# Basic for the Newton

# NS BASIC HANDBOOK

REM
GOSUB
LET
BASIC

# NS BASIC Handbook

December 15, 1995

© NS BASIC Corporation, 1995.
77 Hill Crescent
Toronto, Canada M1M 1J3
(416) 264-5999

Every effort has been made to ensure that the information in this manual is accurate. NS BASIC Corporation is not responsible for printing or clerical errors. Specifications are subject to change without notice.

Mention of third party products and their trademarks is for informational purposes only and constitutes neither an endorsement or recommendation. NS BASIC Corporation assumes no responsibility with regard to the performance or use of NS BASIC or these products.

Special thanks to the following people for help in the testing and development of NS BASIC and the preparation of this manual: Peter Jensen, John Marman and Mark Holmes.

**Canadian Cataloguing In Publication Data**

Henne, George W.P., 1954-
Schettino, John C. Jr., 1961-
Schettino, Elizabeth O., Ph.D., 1961-

NS BASIC Handbook

Includes index.

ISBN 0-9695844-1-5

1. BASIC (Computer program Language). 2. Newton (Computer) - Programming. I. Title.

QA76.8.N48H4 1994005.265C94-931542-7

# LICENSE AGREEMENT

PLEASE READ THIS LICENSE CAREFULLY BEFORE USING THE SOFTWARE. BY USING THE SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS LICENSE. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENSE, PROMPTLY RETURN THE PRODUCT TO THE PLACE WHERE YOU OBTAINED IT AND YOUR MONEY WILL BE REFUNDED.

1. License. The application, demonstration, system, and other software accompanying this License, whether on disk, in read only memory, or on any other media (the "Software"), the related documentation and fonts are licensed to you by NS BASIC Corporation ("NSBC"). You own the media on which the Software and fonts are recorded but NSBC and or NSBC's Licenser(s) retain title to the Software, related documentation and fonts. This License allows you to use the Software and fonts on a single Newton Product (which, for purposes of this License, shall mean a product bearing Apple's Newton logo), and make one copy of the Software and fonts in machine-readable form for backup purposes only. You must reproduce on such copy the NSBC copyright notice and any other proprietary legends that were on the original copy of the Software and fonts. You may also transfer all your license rights in the Software and fonts, the backup copy of the Software and fonts, the related documentation and a copy of this License to another party, provided the other party reads and agrees to accept the terms and conditions of this License.

2. Restrictions. The Software contains copyrighted material, trade secrets and other proprietary material and in order to protect them you may not decompile, reverse engineer, disassemble or otherwise reduce the Software to a human-perceivable form. You may not modify, network, rent, lease, load, distribute or create derivative works based upon the Software in whole or in part. You may not electronically transmit the Software from one device to another or over a network.

3. Termination. This License is effective until terminated. You may terminate this License at any time by destroying the Software and related documentation and fonts. This License will terminate immediately without notice from NSBC if you fail to comply with any provision of this License. Upon termination you must destroy the Software, related documentation and fonts.

4. Export Law Assurances. You agree and certify that neither the Software nor any other technical data received from NSBC, nor the direct product thereof, will be exported outside the United States except as authorized and as permitted by the laws and regulations of the United States. If the Software has been rightfully obtained by you outside of the United States, you agree that you will not reexport the Software nor any other technical data received from NSBC, nor the direct product thereof, except as permitted by the laws and regulations of the United States and the laws and regulations of the jurisdiction in which you obtained the Software.

5. Government End Users. If you are acquiring the Software and fonts on behalf of any unit or agency of the United States Government, the following provisions apply. The Government agrees: (i) if the Software and fonts are supplied to the Department of Defense (DoD), the Software and fonts are classified as "Commercial Computer Software" and the Government is acquiring only "restricted rights" in the Software, its documentation and fonts as that term is defined in Clause 252.227-7013(c)(1) of the DFARS; and (ii) if the Software and fonts are supplied to any unit or agency of the United States Government other than DoD, the Governments' rights in the Software, its documentation and fonts will be as defined in Clause 52.227-

19(c)(2) of the FAR or, in the case of NASA, in Clause 18-52.227-86(d) of the NASA supplement to the FAR.

6. NS BASIC will replace at no charge defective disks or manuals within 90 days of the date of purchase. NS BASIC warranties that the programs will perform generally in compliance with the included documentation. NS BASIC does not warrant that the programs and manuals are free from all bugs, errors or omissions.

7. Disclaimer of Warranty on Software. You expressly acknowledge and agree that use of the Software and fonts is at your sole risk. The Software, related documentation and fonts are provided "AS IS" and without warranty of any kind and NSBC and NSBC's Licenser(s) (for the purposes of provisions 7 and 8, NSBC and NSBC's Licenser(s) shall be collectively referred to as "NSBC") EXPRESSLY DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. NSBC DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN THE SOFTWARE AND THE FONTS WILL BE CORRECTED. FURTHERMORE, NSBC DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE AND FONTS OR RELATED DOCUMENTATION IN TERMS OF THEIR CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY NSBC OR A NSBC AUTHORIZED REPRESENTATIVE SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY. SHOULD THE SOFTWARE PROVE DEFECTIVE, YOU (AND NOT NSBC OR AN NSBC AUTHORIZED REPRESENTATIVE) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

8. Limitation of Liability. Because software is inherently complex and may not be free from errors, you are advised to verify the work produced by the Program. UNDER NO CIRCUMSTANCES INCLUDING NEGLIGENCE, SHALL NSBC BE LIABLE FOR ANY INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES THAT RESULT FROM THE USE OR INABILITY TO USE THE SOFTWARE OR RELATED DOCUMENTATION, EVEN IF NSBC OR A NSBC AUTHORIZED REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU. In no event shall NSBC's total liability to you for all damages, losses, and causes of action (whether in contract, tort (including negligence) or otherwise) exceed the amount paid by you for the Software and fonts.

9. Allocation of Risk: You acknowledge and agree that this Agreement allocates risk between you and NSBC as authorized by the Uniform Commercial Code and other applicable law and that the pricing of NSBC's products reflects this allocation of risk and the limitations of liability contained in this Agreement. If any remedy hereunder is determined to have failed of its essential purpose, all limitations of liability and exclusions of damages as set forth in this Agreement will remain in effect.

10. Support. NSBC may, at its option, provide support services at its standard fees for such services. Such support services will be governed by the limita-

tions of liability under this Agreement.

11. Additional Restrictions: Any upgrade or enhancement of the program subsequently supplied by NSBC may only be used upon the destruction of the prior version, and shall be governed by the terms of this Agreement.

12. Controlling Law and Severability. This License shall be governed by and construed in accordance with the laws of the United States and the State of Delaware, as applied to agreements entered into and to be performed entirely within Delaware between Delaware residents. If for any reason a court of competent jurisdiction finds any provision of this License, or portion thereof, to be unenforceable, that provision of the License shall be enforced to the maximum extent permissible so as to effect the intent of the parties, and the remainder of this License shall continue in full force and effect.

13. Complete Agreement. This License constitutes the entire agreement between the parties with respect to the use of the Software, related documentation and fonts, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter. No amendment to or modification of this License will be binding unless in writing and signed by a duly authorized representative of NSBC.

# CONTENTS

CHAPTER

1

# 1. Introduction

Welcome to NS BASIC for the Newton. NS BASIC is designed to meet the needs of Newton users. It is a simple yet powerful language that can be used to write programs for almost any application.

There is a text file named README.TXT on the supplied disk that contains any late-breaking information about NS BASIC, including updates to the Handbook. Please read it before installing NS BASIC.

If you'd like to get started using NS BASIC right away, then read the Installation section, and then turn to the Getting Started With NS BASIC chapter.

Sample programs are provided with NS BASIC for you to study and use. You can tailor these sample programs to your particular needs. There is a text file named EXAMPLES.TXT on the supplied disk that contains the programs used in this Handbook.

You should be somewhat familiar with the basics of operating a Newton before you start using this Handbook. That is, you should know about opening applications in the Extras Drawer, using the stylus and other Newton features. If you are not comfortable with these terms, review the Newton Handbook.

A basic understanding of operating a desktop computer (Macintosh or IBM Compatible) is needed to install the NS BASIC software.

## 1.1 All About BASIC

BASIC has been around for over 30 years. Over that period, hundreds of interpreters and compilers for BASIC have been developed, and a mountain of application code has been written. Many books continue to be published about the language. BASIC Special Interest Groups exist in a number of forms.

BASIC is somehow good for the soul. As new waves of languages come and go, BASIC still runs almost everywhere: without standards, it adapts to new environments easily and keeps pace with the fancy new languages. The ones that come and go.

Everyone, even Bill Gates, started with BASIC. Somehow, we all keep coming home to it over and over again. It's still the best language for quick programs and simple applications. BASIC interpreters, especially simple ones, can have great charm.

The computer hardware that BASIC is programmed on has turned full circle since the days it was developed. The powerful language to which only the computer scientists and mainframe programmers had access to can now be run on a hand held device.

## NS BASIC

NS BASIC for the Newton is a real programming language. It implements all the commonly used BASIC Statements in a straightforward manner, and has a number of powerful extensions.

As your Statements are entered into NS BASIC, they are compiled into an intermediate representation. When you run a program, each Statement is executed in turn. This type of system is both compiled and interpreted.

NS BASIC Corporation maintains a World Wide Web page at `http://www.nsbasic.com`. If you have a Web browser, check this site for important announcements, technical information, and example NS BASIC programs.

## NS BASIC and the Newton

When you bought your Newton, you probably thought that you'd be able to replace many of your paper-and-pencil tasks with it. You probably also hoped it would be able to function as a small programmable computer. NS BASIC has been designed for this purpose. Using it, you'll be able to create the applications you need, in a language that is easy to use, right on your Newton.

NS BASIC can provide access to all the information that is in your built-in applications. Using it, you can write programs that access your Names, Notes, and Dates information. You can write programs that find birthdays in the next week, by accessing Names information. You can even write programs that transfer Notes to your desktop computer.

NS BASIC can also be used for general purpose programming. Any program that can be written in BASIC can be written in NS BASIC. You can create customized databases, perform complex calculations, or even write games. What sets NS BASIC apart is its accessibility. You don't need to learn a complex new language just to take advantage of the powerful features built into your Newton.

## 1.2 System Requirements

In order to install NS BASIC you will need a Newton device, a desktop computer (Macintosh or PC Compatible,) the Newton Connection Kit or another package installer, and a cable that can be used to connect the Newton to the desktop computer.

NS BASIC can be used with the serial port of the Newton in several ways. If you want to use NS BASIC's serial connection between your desktop computer and your Newton, you will need a serial cable and communications software for your desktop computer. The cable that is supplied with all versions of the Newton Connection Kit is suitable for use with NS BASIC. If you do not have a suitable cable, one may be purchased at your local computer store.

### Newton System Compatibility

NS BASIC version 2.5x is compatible with pre-2.0 Newtons (the MessagePad, MP100, MP110, and MP120 running version 1.3 of the operating system) as well as MessagePads running the 2.0 version of the operating system. NS BASIC 3.0x only works with Newton 2.0.

NS BASIC 3.0 support an expanded set of features. These features are described in this handbook. They do not work with NS BASIC 2.5x. These include:

• the EDIT, MAKEPACKAGE and SETICON Commands

• the APP, AZTABS, AZVERTTABS, CLOSEBOX, LARGECLOSEBOX, DATEPICKER, DIGITALCLOCK, NUMBERPICKER, and PICTUREBUTTON widgets.

• the HEXDUMP, and NOTIFY Functions

- the DO, LOOP, EXIT DO, EXIT FOR, ELSE, END IF, and block IF THEN Statements

Newton 2.0 includes additional functions that do not exist in older versions. These include:

- the 'handwriting font
- the MAKEBITMAP function

Finally, version 2.5x uses a different method of creating run-only programs using the LISTRUN and IMPORT Commands. These are no longer documented in this handbook. Please refer to the Technotes supplied on the software disk for information on using these Commands.

