NorthTek™ System

Programming Manual

Applicable to Sparton AHRS-8, GEDC-6E, and DC-4E.

sparton

NAVIGATION AND EXPLORATION

5612 Johnson Lake Rd • Deleon Springs, FL • 32130
800.824.0682 • 386.985.4631
www.sparton.com/ne • www.spartonnavex.com

REVISION HISTORY:

DATE	DESCRIPTION OF CHANGE	INITIALS
5/11/11	Original	CMN
9/22/11	Refactored, NorthTek™ Moved to standalone document.	CMN
10/5/11	Review comments finalized.	CMN
10/19/11	Spelling corrections and cleaned up artwork.	CMN
6/10/2013	Corrections	MJB
11/26/13	Changed DC-4 and GEDC-6 to add E version	RSW

1 Introduction 3
1.1 What is NorthTek™? 3
1.2 How is NorthTek™ Used? 4
1.3 Organization and Purpose of this Manual 4
1.4 Preparations for using NorthTek™ 4
1.5 References 5
1.6 Document Conventions 5
2 NorthTek™ Programming Language 5
2.1 Environment 6
2.2 NPL Basics 6
2.3 Standard NPL Words 9
2.4 ANSI Forth Differences 9
3 NorthTek™ Application Environment 11
4 NorthTek™ Database and Compute Engine Interface 11
4.1 Internal Database Identifier Operations 12
4.2 General Database Operations. 13
4.3 Streaming Sensor Data Operations. 13
5 NorthTek™ Programming Examples 14
A NPL Command Categories 33
A.1 NPL Arithmetic and Logical 33
A.2 NPL Parameter Stack Manipulation 36
A.3 NPL Input/Output 37
A.4 NPL Variables, Constants 38
A.5 NPL Database 39
A.6 NPL Logic 40

1 Introduction

1.1 What is NorthTek™?

NorthTek™ is a problem solving system for inertial systems. NorthTek™ is the combination of a programming language, a compiler, an interactive command line interpreter, a database, and a real time computation engine. The compiler-interpreter-execution environment provides an interface into both the database and the compute engine. The combination of all these elements is the NorthTek™

Programming System. The name NorthTek™ came from the underlying components of the system, namely a **N**avigation engine, the F**orth** programming language and **Tec**hnical capability, with a bit of poetic license taken with spelling. NorthTek™ allows an application or integration engineer the ability to tailor a NorthTek™ equipped sensor to the operating environment of a product, even in real time as conditions change with the device in which it is incorporated. This gives the user the ability to solve attitude and heading reference problems that previously could not be solved or could only be solved with expensive or difficult to develop solutions.

1.2 How is NorthTek™ Used?

NorthTek™ is used by downloading a program into the sensor and having that program implement custom algorithms in the sensor with tightly coupled access to the sensing environment. That program then allows the sensor output to be both correct and timely for the application environment. This paradigm breaks the normal sensor paradigm where the sensor merely streams raw data and it is up to the integration system to filter and process the raw data to extract the needed information. NorthTek™ can be used, for example, to limit the sensor output to only specific conditions which may be of interest to the system, or to switch from one set of operating parameters or calibration to another based on changing conditions. This capability does not exist in the traditional streaming sensor paradigm.

1.3 Organization and Purpose of this Manual

This manual is intended to be used by Software Developers and Systems Engineers. For Systems Engineers the manual serves as a capabilities overview. This will assist in determining what custom functions may be implemented in NorthTek™ to aid in system partitioning. For Software Developers this manual serves as the reference for developing NorthTek™ applications.

This manual is roughly organized into the following areas:

- Basic NorthTek™ Programming – This is the basics of a programming language and environment
- NorthTek System Environment – This describes the execution environment of NorthTek as to how and when programs get loaded and executed.
- Sensor Data Base Interface – This describes how the NorthTek language and environment interact with the sensor real time computation engine.
- Application Examples – Sample applications that tie the other sections together.

1.4 Preparations for using NorthTek™

As already discussed, this manual is primarily directed at Software Developers. As such this is not a primer on programming. The expectation is that the user is already fluent in one or more programming languages and is familiar with the real time software operating paradigm. This manual is written assuming that basic programming ideas such as variables, loops, function, etc. do not need further explanation. The user is encouraged to download one of the many PC based Forth interpreters and practice on Forth programming. Sparton can provide a PC based Forth interpreter that closely matches the NorthTek™ Programming Language (NPL) interpreter.

1.5 References

Some documents and online references will be helpful to the reader in learning to program applications in NorthTek™. Namely:

- *Starting Forth*, Leo Brodie (Available online at http://www.forth.com/starting-forth/)
- *Thinking Forth*, Leo Brodie (Available online at http://voxel.dl.sourceforge.net/project/thinking-forth/reprint/rel-1.0/thinking-forth.pdf)
- ANSI Forth Standard (ANSI X3.215-1994)
- Other references may be found here: http://www.mpeforth.com/books.htm
- *Remote Function Select(RFS), Interface Design Description*
- *Sparton Inertial Systems, Software Interface User's Manual*

1.6 Document Conventions

Typographic conventions are used in this document to highlight items that may be typed or transmitted to the sensor or output from the sensor.

Descriptive text in this document appears in this font (Calibri 11).

Data transmitted *to* the inertial system appears in ***this font (Courier 11, Bold Italic).***

Data sent *from* the inertial system appears in **this font (Courier 11, Bold, grey background)**.

Information that must be supplied by the user is enclosed in **{}** (curly brace) pairs. The **{contents}** brace pair should be replaced with the user supplied data or will be filled in by the inertial system according to the description contained in the **{}**'s. The curly braces are not part of the transmitted characters. Control sequences, or other non-printable characters are enclosed in **<>** pairs. The following printable shorthand for some non-printable characters is used:

- A carriage return character is shown as **<CR>** or **<cr>.**
- A line feed character is shown as **<LF>** or **<lf>.**
- Control sequences are indicated as **<CTRL-S>.**

Other non-printable characters are spelled out in **<>** pairs, for example **<ESCAPE>** is equal to the escape character (0x1b). The hexadecimal value will be used if needed for clarity. Additional text will be included when the non-printable characters might not be clear to the reader.

The blank or space character will be represented with ~~b~~ (an overstrike b).

2 NorthTek™ Programming Language

NorthTek™ Programming Language (NPL) is a slightly modified Forth programming language. NPL does not strictly adhere to the ANSI standard due to its embedded nature, however the intention is for it to be familiar to someone already proficient in Forth programming and environments. The core language is largely ANSI compliant. There are many references available in print or online for learning to program Forth. An experienced programmer will likely review one of the various references and pick the language up quite quickly. The appendix provides a listing of the words in the dictionary along with the

ANSI equivalent word. This section provides some of the basic concepts of NPL and includes some of the major differences between NPL and ANSI Forth.

2.1 Environment

The NorthTekTM environment is a simplified model. The inertial system provides a command line user interface on the serial port. The user connects to the inertial system with a simple terminal emulator, such as TeraTerm or Hyperterminal. The user can type commands by hand or send the commands as files using the simple file transfer facilities built into these terminal emulators. Note that NorthTekTM is an interpreter that is running as a low priority task in a multitasking real time environment in a resource limited microcontroller. To give the interpreter time to process each line of text, a small amount of timing delay (5 msec) must be added to each line of a file transfer for baud rates above 38.4k. In TeraTerm this option is available on the serial port setup menu.

NorthTekTM prompts the user with an “OK” when a carriage return is entered; this indicates that the interpreter is ready for input. Note that the inertial system sensors support multiple protocols on the same serial port. Several of the protocols are binary and start command sequences with non-printable characters and should not be a problem, however the NMEA protocol starts command sequences with a ‘$’, so if the user accidently begins a line with ‘$’ the interpreter shifts to NMEA mode until the NMEA closing character is received. If this happens, the user should simply enter a newline <CTRL-J> to complete the NMEA command.

As there is no disk or other mass storage, NPL does not implement many of the standard file operations found in a PC based Forth. Instead equivalent methods are provided to save and load programs. Test programs will be stored on the user’s computer and sent to the inertial system as needed using the file send function built into most popular terminal emulators.

2.2 NPL Basics

The reader is strongly encouraged to obtain one of the references and learn the basics of the Forth programming language before proceeding. However if the references are not immediately available this section will serve as a brief introduction. Note that this cannot be a thorough treatise on programming with a new language because as most readers are well aware, doing so would cause the size of this document to reach very large dimensions. This document is not designed to teach a non-programmer how to program. This section is merely a brief overview of NPL. It is suggested that the user have the inertial system and terminal emulator handy and attempt some of the examples. NPL/Forth is an interactive language is typically best learned by experimenting.

NPL is a stack based language and operates similar to an HP[1] type calculator. Any number that is entered is pushed on the operand stack. Any arithmetic, logical or binary operators operate on the items on the operand stack. NPL does not use parenthesis, instead it relies on postfix notation. Postfix notation resolves operator precedence naturally.

[1] HP refers to Hewlett Packard Company and is a registered trademark of the same. No claims are made to any HP technology.

In NPL the dot "." operator prints the top of the stack as an integer (f. does the same but prints as a floating point value). Thus the following sequence first pushes two values then prints them. NPL prints OK when it is ready for more input.

```
1ƀ2ƀ.ƀ.ƀ2ƀ1ƀOK
```

NPL is a simply parsed language. The language is simply a series of tokens separated by white space[2]. In some cases (primarily in string operations there is a slightly different rule that is explained in the formal language description in the references) the rule is relaxed, however for beginning purposes all tokens are separated by white space. There are no rules about how many tokens can appear on an input line or if control structures may span multiple lines. A token is a series of printable alphanumeric characters that contains no spaces.

