Seehau™

Nohau Monitor (NMon)

March 6, 2003

Contents

1 OVERVIEW 5

2 USING NMON 6

Installing NMon 6

Configuration 7

General NMon Configuration 7
NPut Configuration 7
NGet Configuration 8
CPU Configuration 8

API – Application Programming Interface 8

NMon Global Data 8
NMon Functions 9
NPut Functions 9
NGet Functions 9
NLib Functions 10
NSrv Functions 10
NAssert Macros 10

Integrating NMon with Run Time Libraries 11

ARM – Integrate with Run Time Libs 11

Real Time Considerations 11

NMon – Real Time 11
NPut – Real Time 12
NGet – Real Time 12
NAssert – Real Time 12

Not Thread Safe 12

Access from Seehau 13

Target Console Window 13
Data Window 13
Inspect Window 13
Symbols Required when NPUT_NO_EXEC 13

Family Specific Notes 14

ARM - Notes 14

3 FILE REFERENCE 15

4 KNOWN ISSUES 16

IAR Embedded Workbench **16**

Calling printf() & sprintf() from interrupt (IAR) 16

Optimization 16

About This Guide

This guide is valid for NMon version 1.0.X.

Nohau Monitor (NMon) allows some of our emulator families to communicate with a target application while it is executing. Following capabilities are offered by NMon communication:

- Console I/O – for instance can be used for the function printf() to output text on the Target Console Window in Seehau, and for instance can be used for the function getc() to capture keystrokes from the Target Console.
- Application error messages displayed in popup windows in Seehau, including support for <assert.h> style macros.
- Read and write memory during runtime.
- Write to flash (supported by NMon, but not yet directly supported by Seehau.)

NMon consist of a set of source files that files must be linked into the user's target application to make this possible. The package is called NMon, as it is the main module of the package.

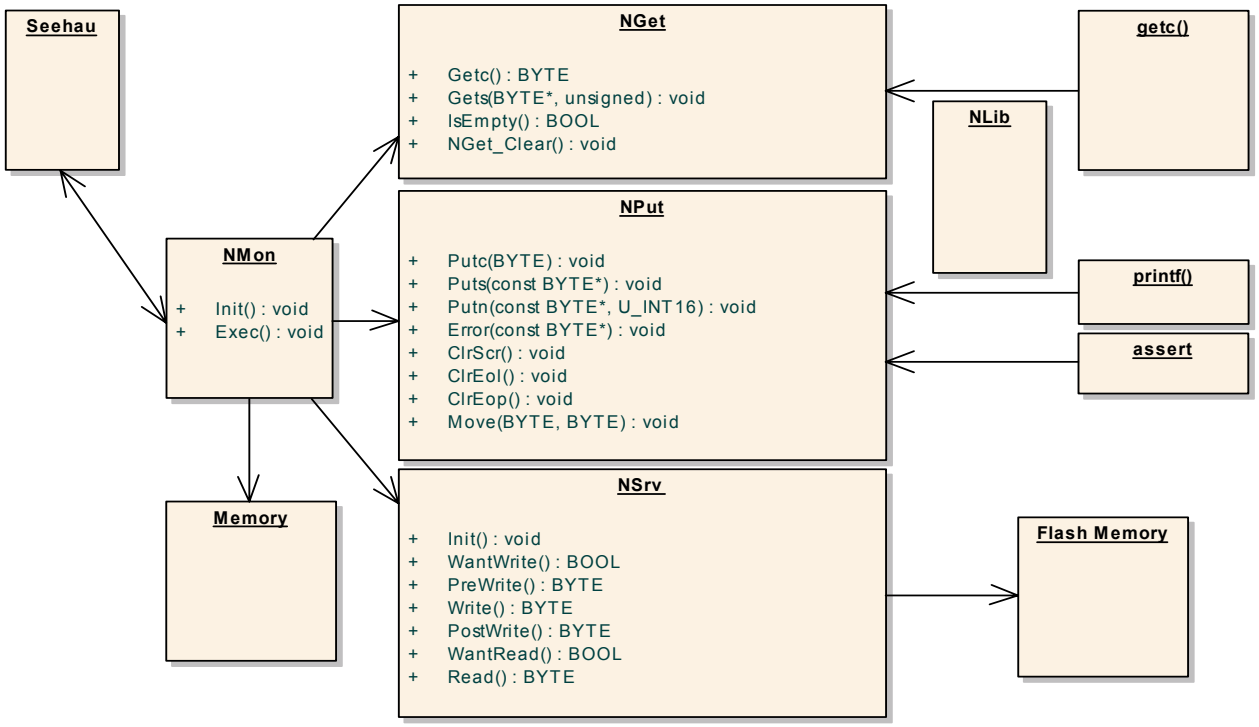
NMon is based on an API – an end user should not use the underlying communication protocol.