### 1.3 Installation

NS BASIC is supplied on a software disk. You must install it onto your Newton using a package installer. The example shown here uses the Newton Connection Kit to install the NS BASIC package. Follow the directions in your package installer software manual when installing NS BASIC.

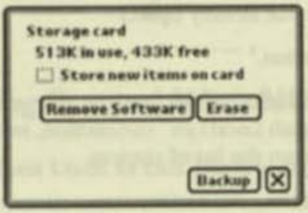
NS BASIC can be installed on your Newton or on a storage card.

#### Preparing to Install on the Newton

Before you attempt to install the software on the internal memory of your Newton, check the available memory in your Newton. Open the Extras Drawer and tap "Prefs". In the list that is displayed, tap "Memory". Verify that the free memory displayed is at least 185k. If you have less than this amount, you should remove some information from your Newton or consider installing NS BASIC on a storage card. Refer to your Newton Handbook's Managing Memory section for more information on removing data and packages from your Newton.

If you have a storage card installed in your Newton, open the Extras Drawer and tap Card.

Verify that the checkbox "Store new items on card" is not checked.


Storage card

513K in use, 433K free

□ Store new items on card

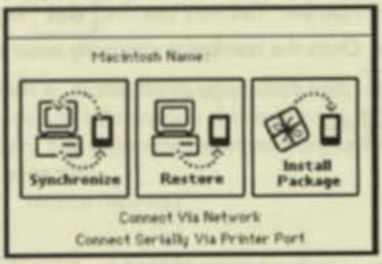
Preparing to Install on a Storage Card

Before you attempt to install the software, check the available memory on your card. Open the Extras Drawer and tap "Card". Verify that the free memory displayed is at least 185k. If you have less than this amount, you should remove some information from your card or consider installing NS BASIC on another storage card. Refer to your Newton Handbook's Managing Memory section for more information on removing data and packages from your storage card.

Verify that the checkbox "Store new items on card" is checked.

Installing The NS BASIC Package

1 Attach your Newton to your desktop computer with an appropriate cable. Insert the NS BASIC disk into your disk drive. Start the Newton Connection Kit software on your desktop computer. Choose "Install Package" From the Connection Window or the Newton Menu.



The connection software will open a window on your desktop computer where you may select a package to install. Select "NSBASIC.PKG" from the disk drive containing the NS BASIC disk.

**2** When the Connection Kit indicates that it is ready for you to open a connection from your Newton, open the Extras Drawer (if it is not already open.)

**3** Tap "Connection."

**4** Choose the kind of connection you are using. If you are using a "Macintosh LocalTalk" connection, select the computer's name from the list of choices.

**5** Tap "Connect."

**6** After your connection kit indicates the installation was successful, you'll see the NS BASIC icon in the Extras Drawer.

REM
GOSUB
LET
BASIC

Entering Your Registration Number

The first time you start NS BASIC on your Newton, you will be asked to enter your Product Registration Number. This number is printed on the outside of the back cover of this Handbook or on the bottom of the box, as well as on the Product Registration Form. This form is the last page of the Handbook. Please tear this form out now, fill it in, and send it to the address printed on the form.

Open the Extras Drawer and tap on the NS BASIC icon. The initial registration screen is displayed, along with the on-screen keyboard. Use the keyboard to tap in your Product Registration Number. You may use the **del** key to make corrections. Once the number is correctly entered, tap the return ↵ key. Your copy of NS BASIC is now installed and ready to use.

CHAPTER

2

# 2. Getting Started With NS BASIC

## 2.1 Conventions Used in this Handbook

The following notation conventions are used in this Handbook:

KEYWORDS Capital letters indicate NS BASIC keywords, symbols, and other text that must be typed exactly as shown. For the purposes of this manual, uppercase text indicates a required part of the Statement syntax. NS BASIC is case-insensitive: keywords are accepted with either uppercase letters, lowercase letters, or any mixture of the two. A keyword such as GOTO may be entered into your programs as goto, Goto, or GOTO.

*placeholders* Italic text indicates a placeholder for types of information that you must supply. In the following Statement, *lineNumber* is italicized to show that the GOTO Statement requires a line number:

GOTO *lineNumber*

In an actual program Statement, *lineNumber* must be replaced with a specific line number, such as:

```
GOTO 40
```

`Examples` This Monaco typeface indicates example program code and information that is printed on your NS BASIC screen. The following example shows a line from a NS BASIC program:

```
10 PRINT "Hello World!"
```

**`User Input`** A bold Monaco typeface is used to indicate something entered by the user in response to a NS BASIC prompt. It distinguishes between an on-screen prompt and user input when both appear in the same example. For instance, **`John`** is entered in response to the "Enter Your

Name:" prompt:

```
Enter Your Name:
? John
```

[Optional] Brackets indicate that the enclosed items are optional. In the following example, brackets are used to show that entering a second item to display on the screen is optional for the PRINT Statement:

PRINT *expression1* [ ,*expression2* ]

Both of these PRINT Statements are legal, since PRINT accepts one or two expressions:

```
PRINT "Hello"
PRINT "Hello","World"
```

| The vertical bar indicates that the items are mutually exclusive. In the following example the bar indicates that the RUN command can either be used with a file name or a line number:

RUN [*fileName* | *lineNumber* ]

Underlined Underlined text indicate that the items are environment variables. In the following example, underlining is used to indicate that PRINTDEPTH is an environment variable:

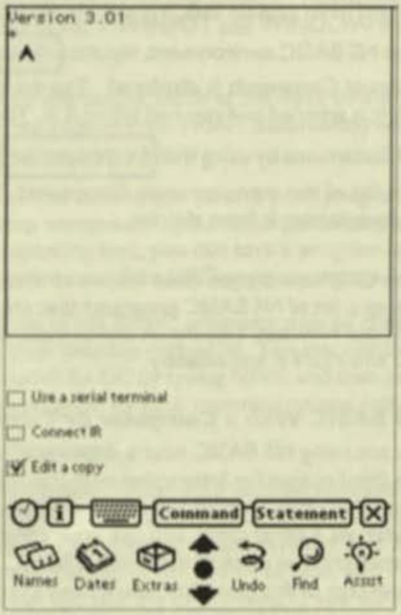
PRINTDEPTH is used to control the amount of information displayed using the PRINT Statement.

## 2.2 Interacting With NS BASIC

NS BASIC provides a powerful environment for programming on the Newton.

You begin working with NS BASIC by opening the Extras Drawer and tapping on the NS BASIC icon.

After briefly displaying an introduction screen, the NS BASIC programming environment is shown:



```
Version 3.01
*
```

□ Use a serial terminal

□ Connect IR

☑ Edit a copy

NS BASIC does not use handwriting recognition for program entry. There are three ways to interact with the NS BASIC environment:

### Using the Newton Keyboard

An on-screen keyboard is displayed when you start NS BASIC. You use the keyboard to enter and edit programs, enter commands that the NS BASIC environment understands, and enter information into running programs in response to input prompts.

You can hide and show the keyboard by tapping the keyboard button ⌨.

### Picking Items Out of a List

Tapping out complex NS BASIC Statements and Commands using the on-screen keyboard can be tedious. Several common Commands and Statements can be quickly entered by selecting them from pop-up lists. To quickly enter a Command to the NS BASIC environment, tap the **Command** button. A list of Commands is displayed. Tap the desired Command. It is entered as if you had typed it in. You can enter several Statements by using the **Statement** button. It displays a list of the more common Statements. You can enter one by selecting it from this list.

Tapping the Overview Button (also known as the belly button) brings up a list of NS BASIC programs that are currently saved on your Newton. Tapping on one of these programs will LOAD and RUN it immediately.

## Using NS BASIC With a Computer or Terminal

When you are using NS BASIC near a desktop computer, you have a third option for interacting with the NS BASIC environment. You may connect your Newton to your desktop computer via a serial cable, and use your computer's keyboard and screen in place of those in NS BASIC. In order to take advantage of this capability, you'll need a serial cable (such as the one supplied with the Newton Connection Kit) and communications software for your desktop computer.

**1** Connect your Newton to your desktop computer using an appropriate serial cable.

**2** Start the communications software on your desktop computer. Examples of compatible communications software include ZTERM for Macintosh, Kermit for Macintosh and PC Compatibles, Procomm, and MicroPhone Pro. Set your host system's serial port to 9600 baud, 8 bits, no parity with software flow control (also known as Xon/Xoff). Also, set it to echo characters locally (also known as half duplex). If there is an option for automatic newline or automatic line feed, enable it.

**3** Tap □ Use a serial terminal . When it is selected, you should see the word "Connected" on your desktop computer's screen. On the next line you will see NS BASIC's asterisk prompt (*). This means the connection is working and NS BASIC is ready for use.

While you are using NS BASIC in this way, it does not duplicate the text display on the Newton. Newton-specific Statements such as HWINPUT and WINDOW will still display on the Newton.

All text input and output (such as the input to INPUT Statements and the output from PRINT Statements) will now be on your desktop computer.

You can use this connection to save your program text on your desktop computer. If your communications software supports capturing text, you can save a program by turning on text capture, and then LISTing your program.

Text versions of NS BASIC programs may be created and edited on your desktop computer. You can transfer these programs to NS BASIC by typing NEW, and then pasting the text of the program to your communications software.

### Possible Problems

When using NS BASIC with your desktop computer, try not to enter information while NS BASIC is outputting text. Your entry may be ignored, or the Newton may freeze. If your Newton freezes (it will ignore all taps, and may not turn off when you use the on/off switch) you will need to press the reset button. Please refer to your Newton Handbook's Tips and Troubleshooting section. As in all cases when you reset your Newton, your information is not lost. When you next start NS BASIC, your program will be there.

### Starting, Stopping, and Resetting

NS BASIC stores each line of your program after you tap the return key. You may close NS BASIC any time, and the next time you start it your current program will still be in the environment. Remember to complete any program Statement you have typed (by tapping the return key) before you close NS BASIC.

When you are programming on a computer, even a Newton, it is possible to get into a state where no input will be accepted. This is called "frozen". When this happens, you can thaw your Newton by pressing the reset button. Please refer to your Newton Handbook's Tips and Troubleshooting section.

## 2.3 Programming in NS BASIC

### The NS BASIC Programming Environment

NS BASIC provides a full featured BASIC programming environment for the Newton. In order to introduce you to these features, an example program will be developed in NS BASIC. Each step in the process will introduce features of the environment. Start NS BASIC and follow along!

### Creating a Program

When you start working on a new program, you should always use the NEW Command. This clears any previous program Statements from the environment. As stated before, you may enter your program via the on-screen keyboard, by picking from the Statement list, or by using an attached desktop computer.

The example program is very simple. It computes the future value of an investment based on the interest rate, amount, compounding term, and number of years invested. We're going to create the program with one small error, so that we can show some of the debugging features of the environment.

Remember to start each new line with a line number. In NS BASIC, line numbers are used to determine the order of Statement execution. You can enter lines in any order, and insert lines by assigning line numbers between existing lines. It is usually a good idea to increment your line numbers by 10 or 20, so you have room to insert new lines later. Line numbers start at 1 and end at 9999.