The NPL interpreter is composed of a set of individual commands called "words". One individual NPL command is called a "word". All of the NPL "words" are grouped together logically into lists called "wordlists". The word "***words***" will cause NPL to print out all the words in all the wordlists in reverse definition order. The wordlists are arranged in a reverse chronological order. The combination of all wordlists is called the "dictionary". When a new word is created is said to be "defined". Each new word that is defined is added to the end of the last wordlist (the first one displayed when "words" is entered).

Tokens are either words in one of the wordlists or numbers. This bears repeating for emphasis. All tokens are either words, i.e. commands in the dictionary, or numbers in the current number base.

In the example above the input line that was typed was "1 2 . . <cr>". NPL's interpretation of this line is to first find the first token "1" and see if it is a word in the dictionary. Not finding it NPL attempts to make a number out of "1" and is successful, so the binary number 1 (after converting the string "1" to a numeric value) is pushed on the stack. The next token, "2" is handled in the same manner, thus pushing 2 onto the operand stack. The third token "." is found in the dictionary and is thus immediately executed, causing the top item on the operand stack to be (destructively) printed. The last token is handled in the same way causing "1" to be printed. Finally the end of the input line is reaches and NPL prints "OK" to indicate that it is ready to interpret some more commands.

There are a few points of confusion for programmers in traditional languages. In traditional languages there are tokens called delimiters. For example in "C" the semi-colon is a delimiter and is used to terminate a command. For example "x=7;" is a valid "C" statement and is equivalent to "x = 7 ; " (xƀ=ƀ 7ƀ;) . "C" does not care if there is white space between the 7 and the ";", nor does it care if there is space surrounding "=". NPL is different in that ";" is a command itself and not syntactical padding, thus it must be surrounded by white space. This is especially confusing for beginners when dealing with comments. In NPL "//" makes the remainder of a line a comment. However it is not a pre-processor token that is parsed at compile time, it is a command that is executed at runtime and must be surrounded by white space, both before and after. For example the following statements:

```
7//comment

7ƀ//comment

7//ƀcomment
```

[2] White space is either a carriage return, newline or a space character. Tab characters are not used in NPL.

While appearing to be legal, and would in fact be legal in "C", these statements cause NPL to emit `HUH?`, indicating that the token cannot be found in the dictionary or made into a number. This is because the first statement has only one token "7//comment", that is neither word nor number. The second statement has two tokens "7", a valid token, and the token "//comment", that is neither word nor number. The third statement has two tokens "7//" and "comment" neither of which are valid tokens. The correct syntax for NPL would be:

```
7ƀ//ƀcomment
```

For the remainder of this document the ƀ symbol will be omitted unless it is particularly helpful or important to enhance readability. Should the reader encounter difficulty with commands that appear as though they should be working, the reader should review this section and recall that all input must be either a word or a number surrounded by white space.

Adding a new word to the dictionary is straightforward. The word colon ":" indicates that the user wishes to define a new word. What follows the ":" is the name of the new word. The name must not include any embedded white space. As an immediate reminder to the preceding section the ":" must be surrounded by white space. A common mistake is to run the ":" and the name of the word together without white space. Thus

```
:mynewword
```

will cause NPL to emit an annoying `HUH?` because there is no word in the dictionary called ":mynewword".

To make our previous example into a word the user would type:

```
: myword 1 2 . . ;
```

The ";" terminates the definition of a new word. The user can then type "***words***" to see that myword has been added to the dictionary. Note that to be successful, all the tokens must be surrounded by white space, including the ":" and the ";"[3]. The user may execute the new word by entering "***myword***" to which NorthTek™ will output:

```
2 1 OK
```

The NPL stack is the mechanism by which arithmetic is performed and parameters are passed. The use of the stack is illustrated in the following examples. These examples utilize one of the fundamental control structures in NPL, the "do" loop. The syntax is straight forward; two values are pushed on the stack, the terminal value, and the initial value. The "do" word marks the top of the loop, and the "loop" word marks the bottom of the loop. The do-loop structure can only be used inside a word definition, that is, while compiling. Inside the loop, the word "i" pushes the current loop index on the stack. Thus

```
: count 10 0 do i . loop ;
```

defines a word that counts from 0 to 9 that prints the loop index each time. Executing count causes the following output:

```
0 1 2 3 4 5 6 7 8 9OK
```

[3] Because ":myword" and ".;" are not words found in the dictionary.

In this example the loop ranges are compiled into the word. When the word is executed the two values are pushed on the stack and the "do" word finds the two required loop parameters on the stack when it begins executing. However the loop indices may be passed to the "count" word when it is executed. Either one or both may be passed as shown in the following examples.

```
: countFrom0 0 do i . loop ;

10 countFrom0 0 1 2 3 4 5 6 7 8 9OK

: countFromMtoN do i . loop ;

20 7 countFromMtoN7 8 9 10 11 12 13 14 15 16 17 18 19OK
```

The dictionary grows with each word that is defined. The user may execute the word "status" to see how much of the free dictionary space remains. The dictionary may be pruned with a word called "forget". Forget releases all the words in the dictionary from the end back to the word passed to forget. If the user has been following with the examples, the dictionary will now include the words "myword", "count", "countFrom0" and "countFromMtoN". The user can clean up the dictionary be executing "forget myword". This is important when learning NPL to avoid a runtime error caused by exceeding the allowable dictionary space.

2.3 Standard NPL Words

The NPL dictionary is described in one of the appendices. Where there is a corresponding ANSI Forth word it is listed as a cross reference and the user can expect the behavior to be as described in the ANSI Forth standard. The Cell size for NPL is 4 bytes. The word size used by w@ and w! is 16 bits.

2.4 ANSI Forth Differences

NPL has, like many Forth implementations, its own minor differences. Some of these differences are minor and will be dealt with on a case by case basis. Some of the more significant differences will be described in this section.

2.4.1 Case

NPL, like most Forth interpreters, is case sensitive. In NPL, all language primitives are in lower case, thus constructs like "do", "if" and "then" are as written here.

2.4.2 Comment lines

NPL allows the use of //ƀ[4] as a comment that continues to the end of the line. The traditional "(ƀ" and " ƀ)" pair may also be used. Examples of comments:
//ƀ comment
(ƀ traditional forth commentƀ)

2.4.3 Expanded Data Types

NPL allows entry of hexadecimal and binary numbers without changing base. Hexadecimal numbers may be entered as 0xDD; binary numbers may be entered as 0bDD, regardless of the current number base of

[4] Reminder: This is a space character, not the letter b.

the interpreter. Thirty two bit floating point values are supported. Floating point values may be input starting with an F and must include a decimal point somewhere in the number. Exponential notation is supported. For example F1.0, F-2.5, F1.234E-6, and F123.4E5 are all valid floating point numbers. Floating point numbers are always entered in base 10, regardless of the current number base. A full floating point library is included as are input/output routines.

2.4.4 Strings during Interpret Mode

Traditional Forth interpreters allow compile time strings with the ." and $" operators. These create a string variable that the "address-of" is provided when the compiled word is executed. NPL allows the use of an interpret-time string that is transient. To create a transient interpret-time string, simply start string with "ƀ and close with ". As with ." and $" there must be a space following the opening " but not one before the closing ". At interpret-time the " operator creates a string[5] up to 80 characters long in a temporary space and leaves the address on the stack. The internal storage for the string is unmodified until another interpret-time string is created or an interpret-time array is created.

2.4.5 Arrays

NPL provides an interpret-time temporary array of up to 16 elements. The words array[and]array are used to create a temporary array. The syntax is ***array[{lower index} {upper index} {lowest element} … { upper element}]array***. Arrays are 0 based indices like other programming languages. At interpret-time, the address of the array is left on the stack. Note that the lower and upper indices are the first two elements in the array. Both fixed and floating point values can be put into the array.

The ***array[*** ,***]array*** pair can be used in a compilation. During compile the operators create the same structure as the interpret-time operators but the data is compiled into the word, and thus is not transient like the interpret-time version of the array that will be overwritten with the next array or interpret-time string.

2.4.6 Define

Define (all lower case in usage) is a word that operates identically to "constant" in a traditional Forth (constant is still available). Define is both faster and takes less memory than constant.

2.4.7 Flow Control

Flow control is actually a inertial system function but nevertheless applies to the NPL environment. When NPL is producing continuous output it may be temporarily suspended by sending ***<CTRL-S>*** and resumed with ***<CTRL-Q>***. This is often helpful in typing a command to disable the streaming output.

2.4.8 Quiet and Loud

Quiet is a word that suppresses all the Forth primitives when "words" is executed. The word loud restores the Forth primitives to the "words" display. Quiet is the default.

[5] The string is a Forth style string with a count byte that contains the number of characters in the string including a trailing null.

2.4.9 Help

NPL contains a "help" facility. Some words contain an extended string that provides further description of the word's behavior. The syntax is "help {word}". If there is a help string associated with that word it, will be printed.

2.4.10 Grep

The NPL wordlist is large. The traditional command "words" to see all the words in the wordlists produces a lot of output. The inertial systems have a lot of variables and it may sometimes be difficult to locate the correct variable. Grep (grep as typed) is available to search for words with only a substring. The syntax is "grep {substring}". The substring and target are compared in a case sensitive way. Thus if the word that is being searched for is thisIsTheVariable and the user believes that "isthe" is part of the variable, the search substring must be "IsThe".

