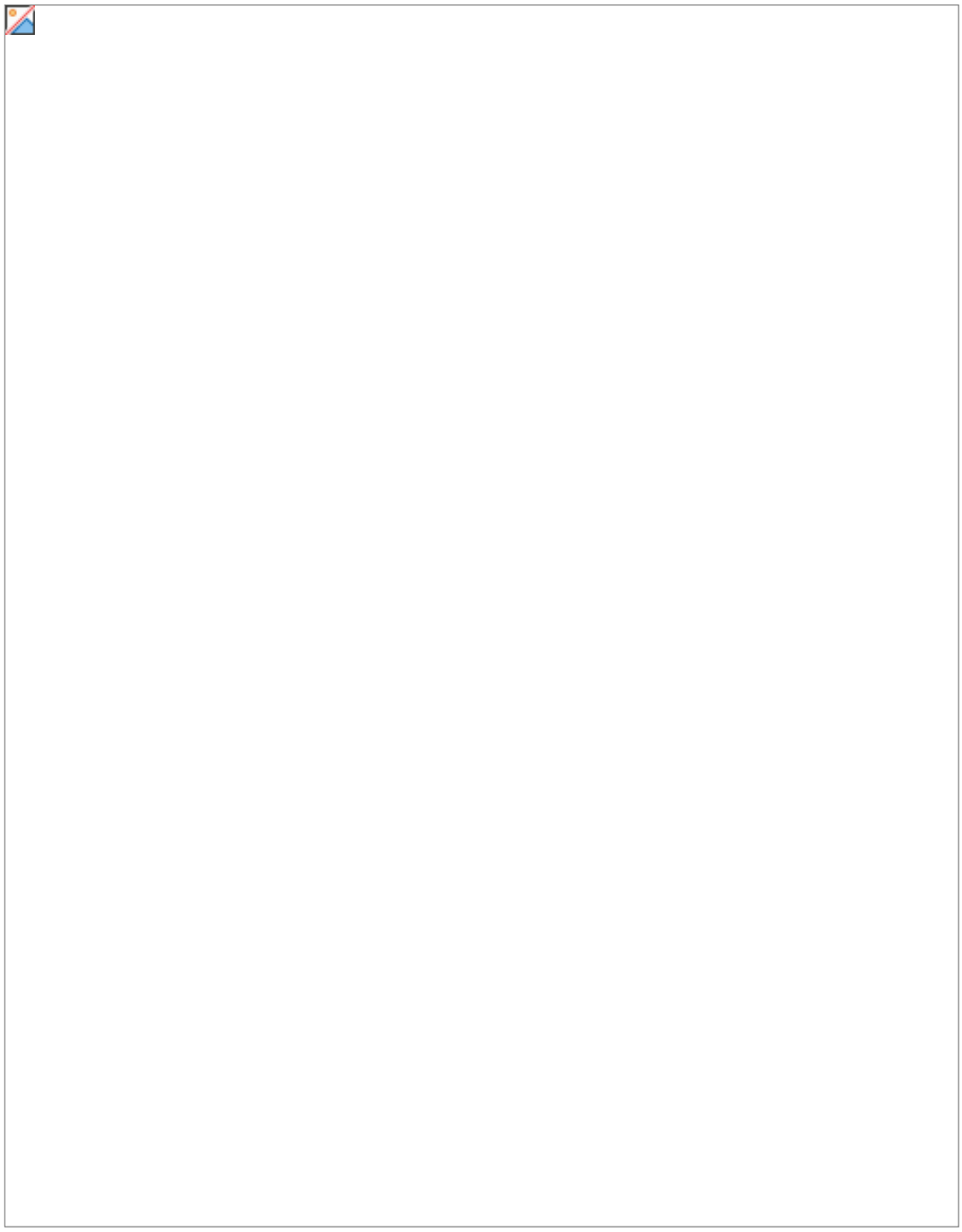
ABZmonPPC
a native PowerPC Debugger
by Alain Birtz
version 0.9
Reference manual

Alain Birtz
650 Grand St-Charles,
St-Paul d'Abbotsford,
P.Q., Canada, J0E-1A0
CompuServe: [72467,2770]
Internet: 72467.2770@compuserve.com

A typical ABZmonPPC debugger screen



ABZmonPPC folder

The ABZmonPPC folder holds six items: the English and French documentation, an important ReadMe text, the **ABZmonPPC_*f*** folder holding the debugger INIT and the ABZmonPPC.help, **PPCdebKey_*f*** folder holding a programmer swtich INIT and it documentation file, and **test_deb_f** folder holding a native application to test the debugger and all assembler source file needed to rebuild the application in MPW. ABZmonPPC can be distributed freely, but please, keep the files together.

Installation

To install the monitor put the 'ABZmonPPC', ABZmonPPC .help and PPCdebKey into the Extension folder or into System folder and restart the computer. ABZmonPPC and ABZmonPPC.help must be in the same folder. If the monitor is correctly loaded you will see this icon at the bottom of the screen:

As soon as you see the monitor icon, the monitor is active, and you can call the monitor immediately (for example, to check how other INITs are installed).

Note: the icon symbol is the one found on some Mac interrupt switch...

If the monitor is not correctly loaded you will see this icon:

You can disable the loading process by pressing the 'Option' and = keys at the startup.

Warning: Pre-System 7.5 must use PPCTraceEnabler extension, otherwise trace and step doesn't work correctly in ABZmonPPC. PPCTraceEnabler is not needed for System 7.5 or higher

Presentation

ABZmonPPC is a low level debugger, a tool to check programming errors. Like other debuggers, ABZmonPPC has standard features such as a break point, a step by step mode, a memory dump, a microprocessor register dump, etc.

But ABZmonPPC has many other functions usually not found in other debuggers. For example, ABZmonPPC has a graphical interface using windows, menus and a mouse. A text can be viewed inside the debugger. You can quit a "frozen" application and continue normally with another application.

Many functions have been added to ABZmonPPC to help the programmer. Inside a code window, for example, you can see an operand, let's say a pointer like $4(r1), directly by option-clicking on the operand. In this example , you can see the pointer's address but also the pointed value. You can save the address (inside the CLIP buffer) and with a simple mouse click, you can open a code or memory dump at this address.

The user can choose the part of the screen where ABZmonPPC will be displayed. The debugger image can be moved on the screen within the debugger, and ABZmonPPC now support color. A second video monitor can be devoted to the debugger. An ABZmonPPC's screen snapshot can be taken with just a key combination.

You can set more than fifty internal parameters, including exception vectors, that controls the way ABZmonPPC works. You can save your menu and window setups for the next time. You can also save the useful addresses or values of the CLIP buffer.

ABZmonPPC is the native port of ABZmon (the low level debugger for 68000 microprocessor family) for PowerPC microprocessor. ABZmonPPC is entirely written in PowerPC assembler.

Since the PowerMac run 68020 program, ABZmonPPC can also work has low level debugger for 68020 processor. ABZmonPPC support standard 68K debugger feature such break point, single step, n-step, line-A trap intercept, etc. However, ABZmonPPC is built for PowerPC, so feature for 68K are limited to the basic one...

How the debugger is invoked

The debugger appear when:

- in your code, you insert one of the Debug instruction. The debugger stops the program after the instruction.

- you set a break point. The debugger stops the program at the instruction under the break point. The instruction is not executed

- a step/trace command is given.

- a stop condition is meet. A such condition appear in the "Stop" menu: changes in some memory segment, register taking some value, program exciting a given instruction.

- on a bus error, address error, illegal instruction, etc...

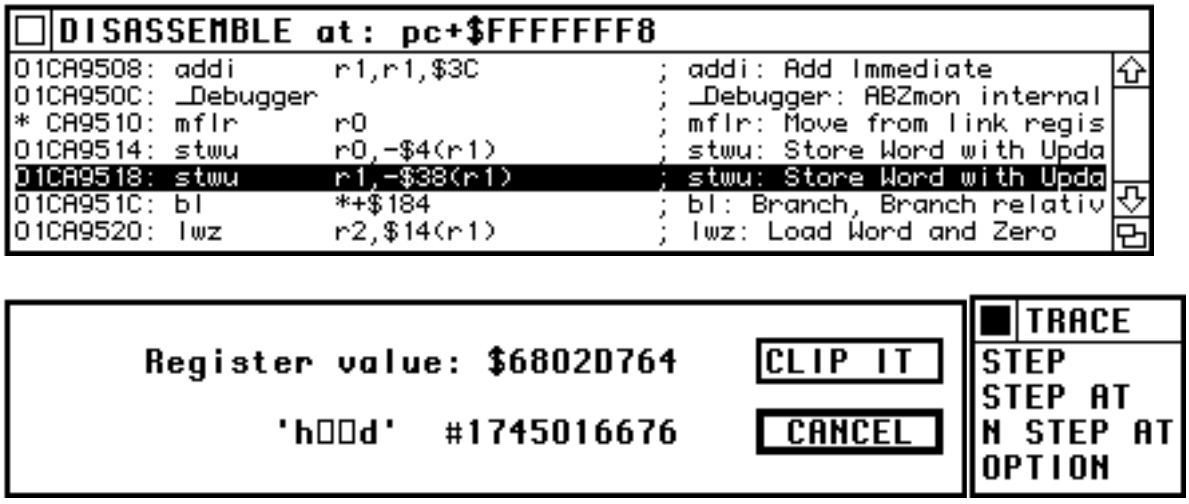
- some Mac have a programmer switch. Hitting this button induce a NMI exception. ABZmonPPC is enterred when the 68K NMI vector is set (default setting).

- the programmer key is hit. 'PPCdebKey' is a little extension that set the programmer key to **control-`** (or control-~). 'PPCdebKey' is included in the ABZmonPPC folder. Resume with a **GO** command.

ABZmonPPC graphic interface

The ABZmon's graphic's interface is similar to the Mac's graphic's interface, but it is not identical. In fact the debugger doesn't use any QuickDraw procedures. ABZmonPPC uses its own screen procedure, and then, does not interfere with the operating system. The interface has menus, windows, dialogs box and uses the mouse like a Macintosh .

Sure, the ABZmonPPC's interface is not as sophisticated as the one built by Apple (the code holds in 50K bytes only), but the user will retrieve almost its familiar environment. Among other differences, one can note that the cut and paste function is not as usual. The debugger transfers only hexadecimal numbers (the most used type of keyboard entries). There are also differences in the window scroll bar. Only up and down arrows are available, and there is no middle thump. However this lack is compensated by a fabulous scroll speed...



Example of window, menu and dialog box in ABZmonPPC

Note that in the dialog box the default button is boldly outlined. Its effect occurs if the user presses **Enter** or **Return**.

The active window

This window has a black close box. The first time you click on a window (menu) it becomes active. You can move the window (active or not) as usual, the mouse being in the title window area. Click into the close box to close the active window. In window with scroll bar, click into the arrow icon to scroll up and down. Alternatively, use the up and down key on the keyboard. if the window has a grow box you can shrink or expand the window as usual.

Sometimes the active window is empty. This means that window content cannot be drawn, probably because ABZmonPPC meets a bus/address error. For example the monitor probably cannot show a DUMP window at $AAAAAAAA since the memory for this address does not physically exist.

In the example above, the TRACE menu is active.

The default address

The default address is set when you click one time in a window and this line become highlighted. By example, a line in a register window is highlighted when you click in the register name area (the value area is reserved to change the register value). The default address is the register value. In the dump window you must click in the dump address area (the other parts are reserved to change the memory). The default address is the hexa value at the beginning of the line. You must click in the address area of a code window since the operand field is used to show the operand value. The default address is also the hexa value at the beginning of the line

The default address is the one shown in most dialogs asking you for an address. For example: to set a break point, highlight an address in any disassemble window and click on the SIMPLE in the SET BREAK menu. To execute the code starting at the address in the R3 register, highlight the R3 line in some register window and click the GO TO item in the CONTROL menu.

In the example above, the fifth line of the DISASSEMBLE window is highlighted, so the default address is $1CA9518.

Debugger mode: PowerPC or 68K

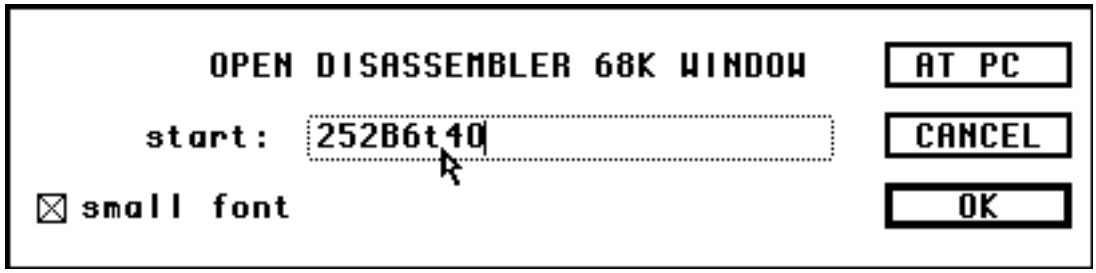
ABZmonPPC operate has both PowerPC and 68K debugger. ABZmonPPC signal mode switch by a dialog like the one below. ABZmonPPC has entered many time to debug PowerPC code. Then a _Debugger instruction in 68K code has called ABZmonPPC.

When a such debugger mode switch occur, a 68K code window is made the front window. If there is no 68K code openned, a new one is created at the address of the PC. If the mode switch from 68K to PowerPC, a PowerPC code is made the front window.

Sometime ABZmonPPC have to guess what the user want to do with some command. By example, the user may wish set a break point in some PowerPC code while the debugger is in 68K mode. If the break point is set directly in the PowerPC code window, ABZmonPPC assume that the break point must be a PowerPC break point. But if the command is issued from the break menu or from a keyboard shortcut, there is no way to know what kind of break point the user want install. In this case, ABZmonPPC follow this simple rule: the command apply to the most recently used code window. So, to be sure sure that your command apply to the right mode, make a code window the front window (just click on the window). The command apply to this kind of code window. The are only few case ABZmonPPC have to made this guess: command given in **OPT DIS** menu (option for disassembler), **SIMPLE** item in the **SET BREAK** menu and **/** and ***** keyboard shortcut (to change PC displacement form and rebuild symbol/label in disassembler).

Dialog

Dialog are used to get parameter and send command to ABZmonPPC. The dialog below is used to open a 68K code window. The default button, the one more darker, is the OK button. This button is selected with **Enter** or **Return** key. The check box named **small font** is crossed, so the window will use small font character. The edit field **start** hold the hexadecimal value 252B6t40. The letter **t** is a mistake. When the user select un button or hit **Enter** or **Return** key, ABZmonPPC do not accept the command and put the arrow cursor near the offending character. The arrow is empty to signal an error.



Everything written in the edit field can be copied into the ABZmonPPC clipboard by the usual **Command-C**. The selected string in the **Clipboard** window is pasted by the usual **Command-V**. The default address is written into the edit filed when the dialog is openned. The default address come from an highlighted line in some window.