Enter the program shown below:

```
10 REM Future Value of an Investment
20 PRINT "Enter starting principal"
30 INPUT principal
40 PRINT "Enter interest rate as % (i.e. 10)"
50 INPUT rate
60 PRINT "Enter term (i.e. 12 for monthly)"
70 INPUT term
80 PRINT "Enter number of years"
90 INPUT years
100 REM Compute final interest
110 rate = rate * 0.01 / term
120 balance = principal
130 FOR y = 1 TO years
140 FOR c = 1 TO term
```

```
150 interest = rate * principal
160 balance = balance + interest
170 NEXT c
180 NEXT y
185 PRINT "Using our calculations:"
190 PRINT "After ";years;" years the balance
is: ";balance
200 REM The easy and fast way
210 PRINT "Using the COMPOUND function:"
215 PRINT compound(rate, years*term) *
principal
220 END
```

## Editing a Program

### Changing Line Numbers

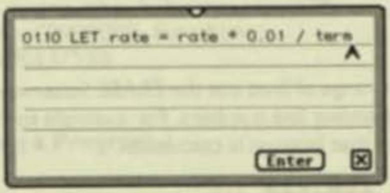
If you've run out of line numbers between Statements (or if you'd just like an orderly program) the RENUM Command will renumber your lines for you. Statements such as GOTO that refer to line numbers will be updated to refer to the new line number, so no additional editing is needed.

Use RENUM on the example program:

```
* RENUM
From 0001 TO 9999 BY 0010 BASE 0010
*
```

### Editing Lines

The traditional way to edit a line in BASIC is to enter a new line with the same line number containing the corrected text. This method works in NS BASIC as well. If you are using a desktop computer, you can use cut and paste to edit the line you wish to change, or even the entire program. On the Newton, there is another way to edit a program line. Tap on the line when it is displayed in the program window and an Edit Box is displayed.



```
0110 LET rate = rate * 0.01 / term
```

You may use the on-screen keyboard to update the Statement, as well as the standard Newton editing gestures for inserting spaces, deleting words, and cut/paste. Tap Enter to save your changes, ☒ to discard them. If you know the line number of the line you want to edit, use the EDIT *lineNo* Command. This will load the specified line into the Edit Box

If you wish to copy a line to another area of your program, you can use the Edit Box. Just tap on the line, and then change the line number shown in the Edit Box to the desired value. When you tap enter, a copy of the line is created at the new line number.

You may use the Edit Box on any line you entered into NS BASIC, even those without line numbers. This is very handy for re-executing Commands, or for entering a line number if you forget to start a Statement with one.

### Using the Newton Clipboard

You can write programs in the Notepad and copy them into the Newton clipboard. If you run NS BASIC and paste them to the main window, and then tap the return key, they will be entered into the currently loaded program. You can also select code in the NS BASIC window and drag it to the lower right hand corner of the Newton. Exit NS BASIC (the BYE command can be used if the close button is covered by the program clipping on the clipboard) and paste the program text into the Notepad.

### Deleting Lines

To delete a line from a program, simply enter that line number again with no Statement. If you wanted to delete the remark line at the start of our example (line 10) you would enter 10 and tap return:

```
* 10
*
```

To delete a range of lines use the ERASE Statement with beginning and ending line numbers. For example to delete the second way that interest is calculated:

```
* ERASE 200,215
*
```

## Examining a Program

You can use the LIST Command to display your program in the text window. If your program contains more lines then can be displayed in the window, it is listed one window at a time, followed by

```
--More--
```

You continue the listing by tapping the return key. You can display specific ranges of line numbers after a LIST Command to see just those lines. LIST the last 5 lines to see the effect of the RENUM Command.

```
* LIST 200,
0200 PRINT "After ";years;" years the balance
is: ";balance
0210 REM The easy and fast way
0220 PRINT "Using the COMPOUND function:"
0230 PRINT compound(rate, years*term) *
principal
0240 END
*
```

## Executing a Program

You use the RUN Command to begin executing your program at the first line number. Let's run our example program:

```
* RUN
Enter starting principal
? 100
Enter interest rate as % (i.e. 10)
? 10
Enter term (i.e. 12 for monthly)
? 12
Enter number of years
? 2
Using our calculations:
After 2 years the balance is: 120
Using the COMPOUND function:
122.039096137556
*
```

## Debugging a Program

The balance we computed should have been the same as the one using the COMPOUND Function. It seems we've got a bug in our program. There are three types of errors possible in NS BASIC: compiler-time errors, run-time errors, and logic errors.

A compile-time error is made when you type in a Statement incorrectly. NS BASIC signals this error immediately after you tap the return key.

```
* 200 prant "hello"
Error 2 -- Statement or syntax invalid
```

This error indicates that there is no PRANT Statement. We misspelled PRINT and NS BASIC signaled the error.

A run-time error is caused when a situation arises that cannot be known at compile-time. For example, you may divide two variables (a/b). When NS BASIC compiles a Statement such as

```
* 200 c = a/b
```

It does not know the values of a and b. When you run this program, if b is a string, there is an error (dividing by a string is undefined.) In this case, NS BASIC will stop executing your program at line 200 and display this message:

```
0200 :Error 29- Expression
```

A logic error is caused when your program is incorrect. It produces results that are wrong or unexpected. This type of error is common and NS BASIC cannot detect it. Unfortunately, our error is a logic error. We did not receive any error messages when we ran it, but the two values displayed at the end should have been the same. We'll have to debug our program!

NS BASIC gives you a number of tools for debugging. You can enable tracing of your program using the TRACE ON Statement. Every Statement executed after TRACE ON will have its line number printed into the display window. You disable tracing using the TRACE OFF Statement. It may be difficult to follow IF THEN ELSE, GOTO, and GOSUBs in your code. You can use tracing to see where a program is going.

The STOP Statement can be inserted into the area of the program that is causing problems. STOP does just that: it stops the program. You may print and change the values of program variables, check the memory statistics, and once you have a good idea of what is happening, you can continue execution at the next line number using CON.

If you are having problems with one section of a program, you can begin execution at that point by using RUN *lineno*. You may also set any necessary variables to any value prior to starting the execution. Use STOP and RUN *lineno* together to debug small parts of complex programs.

We know that our program is computing the interest incorrectly. Add a STOP Statement at line 95 and line 165:

```
* 95 STOP
* 165 STOP
*
```

This way we can check that our initial values are correct, and then we can see how each loop changes these values. RUN the program again, entering the values as shown:

```
* RUN
Enter starting principal
? 100
Enter interest rate as % (i.e. 10)
? 10
Enter term (i.e. 12 for monthly)
? 12
Enter number of years
? 2
Stop at 0095
*
```

When the program STOPs at line 95 your Newton will beep. We can use the VARS Command to view the program variables, and their values:

```
* VARS
PRINCIPAL: 100
Rate: 10
TERM: 12
Years: 2
*
```

Continue the execution of the program using the CON Command. The program will STOP again at line 165. Use the VARS Command again:

```
* CON
Stop at 0165
* VARS
PRINCIPAL: 100
Rate: 0.0083333333333333
TERM: 12
Years: 2
BALANCE: 100.833333333333
y: 1
c: 1
INTEREST: 0.833333333333333
*
```

This looks fine. Continue the program using CON. It STOPs at 165 again since we're in a pair of loops. Use the VARS Command again:

```
* CON
Stop at 0165
* VARS
PRINCIPAL: 100
Rate: 0.0083333333333333
TERM: 12
Years: 2
BALANCE: 101.666666666667
y: 1
c: 2
INTEREST: 0.833333333333333
*
```

Well, here's the problem. It seems the interest we compute every period is the same! That's not how compounding interest is supposed to work – it's supposed to increase. LIST the section of the program that computes interest:

```
* LIST 100, 180
0100 REM Compute final interest
0110 LET rate = rate * 0.01 / term
0120 LET balance = principal
0130 FOR y = 1 TO years
0140   FOR c = 1 TO term
0150     LET interest = rate * principal
0160     LET balance = balance + interest
0165     STOP
0170   NEXT c
0180 NEXT y
*
```

Take a close look at this code, because we know the interest is computed wrong. Notice line 150. It seems we're always computing the interest based on the original principal, not the current balance! Edit line 150 so that it is:

```
* 150 interest = rate * balance
*
```

Also remove the two STOP Statements:

```
* 95
* 165
```

RUN the program again. Notice that the values are the same. We've debugged it!

## Saving and Loading a Program

When you first create a new program (after a NEW Command) all the Statements you type in are saved in a temporary file. This is how NS BASIC saves your work - so that you may quit any time you want, and then come back later to continue where you left off. It also protects your work for those rare occasions that you freeze your Newton developing a program.

When you are ready to save your new program, you must use the SAVE Command. We'll save our example program with the name Compound_Interest.

```
* SAVE Compound_Interest
Compound_Interest saved
*
```

Notice that file names cannot have spaces, but may contain both upper and lower case letters, as well as special characters like underscore (_) and hyphen (-).

When you would like to edit a program you have already saved, you use the LOAD Command. We'll LOAD our just-saved program back into NS BASIC.

```
* LOAD Compound_Interest
*
```

**Note:** When you LOAD a program, NS BASIC performs a "NEW" first. If you have not saved your current program, any changes will be lost.

NS BASIC allows you to edit a copy of your saved program, or edit the original program in place. This option is controlled using the ☑ Edit a copy setting. If you choose to edit a copy, then any changes you make will not be saved until you use the REPLACE Command. This will update the saved version of the currently LOADed program with any changes you have made. If you decide you'd rather not save the changes, you may just LOAD the program again or use the NEW Command to start working on a new program. The advantage of this way of working is that your original file is only updated when you want it to be. The disadvantage is that you may accidentally discard changes you make, by forgetting to use the REPLACE Command. Remember that the NEW and LOAD Commands completely clear all unsaved information from NS BASIC.

Let's try some changes on a very small program, so you can see how NS BASIC works in these two ways. Enter in the following program:

```
* NEW
* 10 a = 5
* 20 b = 10
* 30 c = a/b
* 40 PRINT c
* SAVE Small_Example
Small_Example saved
*
```

Make sure ☒ Edit a copy is checked. Clear all the information out of NS BASIC using NEW, and then LOAD Small_Example:

```
* NEW
* LOAD Small_Example
*
```

Now change line 10 by entering a new line 10 as follows:

```
* 10 a = 100
*
```

LIST your program and confirm that the change is in the program. Clear all the information again using NEW, and then LOAD Small_Example again, and finally LIST it:

```
* NEW
* LOAD Small_Example
* LIST
0010 LET a = 5
0020 LET b = 10
0030 LET c = a/b
0040 PRINT c
*
```

Notice that your change to line 10 was not saved. This is because we did not use the REPLACE Command prior to clearing all the information with the NEW Command. Try making the change to line 10 again, but this time use the REPLACE Command afterwards:

```
* 10 a = 100
* REPLACE
* Small_Example saved.
*
```

Clear all the information again using NEW, and then LOAD Small_Example again, and finally LIST it. Notice that this time, since we specifically saved the changes we made, the new line 10 is in the saved version of the program. If you perform the same actions with "Edit a copy" not checked, you'll find that your changes to line 10 are saved as soon as you make them. There is no need to use the REPLACE Command.

### Saving Packages

Packages are programs that the Newton can execute directly. You can create programs that other people can run, even though they do not own NS BASIC. Additionally, you may want to create package versions of programs they you run frequently, so they will appear in your Extras Drawer. You can also make a program into the Newton backdrop application.

The MAKEPACKAGE statement creates a stand-alone package in the Extras drawer. The name in the Extras drawer is the name the program was SAVEd as. All stand-alone packages use a default icon in the extras drawer:

You can use the SETICON Statement to use a custom icon. The complete name of the package is *programName*.pkg:NS-BASIC. The name displayed in the Extras drawer is *programName*.

We'll create a small program, save it, and then use the MAKEPACKAGE Command to create the stand-alone package. Enter the following program and save it as INVEST:

```
10 REM MAKEPACKAGE Example
20 PRINT "Enter starting principal"
30 INPUT principal
40 PRINT "Enter interest rate as % (i.e. 10)"
50 INPUT rate
60 PRINT "Enter term (i.e. 12 for monthly)"
70 INPUT term
```

```
80 PRINT "Enter number of years"
90 INPUT years
100 REM Compute final interest
110 rate = rate * 0.01 / term
120 PRINT "After ";years;" years the balance
is: "; compound(rate, years*term) * principal
130 END
* SAVE INVEST
INVEST saved.
* MAKEPACKAGE INVEST
*
```

The package will appear in the Extras drawer under Unfiled Icons. You can use a stand-alone program as the backdrop application of your Newton, provided it is stored in the internal memory of the Newton.

