NiceMon

National COP8 Flash Debug Monitor

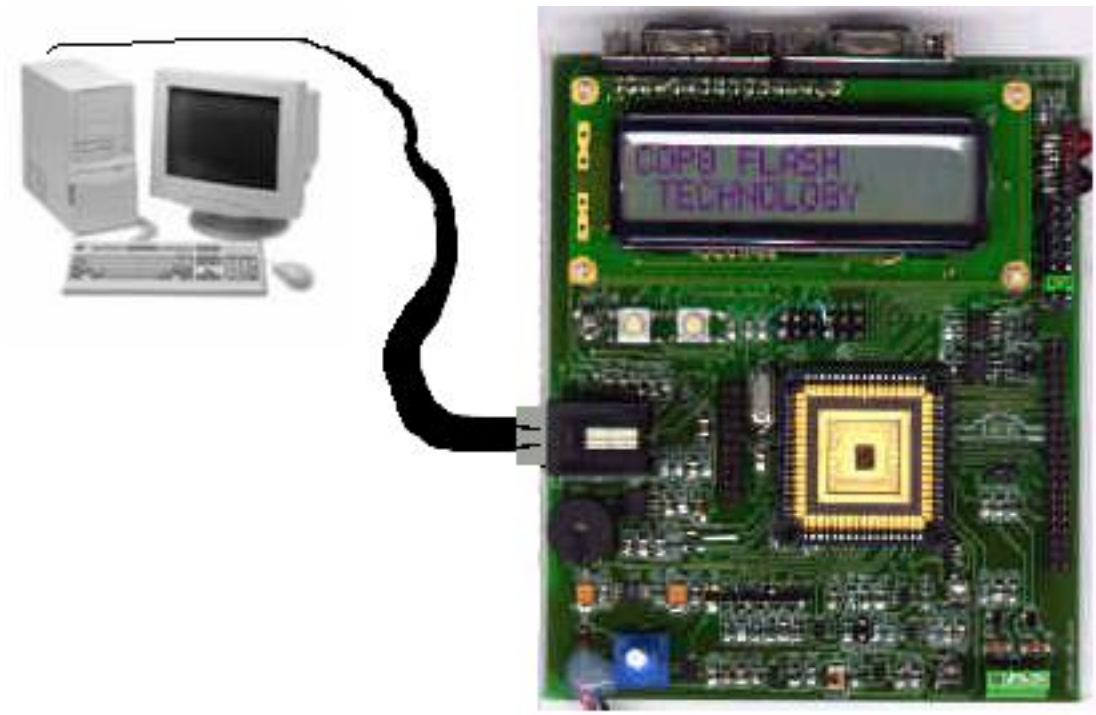
NiceMon: National COP8 Flash Debug Monitor
User Manual $Revision: 1.55 $ $Date: 2001/09/20 23:35:03 $ Edition

