box structure.

### 4.5. MenuBoxUtils.br

*write*(string, nwrds, bitstart, wordstart, bitlimit, font)

Writes the *string* into memory using the specified *font*. The address of the beginning line is *wordstart* and the bit position within the line is *bitstart* (*bitstart*=0 is the first bit). The number of words/scan line is *nwrds* and only *bitlimit* bits of the *string* are written. It returns the number of bits actually written.

*CallBitBlt*(fn,u,dbca,dbmr,dlx,dty,dw,dh,sbca,sbmr,slx,sty,g0,g1,g2,g3)

This routine takes as many arguments as are included in the calling statement and fills them into a table and then invokes BitBlt from the ROM.

*select*(xleft, xright, ybottom, ytop, key, flag)

This routine checks the specified *key* and if the key is released while the cursor is within the indicated region, it returns true. If it is not released before the cursor leaves the region, then it returns *flag*. Therefore, set *flag* to whatever you would like to receive when that condition occurs. The *key* is calculated by (not @#177030) & 7.

### 4.6. MenuKeyboard.br

*GetString*(box, defstring, zone)

*GetString* returns the address of a string allocated from *zone* which contains the information typed in from the keyboard. The indicated *box* is erased and a small blinking box is put in the upper left hand corner. *Defstring* is a default string which was previously allocated from *zone*. If *defstring* is non-zero, it will be displayed in the box as a starting point for keyboard entry. It will be automatically de-allocated from *zone*.

## 5. The Menu Package and the Trident

Bitmap manipulation on the screen, i.e. erasing, flipping the sense, outlining, etc. is done by the routine called *CallBitBlt*(), defined in MenuBoxUtils.BR. This routine uses the BitBLT instruction from the ROM. For most programs this presents no problem, but for software which uses the Trident code, special care must be exercised. The problem is that for PROM microcode versions 23 or before (version 2 for Alto II's), BitBLT does not function properly if the Trident code is loaded in the RAM. For this reason, there are two versions of the *CallBitBlt* procedure. A second version of MenuBoxUtils (MenuBoxUtilsSoft) is provided which uses the SoftBitBlt package.

If wish to use the Trident code and the Menu package at the same time, then you should load your program with MenuBoxUtilsSoft.br (instead of MenuBoxUtils) and the SoftBitBlt package, BitBltA.br and BitBltB.br.

Although this will increase the time needed to erase or blacken a menu box, it will be noticeable only if the boxes being erased or selected are large.

## 6. The Menu Package and Contexts

The Menu Package while scanning the menu or waiting for keyboard input runs an internal procedure called *MenuIdle*. If the menu is being used in a context then blocking can be made to occur simply by setting the external static *MenuIdle=Block*.