### Moving a Program

When you SAVE a program, NS BASIC saves the program in one of two places: either in your Newton's internal memory or on the storage card currently installed in the Newton. Please refer to the Memory and Storage section of your Newton Handbook regarding controlling where new information is stored.

If you have saved a program on a storage card, that program will only be available when that storage card is inserted and store new items on card is checked. If you would like to move a program from one card to another, or from a card to your Newton, or from your Newton to a card, you will need to use the serial connection.

Copy a text version of your program by using the LIST Command. Use copy and paste to save this text on your desktop computer. Use the DELETE Statement to delete the program on your Newton. Close NS BASIC, and use Card in the Extras Drawer to select the desired location for the program. Open NS BASIC and connect to the serial terminal. Clear all memory using the NEW Command, and then use cut and paste to enter the program into the communications software on your desktop computer. Finally, use the SAVE Command to save the program in the new location.

### The Elements of a NS BASIC Program

A program in NS BASIC is a set of numbered Statements or lines. Each NS BASIC program line may consist of the following elements:

```
line-number STATEMENT arguments // comment
```

A line number is any number from 1 to 9999.

A STATEMENT is an instruction for your program. Examples are PRINT, INPUT and IF. The Statement and its arguments determine what action (if any) will be taken by NS BASIC when the line is executed.

Any text following // on a line is a comment, and is ignored by NS BASIC.

## Data Types, Literals, and Variables

The numbers, strings, and other data elements that your program works with have an associated data type. A data type is a way of describing a group of related items. For example, the Integer data type describes all whole numbers.

Literals are just that, literal values you use in your programs. You use them all the time: to set the initial value of a variable, to establish the starting and ending values of a FOR NEXT loop, and so on. You cannot change the value of a literal.

Variables are the named holders of your data. A variable's value may be changed as needed.

NS BASIC supports the following data types:

## Numeric Data Types

There are a several types of Numeric data, but they all share the same behavior. You can generally mix and match among the types of numeric data without difficulty.

| Type | Size | Range | Literal |
|---|---|---|---|
| Integer | 30 bits | -53,6870911 to<br>53,6870911 | 100 |
| Real | 16 bits | $1.5x10^{-45}$ to<br>$3.4x10^{38}$ | 12.5 |
| Double | 32 bits | $5.0x10^{-324}$ to<br>$1.7x10^{308}$ | 1.0e100 |
| Extended | 64 bits | $1.9x10^{-4951}$ to<br>$1.1x10^{4932}$ | 1.0e1000 |

## Boolean Data Types

Booleans consists of two values: TRUE and NIL (false). This data type is used with the IF Statement. It tests the Boolean value of an expression and selects the THEN Statement if it is TRUE, or the ELSE Statement if it is NIL.

## String Data Types

Strings consist of a series of characters. There are a number of Functions that manipulate strings. The concatenation operator (&) is used to join two strings together. The && operator joins two strings as well, but inserts a space between them. A string literal is enclosed in quotation marks:

```
"This is a string literal"
```

## Array Data Types

Arrays are containers. They are lists of values stored with the same name. Each element in the array is referred to by including a number in square brackets after the variable name. Arrays start with a zero element and can have many elements. `ARR[2]` refers to the third element in the ARR array. Each element in an array can be of any type. Array literals are enclosed in square brackets, and each element is separated by a comma. This is an array literal:

```
[1, 2, "Even Strings", 3.14]
```

### Frames

Frames are also containers. They are a collection of zero or more fields enclosed in curly brackets and separated by commas. Each field consists of a field name, followed by a colon and its value. As with arrays, each field can be of any type, including an array or another frame. This is a frame literal:

```
{field1: "String field", number: 12,
realnumber: 3.14}
```

Fields in a frame are referred to by the frame name followed by a period and the field name in the frame. `myField.firstName` refers to the value "John" in this example:

```
10 myField = {firstName: "John", lastName:
"Doe"}
20 PRINT myField.firstName
```

You can add new fields to a frame at any time, simply by assigning the new field a value using the same notation. To add `myNickName` to `myField`, use:

```
30 myField.myNickName = "Johnny"
```

Frames are used extensively for files. Each record in a file is a frame. Refer to the Frames section in the Advanced Topics chapter of this Handbook for more information.

### Symbols

Symbols are internal forms of identifiers. You use symbols to access frame elements, and to create values that are not evaluated. You specify a symbol by preceding it with a '. Symbols may be assigned to variables, used in expressions, and PRINTed.

```
• x='symbolname
• PRINT x
symbolname
```

### Variable Names

A variable is a name that holds a value. The name consists of a sequence of alphabetic and numeric characters. There is no limit to the length of a variable name in NS BASIC, and every character in the name is significant. We tell you this because in some older BASICs you could only use short names. Variable names are not case sensitive, and spaces and other special characters may not be used. Variable names must start with a letter.

NS BASIC keywords may not be used as variable names. For a complete list of keywords, see Appendix B.

The following list shows some variable names that are allowed by NS BASIC:

```
text
LLAMAS           // same as llamas or Llamas
Lemons
W1Spec
world134
```

And some that are not allowed:

```
1table           // starts with a number
X&Ycords         // uses special character &
first counter  // has a space
%correct    // does not start with a letter
size        // this is a NS BASIC keyword
Second_Win_Num  // more special characters
```

## Un-Typed and Typed Variables

If you use a variable name that does not specify a type, then NS BASIC automatically determines the proper type of variable to be used based on the type of the value you assign to it. What this means is that a single variable may hold a string, then a numeric, then a frame, etc.

If you end a variable name with **$**, then that variable will always (and only) contain a string value. Assigning a numeric to one of these variables converts that numeric to a string value first. Array variable names can not end in **$**.

## Expressions and Operators

An expression is a literal, variable, formula or function call that has a value. Here are some examples of expressions:

```
6/3                        // result is 2
5+6/3                      // result is 7
"This" && "that"       // result is "This that"
```

A string expression can be a string literal, a string variable, or it may combine string literals, string variables and substrings to produce a single string value. Similarly, a numeric expression can be a numeric constant, a numeric variable, or a function/variable that produces a single numeric value.

## Arithmetic Operators

NS BASIC allows the following arithmetic operators in this descending order of priority:

| | | |
|---|---|---|
| * | / | Multiplication and Division |
| + | - | Addition and Subtraction |

Parenthesis can be used to change the order of evaluation.

```
* PRINT 2+3*4
14
* PRINT (2+3)*4
20
```

NS BASIC supports floating point arithmetic. All numeric operations are carried out to 32 bit precision and are truncated to 16 digits at the conclusion of the operation. The REMAINDER Function may be used to find the remainder of a division. The DIV Function is used for integer (whole number) division.

Arithmetic operators can only be used with numeric expressions. They may not be used with strings.

## Relational Operators

Relational operators compare two values and return a Boolean value of TRUE or NIL (false). This result can be used to change the flow of a program. Relational operators have a lower priority than arithmetic operators. The relational operators are:

| | |
|---|---|
| = | Equal |
| <> | Not Equal |
| < | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |

In the LET Statement the equal sign is used to assign a value to a variable, not as a relational operator.

### Boolean Operators

Boolean operators tie expressions together, returning a TRUE or NIL answer. Arithmetic and relational operators are evaluated before Boolean operators. Two of the operators, AND and OR, require two expressions. The NOT operator applies to one expression.

The Boolean operators are:

AND Returns TRUE if the two expressions are both TRUE.

OR Returns TRUE if either expression or both of the expressions are TRUE.

NOT Returns TRUE if the expression is false or returns NIL if it is TRUE.

Examples of AND, OR, and NOT:

```
0010 INPUT a
0020 IF a >=1 AND a <=100 THEN PRINT
"Number Between 1 & 100."
```

```
0010 INPUT a
0020 INPUT b
0030 IF a = 10 OR b=10 THEN PRINT "One of the
numbers entered is 10"
```

```
0010 INPUT a
0020 INPUT b
0030 IF NOT a = 10 OR NOT b=10 THEN PRINT "One
of the numbers entered is NOT 10"
```

Boolean operators can be used with any expression that returns a Boolean value. They may not be used in numeric expressions.

## 2.4 Immediate Statement Execution

Commands (such as RUN and LOAD) are always executed immediately as they are entered. Statements entered without a line number are also executed immediately. The results of assignment Statements are available for later use, but the Statements themselves are not saved. There are two uses for immediate Statement execution:

### Simple Calculations

NS BASIC may be used as a full-featured calculator with a very large memory. You may enter several calculations, assigning the results to variables. The variables will hold their values until a NEW Command is issued. For example:

```
* b=5*2
* a=10*b
* PRINT a
* 100
* ; a
* 100
*
```

### Debugging

When you use the STOP or END Statement in your program, you may PRINT the contents of individual variables, use the VARS Command to see all current variables, and even change the values in variables. An example of debugging a program was given in section 2.2.

CHAPTER

3

# 3. NS BASIC Reference

The Reference chapter contains an entry for every Command, Statement, Function, and Widget in NS BASIC, in alphabetical order. The entries are listed in the index grouped under Commands, Statements, Functions, or Widgets.

Each entry in the Reference chapter consists of the following information:

## Name Category

ITEM parameters

**DESCRIPTION**
This section describes the ITEM and its parameters. Details concerning the uses of ITEM are given, as well as any constraints on its use.

**EXAMPLE**
A small program that uses ITEM is listed here.

**OUTPUT**
This section shows the results of running the Example program.

**RELATED ITEMS**
A list of zero or more NS BASIC Commands, Statements, Functions, or Widgets that are related to ITEM. You may often gain a better understanding of ITEM by reviewing the related items.

# ABS Function

ABS(*x*)

FABS(*x*)

### DESCRIPTION

ABS returns the absolute (positive) value of the integer number *x*.

FABS returns the absolute (positive) value of the real number *x*.

### EXAMPLE

```
10 REM ABS Example
20 REM This program returns the positive value
of any number INPUT to it.
30 PRINT "Please enter any number:"
40 INPUT Number
50 PRINT "The absolute value of the number you
entered is = " ; ABS(Number)
```

### OUTPUT

```
Please enter any number:
-20
The absolute value of the number you entered
is = 20
*
```

### RELATED ITEMS

# ADDARRAYSLOT Function

ADDARRAYSLOT(*Array, item*)

### DESCRIPTION

ADDARRAYSLOT extends *Array* by adding a new element to the end. The value of that element is *item*. ADDARRAYSLOT returns *item*.

### EXAMPLE

```
10 REM ADDARRAYSLOT Example
20 a = [1,2,3]
30 PRINT "Please enter any number:"
40 INPUT Number
50 ADDARRAYSLOT(a, Number)
60 PRINT "The new array is  = " ; a
```

### OUTPUT

```
Please enter any number:
30
The new array is = [1, 2, 3, 30]
*
```

### RELATED ITEMS

ARRAYREMOVECOUNT

# ANNUITY Function

ANNUITY (*rate,periods*)

### DESCRIPTION

Calculates the present value factor of an annuity at a given interest *rate* over the specified number of *periods*. The interest *rate* is the rate per period. For example, 12% per year would be  expressed as a monthly rate of 0.01 (12%/12 months = .12/12 = 0.01).

### EXAMPLE

```
10 REM ANNUITY Example
20 REM Compute annuity on monthly basis.
30 PRINT "Annual Interest Rate:"
40 INPUT Rate
50 PRINT "Number of months:"
60 INPUT NumMonths
70 PRINT "Cost of the item:"
80 INPUT Cost
90 PRINT "The cost for all payments is $";
ANNUITY((Rate * 0.01)/12, NumMonths) * Cost
```

### OUTPUT

```
Annual Interest Rate:
12
Number of months:
50
Cost of the item:
1000
The cost for all payments is $39,196.117531105
*
```

### RELATED ITEMS

COMPOUND

# APP Widget

WINDOW *winNum*, *windowSpec*, "APP"

**DESCRIPTION**
The APP widget displays the standard Newton application background. This includes a clock display and close box. When the user taps the close box, theprogram branches to the line specified in the GOTO field of the windowSpec.

