

NiceMon

National COP8 Flash Debug Monitor

Table of Contents

<u>1. Overview</u>	
<u>2. Installation</u>	
<u>3. Command Line Options</u>	
--object.....	
--script.....	
<u>4. User Interface</u>	
<u>Menu Bar</u>	
File.....	
Debug.....	
Watch.....	
Configure.....	
Help.....	
Tool Bar.....	
Status Bar.....	
Main View.....	
Watch Area.....	
Console Area.....	
<u>5. Accommodating NiceMon</u>	
<u>6. Using Console Commands & Scripts</u>	
<u>7. Tutorial</u>	
<u>Writing the firmware</u>	
staying configured for NiceMon.....	
configuring to debug with interrupts.....	
A debugging session.....	
<u>I. Console Commands</u>	
<u>dump</u>	
<u>disassemble</u>	
<u>refresh</u>	
<u>set-port-address</u>	
<u>set-clock</u>	
<u>set-flash-program-option</u>	
<u>set-flash-size</u>	
<u>set-flash-page-size</u>	
<u>set-ram-pages</u>	
<u>set-pin-state-delay</u>	
<u>set-parallel-port-address</u>	
<u>write-option-register</u>	
<u>read-option-register</u>	
<u>load-object-file</u>	
<u>read-flash</u>	

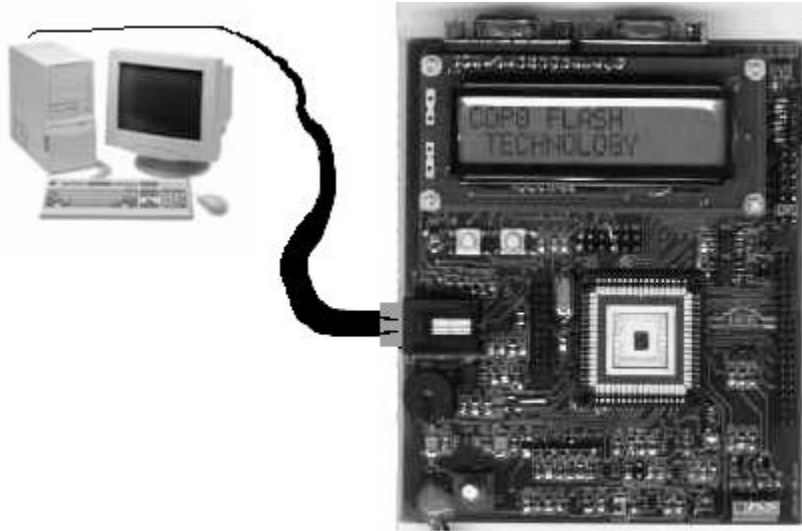
[write-flash](#).....
[read-ram](#).....
[write-ram](#).....
[erase-flash](#).....
[erase-flash-page](#).....
[verify-program](#).....
[flash-program](#).....
[load-object-file](#).....
[write-program-counter](#).....
[read-program-counter](#).....
[write-register-a](#).....
[read-register-a](#).....
[reset](#)
[stop](#)
[run](#)
[step](#)
[mstep](#).....
[running?](#).....
[get-break-point](#).....
[set-break-point](#).....
[clear-break-point](#).....
[add-watch](#).....
[clear-watches](#).....

List of Examples

- 7-1. register macros from debug.inc**.....
- 7-2. using the debug macros.....
- 7-3. interrupt macros from debug.inc.....
- 7-4. debugging an interrupt handler.....

Chapter 1. Overview

The NiceMon debug monitor is a development tool for the National COP8 Flash microcontrollers. Developers are able to debug and test COP8 Flash applications in their target environments from a PC host.



Connection between the host and target system is through the PC parallel port and the target ISP port.

Chapter 2. Installation

- unzip NiceMon anywhere you want
- use this command to run NiceMon

```
java -classpath "c:\nicemon\nicemon.jar" -D"java.library.path"="c:\nicemon" mon.NiceMon
```

Note: the case of `mon.NiceMon` is significant

Chapter 3. Command Line Options

--object

An object can be specified on the command line. Here is an example.