MAIN menu

This is the root menu. It open all other menus. Unlike Mac menu, menus in ABZmonPPC, can be moved and closed. If the MAIN menu is closed, you can re-open it like this: close any menu or window, then press any key or click the mouse anywhere. The MAIN will reappear, and you will be able to open all other menus and window.

The items in this menu open the following menus:

CONTROL: this menu allows to return to the application, to terminate it or to continue at a given address, etc.

OPEN: this menu opens memory dump windows, code windows, message windows, file selector boxes, file text windows...

SEARCH: searches a sequence of letters or hexadecimal values in memory.

MON SPY: each time the debugger is entered, a condition a is ckeck. This menus set or clear such condition (see **JB Condition** section for more on it).

STEPSPY: Like MON SPY menu, but condition is check each time a trace exception occur (initiated by some step command).

STOP: the items of this menu force a break when some conditions on the registers or/and memory occur (for example when the stack pointer increases) or when the program executes some instructions (TW, for example).

BREAK: sets or removes break points.

TRACE: induces the step by step mode. Single step, "n" step, etc.

OPT DIS: each item determines the look of the code window. For example, the type of labels used (locals or the ones from the compiler) or the type of microprocessor (16 or 32 bits)... Two option menus are opened simultaneously.

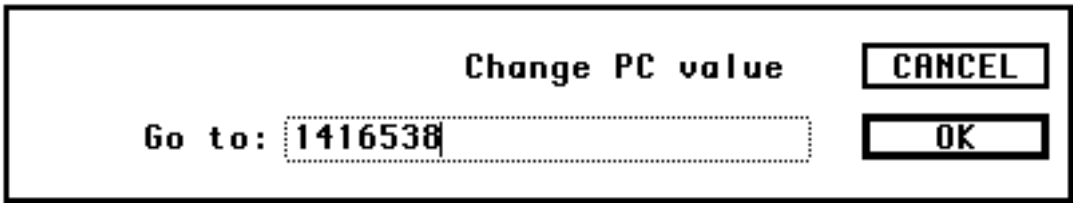
MISC: to save/reload the current menus and window settings, paste an address to the CLIP buffer and set the look of the text windows.

CONTROL menu

This menu allows to return to the current application or terminates it.

GO: quits ABZmonPPC and continues the execution of the program. If the program is stopped by one of the Debug instructions, the execution continues after the Debug instruction. If the program is stopped by a break point, the execution continues at the instruction under the break point. For other king of exception (illegal instruction, unmapped memory) you need to continue at the next instruction: change the **PC** value in the **GPR** window or use the GO TO item).
Keyboard shortcut: **g**, **q**, **G**, **Q**

GO TO ...: quits ABZmonPPC and continues the execution of the program at the address given in the dialog box below. The address $1416538 is the default address (a line of code, at this address, has been selected). When no line is selected, the PC address is used instead.



GOTO NEXT: quits ABZmonPPC and continues the execution of the program at the next instruction. Useful when the debugger is entered because a bus error or an illegal instruction. The faulty instruction is skipped and execution resume at the next instruction.

REFRESH: quits ABZmonPPC and rebuilds the desktop screen. Useful to debug codes using QuickDraw.

FINDER: quits ABZmonPPC and go to the Finder.

RTS: exit the current subroutine. See **The RTS command** below before using this command. Keyboard shortcut: **r**

AE QUIT: send an Apple quit event to the current application. If the application support this Apple event, this can force the application to save the current document before quit.

TO SHELL: quit the current application and return to the command shell: usualy the Finder. Keyboard shortcut: **e** (ExitToShell procedure...)

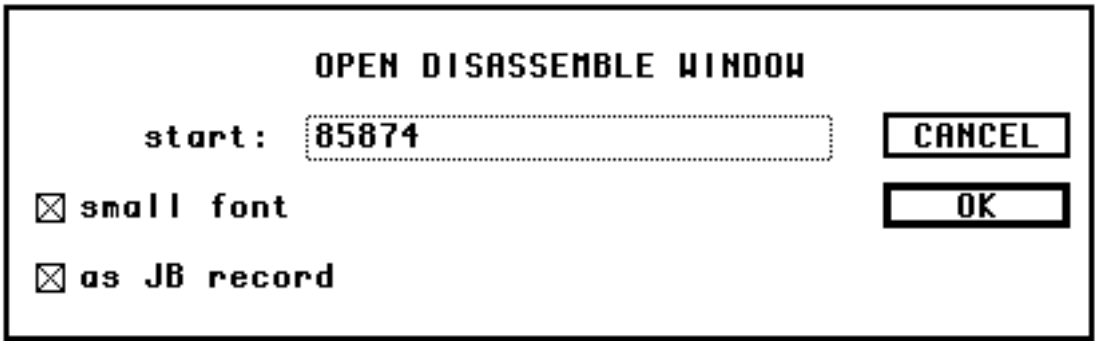
RESTART: restarts the computer (warm start).

The item holding only dash character is used only to separate the other items: the last one is dangerous!

Open menu

This menu opens all non-menu windows in ABZmonPPC.

DISASSEM: opens a PowerPC code window. You must enter the address of the first instruction in the dialog box below. This address can be given indirectly through registers and relative displacements (see **The JB address**). The **small font** check box must be checked to display the text in the window using small fonts. **as JB record** check box must be checked if you want expression in the **start** edit field must be keeping as JB record instead of a straight address. By example, if the expression is PC+8, and PC value is $10000, the window open at $10008. If a step command is given, the PC will probably change to $10004. Then, if **as JB record** check box is checked, the code window show now the first instruction at address $1000C. If **as JB record** check box is not checked, the code window first instruction address still to be $10008.



If the address is not valid, the cursor changes to an empty arrow and the text cursor

appears near the first faulty character. Click the OK button (alternatively, hit Return or Enter key) to open the window and the Cancel button to cancel.

The combination **Command-D** is a shortcut to this item.

There is a more quick shortcut to open a code window: in any code, dump or register simply **option-click** in any line. The window open at the address of this line.

MEM DUMP: opens a memory dump window (in both hexadecimal value and ascII code). A dialog box similar to the one above, asks for the start address.

The combination **Command-M** is a shortcut to this item.

There is a more quick shortcut to open a dump window: in any code, dump or register simply **shift-click** in any line. The window open at the address of this line.

GPR reg: opens a window to show the value of the general purpose register R0 to R31. The **small font** check box, in the dialog below, must be checked to display the text in the window using small fonts.

Shortcut: **Command-R**

CR reg: opens a window to show the bit status of CRO to CR7

FCS reg: opens a window to show the value of the SP, TOC, CTR, LR, PC, CR, XER, RTCU, RTCL, MQ and FPSCR register.

FPR reg: opens a window to show the value of the floating point register F0 to F31.

SPR reg: opens a window to show the value of the special purpose register. Actualy, the Mac OS don't give access to these register. **This item do not open any window**.

BRK PNT: opens a window showing info on all break point.

MESSAGE: opens a message window. Only one message window can be opened at the same time. If the message window is already opened, then ABZmonPPC only make it frontmost window (active).

CLIP: opens a window showing the expression in the CLIP buffer.

PROCESS: opens a window showing all applications, including the background one.

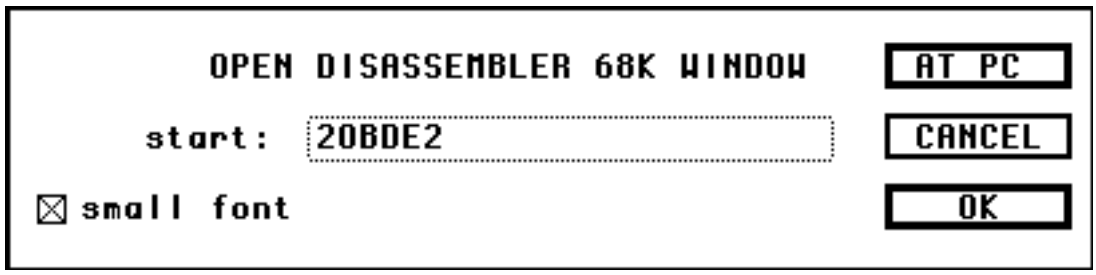
FILE SEL: opens a file selector window. This is the counterpart, in ABZmonPPC, of the Mac dialog you see when choosing "Open" or "Save" as in the File menu. You can hierarchically see and select file and folder.

TEXTVIEW: sees a text of the file selected by the previous item.

CONTEXT: give context information as defined by Code Fragment Manager.

68K DIS: open a 68K code window. The window can be openned such that the first line window is address of the PC (automaticaly updated when the PC change), or at any given address (not updated, but with scroll bar). In the dialog below, the default address is the address got from some highlighted line or the PC address if there is no such

highlighted line. If the botton **AT PC** is selected, the address in the **start** edit filed is ignored.

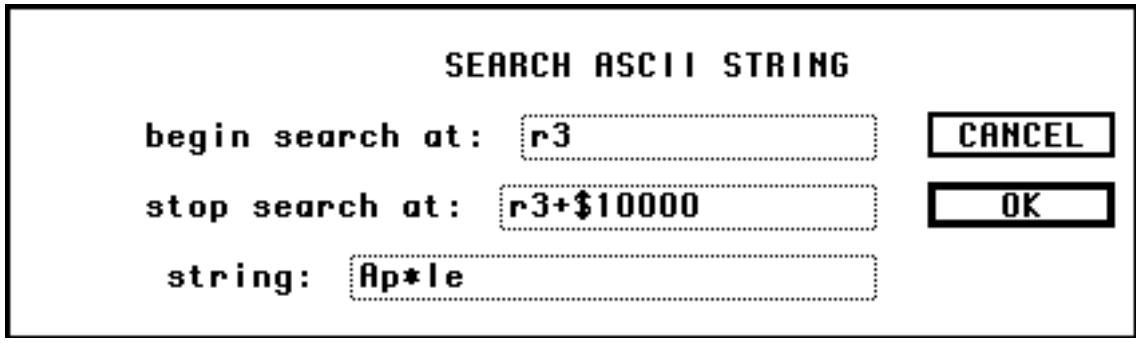


68K REG: opens a window to show the value of the 68K register D0 to D7, A0 to A7, PC, CCR and SR

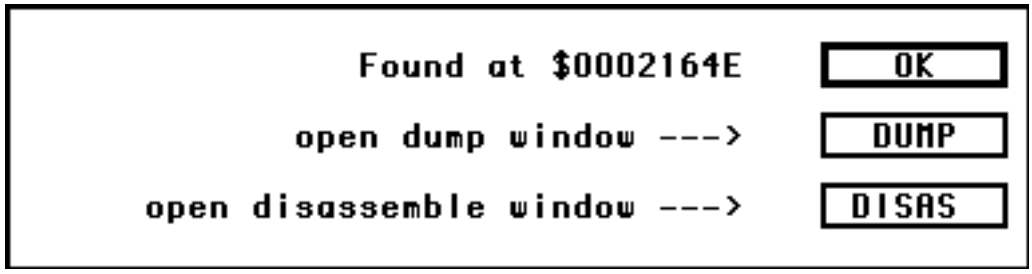
SEARCH menu

To search a sequence of characters or hexadecimal values in memory.

ASCII STR: finds a sequence of letters in memory. In the dialog box below, the first edit field is an expression given the address of the first byte of the memory segment to be scanned. The second field is the address of the last byte in the segment. The third edit field holds the sequence of letters to be found. The symbol * is the wildcard symbol, standing for any character. So, ABZmonPPC searches every word Apple test or part of a word like in attention.



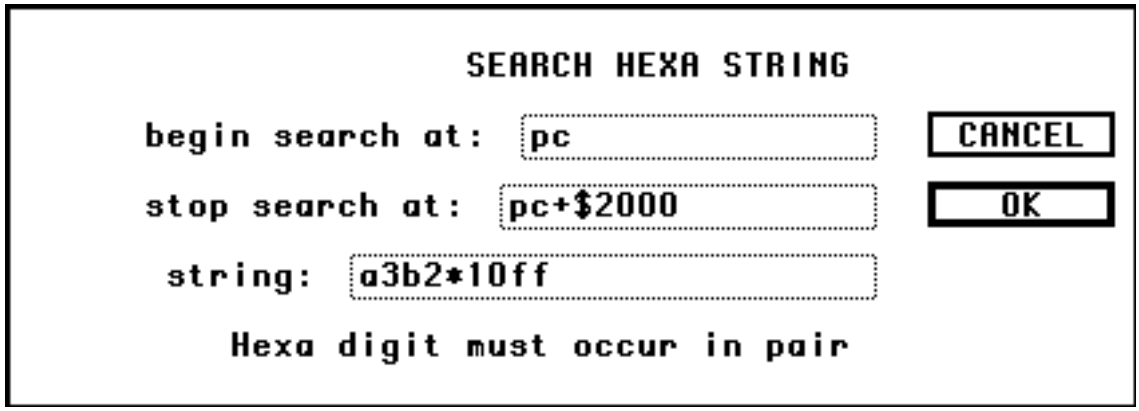
In this example ABZmonPPC has found the word text at the address $2164E. The dialog box below is then shown:



The **DUMP** button opens a memory dump at this address while the **DISAS** button opens a code window. The **OK** button exit without openning any window.

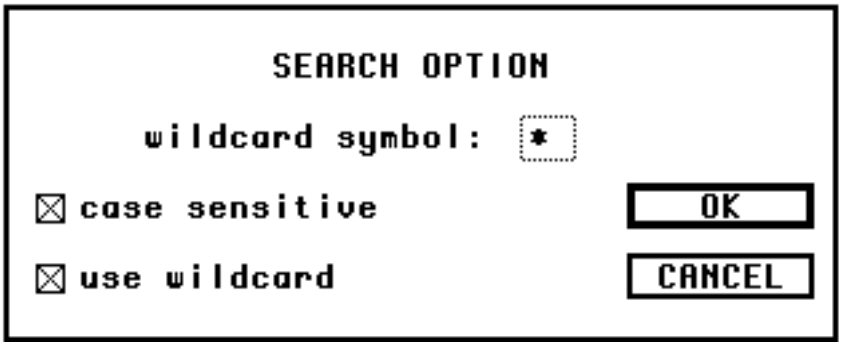
Shortcut: **Command-F**

HEXA STR: finds a sequence of hexadecimal numbers in memory. In the dialog box below, the first edit field is an expression given the address of the first byte of the memory segment to be scanned. The second field is the address of the last byte in the segment. The third edit field holds the sequence of hexadecimal numbers to be found. Note that the wildcard symbol ***** can be also used in a sequence of hexadecimal numbers.



Here the search fail in the segment [PC, PB+$2000]. The dialog below warn you for it:

OPTION: sets some search options. In the dialog box below, the **wildcard symbol** is set in the edit field wildcard symbol. The check box **case sensitive** must be checked if the search is done case sensitive. The check box **use wildcard** must be checked if the search uses the wildcard symbol as a replacement for any character.

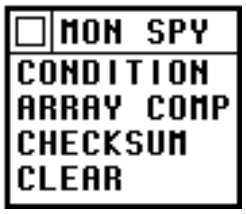


FIND NEXT: continues the search previously initiated.