NMon is currently available on:

- EMUL-ARM

1 OVERVIEW

The figure below shows the Modules in NMon, and how they are connected internally, and how they communicate with the Seehau GUI. (All blocks starting with 'N' are a part of NMon.)



NMon – the protocol and command handler, including reading and writing to memory.

NGet – functionality that sends information from the debugger to the target application – typically for getc() functionality. This means sending keyboard strokes from the Target Console window.

NPut – functionality that receives information from the target application to the debugger – typically for printf() functionality. This means sending output for displaying in the Target Console window.

NSrv – user defined services for reading and writing memory – typically used for writing to flash. The user may need to modify this module.

NLib – a compiler specific module that connects the run time library with the library that comes with the compiler. Basically it replaces some low level input and output routines so that printf() and getc() can be used.

2 USING NMON

Installing NMon

NMon comes in a zip file called NMon_xxx_yyy.zip located in the Seehau Examples directory, where:

- xxx is family – i.e. ARM.
- yyy is NMon version, where 100 is version 1.0.0.

To install NMon, simply open the zip file, and extract the source files. Our suggestion would be to extract the files into a directory called NMon in the target application source directory.

Then rename "NSrv_UserAppl.c" to "NSrv_xxx.c" where xxx would be your company, project, product, etc.

Finally, add files to the makefile, compiler project, etc. The following files must be added (for full functionality):

- **NCom_xxx.c** – family specific low-level communication.
- **NGet.c** – NGet functionality.
- **NMon.c** – required.
- **NPut.c** – NPut functionality.
- **NSrv_xxx.c** – user defined memory read/write.

And, optionally add:

- **NLib_xxx.c** – compiler specific to enable printf() etc. You may have to create this file yourself for your specific compiler.

> **Note**
>
> If NPut is not used, the file needs not to be added. However, it will not add to the size of your executable if NPut is disabled in the configuration.

Configuration

The files “NMon_Cfg.h” and “NMon_Cpu.h” allow configuring NMon to suit a specific target application, or a specific debug session. All NMon features can be enabled/disabled.

Conditional compilation is used to remove parts that are not used. This means that they do not take up memory space – neither RAM nor ROM. This is true also when the corresponding module is compiled and linked into the application.

The following sections list which configuration alternatives that are available for each section.

General NMon Configuration

- **NMON_ALL_ENABLE** – enables all commands, i.e. all basic functionality is available.
- **NSRV_WRITE_ENABLE** – enables customer specific memory write, i.e. flash write.
- **NSRV_READ_ENABLE** – enables customer specific memory read.
- **NMON_BUF_SIZE** – size of the buffer to use for customer specific read / writes.

NPut Configuration

- **NPUT_ENABLE** – NPut is disabled if commented out.
- **NPUT_STORAGE_SIZE** – defines how many bytes are used to store characters in the target to be sent to the debugger. Default = 200. Defining a different size can change the default.
- **NPUT_ERROR_ENABLE** – Enable functions NPut_Error() and macros defined in “NAssert.h”.
- **NPUT_POSITION_ENABLE** – Enable functions for positioning the cursor on the Target Console, i.e. NPut_Move(), NPut_ClrScr(), NPut_ClrEol() and NPut_ClrEop().
- **NPUT_LIB_ENABLE** – Enable call to function NLib_Init() – to allow initializing the interface to the run time libraries (which is not always needed).
- **NPUT_NO_EXEC** – causes the NMon_Exec() not to be required (in fact it is removed) to simplify implementation. There are two notes – (1) this option is not available if NGET_ENABLE or NMON_ALL_ENABLE are defined, (2) output data will not be buffered, instead it will be directly output(if there is a debugger that receives it).