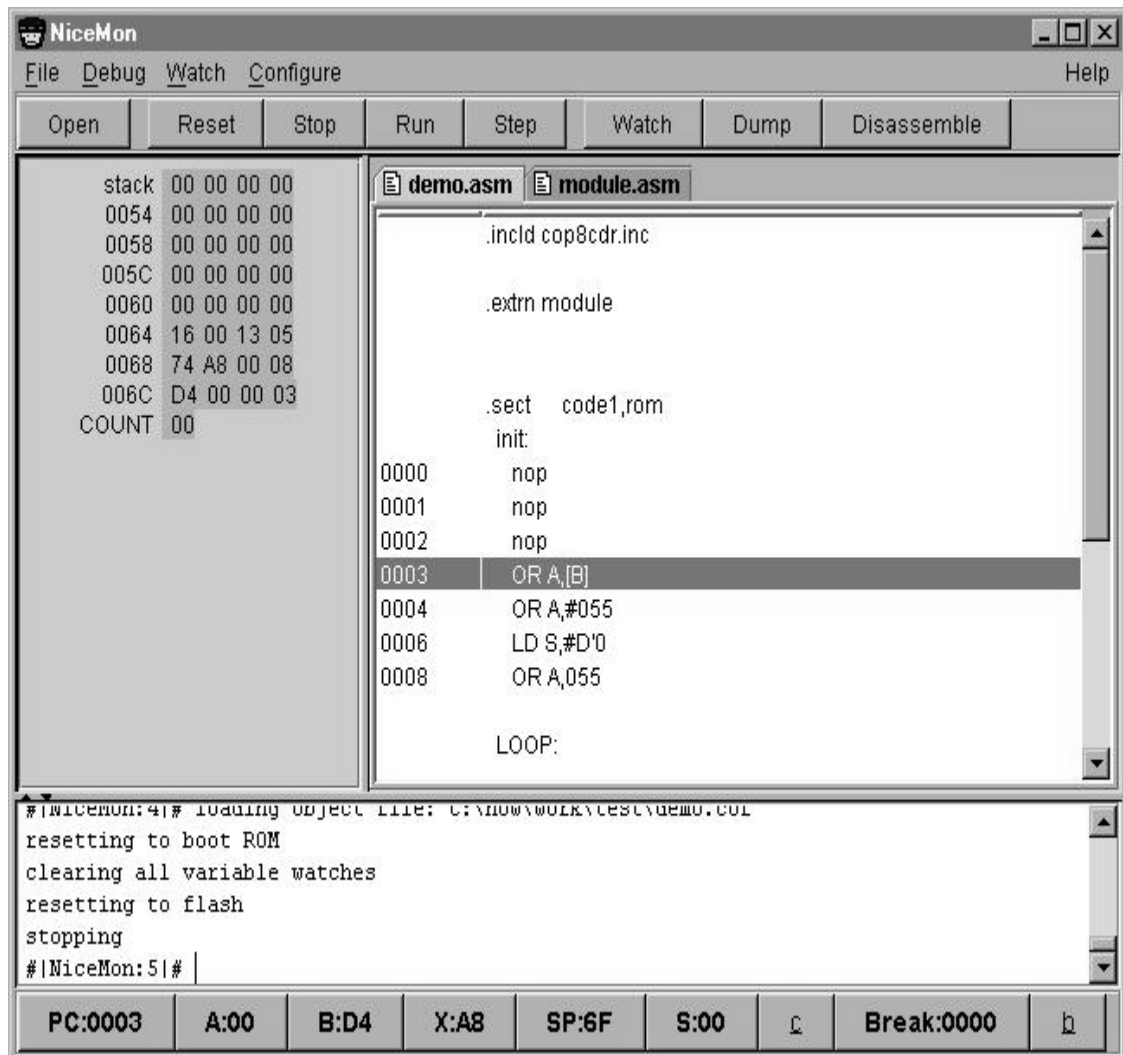
```
java -classpath "c:\NiceMon\NiceMon.jar" -D"java.library.path"="c:\NiceMon" mon.NiceMon --object  
"c:\NiceMon\demo.cof"
```

--script

A script file can be specified on the command line. Here is an example.

```
java -classpath c:\NiceMon\NiceMon.jar mon.NiceMon --script "c:\scripts\custom.scm"
```

Chapter 4. User Interface



This screen shot illustrates the main components of the user interface.