The widget is controlled using the *windowSpec*. These fields are supported:

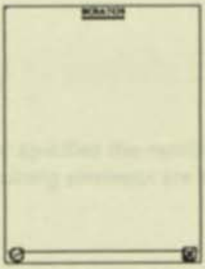
`title`: the title to display for the application.

You may also use these fields in *windowSpec*: `viewFlags`, `viewFont`, `GOTO`

**EXAMPLE**

```
10 REM APP Example
20 w1Spec := {GOTO:100}
30 WINDOW w1, w1Spec, "APP"
40 SHOW w1
50 WAIT 1000
60 GOTO 50
100 REM tapped close box
110 HIDE
120 PRINT "Closed."
```

**OUTPUT**



**RELATED ITEMS**
HIDE, SHOW, WINDOW

# ARRAYREMOVECOUNT Function

ARRAYREMOVECOUNT(*Array, index, numToRemove*)

**DESCRIPTION**

ARRAYREMOVECOUNT deletes elements from *Array* . The first element to remove is given by *index*, and the number of elements to remove is *numToRemove*. ARRAYREMOVECOUNT returns NIL.

**EXAMPLE**

```
10 REM ARRAYREMOVECOUNT Example
20 a = [1,2,3,4,5,6,7]
30 ARRAYREMOVECOUNT (a, 2,3)
40 PRINT "The new array is  = " ; a
```

**OUTPUT**

```
The new array is = [1, 2, 6, 7]
•
```

**RELATED ITEMS**

ADDARRAYSLOT

ARRAYTOPOINTS(*shapeArray*)

**DESCRIPTION**

ARRAYTOPOINTS creates a drawing using an array that specifies the shape of the drawing and the X,Y points for it. The resulting drawing can be displayed in a window using WDRAW once it is converted into a shape with MAKESHAPE().

The first element of *shapeArray* describes the overall shape of the drawing:

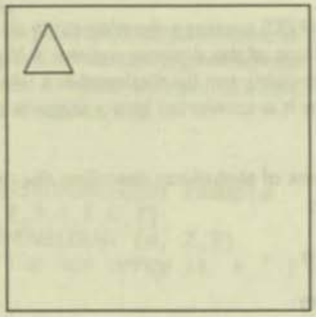
| | |
|---|---|
| 0 | Circle |
| 1 | Ellipse |
| 2 | Small open curve |
| 3 | Closed polygon |
| 5 | Open polygon |
| 6 | Closed curve |
| 7 | Open curve |
| 8 | Line |
| 9 | Triangle |
| 10 | Square |
| 11 | Rectangle |

The second element specifies the number of X,Y points in the shape. The remaining elements are the X and Y values of each point.

**EXAMPLE**

```
10 REM ARRAYTOPOINTS Example
20 shapeArray = [9,4,25,10,10,40,40,40,25,10]
30 points := ARRAYTOPOINTS(shapeArray)
40 shape := MAKESHAPE(points)
50 wspec := {viewBounds: SETBOUNDS(10, 10,
200, 200)}
60 WINDOW w1, wspec
70 SHOW w1
80 WDRAW w1, shape
```

**OUTPUT**



**RELATED ITEMS**

MAKESHAPE, POINTSTOARRAY, WDRAW, WINDOW

# AZTABS

WINDOW *winNum*, *windowSpec*, "AZTABS"

WINDOW *winNum*, *windowSpec*, "AZVERTTABS"

### DESCRIPTION

The AZTABS and AZVERTTABS widgets display the standard Newton selection tabs in the horizontal and vertical orientations.

These widgets are controlled using the *windowSpec*. These fields are supported:

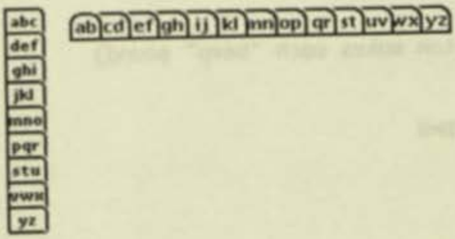
`GOTO`: the line number to goto if tapped.

`viewBounds`: if not supplied, defaults to the top (AZTABS) or left edge centered (AZVERTTABS). Note that if this is specified, the height (`bottom - top`) must be at least 20 for AZTABS, and the width (`right - left`) must be at least 30 for AZVERTTABS.

### EXAMPLE

```
10 REM AZTABS Example
20 w1Spec = {GOTO: 100}
30 WINDOW w1, w1Spec, "AZTABS"
40 SHOW w1
100 REM A selection was made
110 PRINT "Index: "; w1Spec.curIndex; ", Text:
", w1Spec.text
```

### OUTPUT



### RELATED ITEMS

SHOW, HIDE, WINDOW

# BEEP Statement

BEEP *beepsound*

**DESCRIPTION**

Causes the Newton to emit a single "beep" through the Newton's speaker. The actual sound of the beep is selected by *beepsound* as follows:

| | |
|---|---|
| 0 | Alarm Wakeup |
| 1 | Boot Sound |
| 2 | Click |
| 3 | Crumple |
| 4 | Extras Drawer Closing |
| 5 | Extras Drawer Opening |
| 6 | Notepad Scroll Sound |
| 7 | Trill |
| 8 | Highlight Sound |
| 9 | Xylo |
| 10 | Bell |
| 11 | Wakeup |
| 12 | Plunk (Trash) |
| 13 | Poof! |

**EXAMPLE**

```
5 REM BEEP Example
10 FOR i = 0 TO 13
20 BEEP i
30 WAIT 1
40 NEXT i
```

**OUTPUT**

```
(Your Newton makes each "beep" sound)
*
```

**RELATED ITEMS**

# BEGINSWITH Function

BEGINSWITH(*String1*, *String2*)

**DESCRIPTION**
BEGINSWITH returns TRUE if *String1* begins with *String2*. The comparison ignores the case of both strings.

**EXAMPLE**

```
5 REM BEGINSWITH Example
10 target = "YES or NO"
20 IF BEGINSWITH(target,"yes") THEN PRINT "It
starts with yes"
30 IF BEGINSWITH(target,"YES OR") THEN PRINT
"It starts with YES OR"
40 IF BEGINSWITH(target,"No OR") THEN PRINT
"It starts with No OR"
```

**OUTPUT**

```
It starts with yes
It starts with YES OR
*
```

**RELATED ITEMS**
STRCOMPARE, STREQUAL

# BYE

BYE [*val*]

### DESCRIPTION

BYE ends the current program, and then quits NS BASIC. It is valid within a program and as a Command entered using the on-screen keyboard. If *val* is supplied, the value of the expression is returned to the external caller of NS BASIC. See the Advanced Topics section for more information on calling NS BASIC from other Newton applications.

### EXAMPLE

```
10 REM BYE Example
20 PRINT "Quitting NS BASIC"
30 BYE
```

### OUTPUT

```
Quitting NS BASIC
*
```

The above appears momentarily. NS Basic then quits.

### RELATED ITEMS

STOP, END

CEILING(*x*)

**DESCRIPTION**

Returns the next integer greater than or equal to the real number *x*.

**EXAMPLE**

```
10 REM CEILING Example
20 PRINT "Please enter a number:"
30 INPUT Number
40 PRINT "Next largest integer is..." ;
CEILING(Number)
```

**OUTPUT**

```
Please enter a number:
12.31
"Next largest integer is...13
•
```

**RELATED ITEMS**

FLOOR

CHAIN *fileName*[,*lineNumber*]

**DESCRIPTION**

CHAIN causes a NS BASIC program to LOAD a program named by *fileName* from the default store and execute it.. *fileName* may be a string literal or a variable. *Linenumber* is the line from which NS BASIC is to start execution in the new program. The current values of all variables are preserved. You can use this form of CHAIN to break very large programs into sections that are CHAINed in when needed. If no *lineNumber* is given, a NEW is performed. Then NS BASIC starts execution from the beginning of the program. You can use this form of CHAIN to create menu-style programs, where unrelated programs are executed by a menu program using CHAIN.

**EXAMPLE**

```
10 REM Program1
20 PRINT "This is Program 1"
30 CHAIN "Program2"
40 PRINT "Return to Program1"
* SAVE Program1
Program1 saved
* NEW
10 REM Program2
20 PRINT "This is Program 2"
30 CHAIN "Program1",40
* SAVE Program2
Program2 saved
```

**OUTPUT**

```
* Load Program1
* RUN
This is Program 1
This is Program 2
Return to Program1
```

**RELATED ITEMS**

RUN

WINDOW *winNum*, *windowSpec*, "CHECKBOX"

WINDOW *winNum*, *windowSpec*, "RCHECKBOX"

**DESCRIPTION**

The CHECKBOX and RCHECKBOX widgets display a small square box that shows a check mark when selected. The check box can be toggled by the user by tapping on it or the label. CHECKBOX displays the label followed by the check box, and RCHECKBOX shows the check box followed by the label.

These widgets are controlled using the *windowSpec*. These fields are supported:

`viewValue`: TRUE to display a checkmark.

`text`: The label text

**Note:** you can update the state of the CHECKBOX and RCHECKBOX widgets using the following method:

```
U.windowSpec:TOGGLECHECK()
```

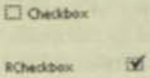
This expression causes the current setting of the checkbox to toggle. For instance, a checked box becomes unchecked.

You may also use these fields in *windowSpec*:
`viewBounds`, `viewFlags`, `viewFont`, `GOTO`, `GOSUB`, `viewFormat`.

**EXAMPLE**

```
10 REM CHECKBOX Example
20 w1Spec = {}
30 WINDOW w1, w1Spec, "CHECKBOX"
40 w2Spec = {viewValue: true}
50 WINDOW w2, w2Spec, "RCHECKBOX"
60 SHOW w1, w2
```

**OUTPUT**



**RELATED ITEMS**

SHOW, HIDE, WINDOW

CHR(*i*)

**DESCRIPTION**

CHR returns the ASCII character equivalent of *i*. You must use an integer with CHR. Refer to Appendix C of this Handbook for a list of useful character codes.

**EXAMPLE**

```
10 REM CHR Example
20 REM This demo asks the user for a number
and then displays the ASCII character
equivalent of it.
30 PRINT "Please enter a number between 1 and
256"
40 INPUT Number
50 PRINT "The Character Equivalent of " ;
Number ; " is " ; CHR(Number)
```

**OUTPUT**

```
Please enter a number between 1 and 256
99
The Character Equivalent of 99 is c
*
```

**RELATED ITEMS**

ORD

CLASSOF(x)

**DESCRIPTION**

CLASSOF returns the class of the variable *x* as a symbol. You can use CLASSOF to check the class of a variable that is INPUT by a user.

The symbols returned for each data type are:

| | |
|---|---|
| Integer | 'int |
| Real | 'real |
| Character | 'char |
| Boolean | 'boolean |
| String | 'string |
| Array | 'array |
| Frame | 'frame |
| Function | 'function |
| Symbol | 'symbol |

**EXAMPLE**

```
10 REM CLASSOF Example
20 a = 5
30 PRINT CLASSOF(a)
40 a = "Hello"
50 PRINT CLASSOF(a)
```

**OUTPUT**

```
Int
String
*
```

**RELATED ITEMS**

# CLOSE Statement

CLOSE[*chan*] | [*chanlist*]

**DESCRIPTION**
CLOSE releases the single file channel *chan* or the list of channels *chanlist* returned from an OPEN or CREATE statement. If *chan* is omitted, all open file channels are released. You cannot use a channel in a GET, PUT, or DEL statement after you CLOSE it.