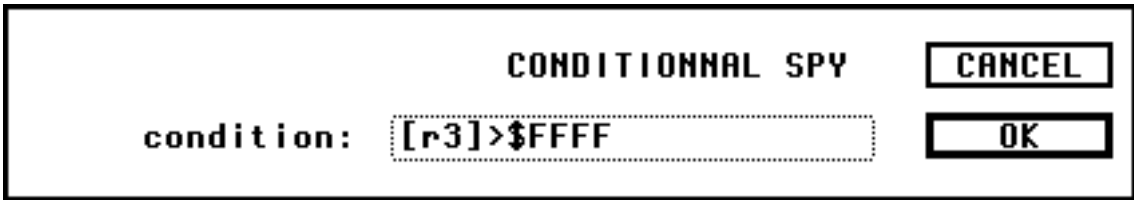
Shortcut: **Command-G**

MON SPY menu

Sometime you may wish to know if a byte or a word in memory has changed since last time you have see the debugger screen. You may also wish to know if a register is becoming negative or zero. Or if a segment has been corrupted. MON SPY menu let you make a command that spy fot such thing.

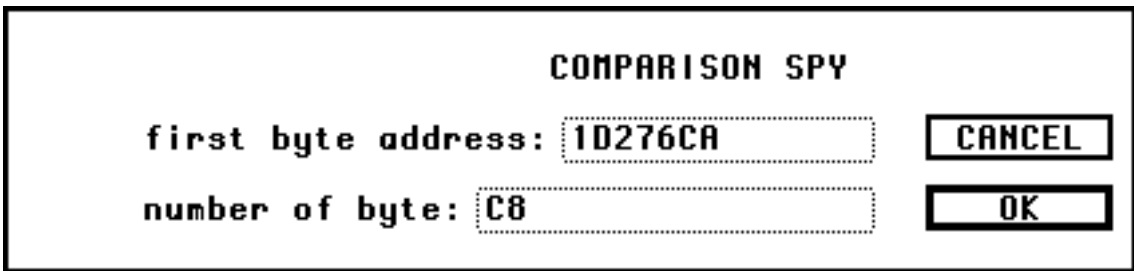


CONDITION: prompt a dialog to set a condition to check each time the debugger in enterred. Warn you when the condition is true. See **JB Condition** section for more on condtion syntax.



The condition in dialog ask the debugger to warn when the word at the address in the register R3 is greater than $FFFF. The next time the debugger will be enterred and the condition will be true, ABZmonPPC will show you the dialog:

ARRAY COMP: warn you if an array of byte in memory has changed. In the dialog below the first field is the address of the first byte to check. The second field is the number of byte to check. When the OK button is selected, the array image is copied into a internal buffer in ABZmonPPC.



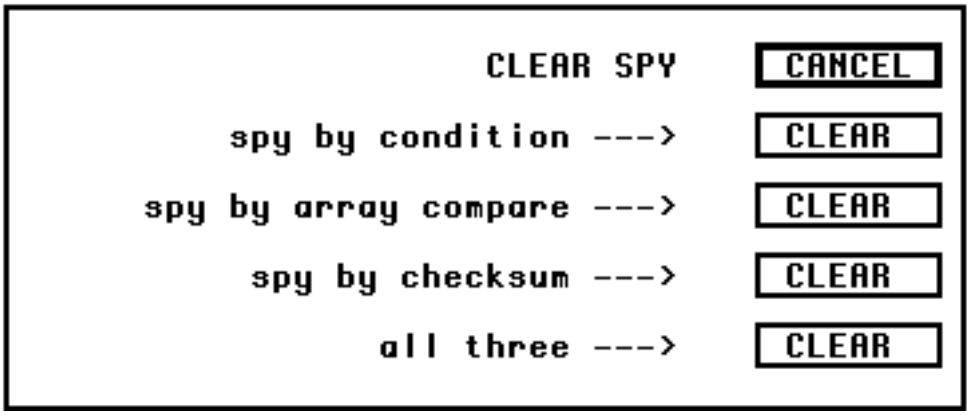
Each time the debugger is entered, the array is in memory is compared to the array copied in the internal buffer. If one or more byte does not match, the debugger warn you:

The size of the internal buffer is set in the S_UP resource of ABZmonPPC INIT (see **Internals variables**).

CHECKSUM: this item is very similar to the previous one. Instead of keeping an image of the segment to check, a checksum is done. The next time the debugger is entered a new checsum is done. If the two checksum does not match, the debugger warn you. Otherwise, the debugger keep silent.

Use CHECKSUM for segment larger than internal buffer used in ARRAY COMP.

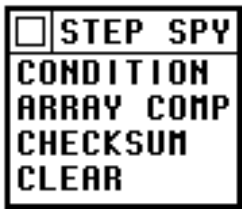
CLEAR: to cancel one or more spy process.



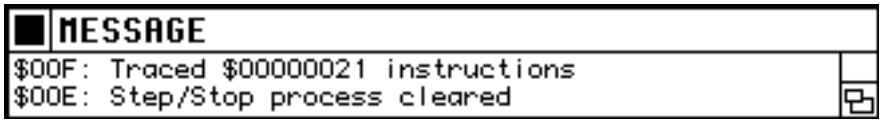
Note: all three MON SPY process can run concurrently.

STEPSPY menu

This menu is identical to the MON SPY menu. Dialog presented are same as above. But, while in MON SPY check is done each time the debugger is enterred, in STEP SPY check is done every time a PowerPC instruction is executed.



ABZmonPPC internaly set a trace mode and check condition or segment change after instruction is executed. If condition is not true or no change is done in segment, ABZmonPPC continue program execution (in trace mode) at the next instruction. Otherwise, the debugger is enterred and a dialog warn the user.To see how many instruction has been executed, look in the message window. The number of instruction is show in hexadecimal.



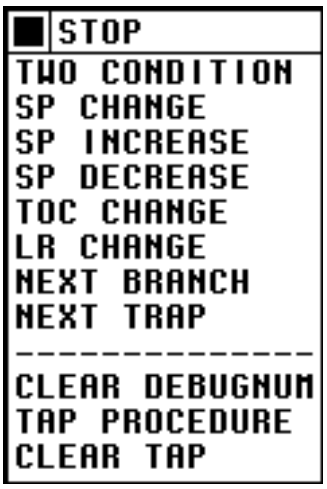
Use STEP SPY with caution. If the condition never come true or if the segment is never changed, ABZmonPPC will continue to execute the program in trace mode, and this is very, very slow!

Note: all three STEP SPY process can run concurrently. In fact MON SPY and STEP SPY run concurrently.

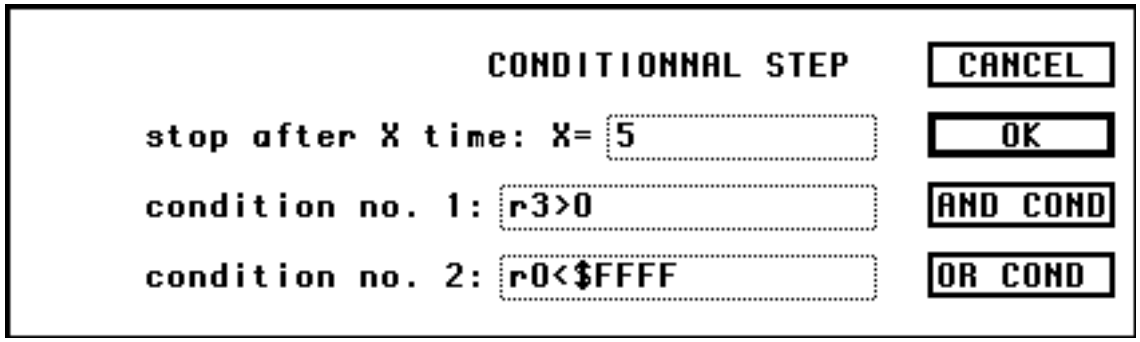
STOP menu

This menu forces a program to break on various conditions. Also, reset DebugNum parameters.

Use item in this menu with caution. If the condition never come true, ABZmonPPC will continue to execute the program in trace mode, and this is extremely slow! When the break is done, ABZmonPPC is entered and a dialog tell you why the break is done. To see how many instruction has been executed, look in the message window. The number of instruction is show in hexadecimal.



TWO CONDITION: set condition to a break when the condition become true.



The first edit field in the dialog above is the number of time the condition become true before breaking. The two next edit field hold the condition. If you you leave a field empty the condition is not used. The **AND** button must be used to force a break when both condition are true. The **OR** button force a break when at least one condition is true. Use the **OK** button when one or both are not used. If both condition are used the **OK** button is same as **AND** button. See **JB Condition** section for more on condition syntax.

SP CHANGE: ABZmonPPC watches the stack pointer SP at every instruction. If the value of SP changed, a break is forced.

SP INCREASE: ABZmonPPC watches the stack pointer SP at every instruction. If the value of SP increases, a break is forced. This feature is useful to get the exit from a subroutine.

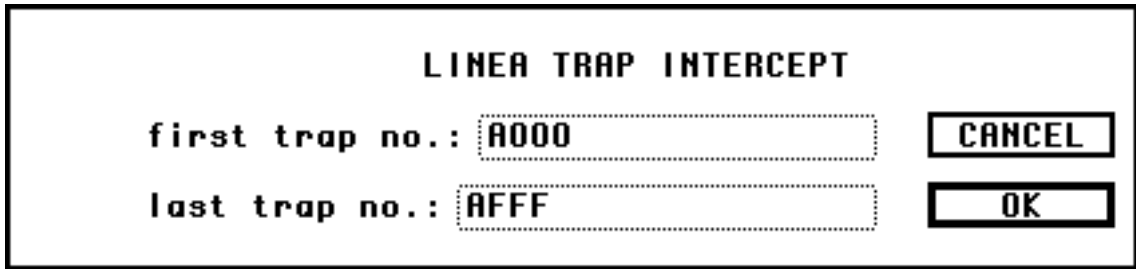
SP DECREASE: ABZmonPPC watches the stack pointer SP at every instruction. If the value of SP decreases, a break is forced.

TOC CHANGE: ABZmonPPC watches the table of content register (R2) at every instruction. If the value of TOC changed, a break is forced.

LR CHANGE: ABZmonPPC watches the link register LR at every instruction. If the value of SP changed, a break is forced.

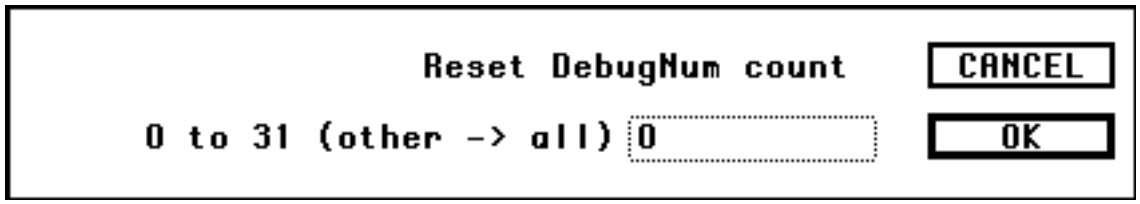
NEXT BRANCH: ABZmonPPC check every instruction until a branch instruction (b, bc, bcctr or bclr) is to be executed. Then, a break is forced.

NEXT TRAP: if the debugger is in PowerPC mode then ABZmonPPC check every instruction until a trap instruction (tw or twi) is to be executed. When this happen, a break is forced. If the debugger is in 68K then ABZmonPPC check every instruction until a lineA trap instruction (OS or ToolBox call) is to be executed. When this happen, a break is forced. You set the range of the lineA trap to be intercepted in the following dialog.



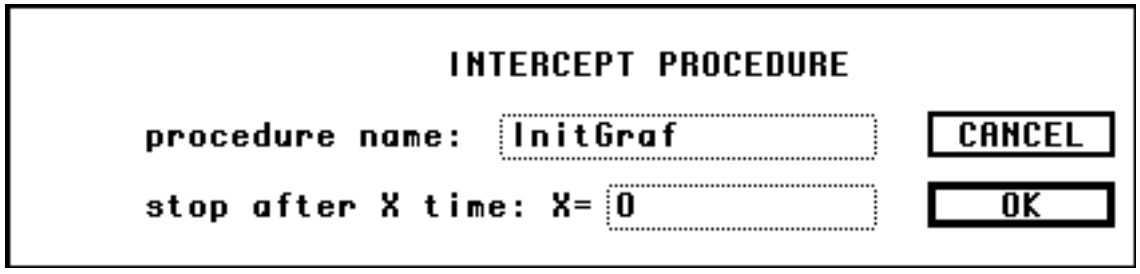
Note: the trap intercept process is cleared each time the debugger is enterred in 68K mode.

CLEAR DEBUGNUM: to reset parameters in one or all DebugNum.



See **DebugNum** section for more on it.

TAP PROCEDURE: a very usefull feature. It is like tapping a phone line. When the target procedure is to be executed, the program is stopped and the debugger is entered. In the dialog below you enter the procedure name (case sensitive) and the number of time to meet the procedure before the debugger was entered (0, the debugger enter the first time, 1, the debugger is enterred only the second time, ect.).

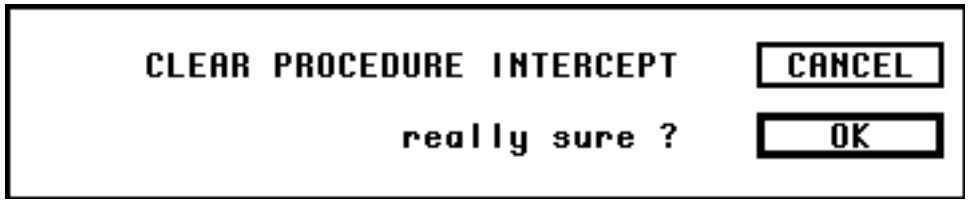


Unlike other **STOP** command, TAP PROCEDURE **do not slowdown** the current process at all.

When the debugger is entered after a procedure intercept, a dialog show for what procedure the stop is done.

There is room for up to 32 procedure interception. When the 32 are filled, the last one is erased and replaced by the new one.

CLEAR TAP: to erase all entry in the TAP buffer.