3 NorthTek™ Application Environment

NorthTek™ is a user interface process executing on the inertial system. It is running at a low priority in the system so that it does not interfere with the real time calculations that are being performed to compute the attitude and heading of the inertial system. NorthTek™, via an NPL program, is used to monitor the heading, change the way the heading is computed, or control the output of the heading information to the user. NorthTek™ has the ability to cause other inertial system protocols to be transmitted (e.g. RFS messages). NorthTek has the ability to modify the calibration, in-field calibration, or boresight matrix in real time to adjust to changing conditions on the host device.

NPL programs are loaded onto the inertial system by simply transmitting the text file to the inertial system after power up. Multiple programs may be loaded as long as the dictionary space is not exceeded. Programs may be loaded, executed and removed using the "forget" command as many times as the user needs with no ill effects on the inertial system. A future feature will allow a NPL program to be permanently stored in the inertial system and automatically executed at power up or upon recovery from low power mode. Currently all user programs are volatile and lost once the inertial system has been powered down or put into low power mode. Once a program is loaded the commands can be executed by simply sending the text string containing the command name.

4 NorthTek™ Database and Compute Engine Interface

A Sparton inertial system contains an embedded database. This database contains both the calibration, configuration and sensor readings in a unified structure. Some of the data items are transient and do not survive a power cycle, inertial system heading is one example. Some of the data items are non-volatile in that when set they will be the same after a power cycle, baud rate is one example. On a different axis some of the non-volatile items are only settable at the factory. The user can set the value temporarily but the factory set value will be recovered after a power cycle or low power cycle. Some values are read-only, such as the device name. Some variables are used as "triggers" that do not carry any real data, but instead cause some action to take place, such as evaluation of the World Magnetic Model, or the sampling of a calibration point. Regardless of whether or not the value is volatile or non-volatile, or factory settable only or not, the inertial system software provides a uniform method for reading and writing the values. A binary method for manipulating the database variables is described in the *RFS Protocol Suite* documentation. NPL provides a more human friendly interactive interface into the database that mirrors the RFS functionality. The list of database variables for a particular device can be found in the Software Interface User's Manual for the specific device and will not be completely enumerated in this document. This document will provide some examples of NPL programs that use common inertial system variables.

4.1 Internal Database Identifier Operations

In an inertial system, each database variable has a name. All the database names are pre-defined at startup and are visible in the dictionary. Executing a variable name causes NorthTek™ to leave an internal database identifier (IDI) on the stack. The user cannot manipulate this value directly; instead, this identifier can be passed to other NPL words to cause an action on that particular variable. The list of words that can operate on a database identifier is:

di.	The current value of the item is printed in a human friendly form.
>default	Sets the item to the factory default value. Note that if the value is non-volatile it will not be stored in permanent storage unless "save" is performed on the database.
di@	Gets an objects value. For Int32's, ordinals, fixed and floating point values, the value returned is the actual value. For arrays, the value returned is the address of the first element. For strings, the value returned is the address of the count byte.
di>	Forces the inertial system to output the value of the variable in RFS protocol with a binary "Value_Is" message.[6]
di?	Returns true or false as to whether or not the variable value is within range limits.
&di	Return the address of a data item. The value is identical to di@ for arrays and strings. For other types the address of the actual database item value is returned. Can be used to read or set an item directly in real time. Caution must be used to ensure that the type of the variable is known and correctly used. This can be a pointer to Int32 or pointer to Byte, so misuse of this pointer can be problematic.
set	Expects an IDI, then a value on the stack. The value can be a single scalar, an array using the array[ ]array words, or a string using the " interpret mode string command. See examples in the later sections for how to use this word. If the variable is non-volatile the NorthTek™ system will commit this variable

[6] Note that a properly written host application can accept NPL commands and RFS messages at the same time. Since NPL uses all printable characters with the exception of carriage returns a front end multiplexer can parse for the SOH character that starts an RFS message and send all intervening characters to the RFS processor up until the ETX. All characters received outside the SOH/ETX pair can be passed to the user terminal. Thus an NPL program can provide human readable output and interaction while RFS output is being supplied either on demand or continuously. On the output side the host application must only ensure that the printable NPL output is not allowed between a SOH and ETX as the sensor will interpret this as part of an RFS packet. This is, in fact, the algorithm used by the sensor to simultaneously accept RFS, NMEA, NPL and Legacy Binary protocols.

	to non-volatile memory after a 5 second delay. The delay allows multiple set commands to be executed in rapid succession and only one non-volatile write to be performed for the combined set operations.
show	Causes NPL to output the RFS "Format" message for this variable.
rec	Set a variable with the same syntax as "set", however no non-volatile variables are immediately committed to non-volatile memory. This command can be used to try out new settings without committing to non-volatile storage. Note however that performing a "set" operation after a "rec" operation will write **BOTH** to non-volatile memory. Also the inertial system writes dynamically computed adjustments into the database on a periodic basis that are intended to be written to non-volatile memory. This will also cause any variables set with "rec" to also be written. Contact Sparton Technical support should you need to use this command to fully understand the ramifications of using this command.

4.2 General Database Operations.

In addition to the words that operate on individual IDI's, other NPL words manipulate the database overall. They are

save	Causes NorthTek™ to commit all non-volatile variables to non-volatile storage.
defaults	Causes NorthTek™ to set all variables to factory default values. **Warning: this will reset all user calibration and configuration to factory default values.**
wmm.	Print the world magnetic model data. Not strictly part of the database.

4.3 Streaming Sensor Data Operations.

NorthTek™ provides some controls to the real time inertial system engine. These controls primarily control the printing of sensor data both raw and processed. Because the raw sensor data is being continuously sampled, it does not get written to the database. Processed sensor data gets written to the database but is constantly overwritten as new values are computed. To allow the user to capture the streaming data NorthTek™ allows the printing of the data with some NPL control words. The streaming control words are of two types. One set enables or disables the streaming output. The other set of words changes the number of samples obtain (the modulus) before raw samples are output. Processed samples are printed at the nominal processing rate. The modulus words allow the user some control over data reporting rate. All the enable/disable words expect a true or false value on the stack to enable or disable the output. The enable/disable words are:

accel.p	Prints the raw accelerometer data. The format is "A:%6d,%6d,%6d,%6d\r\n", with arguments: timestamp,x,y,x
accelp.p	Prints the processes accelerometer data. Format is "AP:%d,%d,%d,%d,%f,%f,%f,%f,%f\r\n", with arguments: timestamp, xraw,yraw,zraw,xprocessed,yprocessed,zprocessed,pitch,roll.
gyro.p	(GEDC-6 only) Prints the raw gyro data. Format is

	"G:%6d,%6d,%6d,%6d\r\n", with arguments temperature,x,y,z (DC-4 only) Prints timestamp and temperature. Format is "G:(%d)%6d\r\n", with arguments timestamp, raw temperature coefficient
gyrop.p	Prints the processed gyro data (GEDC-6 only). Format "GP:%d,%d,%d,%d,%.2e,%.2e,%.2e\r\n", with arguments: timestamp, xraw,yraw,zraw,xprocessed,yprocessed,zprocessed.
mag.p	Prints raw magnetics. Format "M: %d, %d, %d, %d\r\n", arguments timestamp,x,y,z.
magp.p	Prints processed magnetics. Format : "MP:%d,%d,%d,%d,%f,%f,%f,%f\r\n", arguments timestamp,xraw,yraw,zraw,xprocessed,yprocess,zprocessed, yaw(magnetic)
quat.p	Prints quarternian outputs. Format : "QUAT:%f,%f,%f,%f\r\n", arguments w,x,y,z.
s.p	Prints all the raw samples in one line. Format (GEDC-6): "%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d\r\n", arguments are timestamp,magx,magy,magz,accelx,accely,accelz,gyrox,gyroy,gyroz,temperatureCoefficient. Format (DC-4) : Format "%d,%d,%d,%d,%d,%d,0,0,0,%d\r\n", arguments are: timestamp, magx, magy, magz, accelx, accely, accelz, temperatureCoefficient.
compass.p	Prints the computed attitude and heading. Format "C,%d,%7.2f,%7.2f,%7.2f\r\n", arguments are :timeStamp, pitch, roll, yaw

The modulus words expect the modulus value on the top of the stack. The modulus words are:

gyro.mod	Sets the raw gyro data print modulus. Default is 10. This value is volatile and resets to 10 on power up.
accel.mod	Sets the accelerometer raw data print modulus. Default value is 4. This value is volatile and resets to 4 on power up.

5 NorthTek™ Programming Examples

5.1.1 Simple NPL examples

At this point a few detailed examples may be helpful:

5.1.1.1 Print Sensor Heading Example

This example reads the current inertial system heading.

```
    yaw di.<CR>
```

NPL prints:

```
    yaw = 4.433077e+01
    OK
```

5.1.1.2 Setting the baud rate

This example shows how a variable may be set to implement a configuration change. In this case the user is going to change the baud rate. The user should inspect the baud database parameter and values in the appropriate device user manual. In this case the baud rate index = 4 is 9600 baud and baud rate index = 8 is 115200 baud. Assuming the inertial system is already operating at 115200, the user enters:

```
        baud 4 set<CR>
```

(The serial port on the user's terminal needs to be changed to 9600 at this point.)