Menu Bar

The NiceMon menu bar provides the user with access to NiceMon's user interface controls. Here is the list of menus:

File

The *File* menu contains these items.

- *Open*: will load a program to be debugged. Selecting this menu item will open a dialog box prompting the user for a file name. The COP8 Flash is programmed with the application code and the NiceMon debug stub.
- *Program Flash*: will program the device without debugging. Selecting this menu item will open a dialog box prompting the user for a file name. If the OPTION register is not defined by the selected object file it is set to the value specified by the **OPTION value** menu item.
- *Verify Program Flash*: will verify that an object file is properly programmed. Selecting this menu item will open a dialog box prompting the user for a file name. The results of the verify are displayed in the *console* area.
- *Run Script*: will load a Scheme script. Selecting this menu item will open a dialog box prompting the user for a file name. Typically scripts are used to set preferred settings and to automate command sequences.
- *Exit*: will exit NiceMon.

Debug

The *Debug* menu contains these items.

- *Reset*: will issue a soft reset and stop at the reset vector.
- *Stop*: will stop a running program.
- *Run*: will continue execution at the current program counter.
- *Step*: will run and stop after one assembly instruction is executed.

Watch

The *Watch* menu contains these items.

- *Watch*: will add a watch to the watch area. If debug symbols are loaded NiceMon will prompt the user to select a variable to watch. If debug symbols are not loaded NiceMon will prompt the user for a name and address. NiceMon will then prompt the user for the number of bytes to be watched.
- *Clear*: will remove all watch variables from the watch area.
- *Dump*: will dump 0x50 bytes of flash. The starting address is at the program counter.
- *Disassemble*: will disassemble five assembly instructions. The starting address is at the program counter.

Configure

The *Configure* menu contains these items.

- *Clock Frequency*: used to specify the operating frequency of the target microcontroller. This value is used to calculate the flash programming times and the ISP programming delays.

- *Flash Size*: used to specify the flash size of the target microcontroller.
- *Flash Page Size*: used to specify the flash page size of the target microcontroller.
- *Ram Pages*: used to specify the number of RAM pages on the target microcontroller.
- *Pin Delay*: used to specify the delay between parallel port pin transitions. Usually a delay is not required here so it can be left at 0ns.
- *Parallel Port*: used to select the PC parallel port that is connected to the target.
- *Option Value*: used to specify the OPTION value to be used if no value is specified by the user application. This value is only used when a device is programmed (it is not used when debugging).

Here is the default configuration:

clock frequency	10Mhz
flash size	32kb (0x8000)
flash page size	128b (0x80)
RAM pages	8
pin delay	0ns
parallel port	0x0378 (LPT1)
option value	0x07 (watchdog/security disabled and FLEX set

Help

The *Help* menu contains these items.

- *User Manual*: will open a web browser with the NiceMon documentation.
- *Project Web Page*: will open a web browser with the NiceMon Web Page.
- *About*: will open a dialog box with NiceMon version number, credits and technical support contacts.

Tool Bar

The NiceMon tool bar provides the user with quick access to the commonly used commands.

Here are the available tool bar actions.

- *Open*: used to load a program to be debugged. Selecting this menu item will open a dialog box prompting the user for a file name. The COP8 Flash is programmed with the application code and the NiceMon debug stub.
- *Reset*: will issue a soft reset and stop at the reset vector.
- *Stop*: will stop a running program.

- *Run*: will continue execution at the current program counter.
- *Step*: will run and stop after one assembly instruction is executed.
- *Watch*: is used to add a watch to the watch area. If debug symbols are loaded NiceMon will prompt the user to select a variable to watch. If debug symbols are not loaded NiceMon will prompt the user for a name and address. NiceMon will then prompt the user for the number of bytes to be watched.
- *Dump*: used to dump 0x50 bytes of flash. The starting address is at the program counter.
- *Disassemble*: used to disassemble five assembly instructions. The starting address is at the program counter.