**EXAMPLE**

```
10 REM CLOSE Example
20 CREATE chan, "EXAMPLEFile", keyname
30 CLOSE chan
```

**OUTPUT**

```
•
```

**RELATED ITEMS**
CREATE, OPEN

# CLOSEBOX Widget

WINDOW *winNum*, *windowSpec*, "CLOSEBOX"

WINDOW *winNum*, *windowSpec*, "LARGECLOSEBOX"

### DESCRIPTION

The CLOSEBOX and LARGECLOSEBOX widgets display the standard Newton close box (small and large sizes) in the lower right hand corner of the Newton screen.

These widgets are controlled using the *windowSpec*. These fields are supported:

`GOTO`: the line number to goto if tapped.

`viewBounds`: if not supplied, defaults to the lower right hand corner of the screen. Note that if this is specified, it the bounds are relative to the lower right hand corner, so you'll have to use negative numbers for all values to place it anywhere else on the screen.

### EXAMPLE

```
10 REM CLOSEBOX Example
20 w1Spec = {GOTO:100}
30 WINDOW w1, w1Spec, "CLOSEBOX"
40 SHOW w1
50 WAIT 1000
60 GOTO 50
100 REM tapped close box
110 HIDE
120 PRINT "Tapped."
```

### OUTPUT

### RELATED ITEMS

SHOW, HIDE, WINDOW

CLS

DESCRIPTION

CLS causes the contents of the NS BASIC screen to be erased. It will not clear any WINDOWs. You must use the HIDE command to remove them from the Newton's display.

EXAMPLE

```
10 REM Clear Screen Example
20 CLS
```

OUTPUT

```
*
(The Screen is cleared)
```

RELATED ITEMS

HIDE

# COMPOUND Function

COMPOUND(*rate, periods*)

**DESCRIPTION**

COMPOUND calculates the compound interest for a given *rate* over the specified number of *periods*.

**EXAMPLE**

```
10 REM COMPOUND Example. This example assumes
that interest is being calculated monthly.
20 PRINT "Please enter the Interest rate per
year:"
30 INPUT Rate
40 PRINT "Please enter the number of months
you wish interest to be calculated for:"
50 INPUT Period
60 PRINT "The percentage gain is " ;
COMPOUND((Rate*0.01/12), Period)* 100 ; "%"
```

**OUTPUT**

```
Please enter the Interest rate per year:
12
Please enter the number of months you wish
interest to be calculated for:
50
The percentage gain is 164.463182184388%
*
```

**RELATED ITEMS**

ANNUITY

CON

**DESCRIPTION**

CON continues the execution of a NS BASIC program that was halted by a STOP or END Statement. Execution resumes at the next Statement in the program.

If an error halted the program, the CON Statement will also continue execution at the next Statement after the one that caused the error.

**EXAMPLE**

```
10 REM CON Example
20 PRINT "Before Stop"
30 STOP
40 PRINT "After Stop"
```

**OUTPUT**

```
Before Stop
* CON
After Stop
*
```

**RELATED ITEMS**

END, ON ERROR GOTO, RUN, STOP

COS(*x*)

COSH(x)

ACOS(x)

ACOSH(x)

**DESCRIPTION**

COS returns the cosine of the angle *x* in radians.

COSH returns the hyperbolic cosine of the angle *x* in radians.

ACOS returns the arc cosine of the angle *x* in radians.

ACOSH returns the hyperbolic arc cosine of the angle *x* in radians.

**EXAMPLE**

```
10 REM COS Example
20 PRINT "Please enter an angle:"
30 INPUT Angle
40 PRINT "The Cosine of the angle is = " ;
COS(Angle) ; " radians"
```

**OUTPUT**

```
Please enter an angle:
63.7
The Cosine of the angle is = 0.646241795698775
radians
*
```

**RELATED ITEMS**

SIN, TAN

# CREATE Statement

CREATE *chan, fileName, key*

### DESCRIPTION

CREATE makes a new file. Files are stored on the Newton in a similar manner to other computers. The location of the file will be both in your Newton's internal memory and on the storage card (if any) currently installed in the Newton. Please refer to the Memory and Storage section of your Newton Handbook regarding controlling where new information is stored.

Files you create in NS BASIC remain on your Newton until you delete them using the DELETE Statement. You use files to store data that you would otherwise have to re-enter every time you reset your Newton.

A file consists of zero or more frames. Each frame in a file has a *key* field that is used for sorting and searching. This means you must have an entry named *key* in every frame you add to the file with the PUT Statement.

CREATE uses the string in *fileName* as the name of the file. *fileName* may be a string literal or a variable holding a string. It sets the variable *chan* to the number assigned to the file. You use *chan* in subsequent GET, PUT and DEL statements in your program, instead of the file name.

CREATE uses a variable named FSTAT to indicate that the file was either created or not created. FSTAT will be set to one of two values:

| | |
|---|---|
| 0 | File successfully created |
| 1 | File could not be created |

**Note:** You should avoid using a variable named FSTAT for your own purposes.

**EXAMPLE**

```
10 REM CREATE Example
20 REM Creates a file...prompts for some
information, stores then deletes it.
40 CREATE chan, "EXAMPLEFile", keyname
45 IF FSTAT=1 THEN STOP // CREATE error
50 PRINT "Please enter some key data..."
60 INPUT FileKey
70 fileRecord = {}
80 fileRecord.keyname = FileKey // key
90 PUT chan, fileRecord
100 IF FSTAT=1 THEN STOP // PUT error
110 PRINT "Data now in file is..."
120 GET chan,FetchedData,FileKey
130 IF FSTAT=1 THEN STOP // GET error
140 PRINT FetchedData
150 PRINT "Deleting Record From File"
160 DEL chan,FetchedData
```

**OUTPUT**

```
Please enter some data...
? Lemons and Llamas
Data now in file is...
{KEYNAME:"Lemons and Llamas",_uniqueID:0}
Deleting Record From File
*
```

**RELATED ITEMS**

GET, OPEN, PUT, DEL, DELETE

DATA *datalist*

DESCRIPTION

DATA Statements define the information used by the READ Statement. They are not executed by NS BASIC. You may place them anywhere in your program.

You may use any number of DATA Statements in a program. They are accessed in sequential order by the READ Statement. The *datalist* is a comma separated list of literal values. DATA Statements can contain only two types of literals: strings and numerics. String literals must be enclosed in quotation marks. Note that any special characters (\n, etc.) in strings are not evaluated.

EXAMPLE

```
10 REM DATA Example
20 DIM a[10]
30 DATA 4,5,6.5, "This", "Is"
40 DATA "String", "Data", -0.01
50 DATA "1\n2"
60 FOR i = 0 TO 7
70 READ a[i]
80 PRINT a[i]
90 NEXT i
100 READ aString
110 PRINT "1\\n2 the same? ";
STREQUAL(aString, "1\n2")
```

OUTPUT

```
4
5
6.5
This
Is
String
Data
-0.01
1\n2 the same? NIL
*
```

RELATED ITEMS

READ, RESTORE

DATENTIME(*Time*)

DESCRIPTION

DATENTIME returns a string containing the date and time as MM/DD/YY HH:MM. *Time* is the returned value from the TIME Function. If time is NIL, then the current time is used. The format of the date returned will depend on the locale of the Newton being used. Please refer to the Setting Preferences section of your Newton Handbook regarding changing the locale.

EXAMPLE

```
10 REM DATENTIME Example
20 CurTime = TIME()
30 PRINT DATENTIME(CurTime)
```

OUTPUT

```
02/23/94 12:45 pm
*
```

RELATED ITEMS

HOURMINUTE, TIME

WINDOW *winNum*, *windowSpec*, "DATEPICKER"

### DESCRIPTION

The DATEPICKER widget displays the standard Newton date picker. The date or dates can be selected as in the Dates application.

The widget is controlled using the *windowSpec*. These fields are supported:

`selectedDates`: An array of integers (from the TIME() function) representing the selected dates. The first date determines which month is displayed. If no value is supplied, the current month is displayed with the current day selected.

`noSelection`: TRUE if DATEPICKER is display-only

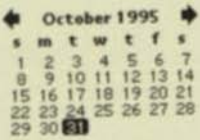
`singleDay`: TRUE if only a single day may be selected

You may also use these fields in *windowSpec*: `viewBounds`, `viewFlags`, `viewFont`, `GOTO`, `GOSUB`, `viewFormat`.

### EXAMPLE

```
10 REM DATEPICKER Example
20 w1Spec = {GOTO:100}
30 WINDOW w1, w1Spec, "DATEPICKER"
40 SHOW w1
50 WAIT 1000
60 GOTO 50
100 REM tapped close box
110 HIDE
120 PRINT DATENTIME(w1Spec.selectedDates[0])
```

### OUTPUT



```
◆ October 1995 ➡
 s  m  t  w  t  f  s
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

### RELATED ITEMS

HIDE, SHOW, TIME, MONTH, WINDOW

DEL *chan, recordFrame*

**DESCRIPTION**

DEL deletes a record specified by *recordFrame* from file *chan*. *chan* is the number of the file returned by the CREATE or OPEN Statements. *recordFrame* is a frame containing at least the key field. It may be a frame returned by the GET Statement

DEL uses a variable named FSTAT to indicate that the record was either deleted or not deleted. FSTAT will be set to one of two values:

| | |
|---|---|
| 0 | Record successfully deleted |
| 1 | Record could not be deleted |

**Note:** You should avoid using a variable named FSTAT for your own purposes.

**EXAMPLE**

```
10 REM DEL Example
20 REM Creates a file...prompts for some
information, stores then deletes it.
40 CREATE chan, "EXAMPLEFile", keyname
45 IF FSTAT=1 THEN STOP // CREATE error
50 PRINT "Please enter some key data..."
60 INPUT FileKey
70 fileRecord = {}
80 fileRecord.keyname = FileKey
90 PUT chan, fileRecord
100 IF FSTAT=1 THEN STOP // PUT error
110 PRINT "Data now in file is..."
120 GET chan,FetchedData,FileKey
130 IF FSTAT=1 THEN STOP // GET error
140 PRINT FetchedData
150 PRINT "Deleting Record From File"
160 DEL chan,FetchedData
```

**OUTPUT**

```
Please enter some data...
? Lemons and Llamas
Data now in file is...
{KEYNAME:"Lemons and Llamas",_uniqueID:0}
Deleting Record From File
*
```

**RELATED ITEMS**

CREATE, OPEN, PUT, GET, DELETE

# DELETE Statement

DELETE *fileName*

RM *fileName*

### DESCRIPTION

DELETE removes the program or file named by the string variable or string literal *fileName* from your Newton. The file is removed from both the internal memory and storage card. If *fileName* does not exist an I/O error will result.

To delete a NS BASIC program add the suffix ".bas" to the program name. To delete a text file from the default store use the suffix ".txt". To delete any files created using the CREATE Statement, enter only the file name (no suffix.) RM is a shorter name for DELETE.

### EXAMPLE

```
DELETE "Llamas.bas"
RM Testfile.txt
DELETE MyProg.bas
10 REM DELETE Example
20 CREATE chan, "Somefile", key
30 DELETE Somefile
```

### OUTPUT

```
*
```

### RELATED ITEMS

SAVE, REPLACE, DIR, LOAD, ENTER, LIST, CREATE, GET, STORE, PUT, OPEN, DEL

# DIGITALCLOCK Widget

WINDOW *winNum*, *windowSpec*, "DIGITALCLOCK"

### DESCRIPTION

The DIGITALCLOCK widget displays the standard Newton time picker. The time can be selected as in the Dates application.

The widget is controlled using the *windowSpec*. These fields are supported:

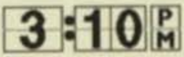
`time`: An integer (from the TIME() function) representing the selected time. The initial value of this field determines the initial display.

You may also use these fields in *windowSpec*: `viewBounds`, `viewFlags`, `viewFont`, `GOTO`, `GOSUB`.

### EXAMPLE