Commands may now be entered at 9600 baud. The user can return to 115200 by entering the command that follows and then changing the user's terminal baud rate back to 115200.

```
        baud 8 set<CR>
```

5.1.1.3 Print Acceleration Example

This example shows how an array is printed. In this case the raw and processed acceleration value.

```
        accelr di.<CR>
        accelr = 00-- -2.080000e+02
        01-- 9.760000e+02
        02-- 1.408000e+04

        OK

        accelp di.<CR>
        accelp = 00-- 1.660156e+01
        01-- 2.832031e+01
    02-- 1.044259e+03
```

5.1.1.4 General Examples

NPL is capable of fixed and floating point arithmetic, bitwise logical operations, and many other useful "calculator" type commands. Here are some simple examples. For the readers convenience the inputs are commented in NPL style.

Simple Arithmetic.

```
        1 1 + . // push 1, push 1 add the top two items
                // from the top of the stack and print
        2 OK
        10 4 - . // push 10, push 4 subtract 4 from 10 and print
        6 OK
```

Arithmetic in other number bases.

```
        hex 1234 . // print 0x1234 in hex
        1234 OK
        1234 decimal . // push 1234,
                      // change NPL I/O base to decimal
                       // print 0x1234 in decimal
        4660 OK
```

5.1.2 NPL Programming Examples – Advanced

This section will provide several NPL examples, starting with the most basic up to some more advanced topics. Given that most software developers like to see well commented examples that provide a sample of single subject, the examples will be just that. Keep in mind that NPL is just a programming language albeit with postfix notation and a simple delimiter model (token-whitespace-token, etc.).

5.1.2.1 Basic NPL Programming

5.1.2.1.1 The Basic "Hello world".

For this example the ƀ will be used to show the required (and non required) whitespaces

```
        :ƀgreet                ƀ//ƀstart a new word called greet
        ."ƀHello World"        ƀ//ƀmake it print the standard
        cr                     ƀ//ƀoutput a carriage return
        ;                      ƀ//ƀfinish the word
        // Now execute it
        greet<CR> Hello World
        OK
```

This example can be done multiple ways to illustrate the free form nature of the language.

```
        : greet ." Hello world" cr ;    // as a single line
        : greet ." Hello World"
        cr ;    // as two lines, note required
                // whitespace between cr and ;
```

5.1.2.1.2 Passing arguments to a function

All operators expect a stack based argument. The arguments can be compile in or passed at execution time. For example to add 2 + 2.

```
        : twoplustwo 2 2 + . ; // add and print 2 and 2
        // execute it
        twoplustwo<cr>4 OK
```

Can also be

```
        : plustwo 2 + . ; // expects the argument on the stack at runtime
        // execute it, note pass one of the arguments now, at runtime
        2 plustwo<cr> 4 OK
        // Finally both args at runtime
        : plus + . ; // expect both
        // execute it
        2 2 plus<cr> 4 OK
```

5.1.2.1.3 if-then-else

The if clause is slightly non-standard. The syntax is if { true stuff} else { false stuff } then { continue on with other stuff}. The if tests if the top of the stack (TOS) is non-zero for true. Thus the following simple example:

```
: test if ." true" else ." false" then cr ;
// execute it
1 test<cr>true
OK
0 test<cr>false
OK
```

Enough said.

5.1.2.1.4 do-loop

The do loop is straight forward; do expects two values on the top of the stack, the starting index and the terminal count. The loop executes over starting value to terminal count -1. Thus the following example prints the integers from 0 to 9. Note that the command i returns the current loop index. The second example shows a nested loop and how to access the index of the outer loop.

```
: count do i . loop ;
// execute it
10 0 count<cr>0 1 2 3 4 5 6 7 8 9 OK
: nested do i 0 do i . j . loop  cr loop ;
// execute it
   5 0 nested //
          // Blank line printed here
0 1
0 2 1 2
0 3 1 3 2 3
0 4 1 4 2 4 3 4
OK
```

5.1.2.1.5 begin-while-repeat

The begin-while-repeat structure implements either a top tested loop (while) or a bottom tested loop (until) simply by placing the while command at the appropriate place in the loop. The syntax is begin {stuff that is always done} while { stuff that is done if the condition at while is true } repeat. The little loop below is a workhorse kind of loop that continues until the user hits escape. The second example is a loop that repeats until any key is pressed.

```
: go
begin
    key dup 27 = 0=   // read a key, make a
                      // copy of it, compare it to
                      // 27 then invert the logic "0="
                      // note the 0= is one token by itself
```

```
while // execute from here to
      // repeat if the key was NOT escape (27)
." You hit" emit cr  // print the key that the user pressed
repeat // back to while
drop   // get rid of the spare key when escape was hit
; // all done
```

Try running the example on the inertial system, by entering the word and then typing ***go<cr>***.

This loop runs until the user presses any key.

```
: go2 begin ?key 0= while ." Hey Hey Hey" cr repeat ;
```

Also try this on the inertial system, by entering the word and typing ***go2<cr>***

5.1.2.1.6 variables

NPL allows the declaration of arrays and constants. The word "variable" followed by a name declares a single 32 bit value. The value can hold and integer or a float32. Using the variable name returns the pointer to the name. The user is required to explicitly read or store using the pointer value.

Arrays are constucted by "allot"ing to the end of a variable. The user must write the access functions to index the array. NPL does not do typed pointers, it is the responsibility of the user to do the correct scaling of pointers to get to the correct addresses. This is identical to the standard Forth language and any quick internet Forth tutorial will provide many examples. Some simple examples follow:

```
variable x // create 32 bit value x

x @ . // read and print x

1234 x ! // store 1234 in x

5678 x ! x @ . // store 5678 in x read it and print it

//  create a buffer that is 10, 32bit values
// variable creates one, allot adds 9 more (not initialized!)
variable buffer 9 allot

// create a word to index the array for read
// expects an index on the top of the stack,
//followed by variable location
: v[]@ 4 * + @ ;

// does the store operation expects ( value variable index -- )
: v[]! 4 * + ! ;
buffer 0 v[]@ . // print the 0th element
1234 buffer 7 v[]! // store 1234 in the 7th element
```

5.1.2.2 Common Sensor User Commands

The table of NPL commands (Appendix A, NPL Command Categories) is at first overwhelming. However like any complex system the user generally uses only a small percentage of available commands. This section describes the most common NPL commands that an interactive human user might need to operate the inertial system.

5.1.2.2.1 Flow Control

NPL allows the user to temporarily suspend any kind of streaming data. The ***<ctrl-s>*** character is used to suspend output and the ***<ctrl-q>*** resumes output. The user may enter commands in between the characters and interact with NPL as usual, however streaming data is suspended. For example the command 1 accel.p causes the inertial system to emit a continuous output of accelerometer data. The user may enter ***<ctrl-s> 0 accel.p <ctrl-q>*** to stop this output. The ***<ctrl-s>/<ctrl-q>*** pair basically allows you to see what you are typing.

5.1.2.2.2 Obtaining Sensor Data

The following commands enable the raw and processed sensor data to be streamed out the user port at a high rate. For these commands the <cr> is not printed.

5.1.2.2.2.1 Accelerometer

1 accel.p // enables print of raw accelerometer readings

// Output format is

// A:%6d,%6d,%6d,%6d

// printing timestamp(ms), x, y and z accelerometer values

0 accel.p // disables

1 accelp.p // enables print of raw/processed acceloremeter readings

// Output format is

// AP:%d,%d,%d,%d,%f,%f,%f

// printing timestamp(ms), x, y and z accelerometer raw values and x,y and z linearized values

0 accelp.p // disables

5.1.2.2.2.2 Gyro

1 gyro.p // enables print of raw gyroscope readings

// Output format is

// G:%6d,%6d,%6d,%6d

// printing timestamp(ms), x, y, z values

0 gyro.p // disables

1 gyrop.p // enables print of raw and processed gyro readings

// Output format is

// GP:%d,%d,%d,%d,%f,%f,%f

// printing timestamp(ms), x, y and z gyro raw values and x,y and z linearized values

0 gyrop.p // disables

5.1.2.2.2.3 Magnetometer

```
1 mag.p // enables print of raw magnetometer readings
// Output format is
// M:%6d,%6d,%6d,%6d
// printing timestamp(ms), x, y and z magnetometer values
0 mag.p // disables

1 magp.p // enables print of raw and processed magnetometer readings
// Output format is
// MP:%d,%d,%d,%d,%f,%f,%f
// printing timestamp(ms), x, y and z magnetometer raw values and x,y and z linearized values
0 magp.p // disables
```

5.1.2.2.2.4 All raw sensor data

```
1 s.p // enables print of all raw sensor data
// Output format is
// %d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d
// mag x,y,z, accel x,y,z, gyro x,y,z and raw temp respectively
0 s.p // disables
```

5.1.2.2.2.5 Computed Roll, Pitch and Yaw

```
1 compass.p // enables print computed attitude and heading
// Output format is
// C,%d,%7.2f,%7.2f,%7.2f
// timestamp(ms), roll, pitch and yaw respectively
0 compass.p // disables
```

5.1.2.2.2.6 Computed Quaternions

```
1 quat.p // enables print of computed quaternions
// Output format is
// QUAT:%f,%f,%f,%f
// quaternion vector (w,x,y,z) respectively
0 quat.p // disables
```