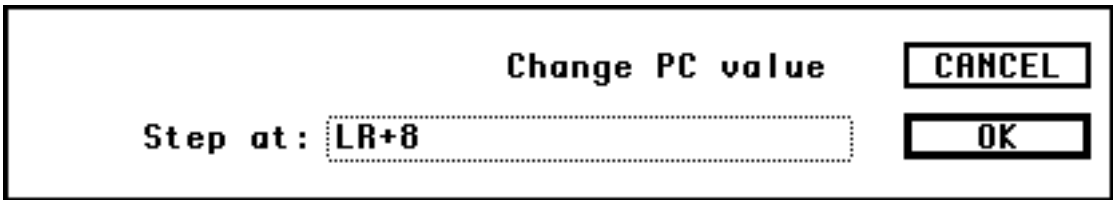
TRACE menu

To initiate step command and set trace option

STEP: executes the instruction pointed by the PC and returns to the debugger. (For both 68K and PowerPC mode)

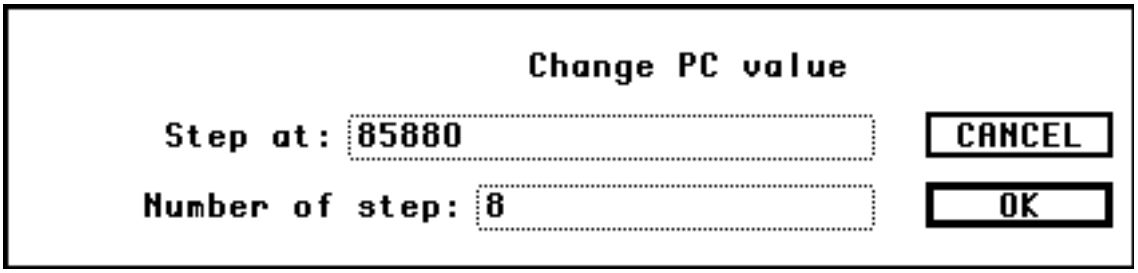
Shortcut: **s**.

STEP AT: executes one instruction at a given address. You enter the address in the next dialog box. (For both 68K and PowerPC mode)



The address can be any valid expression. Here the address is the value of the link register plus 8.

N STEP AT: execute N instruction at the given address. (For both 68K and PowerPC mode). The address and the number of instruction to execute in trace mode are set in the following dialog:

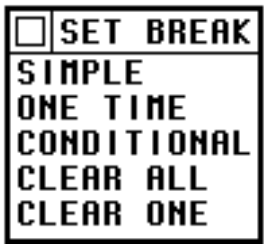


The default address is the PC unless a line is highlighted. In this case the address in this line is used. The number of instruction executed is show in the message window when the debugger is re-entered.

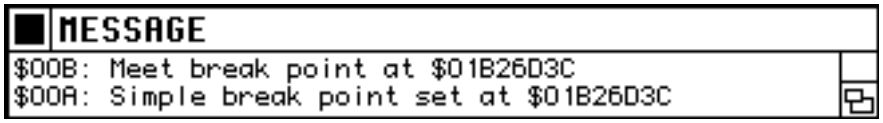
OPTION: no yet implented

BREAK menu

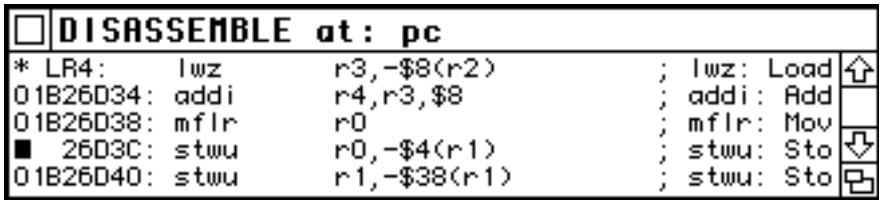
To set or clear break point.



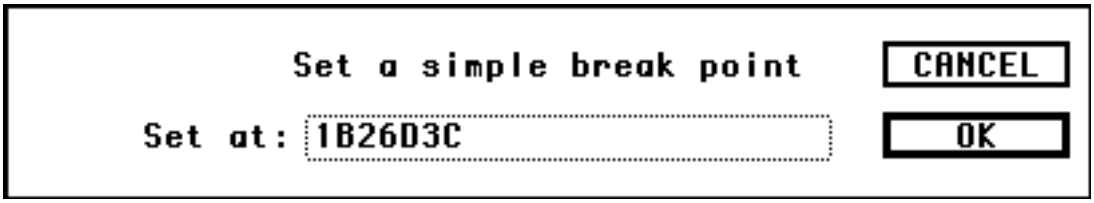
ABZmonPPC let you know when a break point is set by a message in the **MESSAGE** window. A message is also written when later the program is stopped by the break point:



In code window a break point is signaled by a small black box character at the start of the disassembled line. When the break point is not valid the black box hold a white dot. Below the break point is set at $1B26D3C:



SIMPLE: set a break point that break each time. (For both 68K and PowerPC mode)



Shortcut: **Command B**.

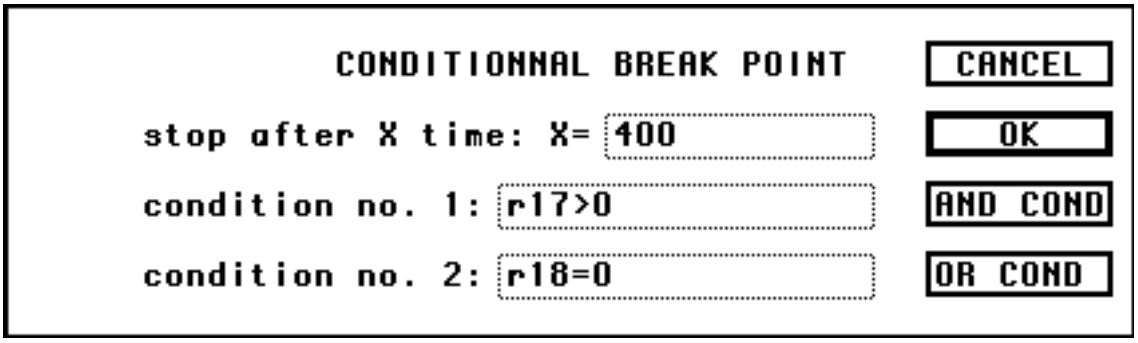
An easy way to set a break point is to highlight, in a code window, the line where the break must be done, before selecting the item. The address of this line appear in the edit filed as default address. Then, to set the break point, you just need hit Return kew or click the OK button.

A more quick way is to to hit the **Command** key while clicking a line in code window.

ONE TIME: very similar to a simple break point. But the break point is automatically removed when the break is done.

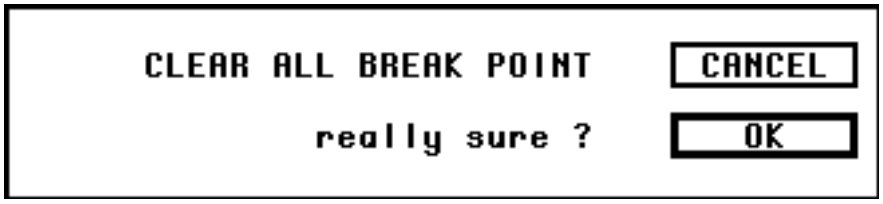
To quickly set a ONE TIME break point, hit the **Command** key while clicking a line in code window.

CONDITIONNAL: set a break point that break when condition become true.

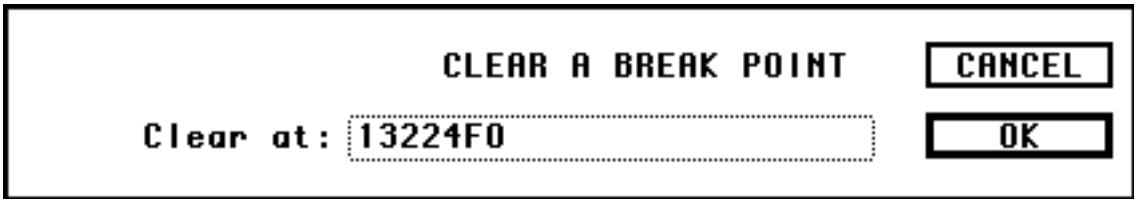


The first edit field in the dialog above is the number of time the condition become true before breaking. The two next edit field hold the condition. If you you leave a field empty the condition is not used. The **AND** button must be used to force a break when both condition are true. The **OR** button force a break when at least one condition is true. Use the **OK** button when one or both are not used. If both condition are used the **OK** button is same as **AND** button. See **JB Condition** section for more on condition syntax.

CLEAR ALL: to remove all break point. (For both 68K and PowerPC mode)



CLEAR ONE: to remove a single break point. (For both 68K and PowerPC mode)



An easy way to remove a break point is to highlight, in a code window, the break code line (the one with the black box character), before selecting the item. The address of this line appear in the edit filed as default address. Then, to remvoe the break point, you just need hit Return kew or click the OK button.

A more quick way is to to hit the **Command** key while clicking the break line in code window.

OPT DISAS menu

To set the form of disassembled line in code window (68K and PowerPC)

REDO SYMBOL: for PowerPC: with the new Code Fragment manager, a procedure in shared library is called by name instead of by address. By example all Mac toolbox procedure in InterfaceLib.xcoff is called by name. ABZmonPPC keep track of all such name (symbol) at startup. A program may build is own set of procedure as shared library. To get the new symbol you need to rebuild the ABZmonPPC symbol buffer. These symbol are used in disassembled line in code window. Unless you suspect an application to create shared library you probably don’t need to rebuild the symbol buffer.

To rebuild the symbol just click in the REDO SYMBOL item, a wait until the dialog “**Now working, please wait...**” diappear.

For 68K: compiler put the name of a routine at the end the routine. The debugger keep track of these names, and show these name in the code window. To get the new names you need to rebuild the ABZmonPPC symbol buffer. You need to rebuild the symbol buffer each time the application to debug is loaded. To rebuild the symbol just click in the REDO SYMBOL item.

Shortcut: ***** (when the PC displacement mode Symbol is selected).

REDO LABEL: (for both 68K and PowerPC mode) label in disassembly are more readable than just plain hexadecimal address. The debugger use local label for all PC reference that sometime need to be rebuild. Suppose by example that you debug an application. The label you are using in this application will probably not be good the next time you debug the application since the application will probably not be loaded at the same address in memory (or some instruction are added or deleted, and PC reference no longer exactly match the label used previously). To rebuild the labell buffer just click in the REDO LABEL item, a wait until the dialog “**Now working, please wait...**” diappear.

Shortcut: ***** (when the PC displacement mode Label is selected).

Rebuilding the address of the label takes only a fraction of seconds (unless you use a very large label buffer), so don't take any chance: use the key '*****' as soon as you suspect your code to be modified or moved...

hexa add: set PC displacement mode as hexadecimal address. PC reference are given with plain hexadecimal address. (For both 68K and PowerPC mode)

Shortcut: **/** (select the next PC displacement mode).

pc+$24: set PC displacement mode as PC offset. PC reference are given with PC symbol ‘*’ plus an offset to PC. (For both 68K and PowerPC mode)

Shortcut: **/** (select the next PC displacement mode).

local label: set PC displacement mode as local label. PC reference are given with local label. (For both 68K and PowerPC mode)

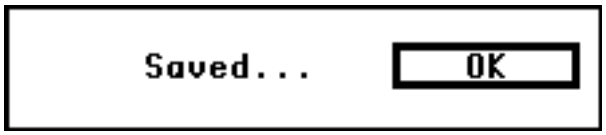
Shortcut: **/** (select the next PC displacement mode).

CFM symbol: for PowerPC: set PC displacement mode as CFM symbol. PC reference are given with shared libarary name plus offset. For 68K: set PC displacement mode as compiler symbol. PC reference are given with compiler routine name plus offset.

Shortcut: ***/*** (select the next PC displacement mode).

MISC menu

SAVE MON: To save the current menu and window setups for the next time.



Shortcut: **Command S**.

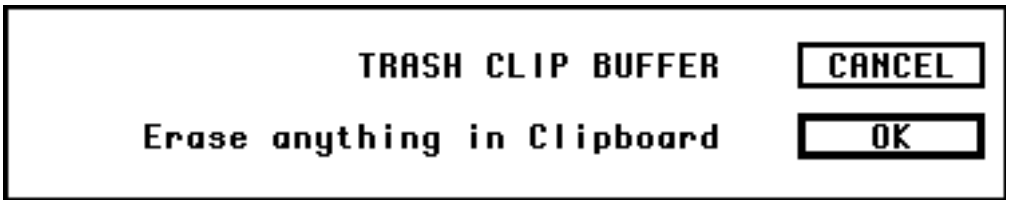
LOAD MON: To restore menu and window default setups or the setup previouly saved.

Shortcut: **Command L**.

CLIP ADR: put the address of the highlighted line into the clip buffer. If the clip buffer is full, the new address replace the last entry in the clip buffer.

Shortcut: **Command S**.

TRASH CLIP: erase all entries in the clip buffer. The dialog below make sure that you don’t make a mistake.



TEXT WRAP: a text window is viewed in two mode. In the first one, a long text line is broken (the text continues at the next row). With the second mode, only the part of the line that fit in the window is show (the remaining is still invisible, but you can scroll this line horizontally with the left and right arrow keys). This item switches from one mode to the other.

TEXT SCROLL: the up/down arrow icons scroll the text in a text window. Scrolling can be done one line at a time or more quickly, one page at a time. This item switches from one mode to the other.

The windows

If the content of a window cannot be displayed, because of a address error in some code window by example, ABZmonPPC leaves the window open and puts a warning in the message window. You must then close this window. Such windows are empty or filled with error message.

Usually, you can select a line by clicking at the start of this line. The line is then video inverted and the address involved by this line becomes the default address (this address appears in the edit file of most dialog boxes). To de-select the line you click another time the selected line.

An active window is identified by its black close box. A window becomes active when you click inside. The keyboard key "**.**" also make the window active. Repeatedly hit the "**.**" key to do a tour of all windows.

To move a window, just put the cursor in the title bar and drag as usual.

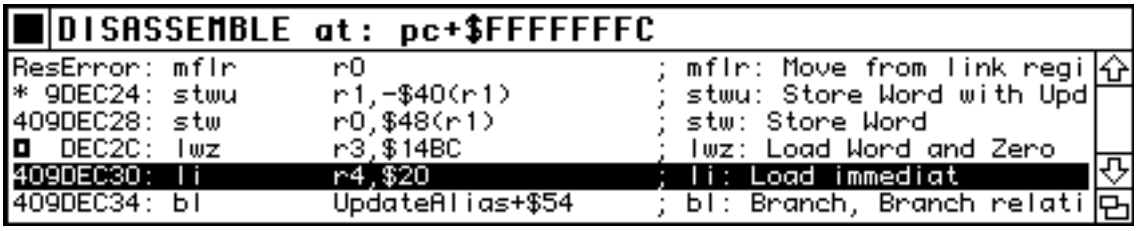
Most of the windows can shrink or grow. Put the cursor in the size box (the two rectangle icons in the right bottom corner) and drag as usual. The size of the window is computed to not broke a character in the middle. Some windows, like the register window, can shrink or grow in only one way.

Click on the arrow icons to scroll the text. If the scrolling is too fast use the keyboard **arrows up** and **down**.

When hit the '**%**' key the active window is continuoulsy refreshed. This is useful to check if some memory location is changed frequently. The cursor changes to a short arrow. By example open a dump window at the address $16A and hit the '**%**' key. The cursor changes to a short arrow and you can see the word at $16A (the timer Ticks) change very quickly. To stop it, press the '**%**' key another time.