Table of Contents

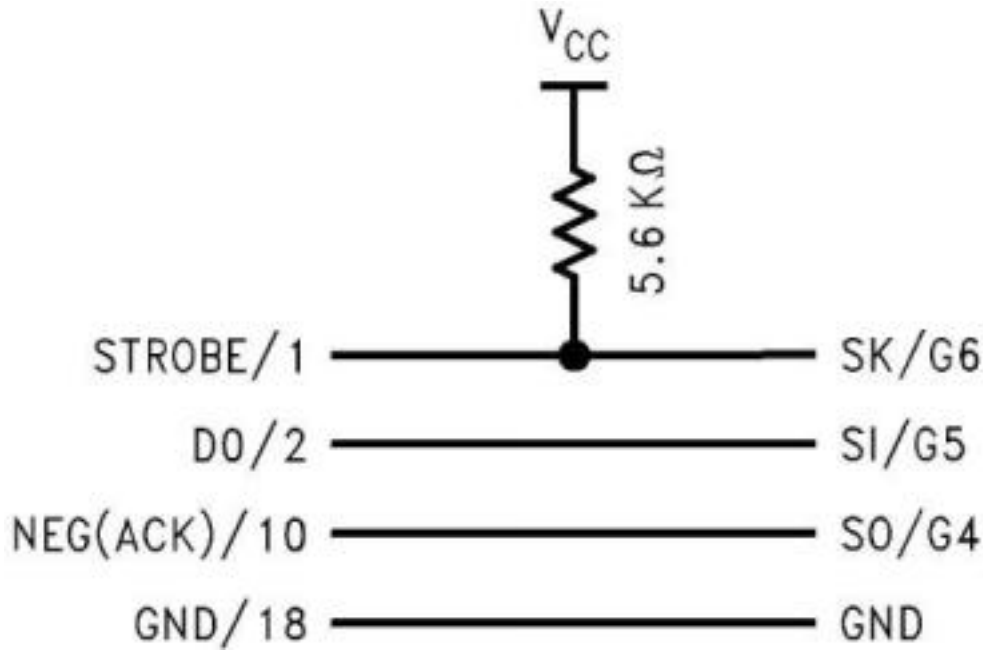
Chapter 1. Overview

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.

NiceMon uses the COP8 Flash' in-circuit programming capabilities to provide developers with emmulator-less debugging.



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


V_{CC}
5.6 KΩ
STROBE/1
SK/G6
D0/2
SI/G5
NEG(ACK)/10
SO/G4
GND/18
GND

Chapter 2. Installation

About the documentation

This document is available in both online[1] and printable[2] formats.
Documentation and software updates are available from Ammunerve[3].
For a list of known bugs and limitations for this version see the ReadMe[4].

The install programs

- Install the Java 2 Run Time Environment (`j2re-1_3_1_01-win.exe`)
- Install the DriverLINX Port I/O Driver (`port95nt.exe`)
- Run the NiceMon Install program (`NiceMon-Setup.exe`).
- Use the Start Menu or Desktop icons to invoke NiceMon.

The command line

Optionally NiceMon can be invoked from the command line.

```
C:\>javaw -jar "C:\COP8\flashNiceMon\NiceMon.jar"
```

Notes

1. index.html
2. NiceMon.pdf
3. http://www.ammunerve.com
4. ../ReadMe.txt

Chapter 3. Command Line Options

–object

A COFF or Intel HEX object file can be specified on the command line. Here is an example.