```
10 REM DIGITALCLOCK Example
20 w1Spec = {GOTO:100, time:
STRINGTOTIME("3:10PM")}
30 WINDOW w1, w1Spec, "DIGITALCLOCK"
40 SHOW w1
100 REM Time changed
110 PRINT TIMESTR(w1Spec.time, 0)
```

### OUTPUT



### RELATED ITEMS

HIDE, SHOW, STRINGTOTIME, TIME, WINDOW

DIM *variable* [*size* ]

DESCRIPTION

DIM sets the number of elements (*size*) for an array (*variable*). All arrays start with the element zero and can have an unlimited number of elements. You access the data in an array element using the expression *variable* [*elementNumber*]. Arrays can have elements of mixed type.

EXAMPLE

```
10 REM Array Example
20 DIM Names[3]
30 Names[0] = "Peter"
40 Names[1] = "Paul"
50 Names[2] = "Mary"
60 PRINT "Contents of the Names Array:"
70 FOR i = 0 TO 2
80 PRINT Names[i]
90 NEXT i
```

OUTPUT

```
Contents of the Names Array:
Peter
Paul
Mary
*
```

RELATED ITEMS

PRINT, LET

DIR

### DESCRIPTION

DIR outputs a sorted listing of the NS BASIC programs and text files currently saved in your Newton's internal memory or on the storage card currently installed in the Newton. Please refer to the Memory and Storage section of your Newton Handbook regarding controlling where new information is stored.

### EXAMPLE

```
DIR
```

### OUTPUT

```
Calculator              BASIC program
Calculator2             BASIC program
HelloWorld              Text File
LlamaCount              BASIC program
*
```

### RELATED ITEMS

SAVE, REPLACE, LOAD

# DIV Function

*x* DIV *y*

### DESCRIPTION

DIV returns the maximum number of times the integer *y* can divide into the integer *x*.

### EXAMPLE

```
10 REM DIV Example
20 REM This program takes two numbers and
computes number of times the 2 numbers can be
divided.
30 PRINT "Please enter two numbers."
40 INPUT Number1,Number2
50 Result = Number1 DIV Number2
60 PRINT "The number of times " ; Number1 ; "
can be divided by " ; Number2; " is " ; Result
```

### OUTPUT

```
Please enter two numbers.
? 7,5
The number of times 7 can be divided by 5 is 1.
*
```

### RELATED ITEMS

REMAINDER, MOD

# DO Statement

DO [WHILE *expression* |UNTIL *expression*]

**DESCRIPTION**

The DO statement begins a loop. The loop ends with a LOOP statement. You may test for the ending condition of the loop in the DO statement by using the WHILE*expression* or UNTIL *expression*.

DO WHILE *expression* will evaluate the Boolean *expression* each time before executing the loop. If *expression* is TRUE, then the loop is executed. If it is NIL, the statement following the LOOP statement is executed.

DO UNTIL *expression* will evaluate the Boolean *expression* each time before executing the loop. If *expression* is NIL, then the loop is executed. If it is TRUE, the statement following the LOOP statement is executed.

You can exit the loop by using the EXIT DO statement within the loop.

**EXAMPLE**

```
10 REM DO Example
20 i = 0
30 DO WHILE i < 10
40   i = i + 1
50   IF i > 5 THEN EXIT DO
60 LOOP
70 PRINT i
```

**OUTPUT**

```
6
*
```

**RELATED ITEMS**

LOOP, FOR, EXIT DO

# DRAW Widget

WINDOW *winNum*, *windowSpec*, "DRAW"

**DESCRIPTION**

The DRAW widget provides a user entry area that accepts ink drawing. The input may be recognized as shapes (this is the default) by setting *windowSpec*.`viewFlags` to `vVisible + vClickable + vGesturesallowed + vShapesallowed`, or just plain ink, by setting *windowSpec*.`viewFlags` to `vVisible + vClickable + vGesturesallowed + vStrokesallowed`

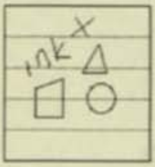
The widget is controlled using the *windowSpec*. These fields are supported:

`viewChildren`: n array of frames describing the drawing
`viewChildren[n].viewBounds`: the `viewBounds` of the nth shape drawn
`viewChildren[n].points`: the `points` of the nth shape drawn
You may also use these fields in *windowSpec*: `viewBounds`, `viewFlags`, `viewFormat`.

**EXAMPLE**

```
10 REM DRAW Example
20 w1Spec = {}
30 WINDOW w1, w1Spec, "DRAW"
40 SHOW w1
```

**OUTPUT**



**RELATED ITEMS**

POINTSTOARRAY, SHOW, HIDE, WINDOW

See the POINTSTOARRAY Reference section entry for an example of extracting the x,y coordinates of the shapes and strokes drawn in a DRAW widget.

# DRAWINTOBITMAP Function

DRAWINTOBITMAP(shape, options, bitmap)

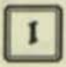
### DESCRIPTION

This function is used to create icons for the PICTUREBUTTON widget and the SETICON Statement. DRAWINTOBITMAP transfers the drawing in *shape* into *bitmap*. Use the MAKEBITMAP function to create *bitmap*. Use one or more of the MAKE functions (MAKELINE, MAKERECT, etc.) to create *shape*. The *options* parameter should be NIL.

### EXAMPLE

```
10 REM PICTUREBUTTON Example
20 shape=[MAKERECT(1,1,30,30), MAKETEXT("I",
12,10,21,21)]
30 myIcon:=MAKEBITMAP(32,32,NIL)
40 DRAWINTOBITMAP(shape, NIL, myIcon)
50 w1Spec = {icon: myIcon, GOTO: 200,
viewBounds: SETBOUNDS(101, 101, 132, 132)}
60 WINDOW w1, w1Spec, "PICTUREBUTTON"
70 SHOW w1
80 WAIT 1000
90 GOTO 80
200 HIDE
210 PRINT "Tapped."
```

### OUTPUT



### RELATED ITEMS

PICTUREBUTTON, MAKEBITMAP, SETICON, WINDOW

# EDIT Command

EDIT *lineNo*

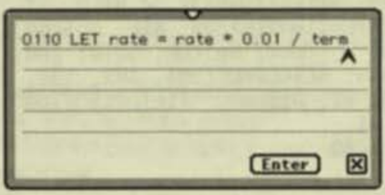
**DESCRIPTION**

EDIT loads the program Statement line specified by *lineNo* into the Edit Box. You may make any desired changes to the line and then update the program by tapping Enter. Tap the close box to discard the changes.

**EXAMPLE**

```
* EDIT 110
```

**OUTPUT**



```
0110 LET rate = rate * 0.01 / term
```

**RELATED ITEMS**

# ENTER Statement

ENTER *fileName*

### DESCRIPTION

ENTER loads a text file named by the string variable or string literal *fileName* from the default store, and attempts to enter each line in the program in exactly the same way lines are typed with the keyboard. A program may be merged with the currently LOADed program in this way. The text file may have been created by another application, sent over in serial mode, or by use of the LIST statement. If a line number from the text file matches a line number of a Statement already in memory, the line from the text file overwrites the one in memory. To enter a program without merging, type NEW before the ENTER Statement.

When you LOAD a program, NS BASIC does not re-interpret the Statements. ENTER can be used to re-interpret a program.

Previously saved NS BASIC programs have a ".bas" after *fileName*. If the file was created by the LIST Command it has a ".txt" after *fileName*.

### EXAMPLE

```
10 REM Simple Program
20 PRINT "Line 1"
SAVE "SimpleProgram"
NEW
10 REM Second Program
20 PRINT "Line 3"
30 PRINT "Line 4"
```

### OUTPUT

```
• ENTER SimpleProgram.bas
• LIST
10 REM Simple Program
20 PRINT "Line 1"
30 PRINT "Line 4"
•
```

### RELATED ITEMS

LIST, LOAD, SAVE

# ELEMENTS Function

ELEMENTS(*frame*)

### DESCRIPTION

ELEMENTS returns a sorted list of all element names in *frame*. It is useful for getting the names of the elements within a frame when they are not known.

The INTERN Function returns an internal reference that may be used in an expression. It may be used along with ELEMENTS to access the values stored in the elements within a frame.

Line 50 of the example below demonstrates the use of INTERN. This line shows that ELEMENTS returns an array of strings representing the names of the elements in a frame, while INTERN converts those strings into a form that is used to access the values stored in the elements of the frame.

### EXAMPLE

```
10 REM Elements Example
20 X={a: 1, d: 4, b: 2, c:3}
30 Y=elements(X)
40 FOR i=0 TO length(Y)-1
50 PRINT Y[i]; x.(INTERN(Y[i]))
60 NEXT I
```

### OUTPUT

```
a1
b2
c3
d4
*
```

### RELATED ITEMS

HASSLOT, INTERN

ELSE

DESCRIPTION
ELSE is used to separate statements to be executed when the expression in an IF THEN statement is TRUE from those to be executed when the expression is NIL.

EXAMPLE
```
10 REM Block IF Example
20 a = 5
30 b = 10
40 IF a=b THEN
50   PRINT a, b
60   PRINT "The numbers are equal."
70 ELSE
80   PRINT ABS(b-a)
90   PRINT "The numbers are this far apart"
100 END IF
```

OUTPUT
```
5
The numbers are this far apart
*
```

RELATED ITEMS
IF THEN ELSE, END IF

END [IF]

**DESCRIPTION**

END causes the program to stop executing without a beep. The program can be continued from the Statement after the END Statement by using the CON Command.

END IF marks the end of the current IF THEN block. END IF is paired with the nearest IF THEN statement proceeding it, when nested IF THEN blocks are used. There must be an IF THEN statement at a lower line number than the END IF statement. An ELSE statement may be used between an IF THEN statement and an END IF statement. See the example for the ELSE statement for an example of using IF THEN, ELSE, and END IF statements in a program.

**EXAMPLE**

```
10 REM END Example
20 PRINT "Line Number 1"
30 END
40 PRINT "Line Number 2"
```

**OUTPUT**

```
Line Number 1
* CON
Line Number 2
*
```

**RELATED ITEMS**

STOP, CON, BYE, IF, ELSE

# ENTER Statement

ENTER *fileName*

**DESCRIPTION**

ENTER loads a text file named by the string variable or string literal *fileName* from the default store, and attempts to enter each line in the program in exactly the same way lines are typed with the keyboard. A program may be merged with the currently LOADed program in this way. The text file may have been created by another application, sent over in serial mode, or by use of the LIST statement. If a line number from the text file matches a line number of a Statement already in memory, the line from the text file overwrites the one in memory. To enter a program without merging, type NEW before the ENTER Statement.

When you LOAD a program, NS BASIC does not re-interpret the Statements. ENTER can be used to re-interpret a program.

Previously saved NS BASIC programs have a ".bas" after *fileName*. If the file was created by the LIST Command it has a ".txt" after *fileName*.

**EXAMPLE**

```
10 REM Simple Program
20 PRINT "Line 1"
SAVE "SimpleProgram"
NEW
10 REM Second Program
20 PRINT "Line 3"
30 PRINT "Line 4"
```

**OUTPUT**