Status Bar

The NiceMon status bar provides the user with quick access to commonly used information. The status bar also serves as a way of modifying these values. A mouse click will open a dialog box prompting the user for a new value.

The NiceMon status contains these items:

- *PC*: The program counter shows the address of the next instruction to be executed.
- *A*: The A accumulator.
- *B*: The B register.
- *X*: The X register.
- *SP*: The stack pointer. The value specified is the value of the stack pointer.
- *C*: The status of the carry bit. Uppercase 'C' is displayed if carry is set and lower case 'c' is displayed if carry is clear.
- *Break*: execution is stopped when it reaches this address. The break check box enables and disables the break point.
- *B*: Uppercase 'B' is displayed if the break point is enabled and lowercase 'b' is displayed if the break point is disabled.

Main View

The main view window has a tab for every source file referenced by the debug information in the loaded object files. If there is a source line that corresponds to the current program counter that line is highlighted.

Watch Area

The *watch area* has a list of variables to be viewed as the program executes. The variable values are updated whenever the program execution stops. New values can be assigned by clicking on the value field.

Console Area

The *console area* is where all messages are logged and where commands can be executed manually. The console is a Scheme (<http://www.schemers.org>) interpreter as implemented by the Kawa Scheme system (<http://www.gnu.org/software/kawa>).

For a list of NiceMon specific Scheme functions see [Console Commands](#).

Chapter 5. Accommodating NiceMon

Here is a list of the resources and configurations required by NiceMon:

- A 0x300 byte block of flash on a page boundary. NiceMon will search for an available block and locate the firmware there.
- 15 bytes of stack space. (Interrupt call overhead + registers saved on stack + ISP function overhead)
- The software interrupt vector. NiceMon will overwrite it.
- The Microwire interrupt vector. NiceMon will overwrite it.
- The VIS location. Address 0x00ff is overwritten with the VIS instruction.
- The Microwire SI/SO/SK PORTG configuration and pins. If the user changes these settings NiceMon will lose control of the application.
- The Microwire peripheral must be left enabled.
- The FLEX bit in the OPTION flash register must be left clear.
- The ISP registers get modified. This means you can't use NiceMon to debug parts of your program that use the ISP registers.

PORTGC	x001 xxxx	leave the clock and data-in as inputs and data-out as output
PORTGD	xx1x xxxx	leave the Microwire clock idle state high
ICNTRL	xxxx 01xx	leave the Microwire interrupt pending flag (uWPND) clear and Microwire enable bit (uWEN) set
CNTRL	xxxx 1xxx	leave (MSEL) Microwire pins activated
PSW	xxxx x1x1	leave both BUSY and GIE set
OPTION	xxxx xxx0	leave FLEX clear

Chapter 6. Using Console Commands & Scripts

Here are some hints to using the command console.

The console is a Scheme (<http://www.schemers.org>) interpreter as implemented by the Kawa Scheme system (<http://www.gnu.org/software/kawa>).

In general, console commands can be run by enclosing the command in round braces.

(<command> [parameters])

Values are specified in decimal. Hexadecimal value can be specified by enclosing the value in double quotes.

This command will read the option register.

(read-flash "ffff")

Commands must be typed in after the last #NiceMon:x# prompt. Most of the time the cursor will be put there automatically but sometimes you have to scroll to the end of the buffer.

The default.scm script (in the NiceMon home direction) is loaded by default when an object file is loaded.

If NiceMon finds a script with the same name as an object file it is automatically loaded at the same time. If you load demo.cof NiceMon will automatically run demo.cof.scm.

Chapter 7. Tutorial

This chapter outlines the simple procedures involved in debugging an application with NiceMon.

Writing the firmware

Here is an example with tips on writing a program that can be debugged with NiceMon

See [Accommodating NiceMon](#) for the list of requirements.

staying configured for NiceMon

Extra care is needed when modifying registers used by NiceMon. Make sure when using sbit or rbit instructions on registers that NiceMon uses that they don't conflict with the settings required by NiceMon.