Disassembler window

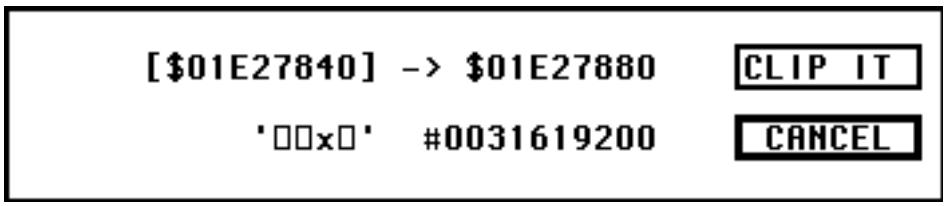
Show PowerPC mnemonic instructions.



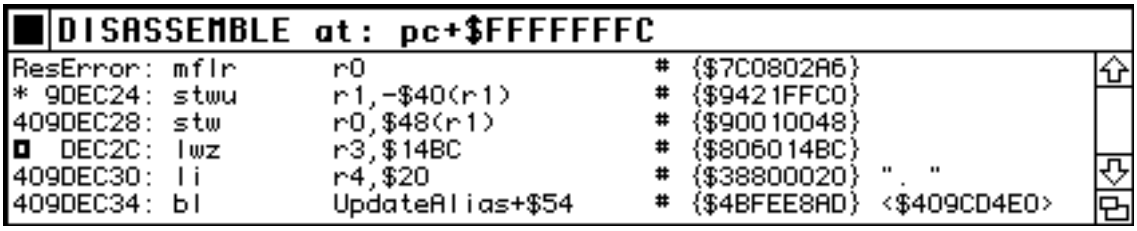
The left column is the address of the instruction disassembled, unless a local label or CFM symbol is used instead (here the symbol is: ResError). See **OPT DISAS menu** for more on it. Click on this column to select the line: the default is then the address of this line.

The next column column is the keyword instruction name. If you click in this column you get a dialog also showing the meaning of the keyword. In the dialog below, the third line keyword. The meaning of this keyword is also show in the last column of the window when first comment mode is selected.

The third column hold the instruction operand. If the mouse is hit on an operand, the operand info dialog is displayed. For integer register or immediat value the value is show in hexadecimal, in ascII and in decimal. For displacement the value is show in hexadecimal and in decimal. For foating point register, the value is show as both hexadecimal and decimal floating point. For register indirect with immediate index, the effective address is show between bracket, and the pointed value is show in hexadecimal, in ascII and in decimal.



The last column is the comment field. Two kind of comment are presented. The first one give the meaning of the instruction keyword, as in the code window above. The second one, as in the code window below, give the instruction opcode between brace, the ascii value of the immediat value operand between double quotes, and PC displacement between < >. Hit the **c** key to toggle beetwen the two mode.



The fourth line begins by a black box char to indicate a break point at this address. When the black box has a white, the break point is invalid. This happen when new code has erased the instruction under the break point (you must then clear the break point as soon as possible, see **BREAK menu**). If the line begins with the character *****, the address of the line is the value of the program counter PC.

The window title reflect the address of the first line in the window. Above, the address of the first line is $409DEC20. Since the PC is $409DEC24, the first line address is PC-4 = PC+$FFFFFFC. When the text scroll (click on the arrow icons or hit on the keyboard **arrows up** and **down**), the title change according to the address of the first line.

If the code is not word aligned in memory, hit the **left** or **right arrow**. The address of the first line decrease or increase by one.

To quickly open a dump window at the address of some line in code window: **Shift** click the line.

To quickly open a code window at the address of some line in code window: **Option** click the line.

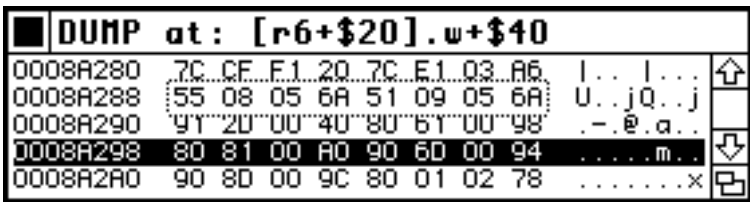
To quickly set a simple break point (see **BREAK menu**) at the address of some line in code window: **Command** click the line.

To quickly set a break point that remove itself (see **BREAK menu**) at the address of some line in code window: **Control** click the line.

When you click in the address area, at the beginning of the line, you select the line. The line is then video inverted and the address at the beginning of the line become the default address. In the first code window above the fifth line is selected and the default address si $409DE30.

Dump window

Show memory content as ascII and hexadecimal number.



The left column is the address of the first byte of line. The middle area is the hexadecimal value of each byte and the right area are the ascII character of each byte or a dot if the a such character doesn't exist. By example, the first byte of the second line is the content of memory at address $8A288. The byte value is $55 and the ascII character $55 is the upper case letter U.

The tiltle reflect the address of the first line in the window. The JB address **[r6+$20].w+ $40** mean this: form an address by adding $20 to the value of the register r6, take the word at this address and add $40. The result is the address of the first line. When the text scroll (click on the arrow icons or hit on the keyboard **arrows up** and **down**), the title change according to the address of the first line.

To scroll horizontally, hit the **left** or **right arrow**. The address of the first line decrease or increase by one.

You can change the content of the memory (RAM only) directly from within a dump window. To change the value of the memory you click inside the line, either in the hexadecimal area or ascII area. The area is then converted to an edit field and you would be able to change hexadecimal value or ascII character. Above, the hexadecimal area in the second line
is converted to an edit field. When an error occur, because an invalid hexadecimal digit or bad number of character in edit field, the mouse cursor change to an empty arrow and the text cursor goes near a faulty character. You must hit Return or Enter to validate the modification. The memory is changed only after the input keyboard of one of these two keys. If you change your mind and wish do not change the memory, simply click outside the window.

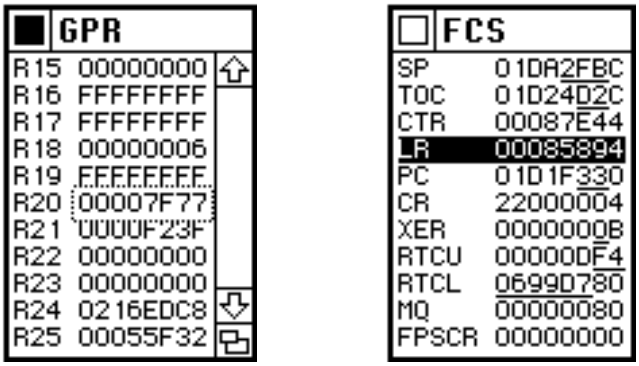
To quickly open a dump window at the address of some line in dump window: **Shift** click the line.

To quickly open a code window at the address of some line in dump window: **Option** click the line.

When you click in the address area, at the beginning of the line, you select the line. The line is then video inverted and the address at the beginning of the line become the default address. In the window above the fourth line is selected and the default address si $8A298.

Register window

Show the GPR registers values and other user level registers values



The first column hold the registers names, the second hold the registers values in hexadecimal. To point out the change in register, ABZmonPPC saves a copy of all registers. When the debugger is entered, a comparison is made between the old and new register values and the difference is underlined in the window. For example, the last two digit of the RTCU register in the **FCS** window above are changed since last time the debugger was enterred.

To select a line, click into the register name. The register value become the default address. In the **FCS** window above, the LR line is selected. The default address is then the value of the Link Register: $85894.

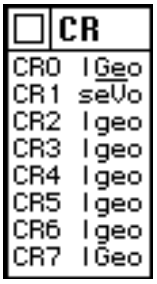
To change a register value, click into the the register hexadecimal value. The hexadecimal area is then converted to an edit field and you can change the register value. In the **GPR** window above the R20 register hexadecimal value is then converted to an edit field. When an error occur, because an invalid hexadecimal digit, the mouse cursor change to an empty arrow and the text cursor goes near a faulty character. You must hit Return or Enter to validate the modification. The register value is changed only after the input keyboard of one of these two keys. If you change your mind and wish do not change the register, simply click outside the window.

To quickly open a dump window for a register in window: **Shift** click the line of this register.

To quickly open a code window for a register in window: **Option** click the line of this register.

Condition register window

Show the Condition Register bits status of CRO to CR7.



When the letter is upper case the bit is set, else the bit is clear. Above the G bit of CR0 is set and the other bit are clear. The meaning of the letter are:

for CR1:

S: Floating-point exception (FX)
E: Floating-point enable exception (FEX)
V: Floating-point invalid exception (VX)
O: Floating-point overflow exception (OX)

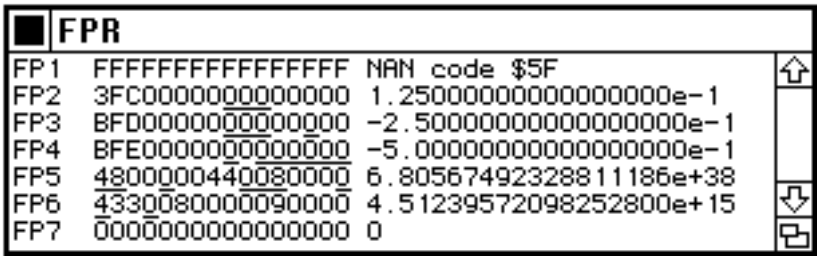
other:

L: Less than [negative] (LT)
G: Greatest than [positive] (GT)
E: Equal [zero] (EQ)
O: Summary overflow (SO)

To point out the change in CRn, ABZmonPPC saves a copy of CR. When the debugger is entered, a comparison is made between the old and new CR value and the difference is underlined in the window. For example, the **G** and **e** bit in the **CR** window above are changed since last time the debugger was enterred.

Floating-point register window

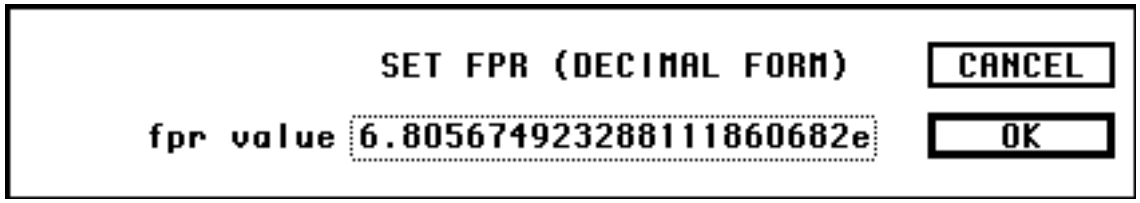
Show the value of floating point register as hexadecimal number and decimal floating-point (scientific notation).



The first column hold the register name, the second hold the register value in hexadecimal and the third the decimal floating-point form of the register value. To point out the change in register, ABZmonPPC saves a copy of all floating-point registers. When the debugger is entered, a comparison is made between the old and new floating-point register values and the difference is underlined in the window. For example, the first and fourth digit of the FP6 register in the **FPR** window above are changed since last time the debugger was enterred.

To change a floating-point register value, click in the hexadecimal area. A dialog appear (below) with the register hexadecimal value of the clicked line. When the change is done, click on the **OK** button to change the register value. When an error occur, because an invalid hexadecimal digit, the mouse cursor change to an empty arrow and the text cursor goes near a faulty character.

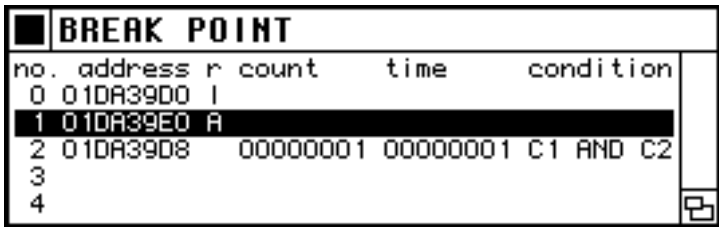
If the mouse button is hit while the cursor is on the floating-point register name or in the decimal floating-point area, a dialog similar to the previous one, but with decimal number instead of hexadecimal number, is show. Use the **OK** button to change the register when the modification is done. When an error occur, because an invalid character, the mouse cursor change to an empty arrow and the text cursor goes near a faulty character.



Here the value of the FP5 register is partially shown. The exponent (+38) is not visible because the dialog is not large enough. When this happen use the **left** or **right arrow key** to scroll horizontaly the text and uncover the invisble part.

Break point window

Show the when and how the break point are set.

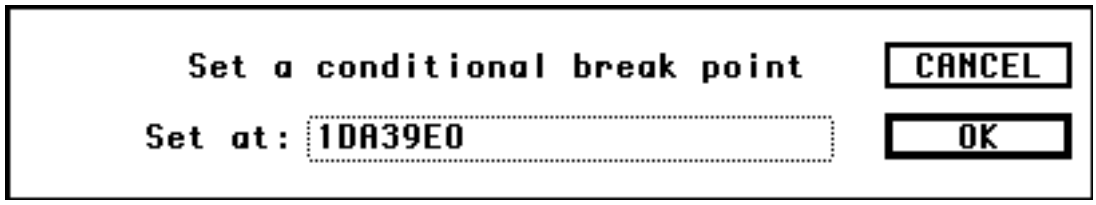


The first column in window is the break point number. The second is the address in memory where thebreak point is set. The third column hold a single character: **A** when the break point automatically remove by itself, **I** when the break point is invalid, and a blank space otherwise. The fifth column is the number of time the program must meet the break point before really stopping. Each time the program cross the break point, the number in the fourth column is decremented by one. When the count is 0, the program is stopped and the debugger is entered. The last column show which and how conditions are used. In the window above the break point no. 0 is an invalid simple break point. The break point no. 1 is a valid simple break point that remove itself. The break point no. 2 use two condition and the count is decremented when both condition are true. The count must be decremented only one time before stopping. So, the stop is done the first time the two condtions become true.

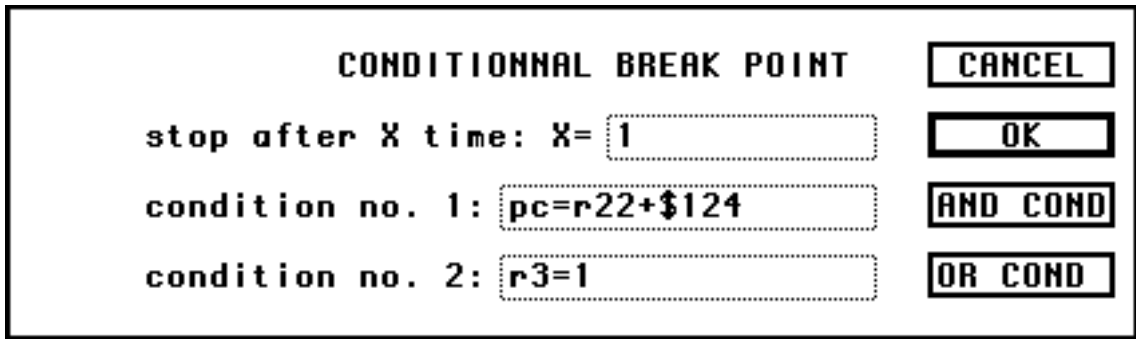
When a break point is not used the line is empty. See **BREAK menu** for more on condition and how to set break point.