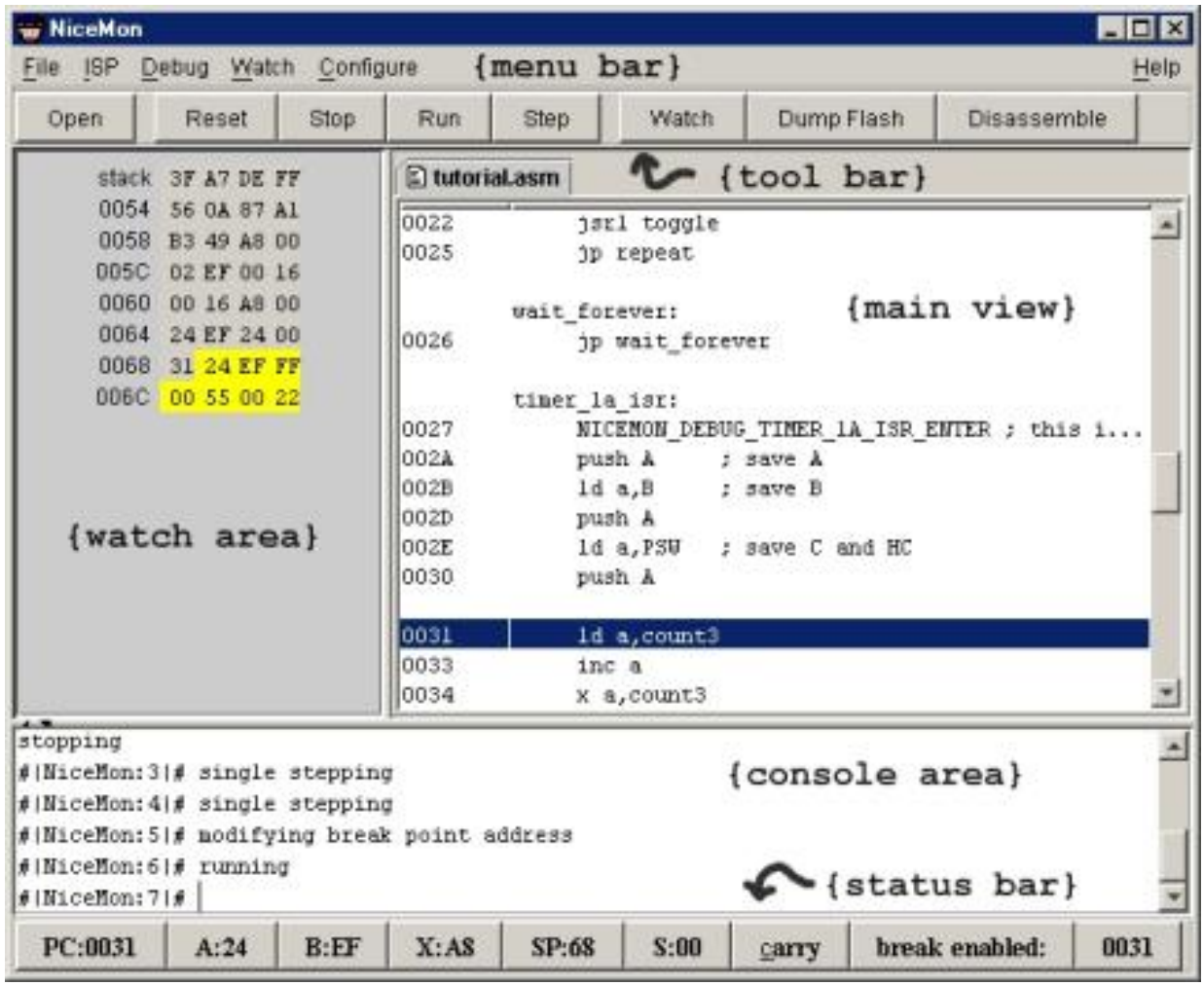
```
C:\>javaw -jar "C:\COP8\flashNiceMon\NiceMon.jar"
    -object "C:\COP8\flashNiceMon\tutorial.cof"
```

–script

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

```
C:\>javaw -jar "C:\COP8\flashNiceMon\NiceMon.jar"
    -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 a description of each menu:

File

The ***File*** menu contains these items.

- ***Open***: will load a program to be debugged (COFF or Intel HEX object file). 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.

- ***Run Scheme Script***: will run a Scheme[1] 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.

ISP

The ***ISP*** menu contains these items.

- ***Program Flash***: will program the device without debugging. Selecting this menu item will open a dialog box prompting the user for a file name (COFF or Intel HEX object file). If the OPTION register is not defined by the 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 (COFF or Intel HEX object file). The results of the verify are displayed in the *console* area.
- ***Default Option Value***: used to to specify the OPTION value to be used if no value is specified by the object file. This value is only used when a device is programmed (it is not used when debugging). The default default value is 0x07 (watchdog/security disabled and FLEX set).

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 Flash***: 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 the default 0ns.
- ***Parallel Port***: used to select the PC parallel port that is connected to the target.

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)

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.
- ***ReadMe file***: will open the ReadMe text file.
- ***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 Flash***: used to dump 0x50 bytes of flash. The program counter is used as the starting address.
- ***Disassemble***: used to disassemble five assembly instructions. The program counter is used as the starting address.

Status Bar

The NiceMon status bar provides the user with quick access to important 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.
- *Carry*: The status of the carry flag. 'Carry' with an uppercase 'C' is displayed if carry is set and 'carry' with a lower case 'c' is displayed if carry is clear. Clicking this button will toggle the carry flag.
- *Break disabled:*: This button will enable and disable the break point.
- <***break address***>: Displays the break address. Click this button to be prompted for a new break address.

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[2] interpreter as implemented by the Kawa Scheme system[3].

For a list of NiceMon specific Scheme functions see *Command Reference*.

Notes

1. http://www.schemers.org
2. http://www.schemers.org
3. http://www.gnu.org/software/kawa

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 stub there.
- 15 bytes of stack space. (interrupt vector 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 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 NiceMon can't be used to debug portions of an application that use the ISP registers. NiceMon does not affect the ISP registers while the application is running but will scramble them once a break point is reached.

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. 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`