5.1.2.3 3D Sensor Calibration

This script performs 3D inertial system calibration. To perform calibration use a dumb terminal to send this to the inertial system (if you see huh? flying by, add some delay to each line transmitted or reduce the baud to 38400). Once the script has been loaded, send the command "cal3D". The script will then prompt the user to capture calibration points. The user should move the inertial system around and select between 4 and 12 points, then hit ESC. The script will then print the magnetic error at a 0.5 Hz rate. Observe the magnetic error until it converges sufficiently then hit the spacebar or any other key. The inertial system will now be field calibrated. The cal3D command may be re-entered as many times as desired without re-loading the macro. The macro needs to be loaded only once while the inertial system remains powered up. There is enough information in the basic programming section and the appendix to analyze this macro sufficiently should the user desire to extend this functionality to other custom user scripts.

```
: cal3D                                    // start compiling a new word, called cal3D
calmode 1 set                                      // set the calibration mode to 3D
." Calibration starting" cr                // tell the user we are starting then <CR>

calCommand  cal_start  set                        //  issue  the  cal_start  command

." Press any key to take next point, ESC to finish" cr  // Tell the user what to do

//    The    code    now    grabs    a    point,    the    user    changes    the
//     inertial     system     position     and     repeats     the     cal3DState
//               from               4-12               times               total.

begin                                             // begin  a  begin-while-repeat  loop
   key 27 = 0=                        // wait for key input compare with ESC
                                      //       invert       the       logic       (0=)
   while                              // only continue if the TOS has a true,
                                      //    i.e.    user    hit    a    key    !=    ESC
      calCommand cal_capture set      // take another point
      250 delay                          // wait 250 msecs to allow point to be counted
       calNumPoints  di.                                 //  print  out  point  number
repeat                                                   //  end  of  begin-while-repeat

// Now the points are captured,
// command the SW to compute the cal values:

calCommand cal_end_capture set        // issue command to end the capture of points

."  Starting  error  settling"  cr                 //  tell  user  what's  going  on
."   Press   any   key   to   terminate"   cr              //   issue   instructions

// The user observes magErr to watch it settle
// at a minimum value (EKit can display every sec or so):

begin                                   // keep printing magErr at .250 sec intervals
  ?key 0=                             // ?key returns false hit a key is hit
                                      //       0=       inverts       so       then…
    while                                        // while tests for no key pressed
     magErr  di.                                          //  print  the  mag  error
     250  delay                                   //  wait  250  msecs  to print  again
repeat                                                   //  end  of  begin-while-repeat
." Calibration done!" cr                    // cellll1-e-braaaate good times, come on!
calCommand   cal_end   set                                    //   cal   computation
```

```
calmode  0  set                                       //  terminate  but  "I'll  be  back"
;                                      // end of compiliation
// To execute the calibration simply type
// cal3D
// Or uncomment the line above and the script will run when it is loaded.
// To unload this program type
// forget cal3D
```

5.1.2.4 Don't Forget forget

The memory space for new NPL words is, unfortunately, not unlimited. As a user begins to experiment with writing NPL words, the wordlist will fill up (enter the status command to see how full). Fortunately there is a way to forget all the words that have been defined by the user. The command ***forge***t followed by a word will erase all defined words back to and including the named word in the wordlist (think reverse linked list model) (you cannot ***forget*** words that are defined before you start your definitions).

5.1.2.5 Four More Special Words

There are a few special words that don't strictly obey the postfix notation of NPL. They are ***:***, ***."***, ***constant***, and ***variable***. (Pronounced : colon, dot-quote, constant and variable). They are unique in that they expect arguments to follow in the input stream.

The word ***:***(colon) expects a (whitespace delimited) token to follow in the stream. It begins compilation of a new word in the wordlist. The reason that the notation is not postfix is that the token can't already be in the wordlist since it is being currently defined. If the as yet non-existent token is encountered by the parser it would just print "huh?" and give up on the line of input. The colon operator gets around this by looking forward and removing the next token in the input line so the basic parser never sees it.

Constant and ***Variable*** have the same behavior as colon, just instead of making the new word an executable word they create a ***constant*** or ***variable*** respectively. Note that constant expects the constant value on the top of the stack before the ***constant*** word is executed.

The word ***."***(dot-quote) is often confusing because it appears to violate the whitespace rules. However once understood it is simple. The ***."*** word is used to compile in a string to output to the user. The ***."*** word looks ahead in the input buffer just like colon, constant and variable do, however instead of parsing the tokens by whitespace, it parses up to the next " (quote) mark. This allows for whitespace in output strings (which is to be expected). Since ***."*** is a command, whitespace must follow it so the basic NPL parser can identify it, but once ***."*** is executing it looks ahead in the input buffer for a closing " mark. Whitespace is not needed before the closing " mark, since ." is doing its own parsing instead of the normal NPL parser. So by example to print the classic "Hello World" in a forth macro the syntax is

."ƀHello World"ƀcr

In this example, whitespace is represented by the literal ***ƀ*** character. The ***cr*** at the end of the line prints a carriage return.

5.1.2.6 Dealing with Arrays

NPL allows setting of arrays. The syntax is

arrayVariable array[{starting_index} {ending_index} {v $_{starting}$ ***v*** $_{s+1}$ ***.. v*** $_{ending}$***}]array set***

5.1.2.7 Advanced Applications

This section describes several advanced applications. To run these programs:

1) Copy the text below to a plain text editor (i.e. Notepad, do NOT use Word,etc.).
2) Save the file to some known location on the computer.
3) Start a terminal program such as Hyperterminal or TeraTerm. (TeraTerm is recommended if technical support is needed.)
4) Configure the terminal program for 115200,8N1. Add 5 milliseconds line delay to the serial port configuration (Hyperterminal : >File->Properties->Settings Tab->ASCII Setup, TeraTerm: Setup->SerialPort). Make sure the new lines are setup for CR.
5) Use the file transfer option to send the text file to the inertial system. (HyperTerminal: Transfer->Send Text File, TeraTerm: File->SendFile.)
6) Scroll back through the download and look for "Huh?". This is NorthTek™'s response to something it can't parse or execute. Not having the 5 msec line delay is a common cause. The "Huh?" response is generally on the offending line and often prints out the offending command.
7) Run the program according to the instructions for each program.
8) When done, unload the program according to the instructions for each program.

5.1.2.7.1 Game

This application plays a simple game. Make sure that the terminal program is set to VT100 mode. After downloading, to run this program enter "***go<CR>***", and follow the commands. To unload this program enter "***forget signon<CR>***".

```
// War games
forget signon

: signon
cr
."  _          _ _        "  cr
." | |__   ___| | | ___   "  cr
." | '_ \ / _ \ | |/ _ \ "  cr
." | | | |  __/ | | (_) |"  cr
." |_| |_|\___|_|_|\___/ "  cr
;

: game?
." Would you like to play a game?" cr
." 1) Marketing Bingo " cr
." 2) Development Sorry " cr
." 3) Engineering Pacheesi " cr
." 4) Sales Poker " cr
." 5) Investment Roulette" cr
." 6) Global Thermonuclear war" cr
;
```

```
// these words emit the VT100 terminal escape sequences
// Consult a VT100 terminal document for details.
: home
// outputs <ESC>[2J<ESC>[H, i.e. clear screen and home
27 emit ." [2J" 27 emit ." [H"
;
: red
// outputs VT100 sequence for red
// <ESC>[5;30;41m
27 emit ." [5;30;41m"
;
: black
// VT100 for black
// <ESC>[0m
27 emit ." [0m"
;


: boom!
home red
."  ____   ___   ___  __  __ _  " cr
." | __ ) / _ \ / _ \|  \/  | | " cr
." |  _ \| | | | | | | |\/| | | " cr
." | |_) | |_| | |_| | |  | |_| " cr
." |____/ \___/ \___/|_|  |_(_) " cr
black
;



: bomb
25 0 do
." ." cr 100 delay
loop
boom!
;

: game
dup [ char 1 ] literal = if  // note [ ] allows interpret mode
." Too much drinking!" cr
else
dup [ char 2 ] literal = if
." Too depressing!" cr
else
dup [ char 3 ] literal = if
." Too boring!" cr
else
dup [ char 4 ] literal = if
." Too Risky!" cr
else
dup [ char 5 ] literal = if
```

```
." No upside!" cr
else
dup [ char 6 ] literal = if
bomb
else
drop ." bad choice " cr
then
then
then
then
then
then
;

: go
home red signon black game?
begin key dup 27 = 0= while
game
repeat
;
home
```

5.1.2.7.2 Setup Magnetic Variation with Position

This application sets up the latitude, longitude, altitude and day and executes the world magnetic model. Finally it prints the computed magnetic variation using the internal world magnetic model.

```
lat  f29.13  set drop
lonG f-81.33 set drop
alt  f20.0   set drop
day  f2011.4 set drop
set_wmm 1 set drop
magvar di.
wmmGD di.
```

5.1.2.7.3 Turtle

This application illustrates evaluating the sensor data with NPL. In this case the device will print a warning if the inertial system is upside down for more than 5 seconds. To run this program after download, type “***turtle<CR>***”. To unload this program type “***forget read_z<CR>***”.

```
// Turn turtle script
// sends an alarm is the inertial system is upside down for more than
5
// seconds.
forget read_z
: read_z
accelr &di 8 + @ // get the raw accel array, read the z channel
;
variable turtleCount
: bump
f0.0 < if
      turtleCount @ 1 + turtleCount !
      else
      0 turtleCount !
      then
;

: turtle 0 turtleCount !
begin
  ?key 0=
  while read_z bump
  turtleCount @ 5 > if ." Alert--Upside down" cr
  7 7 7 emit emit emit // send three alert characters (bells)
  then
  1000 delay
repeat
;
```