To select a line, click into the break point number. The break point address become the default address. In the **BREAK POINT** window above, the break point no. 1 line is selected. The default address is then $1DA39E0.

To modify a break point click in the line of the break point after the column of break point number. To create a new break point, click in an empty line. A first dialog is used to set the break point address.



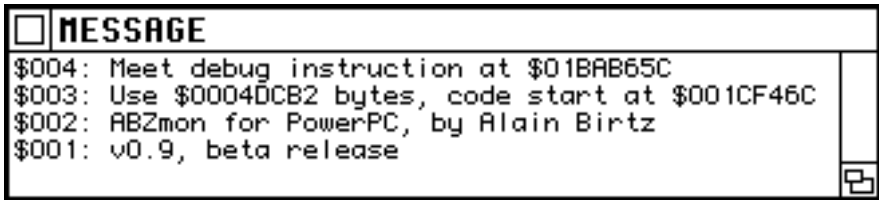
If the **OK** button is selected, a second dialog is used to modify or set the other parameter.



To set a simple break, by example, set the address in the first dialog and in the second dialog, set **X** to 1 and leave other field blank.

Message window

Display the ABZmonPPC warning/error/info messages.



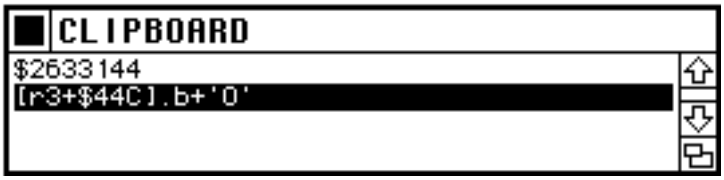
The messages are numbered from 1, in hexadecimal. The first message in window is always the most recent and then, has the highest no. The window can grow to uncover the older messages. There is no arrow icon in window, so you cannot scroll the message. When the maximum size of window is reached, the older messages are lost

(the message are stored in a cycle buffer that hold a maximum of 20 messages).

Only one message window can be opened at the same time. If the message window is already opened, then ABZmonPPC only make it frontmost window (active). If the message window is close and there is an incomming message, the debugger re-open the window, make it front and display the new message.

Clipboard window

Show the strings in the ABZmonPPC clipboard buffer.



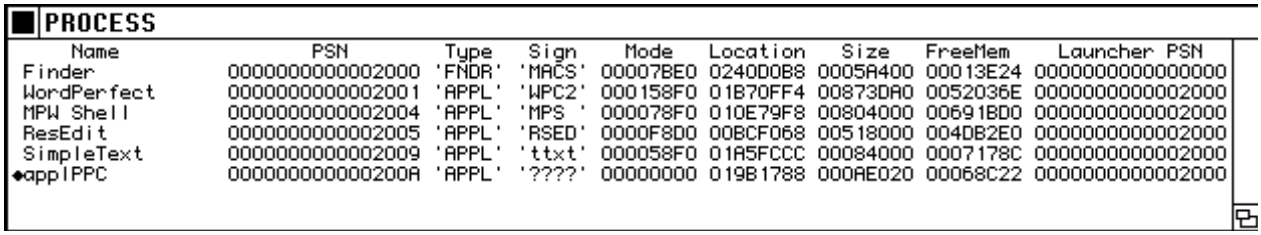
When a line is highlighted in some, the command **CLIP ADR** of the **MISC** menu (or the keystroke **Command C**) store the default address of this line into the ABZmonPPC clipboard. The button **CLIP** in the **Calculator** dialog store the edit string into the CLIP buffer. Every string in any edit field is stored into the clipboard by the keystroke **Command C**.

When a string is highlighted in the clipboard window, the keystroke **Command V** replace the edited string by the highlighted string .A dialog is open with either the highlighted string in the clipboard window or the default address.

When the clipboard buffer is full (40 entries), the new string replace the last stored string. The TRASH CLIP item the MISC menu erase all entries in the CLIP buffer.

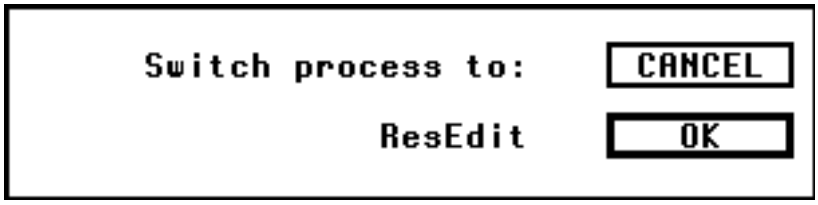
Process window

Show the current active process (application).



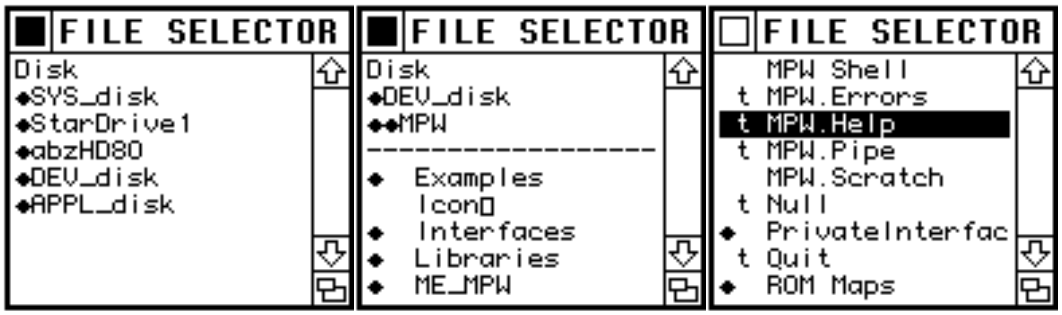
The first line in window hold the column title. The first column, **Name**, is the name of the application. The second column, **PSN**, is the process serial number. The third, **Type**, is the application type. The fourth, **Sign**, is the creator signature. The column **Mode** gives some information from the application "SIZE" resource. The starting address of the application code holds in **Location** column while the **Size** column gives the application size. **FreeMem** shows the amount of memory available for the application and **Laucher PSN,** the number of the parent application (generally the Finder). The current running application (here applPPC) is marked by a carret symbol just before the application name.

To leave the current application, and swich to an other application, click one time in the application line. Below, the fifth was selected, so the application ResEdit will be activate.



File selector window

These windows select a file (this is like the standard file selector box, in the Mac, you can see when you open a document).



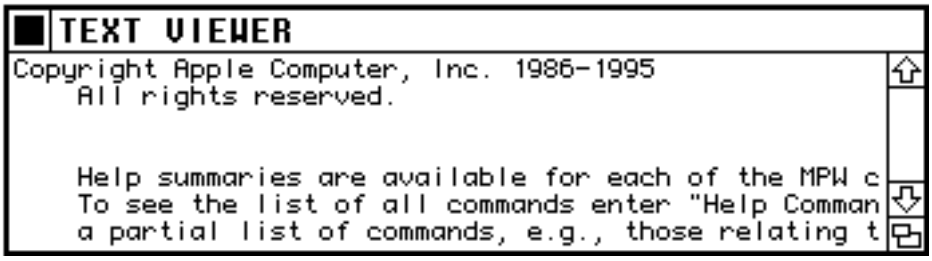
The first time you open a file selector window you see the list of all connected disks, including the one shared. Look at the first window above. The caret symbol tells the line is a disk name or a folder name. If you click on the disk DEV_disk name, the window show the root level of this disk. A folder in this disk is named MPW. A mouse click on the MPW name show the content of this folder, as in the middle window. This window displays, after the dashed line, the content of the second hierarchical file level of the disk DEV_disk. Scrolling up the window content let you see other file and folder at this level, like in the third window. A text document is identified by the the letter **t**, before the file name. To select a document, click anywhere in the document line.

Briefly, the lines before the dashed lines give the folders' hierarchy while the lines after the dashed lines give the content of the last folder in the hierarchy. You click on the lines **after** the dashed lines to select a document or to open the folder (this folder moves before the dashed line and becomes the last folder in the hierarchy). You click on a line **before** the dashed lines, in the folders' hierarchy, to open (after the dashed line) the folder or the disk of this line. For example, to return to the root level, as in the first window, click on the first line of the second window, named Disk.

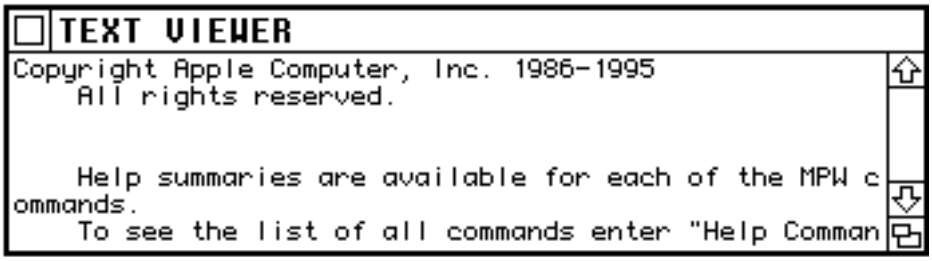
When a document line is selected, like MPW.Help in the third window, you can read this document. Go on the **OPEN** menu and select **TEXTVIEW**.

Text viewer window

Show the content of the selected document in a **FILE SELECTOR** window.



In the window above you cannot see the fifth line entirely becouse the window is not large enough. Go to the MISC menu and select the TEXT WRAP item. The text look now like the window above.

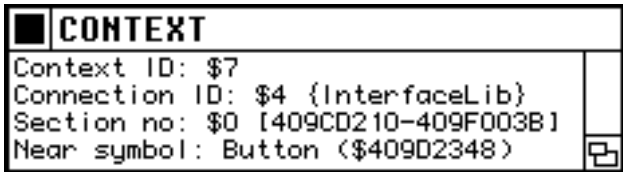


You can also scroll horizontally the first line of the window with the left and right arrow keys.

Note: all documents can be viewed, not only text window. But non-text hold many garbage characters show as empty rectangle, and so are meaningless outside text section.

Context window

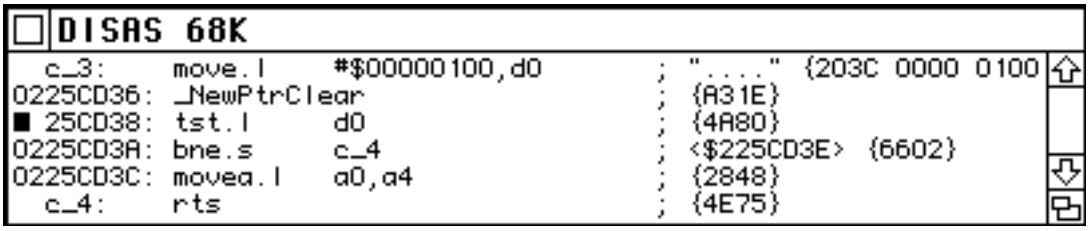
Give context information as defined by Code Fragment Manager.



The first line is the content ID of the code to be executed. The second line show the connection ID and connection name. The third line give the section number and where this section is located in memory. The last line is the name of the procedure and the address of start of the procedure code.

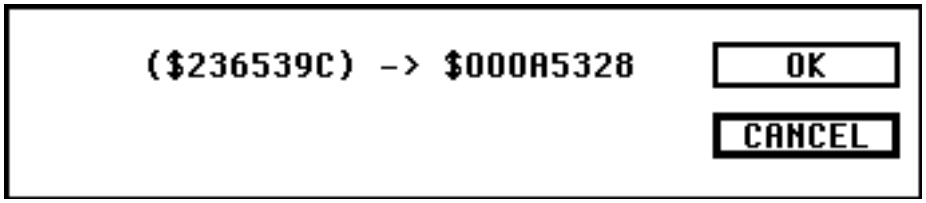
68K DIS window

Show 68K mnemonic instructions.



The left column is the address of the instruction disassembled, unless a local label or compiler symbol is used instead (here the lavel are c_3 and c_4). See **OPT DISAS menu** for more on it. Click on this column to select the line: the default is then the address of this line.

The next column column is the keyword instruction name. The third column hold the instruction operand. If the mouse is hit on an operand, the operand info dialog is displayed. By example, the dialog below is prompted when the mouse hit the operand in the line **pea -4(a5)**. $236539C is the value of **a5** minus 4 and $A5328 is the value at the address $236539C



The last column is the comment field. The second one, as in the code window below,

give the instruction opcode between brace, the ascii value of the immediat value operand between double quotes, and PC displacement between < >.

The third line begins by a black box char to indicate a break point at this address. When the black box has a white, the break point is invalid. This happen when new code has erased the instruction under the break point (you must then clear the break point as soon as possible, see **BREAK menu**). If the line begins with the character *****, the address of the line is the value of the program counter PC.

If the window is openned at PC (see **68K DIS** in the **Open menu**) the first line of the window is always the instruction code at the PC address. This line is updated each time the PC value change. There is not scroll bar for this window. The other kind of 68K code window is not updated when the PC change, but have a scroll bar. If the code is not word aligned in memory, hit the **left** or **right arrow**. The address of the first line decrease or increase by one.

To quickly open a dump window at the address of some line in code window: **Shift** click the line.

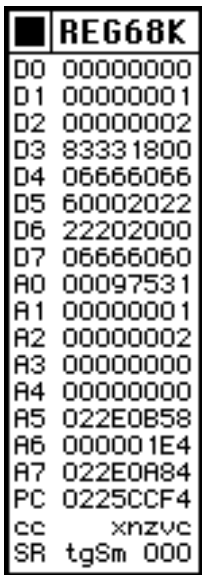
To quickly open a code window at the address of some line in code window: **Option** click the line.

To quickly set a simple break point (see **BREAK menu**) at the address of some line in code window: **Command** click the line.

When you click in the address area, at the beginning of the line, you select the line. The line is then video inverted and the address at the beginning of the line become the default address.

68K REG window

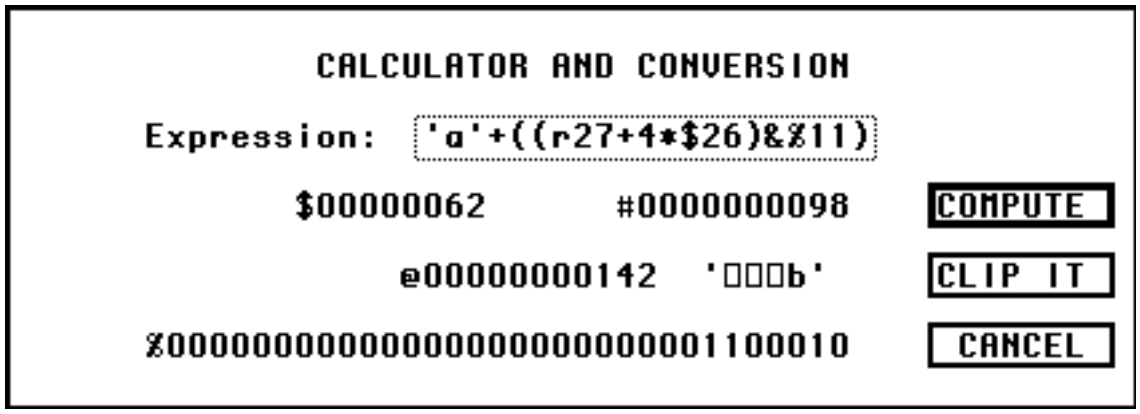
Show the value of the 68K register D0 to D7, A0 to A7, PC, CCR and SR



The last two row show the bit value of the SR. The cc' row show the low byte of the SR (the CCR). The SR row show the high byte of SR. When the bit is set, the corresponding letter is upper case. When the bit is clear, the letter is lower case.