```
; 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 NICEMON_CNTRL value
        .mloc NICEMON_CNTRL_MASK
        .mloc NICEMON_CNTRL_VALUE
        NICEMON_CNTRL_MASK  = B'11110111
        NICEMON_CNTRL_VALUE = B'00001000
        .ifdef DEBUG_NICEMON
            ld CNTRL,#((value AND NICEMON_CNTRL_MASK) OR NICEMON_CNTRL_VALUE)
        .else
            ld CNTRL,#value
        .endif
    .endm

    .macro NICEMON_ICNTRL value
        .mloc NICEMON_ICNTRL_MASK
        .mloc NICEMON_ICNTRL_VALUE
        NICEMON_ICNTRL_MASK  = B'11110011
        NICEMON_ICNTRL_VALUE = B'00000100
        .ifdef DEBUG_NICEMON
            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
.incld debug.inc



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.

NiceMon will enable interrupts when it returns control to the user program. This causes a problem when trying to single step through an interrupt sub routine. When NiceMon returns from a single step it automatically re-enables interrupts, even if 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.

```
; 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 defined for this
; macro to have an effect.
;
; DEBUG_TIMER_1A_ISR = 1
    .macro NICEMON_DEBUG_TIMER_1A_ISR_ENTER
        .ifdef DEBUG_NICEMON
        .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 NICEMON_DEBUG_TIMER_1A_ISR_EXIT
        .ifdef DEBUG_NICEMON
        .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.

```
timer_1a_isr:
    NICEMON_DEBUG_TIMER_1A_ISR_ENTER ; this is required 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 T1PNDA,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
NICEMON_DEBUG_TIMER_1A_ISR_EXIT ; this is required to
                                ; debug the interrupt handler
reti
```

A sample debugging session

Once NiceMon has started up you'll want to make sure everything is configured appropriately for your target. Do this using the **Configure** menu.

Here is the check list to go through before starting a debug session:

- Make sure the target is connected and powered.
- Make sure the target is executing from boot ROM. This is the default for a blank device.
- Make sure the correct parallel port is selected. The default is LPT1.
- Make sure the correct target operating frequency is selected. The default is 10Mhz.
- Make sure the flash and page sizes are correct. The defaults are 32k bytes and 128 bytes.

Start a debugging session by pushing the tool bar's **Open** button, and selecting `tutorial.cof` form the NiceMon distribution directory. If everything works NiceMon will program the device, reset and set the program counter to `0000`. You will see **PC: 0000** at the very bottom left. The watch area will show a chunk of the stack and the main view area will show `tutorial.asm` with the assembly source for address `0000` highlighted.

Now you're ready to try things out. Pushing the **Step** button will execute individual instructions. Hitting the **Run** button will run until you hit **Cancel** or a break point is reached. You can set a break point by selecting the **—-** button in the bottom right corner. The break point can be disabled/enabled using the button next to it.

You can look at the contents of RAM by adding a `watch` to the watch area. Hitting the tool bar's `watch` button will open dialog boxes prompting you for the required information. Once the `watch` has been added contents of RAM can be modified by clicking the value in the watch area.

You can view the contents of flash by hitting the tool bar's `Dump Flash` button. This will show `0x50` bytes of data starting from the current program counter. Hitting the `Disassemble` button will disassemble the next five instructions.

If you want to see the contents of different flash addresses you can either change the program counter (by hitting the PC:xxxx button) and use the tool bar button as before or you can use the console to manually invoke the `disassemble` and `dump` commands. Here is an example:

```
#|NiceMon:12|# (disassemble "30" 3)

0030  push A
0031  ld A,0F2
0033  inc A
#|NiceMon:13|# (dump "12" 4)

0012 13 BC E6 10 BC E7 27 BC-EF 15 BD EE 7C AD 00 51
0022 AD 00 5A F9 FF BD EF 6C-67 9D FE 67 9D EF 67 9D
0032 F2 8A 9C F2 BD EF 6D 9D-DC 96 01 9C DC 8C A0 9F
0042 EF 60 80 7F 60 40 7E 8C-9C FE 8C BD EF 7C 8F D0
#|NiceMon:14|#
```

Project settings

Script files can be used to configure project settings. See *Using Console Commands & Scripts* for a description of the special script files.

The `tutorial.cof-pre.scm` and `tutorial.cof-post.scm` script files from the NiceMon distribution are examples of how project settings can be initialized.

Typically the <*`object file name`*>`-pre.scm` will have commands to initialize NiceMon with the required settings.