5.1.2.7.4 Criteria Limited Output

This script illustrates an application where the inertial system value is only output if the heading varies by more than 10 degrees. This would be useful in an application where the inertial system is remotely mounted and has a limited bandwidth channel in which to communicate, or where multiple sensors must share a busy channel. After downloading, to run this program, type "***filter<CR>***". To stop the execution, press any key. To unload this program type "***forget lastYawT<CR>***".

```
// Read the inertial system, only output a value if the heading
changes
// more than 10 degrees
forget lastYawT

variable lastYawT

: deviation
// make a copy, compare to last value
dup lastYawT @ f- fabs
f10.0 f> // compare absolute value of difference in last two readings
         // to see if it is greater than 10 degrees.
// if true store and output the value
if
dup lastYawT ! f. cr // make a copy, store in lastYawt, print value
7 7 7 emit emit emit // send 3 alerts
else
drop  // drop the new sample, change not big enough
then
;


: filter
// first output the current heading
yawt di@ dup f. lastYawT !
begin ?key 0=
while
   yawt di@ deviation // read the current yaw, send to function
   500 delay
```

```
repeat
;
```

5.1.2.7.5 Real Time Field Calibration Swap

This application example shows how to modify the inertial system operation in real time. In this case the magFieldCalX value is being changed according to an operational mode in the host device. The "hard-coded" numbers represent an alternate calibration value obtained during system setup. If the end user has a need for this type of operation, please contact Sparton Technical Support for programming application assistance for this type of problem. The inline comments can be examined for how this program operates.

```
// change field calibration on the fly
forget storeCal // this unloads a previous version of the program
// if it is being re-downloaded.

// create a variable to store the current calibration
// 4 elements, so it is an array.
variable storeCal 4 allot

// This stack diagram indicates that the divisor and offset are on
// the stack before calling this word.
// ( divisor offset -- )
// This word formats the storeCal array to be compatable
// with the "set" database word
//
: store
0 storeCal !          // store 0 in first position
1 storeCal 4 + !      // 1 in the second postion, therefore we are
                      // going to set array positions 0..1
storeCal 8 + !        // store the offset at position 2 in storeCal
                      // this is position 0 in the database array
storeCal 12 + !       // store the divisor at position 3 in storeCal
                      // this is position 1 in the database array
;

// start expects no argument
// It reads the current magFieldCalX value
// and stores it in the storeCal array for
// later restoration
: start
// stores the current cal
 magFieldCalX &di // get the pointer to magFieldCalX array
dup @              // duplicate the pointer, read the first value
swap 4 + @         // swap pointer for value, advance and read
                   // second element in the array
swap               // reverse the order so they are in correct order
                   // for the store word
store              // store in the storeCal save array
;
```

```
// create an alternate magFieldCalX array
variable alternate 4 allot
// Initialize the array with desired values.
0 alternate !
1 alternate 4 + !
f-10.0 alternate 8 + !
f0.5 alternate 12 + !

// This word is used to store the current field cal X
// Then replace it with the alternate.
: on
  start
  magFieldCalX alternate set drop  // note drop is required because
                                   // set returns a T|F success value
;

// This word restores the magfield cal to the original value.
: off
  magFieldCalX storeCal set drop
;

// This word is used to restore the original field to
// some know values, in case of an operator mistake.
: restore
F24.1458 f1.079998 swap store
magFieldCalX storeCal set
;

// Program that prints out the heading until a key is pressed.
// The user should type "on" or "off" then heading
// to see the effect of the two different field calibrations.
: hdg
begin ?key 0= while
yawt di. 500 delay
repeat
;
```

5.1.2.7.6 TARE Command

This program allows the inertial system to be mounted in any configuration within the host device and then adjusted so that the roll, pitch and yaw outputs are aligned with the device, not the inertial system. This allows totally arbitrary placement of the inertial system in the host device. All that is required is to position the inertial system in the desired position, then send this script, wait until the OK prompt is returned as the script takes 5 seconds to run, then wait 5 seconds for the database to be stored, then reset the device.

```
// TARE function
forget matrix
// Holding matrices for intermediate computations.
variable matrix 9 allot
variable matrix2 9 allot
matrix clear(m)
// This performs the actual computation
: compute
// form a matrix of the cp2, cp1 and acceleration estimate
cp2 di@ cp1 di@ accelEst di@ matrix buildMatrix
// Transpose the matrix so the coefficients are in the columns
matrix matrix2 T(m)
// Now take the inverse, resultant matrix is the boresight matrix
matrix2 matrix inv(m)
;
// matrix cr m.
// This logic is a temporary array to copy rows from the
// result matrix into the 3 vectors that make up the boresight
// matrix
// the copy array
variable copyarray 5 allot
// a word to compute array locations.
: index 4 * copyarray + ;
// Copies a an array to the copy array
// after putting the 0 and 2 index bounds into the copy array.
: cp 0 0 index ! 2 1 index !
dup @ 2 index ! 4 + dup @ 3 index ! 4 +  @ 4 index !
;
// Use the copyarray and the cp function to put in each
// row of the boresight matrix from the computed matrix.
: copyit
matrix cp boresightMatrixX copyarray set
// note 3 4 * + moves up to the second row of a 3x3 matrix
matrix 3 4 * + cp boresightMatrixY copyarray set
// note 6 4 * + moves up to the third row of a 3x3 matrix
matrix 6 4 * + cp boresightMatrixZ copyarray set
;
// Print the resultant matrix and the new boresight matrix
// for error check.
: printit
matrix cr m.
boresightMatrixX cr di.
boresightMatrixY cr di.
boresightMatrixZ cr di.
;

// This word computes the boresight matrix in place.
: doit
orientation 0 set   // set the orientation back to default
5000 delay          // wait 5 seconds to allow the estimates to catch
                    // up
compute             // compute new boresight matrix
copyit              // put back in the database
printit ;           // sanity check printout

doit                // run this macro automatically at download
```

```
forget matrix        // automatically unload this program
```

5.1.2.8 Programmer to Programmer

This section is addressed to the programmer, software engineer or other software professional that wants to use NPL or Remote Function Select (RFS) in the inertial system.

- NPL is a good method for experimenting with the inertial system manually. The ability to make words quickly allows rapid experimentation.
- RFS is the best method going forward for an automatic interface. There is a little more work to create the packet structure but once it is done you will be able to adapt to new variables almost instantly and also interoperate with other products in the product line. Also you can design your own message content so everything comes in one message.
- With NPL, you can shorthand anything quickly. If a variable is someVeryLongNameThatsAPainToType and you plan on working with the variable for a while, you can quickly type "***: it someVeryLongNameThatsAPainToType ;***" and then use ***it*** whenever you want the long thing.
- Take note of the first few lines of each sample application. Note that each contains a line that forgets the first definition in the application. This is present so that when the application is reloaded over and over during test, it automatically cleans out the previous instance of the application. On the first load the first definition is not found, so NPL prints "can't find it", but this is harmless. This is good practice and keeps from overflowing the dictionary space.
- The NPL operand stack is of finite depth. It is sufficiently deep for virtually all applications however it can be overflowed quickly in a loop, for example. Take care to keep the stack size small (< 32 items). The ".s" operator prints the stack contents. Note that the "set" and "rec" words return a flag indicating success or failure. A common mistake is to forget to "drop" the result if there is no plan to test it. This often leads to a stack overflow and a lockup of the interpreter and/or inertial system.
- The NPL interpreter operates in a virtual memory space, access to flash memory or hardware peripherals is not possible.
- You can work in any number base up to 37. The base variable contains the current number base for i/o. ***Decimal*** and ***hex*** put in the two standards. Decimal is defined as "***: decimal 10 base ! ;***", hex is defined as "***: hex 16 base ! ;***". So you can use whatever base you want. Binary is helpful. You can get the current number base with ***base @.*** (Remember that ***.s*** prints the stack but always in decimal and hex so this is a good sanity check when playing around in the number base). So to work in binary do ***2 base !***. Note that the word ***decimal*** is necessary because once you are in base 2, ***10 base !*** does not return you to decimal it puts you in base 10_2 (which is 2!, to accomplish this you would have to do 1***010 base !***). This is why "***decimal***" exists. By example ***hex aa 2 base ! . // prints 10101010***.
- The "***status***" word tells you how much NPL memory you are using. If you are doing something exotic, run that once in a while and if you are overflowing the memory funny things might be happening.