NGet Configuration

NPut is required for NGet to be enabled.

- **NGET_ENABLE** – NGet is disabled if commented out.
- **NGET_STORAGE_SIZE** – defines how many bytes that are used to store received characters in the target. Default = 100. Defining a different size can change the default.

CPU Configuration

CPU Configuration is made in file "NMon_Cpu.h".

- **NMON_CPU** – Select which CPU is the closest match for the target system.

API – Application Programming Interface

There are functions internally in NMon that are not included here. Note that most of the functions may be disabled in the configuration and not be available.

NMon Global Data

The global data is used by the NSrv module when reading, writing, erasing, etc., and will be updated before any NSrv functions is called.

BYTE NMon_Addr_Curr* – Current address for memory access, increment when reading/writing.

BYTE NMon_Addr_Last* – Last (highest) address for memory access.

ADDRESS NMon_Verify_Addr – If verify fails, the address shall be stored here.

BYTE NMon_Buf[NMON_BUF_SIZE] – Buffer for reading/writing data.

NMon Functions

void NMon_Init(void) – Initialize NMon, must be called at application startup to make sure that all NMon modules are initialized.

void NMon_Exec(void) – Execute NMon, needs to be called from low priory function, as quite long busy waits could be caused.

BOOL NMon_Terminate(void) – Flag to determine if current action be terminated. To be called from NSrv during lengthy operations.

NPut Functions

void NPut_Putc(BYTE ch) – Display one character in the Target Console.

void NPut_Puts(const BYTE str)* – Display zero terminated string in the Target Console.

void NPut_Putn(const BYTE str, U_INT16 count)* – Display count characters from string in the Target Console.

void NPut_Error(const BYTE str)* – Output error message in popup window in Seehau and the Target Console.

void NPut_ClrScr(void) – Clear screen, i.e. Target Console window.

void NPut_ClrEol(void) – Clear end of the current line in the Target Console window.

void NPut_ClrEop(void) – Clear end of the page in the Target Console window, this is the rest of the current line and all lines below.

void NPut_Move(BYTE x, BYTE y) – Move the cursor in the Target Console window to column *x* and row *y*.

void NPut_Echo(BOOL on) – Turn echo on/off for user keystrokes in the TargetConsole.

NGet Functions

BYTE NGet_Getc(void) – Get one (next) character from the Target Console. Wait until available, if not already available.

void NGet_Gets(BYTE pBuf, U_INT16 count)* – Get count (next) characters from the Target Console. Wait until available, if not already available.

BOOL NGet_Available(void) – Flag to determine if there are any characters available. Return TRUE if available.

void NGet_Clear(void) – Clear all the characters in the input buffer and in the debugger.

NLib Functions

void NLib_Init(void) – Initialize the run time libraries interface module. This may be implemented by the user.

NSrv Functions

Note that the NSrv functions operate on the NMon global data. This means that when writing in NSrv_Write(), read data from NMon_Buf and write to NMon_Addr_Curr – NMon_Addr_Last.

void NSrv_Init (void) – Initialize user defined memory read/write. Note that this function may be called at startup and after startup.

BOOL NSrv_WantWrite(void) – Flag to determine if the targets specific code will do the write? If yes, return TRUE.

BYTE NSrv_Write(void) – Do the actual write.

BOOL NSrv_WantRead(void) – Flag to determine if the targets specific code will do the read? If yes, return TRUE.

BYTE NSrv_Read(void) – Do the actual read.

NAssert Macros

These are #define MACROS with functionality similar to the classic <assert.h> functionality, and with some inspiration from the Eiffel Language (precondition and postcondition), plus some ideas of our own (never_be_here and not_implemented).