```
;; configure PC host settings
(set-parallel-port-address "378") ; LPT1
(set-pin-state-delay 0) ; no delay required for my PC

;; configure settings for COP8-EVAL-FSK1
(set-operating-frequency 10000000) ; 10Mhz
(set-flash-size "8000") ; 8k bytes of flash
(set-flash-page-size "80") ; 80 bytes per page
(set-ram-pages 8) ; S register can be 0 to 7
```

Typically the <*`object file name`*>`-post.scm` will add watches and define functions that will be usefull when debugging this object file.

```
;; add some watches
(add-watch "COUNT1" "f0" 1)
(add-watch "COUNT2" "f1" 1)
(add-watch "COUNT3" "f2" 1)
(add-watch "timer low" "ea" 1)
(add-watch "timer high" "eb" 1)
(add-watch "PORTD" "dc" 1)

;; display a note to self
(display "this is a nice place to display a message")
(newline)

;; a little function that sets COUNT1 and COUNT2
```

```
;; to 1 so you can step out of the delay function
(define (step-out)
  (write-ram "f0" 1)
  (write-ram "f1" 1)
  (refresh))
```

Chapter 7. Using Console Commands & Scripts

Here are some hints to using the command console.

The console is a Scheme[1] interpreter as implemented by the Kawa Scheme system[2].

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.

```
#|NiceMon:1|# (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.

NiceMon will automatically run script files that can be customized for various targets and host configurations. Here is a sequence of events illustrating when the special scripts are run.

- NiceMon is started
- `default.scm` is run

  This script is part of the NiceMon distribution. It is automatically run every time NiceMon is started.

- `user-default.scm` is run if it exists

  Users can create this script if they need commands to execute every time NiceMon is started. NiceMon will look for this file in the NiceMon home directory.

- an object file is opened for debugging
- `default-pre.scm` is run

  This script is part of the NiceMon distribution. It is automatically run before any object file is opened for debugging.

- `user-default-pre.scm` is run if it exists

  Users can create this script if they need commands to be executed before any object file is opened for debugging. NiceMon will look for this file in the NiceMon home directory.

- *<object file name>*`-pre.scm` is run if it exists

Users can create scripts to be run before an object file is opened. This script must be located in the same directory as the object file. For example: c:\test\tutorial.cof-pre.scm would be run before c:\test\tutorial.cof is opened for debugging.

- the object file is programmed for debugging
- `default-post.scm` is run

  This script is part of the NiceMon distribution. It is automatically run after an object file is opend for debugging.

- `user-default-post.scm` is run if it exists

  Users can create this script if they need commands to be executed after any object file is opened for debugging. NiceMon will look for this file in the NiceMon home directory.

- *<object file name>*`-post.scm` is run if it exists

  Users can create scripts to be run after an object file is opened for debugging. This script must be located in the same directory as the object file. For example: c:\test\tutorial.cof-post.scm would be run after c:\test\tutorial.cof is opened for debugging.

Notes

1. http://www.schemers.org
2. http://www.gnu.org/software/kawa

. Command Reference

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-operating-frequency

Name

`set-operating-frequency` — Specifies the target operating frequency (in Hz). This value is used to calculate the flash programming times and the ISP programming delays.

Syntax

set-operating-frequency [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]

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

reset-to-user-application

Name

`reset` — Resets and gives control to the user application.

Syntax

reset-to-user-application

reset-to-monitor

Name

`reset` — Resets and gives control to the debug monitor.

Syntax

reset-to-monitor

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]

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