- The "***begin ?key 0= while {stuff } repeat***" construct is your friend. It makes a loop till you hit any key.
- There is a ***delay*** function that takes msecs as an argument. It comes in handy.
- Prototype a bunch of words that do the functions you want, then store them as text files and send them with the terminal emulator (be careful at 115200 as you can overflow the input stream, it is helpful to set a few (5) msec line delay in the terminal emulator when sending files of more than a few lines.) That way you won't have to type the same things over and over again to do the same things. This is especially time-saving if you are doing the same thing to a bunch of different devices (like testing or calibrating).
- "***words***" shows all the words in the wordlist minus the core words. Type ***loud*** to allow words to show all the words in the base wordlists. Type ***quiet*** to go back to the minimum set.
- "***grep***" will search even the hidden wordlists regardless of the state of ***loud*** or ***quiet***, remember that everything is case sensitive so if you "***grep points***" but the variable is ***myPoints*** it won't find it. Try "***grep oints***" or some other part of the string. There is no "-i" option on NPL grep.
- There are some extensions to Forth in NPL. You can enter ***" string"*** interactively (but not compiled, there are other words for compiled strings). What gets left on the stack is a pointer to a counted string. This is occasionally useful. Similarly the word ***array[v1 v2 v3]array*** leaves a pointer on the stack to the array v1,v2,v3. It's transient so the next array or string declared this way overwrites the previous one (same way with ***" string"***). The arrays need to be small (< 16 elements, strings < 80 characters.) They use the same temporary space so you can only have one at a time lying around. The words ***array[*** and ***]array*** can be used in a compiled word. For compiled words ***$"*** closed with " creates a compile time string.
- The "***set***" word automatically determines the variable type from the variable identifier. It is up to you to make sure to provide the correct data. The syntax is ***variableName value set***. The value is an integer for integers and ordinals, a float (contains F and .) for floats. A string (***" string")*** for strings, and the array syntax for arrays. So for example to set a 3 element array call bob: ***bob array[0 2 v1 v2 v3] array set***, pushes the variable info, uses the interactive array function to create an array with the values 0 2 v1 v2 and v3 and passes this array to set. ***Set*** expects the first two array elements to be the starting and ending index of the values in the array you want to set. The remaining part of the array is the values to ***set*** with.
- You ***can*** crash the software with NPL (it is not Java, after all), so if you are playing around you might need to have access to the power switch on your device. If you stick to the mainline commands that typical users are using everything will be ok, but if you attempt to divide by 0, index way beyond the end of an array, access memory that's not in the NPL variable space, etc. etc. it is designed to, and will crash hard enough that you have to reset. You won't lose any permanent calibration data, but you can make it stop operating properly. It is a useful tool, but if NPL was a car, it would be a Shelby Cobra not a 12 airbag, 97 horsepower hybrid.
- Sparton may be able to help with complex applications where NPL may be used in the solution. Contact someone at Sparton Tech Support if you have such a need.

Appendices

A NPL Command Categories

NPL is a full programming language in addition to being a command line interpreter. Forth commands exist in multiple categories as described in this section. The user is directed at the various Forth tutorials on the Internet for a primer on basic Forth concepts, for example this is a good reference http://www.forth.com/starting-forth/ .

Each command is shown with a stack diagram that shows the state of the parameter stack before and after the command. The syntax of the diagram is (before – after) with the top of the stack shown on the right hand side of each before or after set. It is usual practice to have white space around the parenthesis so that the stack diagram forms a comment.

A.1NPL Arithmetic and Logical

NPL contains a full set of arithmetic and logical operators for both fixed and floating point (32 bit) values. Each command is described briefly below:

A.1.1 Integer Commands

Command	Stack Diagram	Notes	ANSI Equivalent
/	(a b – a/b)	Divide the second stack item by the item on the top of the stack.	
s/	(a b – a/b)	Signed divide of the top to items on the stack.	/
*	(a b – a*b)	Multiply the top two stack items.	
s*	(a b – a*b)	Signed multiply of the top two items.	*
–	(a b – a-b)	Subtract the top stack item from the next item up on the stack.	-
+	(a b – a+b)	Add the top two stack items.	+
>>	(a b – a>>b)	Shift the 1 deep stack item to the right by the number of bits specified by the value on the TOS.	RSHIFT
<<	(a b – a<<b)	Shift the 1 deep stack item to the left by the number of bits specified by the value on the TOS.	LSHIFT
mod	(a b – a%b)	Leave the remainder after division (modulus).	MOD
/mod	(a b – a/b a%b)	Leave the quotient and remainder on the stack as the result of an integer division.	/MOD
xor	(a b – a^b)	Exclusive or the top two stack items.	XOR
not	(a – ~a)	Take the ones complement of the item	NOT

		on the top of the stack.	
`or`	`( a b – a|b)`	Bitwise “**`or`**” the top two stack items.	OR
`and`	`(a b – a&b)`	Bitwise “**`and`**” the top two stack items.	AND
`+!`	`( v addr -- )`	Add the value 1 deep in the stack to the value stored at the address on top of the stack. Can be used to increment/decrement variables easily.	+!
`abs`	`(a – abs(a))`	Take the absolute value of the top of the stack.	ABS
`>`	`( a b – T|F)`	Compare the top two items, return true if the 2nd item is greater than the 1st item, else return false.	>
`<`	`( a b – T|F)`	Compare the top two items, return true if the 2nd item is less than the 1st item, else false.	<
`negate`	`( a -- -a)`	Take the two’s complement of the top of the stack.	NEGATE
`0=`	`( v – T|F)`	Compare the top of the stack to 0 and if so return true, else return false.	0=
`0>`	`( v – T|F)`	Return true if the item on the top of the stack is greater than 0.	0>
`0<`	`( v – T|F)`	Return true if the item on the top of the stack is less than 0.	0<
`=`	`( a b – T|F)`	Compare the top two stack items and return true if they are equal, else return false.	=

A.1.2 Floating Point Commands

Command	Stack Diagram	Notes	ANSI Equivalent
`pi`	`( -- 3.14159 )`	pi	
`d>r`	`( fd – fr )`	degrees to radians	
`r>d`	`( fr – fd )`	radians to degrees	
`exp`	`( f – e`^{x}`)`	computes e^x	FEXP
`log`	`( f – log(x) )`	computes log(x)	
`floor`	`( f – floor(f) )`	floating point floor function	FLOOR
`ceil`	`( f – ceil(f) )`	floating point ceiling function	
`pow`	`( fa fb – fa`^{fb}` )`	computes x^y	
`atan`	`( f – atan(f) )`	arc tangent (radians)	FATAN
`acos`	`( f – acos(f) )`	arc cosine (radians)	FACOS
`asin`	`( f – asin(f) )`	arc sine(radians)	FASIN
`tan`	`( f – tan(f) )`	tangent (radians)	FTAN
`cos`	`( f – cos(f) )`	cosine (radians)	FCOS
`sin`	`( f – sin(f) )`	sine (radians)	FSIN
`fround`	`( f – round(f) )`	round floating point number	FROUND
`ftrunc`	`( f – trunc(f) )`	truncate a floating point number	
`f>i`	`( f – i)`	convert floating point to signed	

		integer	
`i>f`	`( i – f)`	convert signed integer to floating point	
`-f`	`( f -- -f)`	negate a floating point number	FNEGATE
`fabs`	`( f – fabs(f) )`	take the absolute value of a floating point number	FABS
`f/`	`( f1 f2 – f1/f2)`	floating point divide	F/
`f*`	`( f1 f2 – f1*f2)`	floating point multiply	F*
`f-`	`( f1 f2 – f1-f2)`	floating point subtract	F-
`f+`	`( f1 f2 – f1+f2)`	floating point add	F+
`f>`	`( f1 f2 – t|f )`	Returns true if f1 > f2	F>
`f<`	`( f1 f2 – t|f )`	Returns true if f1 < f2	F<
`f=`	`( f1 f2 – t|f )`	Returns true if f1 = f2	F=
`f<=`	`( f1 f2 – t|f )`	Returns true if f1 <= f2	
`f>=`	`( f1 f2 – t|f )`	Returns true if f1 >= f2	
`dot`	`( v1 v2 vr – )`	Takes the dot product of v1 and v2 and places the result in vr.	
`dot3d`	`(m v vr – )`	Takes the three dimensional dot product of matrix m and vector v, placing the result in vector vr.	
`cross`	`( v1 v2 vr – )`	Takes the cross product of v1 and v2, placing the result in vr.	
`v*v>v`	`( v1 v2 vr – )`	Multiplies the elements of v1 by the elements of v2, placing the result in vr.	
`neq`	`( float – T|F)`	Returns true if the value is not equal to zero (-1e-20 < value < 1e-20).	
`norm`	`( v1 – )`	Takes a 4 element vector (x,y,z,magnitude) and divides x, y and z with the magnitude, and sets the magnitude to 1.0 thus turning it into a unit vector. This function assumes that the magnitude has already been computed and assigned which can be done with “v[3]=|v|”.	
`norm4d`	`( v1 vr – )`	Performs a 4D normalization on v1, placing the result in vr.	
`buildMatrix`	`( v1 v2 v3 m –)`	Builds a matrix from v1, v2 and v3. Each vector forms the rows of matrix m.	
`genRotMatrix`	`( mr v1 theta – )`	Generate rotation matrix mr, from vector v1 and angle theta.	
`m*v>v`	`( m v – )`	Multiply matrix by vector, place result in vector.	
`m*v>r`	`( m v1 vr – )`	Multiply matrix m by vector v1, place result in vector vr.	

`s*v>r`	`( float v1 vr - )`	Multiply float value by each element of v1, place result in vector vr (4 element vectors).
`s*m>r`	`( float m mr - )`	Multiply float value by each element of matrix m, putting result in matrix mr.
`v>v`	`( v1 v2 - )`	Copy vector v1 to vector v2 (4 elements).
`v=v`	`( v1 v2 - T|F)`	Returns true if the two vectors are equal (4 elements).
`m>m`	`( m1 m2 - )`	Copies matrix m1 to matrix m2.
`T(m)`	`( m mr - )`	Transpose matrix m, place result in matrix mr.
`clear(v)`	`( v - )`	Makes all 4 elements in vector v = 0.0.
`v+v>r`	`( v1 v2 vr - )`	Add vector v1 to vector v2, place result in vector vr.
`v-v>r`	`( v1 v2 vr - )`	Subtract vector v2 from vector v1, place result in vector vr.
`|v|`	`( v - m )`	Compute the magnitude of vector v, return the value on the top of the stack.
`v[3]=|v|`	`( v - )`	Compute the magnitude of vector v, placing the result in the 3rd index (4th position).
`|x,y|`	`( x y - v )`	Compute the 2d magnitude of the two element vector created by x and y, return the resut on the top of the stack.
`clear(m)`	`( m - )`	Set all matrix element equal to 0.0
`m=I`	`( m - )`	Set matrix m equal to the identity matrix.
`m*m>r`	`( m1 m2 mr - )`	Multiply matrix m1 by matrix m2, place the result in matrix mr.
`inv(m)`	`( m mr - )`	Compute the inverse of matrix m, place the results in matric mr.
`m.`	`( m - )`	Print matrix m in free format .
`m.f`	`( w f m - )`	Print matrix m in %w.f format.
`v.`	`( v - )`	Print vector v in free format.
`v.f`	`( w f v - )`	Print vector v in %w.f format.