The calculator

The calculator is a dialog box that makes calculations and conversions in hexadecimal, decimal, octal, binary numbers and in ascII characters. To call the calculator, simply hit the key "**=**".



In the Expression edit field you enter the expression to compute. The calculation is done when you hit the **Return** or **Enter** key or the **COMPUTE** button. When an input error occurs, the cursor changes to an empty arrow and goes near an offending character. The **CLIP IT** button adds the computed value into the CLIP buffer. The computed value is also kept in memory, and used as a default value for the calculator when there is no line selected in a window. Use the **CANCEL** button to close the calculator.

The calculator shows the five conversions in the dialog box using a leading symbol to point out the kind of number. The ascII conversion, between double quotes, uses empty rectangle for unprintable character.

As an input, the calculator accepts hexadecimal numbers (with leading $), decimal numbers (without leading #), octal numbers (with leading @) and binary numbers (with leading %). A number without leading is assumed in base 16. This default value can be changed in the **S_UP** variable **Default base number**. The calculator also accepts numbers in ascII forms, like "abCd" or 'zx' between single or double quotes, of one to four letters. Each letter counts for a byte in a long integer (four bytes). In the last two examples the long integer values are $61624364 and $00007A78.

The calculator uses the following operator, given in ascending priority order, as defined by the C language:

- ~	minus unitary and biwise not
/ *	division and multiplication
+ -	addition et substraction
<< >>	left and right logical bitwise shift
&	bitwise 'and'
^	bitwise 'xor'
|	bitwise 'or'
()	parenthesis

When the calculator is opened, the default value is the last value used by the calculator, unless a line is selected in any window. In this case the default address for this line becomes the default value for the calculator. To make a conversion of a register, for example, just select the register and press the key **'='**.

The JB addresses

JB addresses are addresses given with register indirections and displacements. There are many forms accepted by the debugger. The general form is:

[Rn+disp1].s+disp2

where **Rn** is a PowerPC register (R0-R31, SP, PC, LR, CR, TOC, CTR, XER, MSR), **disp1** and **disp2**, the displacements and **.s** the size of the element in memory (.b -> byte, .h -> half word, .w -> word). If .s is not used, the size word is used by default. **Rn**, **disp1** and **disp2** are optional.

Here are some examples of JB addresses:

[PC+5000E]+20

if PC = 20000 and the value of the word integer at the address 7000E (it was 20000+5000E) is 44444, then the JB address is 44464 (it was 44444+20)

[22222].b-20

if the byte at the address 22222 is 84 then the JB address is 64 (it was 84-20)

[R6]

if R6 = 44444 and if the word integer at this address is 12345 then the JB address is 12345

SP+20

if the stack pointer holds 66666 then the JB address is 66686 (it was 66666+20)

R4

the JB address is the address of the R4 register

E3678

the JB address is E3678

The JB conditions

JB conditions are expressions holding two JB addresses. The comparisons are done on non-signed integers only. The general form is:

JB1 op JB2

where **JB1** and **JB2** are JB addresses and **op** one of the operators:

<	>	strictly smaller, greater
<=	=<	smaller or equal
>=	=>	greater or equal
=		equal
<>	><	not equal

Here are some examples:

R5>6

To respect this condition, the R5 register value must be strictly greater than 6.

[22222].w=[22224].w

To respect this condition, the word (4 bytes) at the address 22222 must be the same as the one at the address 22224.

[PC].w<=C0000000

To respect this condition, the current instruction must be a TWI instruction (Trap Word Immediate).

The RTS command

The RTS stand for ReTurn from Subroutine. RTS is an opcode for MC68000 microprocessor. In 68000 assembly a subroutine is called by BSR and JSR instructions. The subroutine terminate by an RTS instruction. The RTS instruction tell the microprocessor to resume at the instruction after the BSR or JSR.

In PowerPC assembly a subroutine is called by branch instructions BL, BCL, BCCTRL or BCLRL. The L at the the end of these opcodes tell the microprocessor to save the return address, that is the address after of the instruction after the branch instruction, into the link register LR. At the end of the subroutine, the value store in LR is used to resume at the instruction after the branch instruction. The RTS command do exactly the same thing. When you give a RTS command, the debugger execute the remainning of the current subroutine and stop just after the branch instruction that called the subroutine.

A program is a routine that call a subroutine that itself call an inner subroutine, etc. When the debugger is enterred, the PC is probably very deep in the hierarchy of subroutines. To return to an higher level you cannot just send many RTS command. The first RTS command work correctly. The debugger stop after the caller instruction. But the second RTS command re-execute the same subroutine, and this is surely not what you wish. To understand what is happenning (and what to do to execute the higher level subroutine) we must remember how a subroutine is called in PowerPC assembly. The PowerPC architecture define a standard way to call a procedure. The code below show how a program call the a subroutine
named SUBROUTINE_A.

```
                bl      SUBROUTINE_A        # save address of nop
                                            instructiom into LR and

        branch to SUBROUTINE_A

                nop                         # instruction that do
                                            nothing

                ...                         # many other instructions

SUBROUTINE_A:   ...                         # instructions for the
                                            subroutine
                blr                         # branch to nop instruction
```

The last instruction of the subroutine, BLR, the microprocessor to branch at the instruction pointed by the value of LR. The value of LR is the address of the NOP, saved at the BL instruction. Now what to do, if inside SUBROUTINE_A, we need to call an other subroutine? We cannot simply use an other BL instruction, since the current of LR, the return address of SUBROUTINE_A, will be loose. The mechanism defined in PowerPC architecture to call many subroutine hold in two very small piece of code called **prolog** and **epilog**.

At the beginning of the procedure, the prolog save the value of the LR in stack. At the end of procedure the epilog restore the LR value. The return address is never loose. The code below show what happen when the subroutine SUBROUTINE_A call itself an other subroutine SUBROUTINE_B.

```
                bl      SUBROUTINE_A        # save address of nop
                                            instructiom into LR and

        branch to SUBROUTINE_A

                nop                         # instruction that do
                                            nothing

                ...                         # many other instructions

SUBROUTINE_A:   mflr    r0
                stwu    r0,-4(sp)           # save LR
                stwu    sp,-56(sp)          # stack space needed
                                            for  cross-TOC call
                bl      .SUBROUTINE_B
                nop

                lwz     r0,56(sp)
                mtlr    r0                  # restore LR
                addi    sp,sp,4+56          # clean stack

                ...                         # instructions for the
                                            subroutine
                blr                         # branch to second nop

SUBROUTINE_B:   ...                         # instructions for the
                                            subroutine
                blr                         # branch to first nop
```

As you see, before to call SUBROUTINE_B, the subroutine SUBROUTINE_A save the LR in the stack (with other value like the TOC register for cross-TOC call). After the execution of SUBROUTINE_B, the saved LR is restored and SUBROUTINE_A can return correctly.

Suppose now that inside SUBROUTINE_B you send a RTS command. The debugger execute the ramainning of the subroutine and resume at the second NOP (inside SUBROUTINE_A). For a short peroid of time, namely, before the 3 instruction after the BL .SUBROUTINE_B instruction (NOP, LWZ and MTLR), the LR is a scrap register. You cannot send an other RTS command. But after the MTLR instruction, the LR hold the return address of SUBROUTINE_A, it is safe to send the second RTS command.

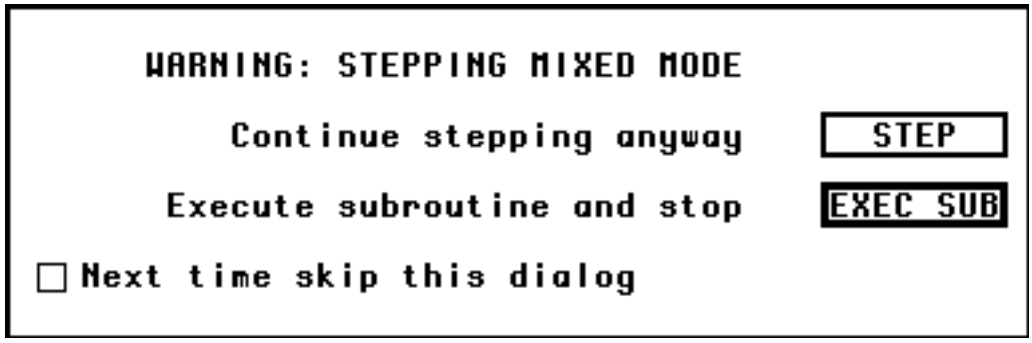
What you have to do to exit SUBROUTINE_A when you are SUBROUTINE_B is send a first RTS command, then trace the 3 instructions for which the LR is not a valid return address. As soon as the LR change, it is safe to send an other RTS command. And you can repeat to jump to return to the procedure that call SUBROUTINE_A itself. Trace the 3 next instructions (until the LR change value) then send an other RTS command.

For some procedure, it may have more than 3 instructions to trace. Also some compiler may use different epilog code. So you can have to trace 4 or r instructions until the LR was loaded with a new (valid return) address. But as soon as the LR change, it is safe to send the RTS command.

To send an RTS command, use the **RTS** item in the **Control** menu. Or just it the **r** key on the keyboard.

The peril of Trace in MixedMode

The MixedMode manager is invoked when a procedure in 68K is to be executed by the PowerPC processor. Two Mac OS procedure are called to translate the 68K code into PowerPC code: **CallUniversalProc** and **CallOSTrapUniversalProc**. These two procedures soon call the **twi** instruction to emulate 68K code. The MacOS exception handler seem have trouble to trace correctly the **twi** instruction or at least, tracing in MixedMode. A way to circonvent the problem is to never let the user enter in the MixedMode code (the MacsBug way!). ABZmonPPC let you enter in MixedMode, but warn you before. When you are tracing **CallUniversalProc** and **CallOSTrapUniversalProc**, the debugger show you the following dialog.



The **EXEC SUB** execute the subroutine (sending a **RTS** command to the debugger) and resume into ABZmonPPC at the end of the procedure.
With the **STEP** you continue to stepping as usual. Be warn that some procedure into the MixedMode enter then in infinite loop and never exit. The **Next time skip this dialog** check box must be turned on if you wish not to see this dialog again. In this case the count in **N STEP AT** (in the **TRACE** menu), **TWO CONDITION** in (**STOP** menu), or in Conditionnal Break point are not updated. If the check box is off and the debugger is in trace mode (when the ABZmonPPC is not shown), the current program is stopped and the debugger is entered:

Any further trace command will then redisplay the **WARNING** dialog above.

DebugNum

This procedure works like **Debugger** or **DebugStr**, but instead of stopping the program and call the debugger just after this instruction, **DebugNum** stops the program only after a number of time. **DebugNum** needs three parameters. The first, called '**count**', is a word integer. If **count**=0, the stop occurs the first DebugNum is meet. If **count**=1, the first time DebugNum is meet, nothing happen. The stop occurs only the next time. Each time the PC is at a **DebugNum** instruction, the **count** parameter is decrement by one. When **count** is 0, the debugger is called. The second parameter, called '**Number**', is a word, between 0 and 31, identifying the **DebugNum** call. You can then use up to 32 different **DebugNum**s. The third parameter, '**pString**' is a pointer to a Pascal style string. This string is prompted by debugger when **count** is 0.

The assembly code to call this procedure is

```
############

        li          r3,7                        # count
        stwu        r3,-4(sp)
        li          r3,6                        # DebugNum no. -> 'Number'
        stwu        r3,-4(sp)
        lwz         r3,test_str[TC](rtoc) # the string to show -> 'pString'
        stwu        r3,-4(sp)
        _DebugNum

############

        tc          test_str[TC],test_str_[RO]
        csect       test_str_[RO]

        string  Pstring

        dc.b        'A Pascal style string example'

        align   3

############
```

Re-initialisation of DebugNum

ABZmonPPC holds in memory 32 word integers, associated to the 32 **DebugNum**s. These 32 integers hold the number of time the **DebugNum**s are meet.When a word integer match the '**count**' parameter (previouly we have say that '**count**' is 0) ABZmonPPC stops the program, else the word integer is increased by 1.

Suppose that your program ends before the word integer match the '**count**' parameter. The next time you will use **DebugNum** (for the same '**number**', after having rebuilt your program, by example) ABZmonPPC will continue to use the previous value of the 32 word integers. If you test a new version of your program you may wish use a fresh copy of one or all of these 32 word integers. In this case go to the STOP menu and click into the CLEAR DEBUGNUM item. The dialog prompted allow you to reset one ar all **DebugNum**s.