Also make sure that registers initialized with X A,## don't conflict with NiceMon settings either. This can be a little tricky. The constant used to initialize the register may need to be different depending on whether NiceMon is used or not.

A convenient way to deal with this problem is to use macros that are conditionally defined for use with or without NiceMon. Example macros are given in `debug.inc`

Example 7-1. register macros from `debug.inc`

```
; These macros can be used instead of
; loading a constant into a register used
; by NiceMon.
;
; These macros ensure that the configuration will won't
; disturb NiceMon.

    .macro NICE_MON_CNTRL value
        .mloc NICE_MON_CNTRL_MASK
        .mloc NICE_MON_CNTRL_VALUE
        NICE_MON_CNTRL_MASK = B'11110111
        NICE_MON_CNTRL_VALUE = B'00001000
        .ifdef DEBUG_NICE_MON
            ld CNTRL,#((value AND NICE_MON_CNTRL_MASK) OR
NICE_MON_CNTRL_VALUE)
        .else
            ld CNTRL,#value
        .endif
    .endm

    .macro NICE_MON_ICNTRL value
        .mloc NICE_MON_ICNTRL_MASK
        .mloc NICE_MON_ICNTRL_VALUE
        NICE_MON_ICNTRL_MASK = B'11110011
        NICE_MON_ICNTRL_VALUE = B'00000100
        .ifdef DEBUG_NICE_MON
```

```

                                ld ICNTRL,#((value AND NICEMON_ICNTRL_MASK) OR
NICEMON_ICNTRL_VALUE)
                                .else
                                ld ICNTRL,#value
                                .endif
                                .endm

                                .macro NICEMON_PSW value
                                .mloc NICEMON_PSW_MASK
                                .mloc NICEMON_PSW_VALUE
                                NICEMON_PSW_MASK = B'11111010
                                NICEMON_PSW_VALUE = B'00000101
                                .ifdef DEBUG_NICEMON
                                ld PSW,#((value AND NICEMON_PSW_MASK) OR
NICEMON_PSW_VALUE)
                                .else
                                ld PSW,#value
                                .endif
                                .endm

                                .macro NICEMON_PORTGC value
                                .mloc NICEMON_PORTGC_MASK
                                .mloc NICEMON_PORTGC_VALUE
                                NICEMON_PORTGC_MASK = B'10001111
                                NICEMON_PORTGC_VALUE = B'00010000
                                .ifdef DEBUG_NICEMON
                                ld PORTGC,#((value AND NICEMON_PORTGC_MASK) OR
NICEMON_PORTGC_VALUE)
                                .else
                                ld PORTGC,#value
                                .endif
                                .endm

                                .macro NICEMON_PORTGD value
                                .mloc NICEMON_PORTGD_MASK
                                .mloc NICEMON_PORTGD_VALUE
                                NICEMON_PORTGD_MASK = B'11011111
                                NICEMON_PORTGD_VALUE = B'00100000
                                .ifdef DEBUG_NICEMON
                                ld PORTGC,#((value AND NICEMON_PORTGD_MASK) OR
NICEMON_PORTGD_VALUE)
                                .else
                                ld PORTGD,#value
                                .endif
                                .endm

```

If the debug macros are used, your program only needs to specify the constants required by your program. The macro will make sure the NiceMon configuration is not disturbed.

Adding the statement `DEBUG_NICEMON = 1` will direct the macros to alter the configuration.

```
DEBUG_NICEMON = 1
.incl d debug.inc
```

Example 7-2. using the debug macros

```
NICEMON_CNTRL B'10100000

    ld TMR1LO,#L(10000)
    ld TMR1HI,#H(10000)
    ld T1RALO,#L(5000)
    ld T1RAHI,#H(5000)
    ld T1RBLO,#L(10000)
    ld T1RBHI,#H(10000)

NICEMON_PSW B'00010001

    sbit T1C0,CNTRL ; start timer 1a
```

configuring to debug with interrupts

NiceMon requires interrupts to operate properly. So your interrupt sub routines must coexist with NiceMon. This fact puts constraints on your program.