A.2NPL Parameter Stack Manipulation

These commands allow manipulation of the parameter stack.

Command	Stack Diagram	Notes	ANSI Equivalent
`drop`	`( v - )`	Drops the top item from the stack.	DROP
`.s`	`( - )`	Prints the stack contents, non-	.S

		destructively	
pick	(..v-..vth-item)	Copy the vth item in the stack to the top of the stack.	PICK
2drop	(v1 v2 –)	Drop the top two items from the stack.	2DROP
nip	(v1 v2 v3 – v1 v3)	Removes the 2nd item down in the stack.	NIP
2dup	(v1 v2 – v1 v2 v1 v2)	Duplicates the top two items on the stack.	2DUP
over	(v1 v2 – v1 v2 v1)	Makes a copy of the 2nd item in the stack to the top of the stack.	OVER
dup	(v1 – v1 v1)	Makes a copy of the top item on the stack.	DUP
swap	(v1 v2 – v2 v1)	Swaps the top two stack items.	SWAP
rot	(v1 v2 v3 – v2 v3 v1)	Rotates the top three items so that the 3rd deep item is now on the top of the stack.	ROT

A.3NPL Input/Output

These are commands to do input and output to/from the user.

Command	Stack Diagram	Notes
accelp.p	(T|F –)	Turns on/off the processed accelerometer data.
magp.p	(T|F –)	Turns on/off the processed magnetometer data.
gyrop.p	(T|F –)	Turns on/off the processed gyro data.
accel.p	(T|F –)	Turns on/off the accelerometer data.
mag.p	(T|F –)	Turns on/off the magnetometer data.
compass.p	(T|F –)	Turns on/off the inertial system data.
s.p	(T|F –)	Turns on/off the raw sample data. Outputs all raw samples.
gyro.p	(T|F –)	Turns on/off the raw gyro data.
db.	(–)	Print the entire database contents.
d.p	(T|F –)	Enable or disable database internal debug print statements. (Debug port output only.)
loud	(–)	Enables display of the basic forth words when “words” is executed.
quiet	(–)	Disables display of the basic forth words when “words” is executed.
echo!	(T|F –)	Turns on/off NorthTek™ responses such as echoed characters, “OK” and “Huh”. On by default.
(	(–)	Begin a comment
."	(–)	Compile Only – compile a string that will be output when this word is executed.
?key	(– T|F)	Returns true if there is a keystroke pending in the input buffer. Returns immediately.

`bl`	`( – 0x20 )`	Pushes a blank (0x20) on the stack.
`key`	`( – keyvalue)`	Waits until a character has been received, then returns the character.
`emit`	`( v – )`	Ouput the number on the top of the stack as a character.
`//`	`( – )`	Ignore the remainder of this line.
`grep`	`( – )`	Reads the token following grep, attempts to locate commands in the wordlists that contain this substring. Case sensitive.
`words`	`( – )`	Display the list of words in the set of wordlists.
`.s`	`( – )`	Non-destructively print the stack contents. Note: Always prints in decimal and hex regardless of the current i/o number base.
`ff.`	`(w d f – )`	formatted floating point print %w.df format.
`e.`	`(f – )`	floating point print in %e format
`f.`	`( f – )`	floating point %f format
`.`	`( v – )`	Remove and print the top item from the stack using the current number base.
`decimal`	`( – )`	Change the current number base to decimal.
`hex`	`( – )`	Change the current number base to hex.
`base`	`( – addr)`	Returns the address of the variable that holds the current number base.

A.4NPL Variables, Constants

These are commands to declare and manipulate constants and variables.

Command	Stack Diagram	Notes	ANSI Equivalent
`constant`	`( value – )`	Reads the token just past constant, and makes a constant with that name and the value found on the stack. For example ***`1234 constant myValue`*** Creates a constant whose value is 1234.	CONSTANT
`variable`	`( – )`	Creates a 32 bit variable, initialized to 0 using the name that follows the word variable. When the variable name is executed it returns the address of the variable.	VARIABLE
`allot`	`( v – )`	Allocate the number of 32 bit words to the variable just allocated. For example ***`variable buffer 3 allot`*** creates a 4 element array called buffer. When buffer is executed it returns the address of the first element. The values are NOT initialized to 0.	ALLOT[7]

[7] ANSI ALLOT allocates bytes, but the function is the same. NPL allocates in 32 bit chunks.

`w!`	`( value addr - )`	Stores the 16 bit value that is 1 deep in the stack at the location that is on the top of the stack.	
`w@`	`( addr - value)`	Reads the 16 bit value at the address on the top of the stack and puts in on the stack.	
`c@`	`( addr - value)`	Reads the 8 bit value at the address on the top of the stack and puts in on the stack.	C@
`c!`	`( value addr - )`	Stores the 8 bit value that is 1 deep in the stack at the location that is on the top of the stack.	C!
`!`	`( value addr - )`	Stores the 32 bit value that is 1 deep in the stack at the location that is on the top of the stack.	!
`@`	`( addr - value)`	Reads the 32 bit value at the address on the top of the stack and puts in on the stack.	@

A.5NPL Database

These are database commands.

Command	Stack Diagram	Notes
`set`	`(IDI value - result)`	Expects an internal database identifier on the stack, followed by a value. Sets the item to the value given. See advanced programming section on how to handle arrays. See the Software Interface Users Manual applicable to the product being used, for the list and details of variables that may be set. The result is 0 if the set did not complete properly, for example if the value was out of range. If the set operation completed properly the return value will be not equal to 0. If the value set is non-volatile causes the non-volatile variables to be written back to non-volatile memory. Note that there is a 5 second delay before variables are written to allow multiple commands to be collated into one single non-volatile write operation.
`rec`	`(IDI value - result)`	Expects an internal database identifier on the stack, followed by a value. Sets the item to the value given. See advanced programming section on how to handle arrays. See the Software Interface Users Manual applicable to the product being used, for the list and details of variables that may be set. The result is 0 if the set did not complete properly, for example if the value was out of range. If the set operation completed properly the return value will be not equal

		to 0. Does not cause the non-volatile items to be written. However the inertial system periodically writes updated calibration to non-volatile memory and would include values set by "rec" in that process.
`di.`	`( IDI -)`	Expects an internal database identifier on the stack, prints the current value.
`di@`	`( IDI – v )`	Returns the value of an IDI scalar, or a pointer to an IDI string or array.
`&di`	`( IDI – address )`	Returns the address of the data for the given variable.
`di>`	`( IDI – )`	Causes the inertial system to output the RFS "Get_Value" message for the given variable.
`save`	`( – )`	Save any changed variables to non-volatile memory.
`>default`	`( IDI- )`	Set the database item to its default value.
`defaults`	`( – )`	Set all values to default value. Warning: Erases all calibration and configuration data. Factory use only.
`++`	`( IDI -- )`	Increments the value of a scalar IDI that is on the stack.
`show`	`( IDI – )`	Causes the inertial system to output the RFS "Format" message for the given variable.
`dvid@`	`( IDI – v )`	Return the internal RFS vid number for the given variable. This will not match the external user VID number.
`wmm.`	`( – )`	Print the current world magnetic model.
`wmmeeload`	`( – )`	Force a World Magnetic Model to be loaded from EEPROM.

A.6NPL Logic

These are logic commands.

Command	Stack Diagram	Notes	ANSI Equivalent
`j`	`( – v)`	Pushes the outer loop index from a nested do loop onto the stack. Compile Only.	J
`i`	`( – v)`	Pushes the inner loop index onto the stack. Compile Only.	I
`repeat`	`( – )`	Closes a begin-while-repeat loop. Compile Only.	REPEAT
`while`	`( – )`	Forms the middle test of a begin-while-repeat loop. Compile Only.	WHILE
`begin`	`( – )`	Forms the top of a begin-while-repeat loop. Compile Only.	BEGIN
`loop`	`( – )`	Closes a do-loop structure. Compile Only.	LOOP
`do`	`( – )`	Opens a do-loop structure. Compile Only.	DO
`else`	`( – )`	Begins the else clause of an if-then-	ELSE

		else. Compile Only.	
`then`	`( – )`	Closes an if-then-else structure. Compile Only.	THEN
`if`	`( – )`	Opens an if-then-else structure. Compile Only.	IF
`:`	`( – )`	Take the token immediately (but with whitespace, remember!) and make a new command in the wordlists with this name, begin compiling.	:
`;`	`( – )`	Finishes compiling, finalize the word that was started with : in the current wordlist.	;
`forget`	`( – )`	Truncate the wordlist back to the token immediately following the forget word.	FORGET
`delay`	`( v – )`	Pauses a program for the given number of milliseconds.	