ABZmonPPC special keys

command '**d**' key -> open disassemble window
command '**m**' key -> open memory dump window
command '**r**' key -> open GPR register winfow
command '**b**' key -> set a simple break point
command '**s**' key -> save current window setup and ABZmonPPC variable
command '**l**' key -> reload current window setup and ABZmonPPC variable
command '**f**' key -> search ascII partern
command '**g**' key -> search next occurrence
command '**-**' key -> open stat window
command '**6**' key -> open 68K register window
command '**c**' key -> copy highlighted address to (ABZmonPPC) clipboard
command '**0**' key (zero) -> Enter to the other debugger (MacsBug ?)
option ESC key -> re-initialise mouse

'**up arrow**' key -> scroll down
'**down arrow**' key -> scroll up
'**left arrow**' key -> scroll left
'**right arrow**' key -> scroll right
ESC key -> show user screen

The following key are are upper/lower case unsensitive

'**g**' key -> return to user program (go)
'**q**' key -> return to user program (quit)
'**s**' key -> step one instruction
'**e**' key -> ExitToShell (cancel current process)
'**=**' key -> calculator dialog
'*****' key -> reset the lable/symbol
'**/**' key -> change disassembled displacement form: hexa address, *+$24, local label, CFM symbol
'**c**' key -> toggle comment form in disassemble window
'**?**' key -> help window
'**.**' key -> use repeatedly to do a tour of all windows
'**%**' key -> for the current window to redraw continiously
'**r**' key -> do a PowerPC RTS

To simulate the mouse moves and clicks use theses keys

option '**left arrow**' simulate mouse left move
option '**right arrow**' simulate mouse right move
option '**up arrow**' simulate mouse up move
option '**down arrow**' simulate mouse down move
command-option '**left arrow**' mouse left move and button down
command-option '**right arrow**' mouse right move and button down
command-option '**up arrow**' mouse up move and button down
command-option '**down arrow**' mouse down move and button down
command-shift '**3**' key -> take screen snapshot

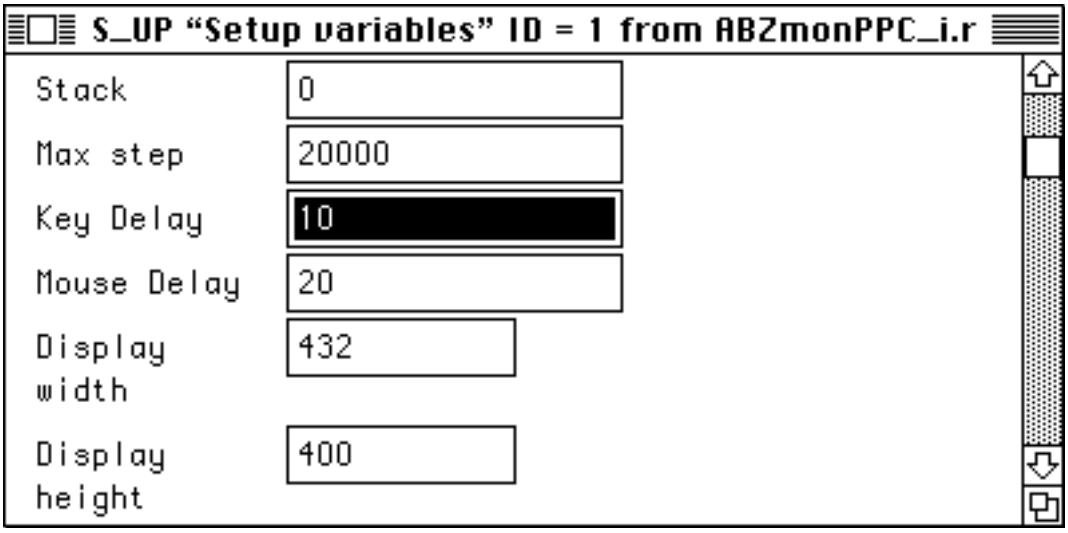
Special key used in edit field only (dialog and dump/register window)

'**Tab**' key -> go to next edit field
'**Return**' key -> simulate default button
'**Enter**' key -> simulate default button
'**Esc**' key ed_exit -> simulate default button
'**left arrow**' key -> move left one char the text cursor
'**right arrow**' key -> move right one char the text cursor
'**backspace key**' -> backward delete
'**delete**' key -> forward delete
command 'c' key -> copy to (ABZmonPPC) clipboard
command 'v' key -> paste to (ABZmonPPC) clipboard

Internal variables (PowerPC)

These variables are used to change some ABZmonPPC features like the video monitor where the debugger appears, the ABZmonPPC screen size, the look of the code window, the size of the font used in the menus, etc. Some variables must be manipulated with caution, like the one in the exception table.

You change these variables with ResEdit. Open the S_UP number 1. The window show the name of the variables and their default values:



Here is the list of the variables and their meanings:

The first 16 are the exception handle by the Mac OS. When a bit is set (the value 1 is selected) the exception is signaled by the debugger. Otherwise the ABZmonPPC ignore the exception and let Mac OS deal with it (generate a System error message).

Unknown: This exception code is defined for completeness only

Illegal: The processor attempted to decode an instruction that is either illegal or unimplemented (ABZmonPPC use it, leave it to 1)

Trap: The processor decoded a trap type instruction that is not used by the system software (ABZmonPPC use it, leave it to 1)

Access: A memory reference resulted in a page fault because the physical address is not accessible

Unmapped: A memory reference was made to an address that is unmapped

Excluded: A memory reference was made to an excluded address'

Read-only: A memory reference was made to an address that cannot be written to

Page: A memory reference resulted in a page fault that could not be resolved

Privilege: The processor decoded a privileged instruction but was not executing in the privileged mode

Trace: This exception is used by debuggers to support single-step operations

(ABZmonPPC use it, leave it to 1)

Brk. instr: This exception is used by debuggers to support breakpoint operations

Brk. data: This exception is used by debuggers to support breakpoint operations

Integer: This exception is not used by PowerPC processors

Floating: The floating-point processor has exceptions enabled and an exception has occurred'

Stack overf: The stack limits have been exceeded and the stack cannot be expanded

Termination: The task is being terminated

Stack: the internal ABZmonPPC stack size. Not already implented. Leave it to 0

Max step: the maxmum number of step to do in trace mode before the debugger appear. A value 0 mean that ABZmonPPC never stop the trace until the number of step specified by **n-step** command is reached or the condstions come true in **conditional step** command. Since trace mode is very slow, this is a good protection againt waiting for nothing

Key Delay: delay to repeat the key when the key is still down.

Mouse Delay: delay to accept a second click of the mouse.

Display width: the width of the ABZmonPPC screen. Never let this value smaller than 432. Some dialog boxes have this width.

Display height: the height of the ABZmonPPC screen. Never let this value smaller than 240. Some dialog boxes have this height.

Display X: the X coordinate of the left upper of ABZmonPPC screen. A value of 32 puts the left upper of ABZmonPPC screen at 32 pixels of the left side of the video monitor.

Display Y: the Y coordinate of the left upper of ABZmonPPC screen. A value of 40 puts the left upper of ABZmonPPC screen at 40 pixels of the upper side of the video monitor.

Maximum number of window: the maximum number of windows that can be opened at the same time in the debugger.

Use big font in Menu window: if 0, the text in menus appear with small font and if -1, a bigger font is used instead.

Use big font in Message window: if 0,the text in message window appear with small font and if -1, a bigger font is used instead.

Don't report OS event: a source of trouble occurs if the OS update window or cursor when the ABZmonPPC screen is displayed. One way to circumvent is to clear the low memory variable SysEvtMask ($144). No event is then reported. When the bit is 0, the debugger clears SysEvtMask when ABZmonPPC screen is displayed and restores SysEvtMask when it returns. If the bit is 1, the debugger doesn't change the low memory variable. Used only when 'Low level input' bit is set.

Mouse re-center: the Mac OS don't report the mouse move outside the rectangle screen. When the bit is set to 1, the debugger reset the mouse to the center when the mouse is near the rectangle border. The ABZmonPPC cursor can then move in any direction. Used only when 'Low level input' bit is set.

Mouse not coupled: when the low memory variable CrsrCouple ($8CF) is set the cursor moves according to the mouse movement. Since ABZmonPPC uses its own cursor, there is no need for the mouse to be coupled with the Mac cursor. If the bit is 1 the debugger clears CrsrCouple when ABZmonPPC screen is displayed and sets CrsrCouple when it returns. In this way the Mac cursor is not moved. You can clear the bit if the Hide/Show cursor bit is 1. Used only when 'Low level input' bit is set.

Hide/Show cursor: the QuickDraw cursor is handle at the interrupt time. Sometimes, it happens that the cursor is changing when ABZmonPPC takes control of the screen. Back to the program, there will be a mix-up of the old and the new cursor image. The result is garbage on the screen where cursor sits. If the bit is 1, ABZmonPPC corrects it automatically. If you work on routine modifying the cursor, let this bit to 0. ABZmonPPC will not interfere.

Use wildcard in Search: the searches (issued of the SEARCH menu) are done using the wildcard symbol if this variable is -1; if the variable is 0, ABZmonPPC does not use wildcards.

Wildcard symbol in Search: the wildcard symbol used for the search. The usual symbols are '*****' and '**?**'

Case sensitive in Search: the searches are done by distinguishing upper and lower case letters when this variable is -1. If the variable is 0, ABZmonPPC does not make the difference between upper and lower case letters.

The following variables determine the look of the code windows.

Tab length: the fields in a line of code are separated by tabs of length given by this variable.

Disp. form: sets the form (by default) of the operand displacement:
0 -> $1234,(pc)
1 -> *+32
2 -> local_label+22
3 -> CFM_symbol+22.

Hex prefix is Ox: if 0 the hexadecimal number has a leading symbol $, else the C style prefix 0x is used

Default number is decimal: if 0 the default base number in disasembly is hexadecimal, else the base number 10 is used.

Add first comment: if non-zero the the hexadecimal code of the instruction, the branch address and the immediat value of the instruction are added at the the end the disassembled line.

Add second comment: if non-zero the the meaning of the instruction mnemonic is added at the the end the disassembled line.

Use simplified: code instruction are translated to simplified mnemonic when this variable is non-zero. Otherwise standard mnemonic are used

Add instruction adr: when non-zero the instruction address in memory is show at the start of the disassembled line.

Add instruction code: when non-zero the instruction code is show in the first comment (see above).

Two blank before label: two blank carracter are added before the label name, when this variable is non-zero. This help to locate the lable in the code window.

Only 601: when non-zero, only PowerPC 601 instruction is disassembled, not the 620 instruction.

PC (disp) symbol: this symbol replace the address of the instruction for branch target address.

Comment 1 symbol: to mark the beginning of the first comment

Comment 2 symbol: to mark the beginning of the second comment

Register name set: 1 -> lower case name, 2-> uppercase name, 3-> upper case with SP and RTOC standing for r1 and r2

Local label buffer size: the size of buffer to store the local label and their addresses.

NB instr. between 2 label: estimated number of instruction between two label.

Percentage after central instruction: label are taken before and after the central instruction. the value 50 in this variable mean there are as much label before and after central instruction.

CFM symbols buffer size: the size of buffer to store the CFM symbols info.

Mon comparison segment length: the maximum length of the segment used in ARRAY COMP of MON SPY menu.

Step comparison segment length: the maximum length of the segment used in ARRAY COMP of STEP SPY menu.

Default number base: the number base used in edit field, for number shown without leading (base) symbol ($, #, @, %)

The next 6 variables set the debugger screen color. The foreground value 30000, -1, -1 and the background value 0, 0, 0 show the ABZmon image in blue with black character.

Foreground Red: the debugger screen foreground red depth.

Foreground Green: the debugger screen foreground green depth.

Foreground Blue: the debugger screen foreground blue depth.

Background Red: the debugger screen background red depth.

Background Green: the debugger screen background green depth.

Background Blue: the debugger screen background blue depth.

Turn to BW to display image: when non-zero, ABZmonPPC switch the screen resolution to 1 bit mode, and then display the debugger image. This method completly bypass the Quickdraw. With zero value, ABZmonPPC use CopyBits procedure to show the color debugger image.

Save screen image: The screen image under the ABZmonPPC screen is saved if this variable is set to -1. If you use only one video monitor you must use this variable value.

If you use a second monitor, it is not necessary to keep this image, you save a bit of memory and more, you see the ABZmonPPC screen even after the debugger has quit.

Screen monitor no: if you have more than one video monitors connected to your computer, you can choose the monitor for the ABZmonPPC screen. If you set the value of this variable to 0, the ABZmonPPC screen appears on the main video monitor . The first, second, third... video monitors are selected for the value 1, 2, 3...

RowBytes adjustment: internal use. Leave it to 0

Get BW pixel map from slot: the pixel map is copied directly from the video slot when this bit is set. Otherwise the pixel map is get from the
GDevice for the selected video monitor. Keep it to 0 unless you get some problem with not standard video monitor.

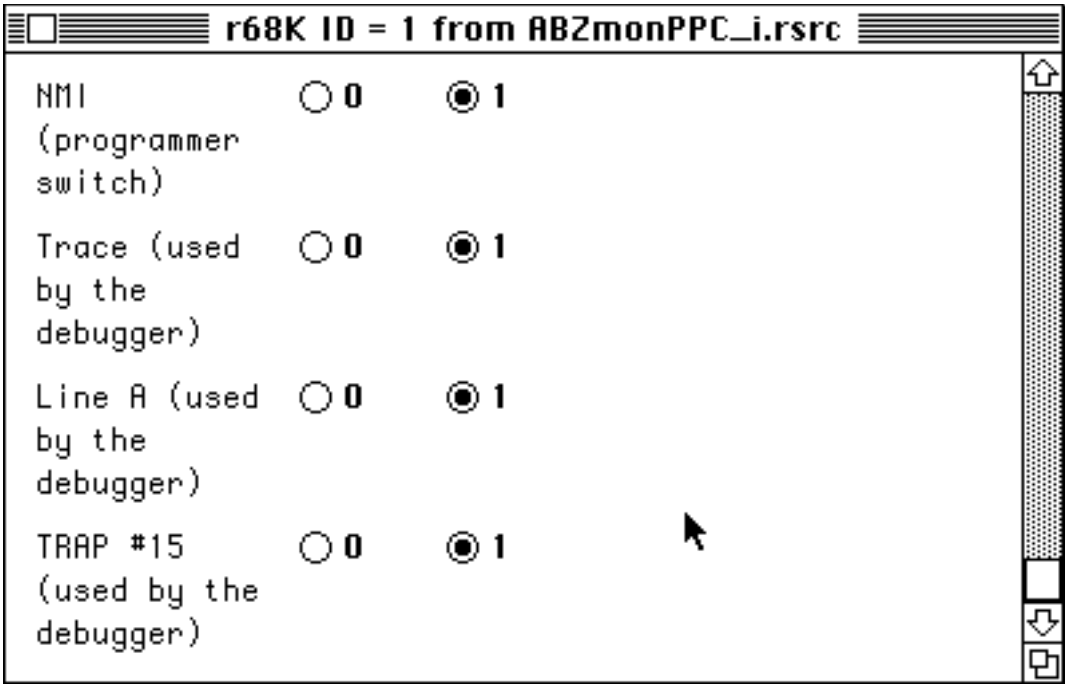
Text scroll one page: in a text window, the scrolling is done one page at a time (window size), if the variable is -1 and one line at a time, if the variable is 0.

Text window wrap: in a text window, a line too long to fit in the window is broken and continues on the next row when this variable is set to -1. If the variable is 0, the remaining of the line is not shown.

Internal variables (68K)

These variables are used to set the 68K exceptions that ABZmonPPC must use as 68K debugger. If the exception is set to 1, the exception is handle by ABAmonPPC, otherwise the exception is handle by the system or by an other debugger loaded before ABZmonPPC (like MacBug)

You change these variables with ResEdit. Open the r68K number 1. The window show the name of the variables and their default values:



The exception listed in this resource are the most frequently involved by programming error. You can set all of them when the virtual memory is not not active. When the virtual memory is used, you must clear 'Bus error' exception, since this vector is used during memory page swap.

ABZmonPPC need some vector set to work correctly. The TRAP #15 exception is needed to set 68K break point. The Line A exception is needed to enter to the debugger by calling _Debugger or _DebugStr. Stepping is done via the Trace exception, and Line A is also used to step Mac OS toolbox procedure. NMI exception must be set if you want use the hardware programmer switch button on some Mac.

Bug Report

ABZmonPPC has been tested on PowerMac 6100, Performa 6200 and Quadra 950 with Apple PowerPC 610 upgrade card. If you cannot correctly install ABZmonPPC or if you find any bug, please, leave me a message (the kind of computer you use, when this bug appears,...).

CompuServe: [72467,2770]
Internet: 72467.2770@compuserve.com

Alain Birtz
650 Grand St-Charles,
St-Paul d’Abbotsford
P.Q., Canada, J0E-1A0

Table of contents