When NiceMon programs the device for debugging it will place a VIS instruction at 0x00ff. This ensures NiceMon gains controls when a break point is reached or commands are received from the host. (remember: when interrupts occurs execution jumps to 0x00FF)

This VIS instruction will cause a problem in some situations. The most common case is when the user program save the processor context before executing the VIS. The problem can be avoided by executing the VIS at 0x00FF and saving context within each interrupt handler.

NiceMon will enable interrupts when it returns control to the user program. This causes a problem when trying to single step through in interrupts sub routine. When NiceMon returns from a single step it automatically re-enables interrupts, even it interrupts were disabled before NiceMon gained control. The effect of this is that subsequent interrupts will re-enter and restart the interrupt handler.

The interrupt can be debugged if the interrupt source is disabled. The debug.inc include file has a macro that can be used to debug the timer 1a interrupt handler.

Example 7-3. interrupt macros from debug.inc

```
; This macro can be used to enable debugging of
; the timer 1a interrupt handler.
;
; If the timer interrupt is disabled, break points
; can be set in the interrupt handler.
;
; NiceMon will enable global interrupts so disabling
; the timer interrupt ensures that subsequent interrupts
; don't mess with the debugging.
;
; The symbol DEBUG_TIMER_1A_ISR must be define for this
; macro to have an effect.
;
```

```

; DEBUG_TIMER_1A_ISR = 1
    .macro NICE_MON_DEBUG_TIMER_1A_ISR_ENTER
        .ifdef DEBUG_NICE_MON
        .ifdef DEBUG_TIMER_1A_ISR
            rbit T1ENA,PSW ; disable timer 1a interrupt
        .endif
        .endif
    .endm

; This macro is used to re-enable
; the timer 1a interrupt
    .macro NICE_MON_DEBUG_TIMER_1A_ISR_EXIT
        .ifdef DEBUG_NICE_MON
        .ifdef DEBUG_TIMER_1A_ISR
            sbit T1ENA,PSW ; enable timer 1a interrupt
        .endif
        .endif
    .endm

```

The tutorial.asm example shows how this macro can be used.

Example 7-4. debugging an interrupt handler

```

timer_1a_isr:
    NICE_MON_DEBUG_TIMER_1A_ISR_ENTER ; this is require to debug the
interrupt handler
    push A      ; save A
    ld a,B     ; save B
    push A
    ld a,PSW   ; save C and HC
    push A

    ld a,count3
    inc a
    x a,count3
    rbit T1PND,PSW ; reset timer 1a pending flag
    ld A,PORTD
    xor A,#001 ; toggle port D pin 0
    x A,PORTD

    pop A      ; restore C and HC
    rc
    ld B,#PSW
    ifbit 7,A
    sbit 7,[B]
    ifbit 6,A
    sbit 6,[B]
    pop A      ; restore B
    x A,B
    pop A      ; restore A
    NICE_MON_DEBUG_TIMER_1A_ISR_EXIT ; this is require to debug the

```

```
interrupt handler
    reti
```

A debugging session

Invoke NiceMon as directed in the [Installation](#) chapter.

Use the **Configure** menu to set the appropriate settings for your target environment. The effects and default values of these options are described in the [Configure](#) chapter.

Once you have your program written and compiled you can open the generated COFF file by selecting **open** from the **File** menu.

If the object file was loaded properly NiceMon will open the source files and scroll to address 0x0000. Now you can step through your program, set break points and watch RAM variables.

I. Console Commands

dump

Name

`dump` — Display contents of flash. The count parameter specifies how many lines are displayed.

Syntax

```
dump [address] [count]
```

disassemble

Name

`disassemble` — Disassemble the contents of flash. The count parameter specifies the number of instructions to disassemble.

Syntax

```
disassemble [address] [count]
```

refresh

Name

`refresh` — Refresh the user interface.

Syntax

```
refresh
```

set-port-address

Name

`set-port-address` — Use this function to set the communication port address.

Syntax

`set-port-address` [address]

set-clock

Name

`set-clock` — Specifies the operating frequency (in Hz).