However, note that there are no real differences between the macros below (except some take the x parameter), it's only a readability difference, and they get different error messages in Seehau.

N_ASSERT(x) – check that the condition 'x' is not zero.

N_PRECONDITION(x) – check that the condition 'x' is not zero when a function for parameter to function.

N_POSTCONDITION(x) – check that the condition 'x' is not zero when function is about to return.

N_NEVER_BE_HERE() – never execute this line.

N_NOT_IMPLEMENTED() – this is functionality not yet implemented.

Integrating NMon with Run Time Libraries

NMon can be integrated with run time libraries for most compilers if so desired.

ARM – Integrate with Run Time Libs

ARM Compiler - See file NLib_ARM.c.

IAR Compiler - See file NLib_IAR.c.

GNU Compiler - NMon currently does not support the GNU compiler. (This is possible for the programmer to add.)

Real Time Considerations

NMon – Real Time

Calling NMon_Exec() implements performing actual communication in the normal case (i.e. unless NPUT_NO_EXEC defined).

It should be called in a low priority process at a relatively high frequency, as the debugger may otherwise timeout in its attempt to communicate. The time for timeout in the debugger will normally be somewhere around 200 ms.

However, the performance in the communication affects the overall performance of Seehau drastically, when a lot of data is displayed in data windows etc. So it is desired to call as often as possible, which could normally be done in the Idle loop, or the lowest prioritized task.

Each application will have to determine how to best call NMon.

After receiving a memory read/write command, NMon_Exec() will not return until the operation is completed. Maximum block size is normally 1000 bytes or so. ARM transfers four bytes per ms, which give a maximum communication time of 250 ms. Note that memory read/write is entirely controlled by the user of the debugger – if data windows show small amounts of data, the time to complete a NMon_Exec() call may be 10 ms.

> **Note**
> NMon can be used in dedicated Flash programming applications, in which case handshaking is not used, and the speed for communication is radically faster.

Finally, there will be very small overhead from NMon_Exec() for NPut and NGet as no busy waits are used for these two areas of functionality in NMon_Exec().

NPut – Real Time

There are two cases for NPut routines – with NPUT_NO_EXEC defined and without.

With NPUT_NO_EXEC defined, data will be communicated before the NPut function returns, i.e. it will take considerable time.

Without NPUT_NO_EXEC defined, data will be buffered, and return will follow immediately. (Please note that the N_ENTER_CRITICAL_SECTION describe below may affect timing.)

NGet – Real Time

Data received from the debugger is buffered in the target, so if NGet_Available() returns TRUE, a subsequent call to NGet_Getc() will return immediately. However,if the data becomes available in the target, the call to NGet_Getc() will not return until it does.

NAssert – Real Time

Note that NAssert macros will affect real time if the condition is unfulfilled (error). In this case, already existing NPut data to be flushed, and then the assert information will be stored and flushed. This could take up to 60 ms. Make sure that nothing dangerous or harmful can happen as a consequence of this delay. This busy wait will also occur when a NAssert macro is called from a high priority task. The reason being is that we want to get the information out, even if the target application is about to crash.

Not Thread Safe

NMon is by default not thread safe (and is not entirely thread safe in all situations). For most situations, this is probably not a big problem. However, a provision to improve the situation is available. A macro pair is "called" before and after a critical is entered, which by default is translated to "nothing" (see file "NMon_Cfg.h"):

- N_ENTER_CRITICAL_SECTION
- N_EXIT_CRITICAL_SECTION

By providing implementation for these, some of the elements will be protected – most notably NPut (i.e. printf() functionality). A normal action would be to disable / enable interrupts. Another action might be to connect them to a semaphore or similar. NMon guarantees not to have nested "calls" to these macros. However, the consequence of not being thread-safe is small. Basically, text in the Target Console may be scrambled.

> **Note!**
>
> NGet (i.e. getc() functionality) cannot easily be made thread safe, so for predictable results, it should be accessed from one task only, or calls to it need to be protected somehow. Defining N_ENTER_CRITICAL_SECTION has **no affect** on NGet.

Access from Seehau

Note that communication with NMon will not be very efficient for most systems. It is quite possible to overload Seehau by using too many data windows and inspects to that utilize the NMon.

Also, note that the NGet in particular is designed for small quantities of data transfers – i.e. keyboard strokes.

Target Console Window

NPut and NGet can be accessed using the Target Console window.

Data Window

The MONITOR data space is available in the Data Windows. Note that when the target is not executing, data is read/written as done by default.

Inspect Window

The Inspect Window allows selecting “Update During Runtime” which will cause data to be updated periodically using NMon communication.

Symbols Required when NPUT_NO_EXEC

Symbols are not required for Seehau to be able to communicate with NMon. However, if NPUT_NO_EXEC is defined, symbols are required. (The reason being is that there is no complete command handler available in NMon if NPUT_NO_EXEC is defined since there is no execution that drives it.)

Family Specific Notes

ARM - Notes

Both ARM and Thumb mode are supported. The Debug Comms Channel of the ARM core is used to implement NMon communication for ARM. The instructions to access those are only available in ARM mode.

No special method need be used to compile NMon for ARM Mode. However, depending in compiler used, we know of two methods of compiling for Thumb mode.

1. Select compiler – when there is a separate compiler for ARM and Thumb. Examples include the ARM and GNU compilers. Compile DCC.c in ARM mode, all other files in Thumb mode.

2. Function attribute – each function can be specified to be either ARM or Thumb. One example is the IAR compiler. For these compilers, the ARMMODE macro should be defined in DCC.h which should make it automatic to compile the function for Thumb. (Note that “interworking” need to be enabled to make it automatic.)

3 FILE REFERENCE

This is an alphabetical chart that lists of all files in NMon where “Include This” means that the file very well could be included in a target application source file and “User modifiable” means that the user should normally modify the file.

File Name	Description	Usage
DCC.c / DCC.h	ARM specific files for low-level DCC access. The sole purpose file is to allow compiling NMon in Thumb mode. The functions in DCC.c must be executed in ARM mode.	
NAssert.h	Defines <assert.h> style macros for the target application.	**Include this**
NCom.h	Interface to low-level communication routines.	
NCom_XXX.c	Implementation of low-level communication routines for specific CPU (and possibly communication method).	**User modifiable.** Note that the user may have to create this file for a specific compiler.
NMon_CPU.h	Configuration related to specific CPU (and hardware).	**User modifiable.**
NDefs.h.	Definitions used in NMon – data types etc.	
NGet.h / NGet.c	Prototypes & implementation of NGet functionality.	
NLib_XXX.c	Implementation of integration with specific compiler’s run time library.	
NMon.c	The main command handler.	
NMon.h	Prototypes for user functions and global data in NMon.	
NMon_Cfg.h	Configuration of NMon.	**User modifiable**
NMon_Chk.h	Checks that the defined configuration is allowed.	
NMon_Cmd.h	Commands, parameters, return values and other definitions.	
NMon_Cpu.h	CPU/Family specific definitions.	
NMon_Sys.h	General include file for non-user functions.	
NPut.h / NPut.c	Prototypes & implementation of NPut functionality.	
NSrv.h	Prototypes for user defined functions of NSrv.	
NSrv_UserAppl.c	Contains user specific memory read / write.	**User modifiable.** (Should be renamed.)

4 KNOWN ISSUES

IAR Embedded Workbench

Calling printf() & sprintf() from interrupt (IAR)

For some reason, on the ARM IAR compiler, calling printf() or sprintf() from within an interrupt causes NMon communications to fail from there after. In one sample, we called printf() exactly once after 10 seconds of good communication, and communication failed after that. Calling other routines, such as strcpy() had no such affect.

- Calling the low level output functions, such as NPut_Puts() caused no problem.
- Calling the low level output functions, such as NPut_Puts() caused no problem.
- The problem occurred even if printf() was not connected to NPut.

We don’t know why this happens yet, but please beware on other compilers as well.

Optimization

Semihosting (at least read memory) does not seem to work with max size optimization.