Syntax

`set-clock` [frequency]

set-flash-program-option

Name

`set-flash-program-option` — Use this command to set the default OPTION register value when programming a device.

Syntax

`set-flash-program-option` [value]

set-flash-size

Name

`set-flash-size` — Use this command to specify the size of the flash on the target microcontroller.

Syntax

`set-flash-size` [size]

set-flash-page-size

Name

`set-flash-page-size` — Use this command to specify the size of a flash page on the target microcontroller.

Syntax

`set-flash-page-size` [size]

set-ram-pages

Name

`set-ram-pages` — Use this command to specify the number of RAM pages on the target microcontroller.

Syntax

`set-ram-pages` [count]

set-pin-state-delay

Name

`set-pin-state-delay` — Use this command to specify the delay time between parallel port pin state changes. The time is specified in nanoseconds.

Syntax

`set-pin-state-delay` [time]

set-parallel-port-address

Name

`set-parallel-port-address` — Use this command to specify the parallel port address.

Syntax

`set-parallel-port-address` [address]

write-option-register

Name

`write-option-register` — Use this to set the OPTION register.

Syntax

`write-option-register` [value]

read-option-register

Name

`read-option-register` — Returns the OPTION register value.

Syntax

`read-option-register` [value]

load-object-file

Name

`load-object-file` — Loads the specified object file.

Syntax

`load-object-file` [filename]

read-flash

Name

`read-flash` — Reads the data from a given flash address.

Syntax

`read-flash` [address]

write-flash

Name

`write-flash` — Writes data to flash.

Syntax

`write-flash` [address]

read-ram

Name

`read-ram` — Reads the data from a given RAM address.

Syntax

`read-ram` [address]

write-ram

Name

`write-ram` — Writes the data to a given address in RAM.

Syntax

`write-ram` [address]

erase-flash

Name

`erase-flash` — Erases all of the COP8 Flash flash.

Syntax

`erase-flash`

erase-flash-page

Name

`erase-flash-page` — Erases the flash page of a given address.

Syntax

`erase-flash-page` [address]

verify-program

Name

`verify-program` — Use this command to verify that an object file was programmed properly.

Syntax

`verify-program` [file name]

flash-program

Name

`flash-program` — Programs the target with the specified object file. This command does not load the NiceMon debug stub.

Syntax

`flash-program` [file name]

load-object-file

Name

`load-object-file` — Loads an object file to be debugged.

Syntax

`load-object-file` [file name]

write-program-counter

Name

`write-program-counter` — Set the value for the program counter.

Syntax

`write-program-counter` [address]

read-program-counter

Name

`read-program-counter` — Returns the current address in the program counter

Syntax

`read-program-counter`

write-register-a

Name

`write-register-a` — Assigns a value to the A accumulator.

Syntax

`write-register-a` [value]

read-register-a

Name

`read-register-a` — Returns the value in the accumulator.

Syntax

`read-register-a`

reset

Name

reset — Issues a soft reset.

Syntax

reset

stop

Name

stop — Stops a running program.

Syntax

stop

run

Name

run — Continues execution at the current program counter.

Syntax

run

step

Name

`step` — Runs and stops after one assembly instruction is executed.

Syntax

`step`

mstep

Name

`mstep` — Multiple step.

Syntax

`mstep` [count]

running?

Name

`running?` — Tests whether the user program is running.

Syntax

`running?`

Returns true if the target application is running or false if the program is halted.

get-break-point

Name

`get-break-point` — Returns the break point address.

Syntax

`get-break-point`

Returns a negative value if no break point is set.

set-break-point

Name

`set-break-point` — Makes execution stop when it reaches the address specified.

Syntax

`set-break-point` [address]

clear-break-point

Name

`clear-break-point` — Removes a previously set break point.

Syntax

`clear-break-point`

add-watch

Name

`add-watch` — Adds a variable watch to the watch list.

Syntax

`add-watch` [name] [address] [size]

clear-watches

Name

`clear-watches` — Clears the watch area of all variable watches.

Syntax

`clear-watches`