```
* ENTER SimpleProgram.bas
* LIST
10 REM Simple Program
20 PRINT "Line 1"
30 PRINT "Line 4"
*
```

**RELATED ITEMS**

LIST, LOAD, SAVE

# ENVIRON Statement

ENVIRON *variableName* = *value*ENV(*variableName*)

### DESCRIPTION

The ENVIRON Statement allows you to create environment variables that retain their value even after closing NS BASIC or resetting your Newton.

ENV(*variableName*) returns the value currently stored in environment variable *variableName*. The STATS Command also shows a complete list of all environment variables and their current values.

To remove an environment variable, set its value to NIL or leave the right hand side of the Statement empty. It will be removed the next time you close NS BASIC.

### SPECIAL NS BASIC ENVIRONMENT VARIABLES

You can control just how much information is printed for arrays and frames using the PRINTDEPTH environment variable. The default is 1, and valid values are 0 (no information is printed for arrays and frames) to any desired depth. When a variable that is an array or frame is used in a PRINT Statement the individual elements of the array or items of the frame will or will not be printed based on PRINTDEPTH.

You can read or write data to the serial port using NS BASIC. The location for input and output is controlled using the environment variable IO. If IO is set to "SCREEN" then INPUT and PRINT Statements refer to the Newton's screen. If IO is set to "S0" then INPUT and PRINT Statements refer to the Newton's serial port. If IO is set to "IR" then INPUT and PRINT Statements refer to the Newton's infrared port. If you issue an INPUT Statement to the serial or infrared port, the current inputPrompt is sent out the port. The input that is read must be terminated by a CR character. PRINT Statements that are output to the serial or infrared port are terminated by a CR character. Control characters can be sent as part of the output stream. For example:

```
PRINT CHR(27)
```

will send the ESC character.

Note that the Newton needs a few seconds to switch its output from/to the serial or infrared port.

The environment variable inputPrompt contains the character that is displayed by an INPUT Statement when it is prompting for input. This is especially useful for communications: setting inputPrompt to "" will make the INPUT Statement display no character prompt at all. The default value for inputPrompt is "? " and it is set to this when starting NS BASIC.

You may reset inputPrompt to its default setting using the following statement:

```
ENVIRON inputPrompt="? "
```

The environment variable s0 contains a frame that is used to control the characteristics of the serial port. Any changes you make to the elements of s0 are used the next time serial communications are established using the IO environment variable. Changes you make will not affect the current serial connection. This means you must set the s0 environment variable before you set IO to "S0".

The elements of the s0 frame are set to their default values each time NS BASIC is started.

Serial Port Settings

| | | |
|---|---|---|
| s0.connect | The message that is sent when establishing a connection. | "Connect."& CHR(10) & "*" |
| s0.unPend | The character that is used to terminate an input field.<br>**Note**: unPend must be a character. You may use special characters by using the CHR Function. | CHR(13) |
| s0.byteCount | The number of characters to accept before automatically unPending.<br>**Note**: Either s0.unPend or s0.byteCount must be NIL | NIL |
| s0.data | The data representation, in an array [bps,dataBits,stopBits,parity] | [9600, 8, 1, "no"] |
| s0.data[0] | The transmission speed (bps). Allowed values are 300, 600, 1200, 2400, 4800, 7200, 9600, 12000, 14400, 19200, 38400, 57600, 115200, 230400 | 9600 |
| s0.data[1] | The number of data bits. Allowed values are 5, 6, 7, 8. | 8 |
| s0.data[2] | The number of stop bits. Allowed values are 0, 1, 2. | 1 |
| s0.data[3] | The parity. Allowed values are "no", "even", "odd" | "no" |
| s0.inputform | The pre-processing style for input. Set to 'raw to receive data of class 'binary (such as Unicode.) Set to 'string for normal input. | 'string |

The environment variable IR contains a frame that is used to control the characteristics of the infrared port. Any changes you make to the elements of IR are used the next time infrared communications are established using the IO environment variable. Changes you make will not affect the current infrared connection. This means you must set the IR environment variable before you set IO to "IR".

The elements of the IR frame are set to their default values each time NS BASIC is started.

Infrared Port Settings

| | | |
|---|---|---|
| IR.connect | The message that is sent when establishing a connection. | "Connect."&<br>CHR(10) &<br>"*" |
| IR.unPend | The character that is used to terminate an input field.<br>**Note**: unPend must be a character. You may use special characters by using the CHR Function. | CHR(13) |
| IR.byteCount | The number of characters to accept before automatically unPending.<br>**Note**: Either IR.unPend or IR.byteCount must be NIL | NIL |
| IR.CharDelay | The number of Ticks to pause between the transmission of characters. | 0 |

**EXAMPLE: USING YOUR OWN ENVIRONMENT VARIABLES**

```
• ENVIRON pie=3.1415926
• PRINT ENV("pie")
• ENVIRON pie=
• STATS
```

**OUTPUT**

```
3.1415926
(STATS shows that there is currently a pie
environment variable with the value NIL)
```

### EXAMPLE: USING THE SERIAL PORT FOR OUTPUT

```
10 REM List contents of spreadsheet to the
serial port
12 ENVIRON io="s0"
15 OPEN ch,"qfig:donv"
20 GET ch,f1
30 GET ch,f2
50 x=ELEMENTS(f1.values)
60 FOR i=0 TO length(x)-1
70 y=INTERN(x[i])
80 PRINT x[i],f1.values.(y),
f2.data.(y).formula
100 NEXT i
110 ENVIRON io="screen"
```

### EXAMPLE: USING THE SERIAL PORT FOR INPUT

```
10 REM collect data from serial port
20 f ={viewBounds:SETBOUNDS(100,100,130,110)}
30 tr = 0
40 translist = ""
50 CLS
60 WINDOW w1,f
70 SHOW w1
80 ENVIRON io="s0"
90 REM get a transaction
100 INPUT trans
110 tr = tr+1
120 translist = translist & trans & CHR(13)
130 WPRINT w1, tr
140 IF trans <> "BYE" THEN GOTO 90
150 ENVIRON io="screen"
160 PRINT translist
170 HIDE w1
```

### RELATED ITEMS

# ERASE

ERASE *from,to*

**DESCRIPTION**

Deletes lines of the currently loaded program starting at *from* and ending at *to*. ERASE can not erase itself.

**EXAMPLE**

```
10 REM ERASE Example
20 ERASE 30, 40
30 PRINT "Line 30"
40 PRINT "Line 40"
50 PRINT "Line 50"
```

**OUTPUT**

```
Line 50
*
```

**RELATED ITEMS**

EXIT DO

EXIT FOR

### DESCRIPTION

EXIT leaves a loop at any point. If you have a specific condition that ends loop processing, you use EXIT to terminate the loop and begin execution at the statement followin the loop. You can use EXIT instead of a GOTO statement in this case.

EXIT DO causes the statement following the LOOP statement for the DO loop to be executed next, ending the DO loop.

EXIT FOR causes the statement following the NEXT statement for the FOR/NEXT loop to be executed next, ending the FOR/NEXT loop.

### EXAMPLE

```
10 i=0
20 DO
30   i=i+1
40   IF i>5 THEN EXIT DO
50 LOOP UNTIL i=10
60 PRINT i
```

### OUTPUT

```
6
*
```

### EXAMPLE

```
10 FOR i = 1 TO 10
20   IF i>5 THEN EXIT FOR
30 NEXT i
40 PRINT i
```

### OUTPUT

```
6
*
```

### RELATED ITEMS

FOR, NEXT, DO, LOOP, GOTO

EXP(*x*)

EXPM1(*x*)

**DESCRIPTION**

EXP returns the natural (base -e) exponential for the real number or integer *x*.

EXPM1 returns EXP(*x*)-1.

**EXAMPLE**

```
10 REM EXP Example
20 PRINT "Please enter a number"
30 INPUT Number
40 PRINT "The Natural exponential is " ;
EXP(Number)
```

**OUTPUT**

```
Please enter a number
? 7
The Natural exponential is 1,096.63315842846
*
```

**RELATED ITEMS**

# FLOOR Function

FLOOR(*x*)

**DESCRIPTION**

FLOOR returns the integer less than or equal to the real number *x*.

**EXAMPLE**

```
10 REM FLOOR Example
20 PRINT "Please enter a number"
30 INPUT Number
40 PRINT "Next Smallest integer is..." ;
FLOOR(Number)
```

**OUTPUT**

```
Please enter a number
? 12.31
"Next Smallest integer is..12
*
```

**RELATED ITEMS**

CEILING

# FOR Statement

FOR variable =expression1 TO expression2 [STEP expression3 ]

### DESCRIPTION

The FOR statement first sets *variable* to *expression1*. It starts counting up to *expression2* by adding *expression3* to the *variable* at the end of every cycle. If *expression3* is a negative number the counter will count down from *expression1* to *expression2* in *expression3* increments. If *expression3* is omitted NS BASIC assumes the default value of 1. *Expression3* cannot be zero.

A FOR Statement must have a corresponding NEXT Statement somewhere after it in the program in order to make the loop complete. FOR loops may be "nested" or placed within one another. Any number of FOR loops may be nested within each other.

The final value of *variable* is equal to the first number the loop reaches beyond *expression2*.

You can exit the loop by using the EXIT LOOP statement within the loop.

### EXAMPLE

```
10 REM FOR Loop Example
20 FOR i = 1 TO 10 STEP 3
30   FOR j = 1 to 2
40     PRINT i,j
50   NEXT j
60 NEXT i
```

### OUTPUT

```
1      1
1      2
4      1
4      2
7      1
7      2
10     1
10     2
*
```

### RELATED ITEMS

DO, NEXT, EXIT FOR

FUNCTION *functionName(args) expression*

DEF FN *functionName(args)* = *expression*

**DESCRIPTION**

FUNCTION and DEF FN define a user function. *FunctionName* is a valid NS BASIC variable name and *expression* is a valid NS BASIC expression or NewtonScript code. *args* are parameter variables that are used in *expression*. User functions retain their values in the same manner as any other variable. Use of functions can greatly speed up your code.

**Note:** To use NewtonScript in *expression*, you'll need a NewtonScript Manual. *Programming for the Newton*, by McKeehan and Rhodes and published by AP Professional is a good source of NewtonScript documentation.

Variables within your NS BASIC program are available within *expression*, even if they aren't passed in via *args*: preface them with "U.".

To call a user function, use:

U:*functionName(args)*

### EXAMPLE

```
10 REM FUNCTION Example
20 DEF FNS(starttime)=(TICKS()-starttime)/60
30 FUNCTION tot(b) BEGIN LOCAL x:=0; FOR i:=0
TO LENGTH(b)-1 DO x:=x+b[i]; x END
40 iterations=1000
50 a=ARRAY(iterations, 25)
60 GOSUB 90 //sum using NS BASIC loop
70 GOSUB 170 //sum using function
80 STOP
90 REM sum using NS BASIC loop
100 tm=TICKS()
110 x=0
120 FOR i=0 TO LENGTH(a)-1
130 x=x+a[i]
140 NEXT i
150 PRINT "Method 1:", U:fns(tm)
160 RETURN
170 REM sum using function
180 tm=TICKS()
190 x=U:tot(a)
200 PRINT "Method 2:", U:fns(tm)
210 RETURN
```

### OUTPUT

```
Method 1: 23.8833333333333
Method 2: 0.0833333333333333
Stop at 0080
•
```

### RELATED ITEMS

# GAUGE Widget

WINDOW *winNum*, *windowSpec*, "GAUGE"

### DESCRIPTION

The GAUGE widget provides a display of a relative value (i.e., the battery gauge). You can set the initial value of the GAUGE, and update the value within a program.

The widget is controlled using the *windowSpec*. These fields are supported:

`viewValue`: The current setting (0-100% filled)

You may also use these fields in *windowSpec*: `viewBounds`, `viewFlags`, `viewFormat`.

Whenever you change the viewValue of a GAUGE, you must use

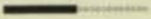
WPRINT *winNum*, ""

to update the display of the GAUGE.

### EXAMPLE

```
10 REM GAUGE Example
20 w1Spec = {viewValue:0}
30 WINDOW w1, w1Spec, "GAUGE"
40 SHOW w1
50 FOR i = 1 TO 100
60 w1Spec.viewValue = i
70 WPRINT w1, ""
80 NEXT i
```

### OUTPUT



(at the half-way point)

### RELATED ITEMS

SHOW, HIDE, WINDOW, WPRINT

# GET Statement

GET *chan, variable*[, *key*]

**DESCRIPTION**

GET retrieves information from file *chan*. *Chan* is a number returned from the OPEN or CREATE Statement. *Variable* is the variable in which the data retrieved from the file is placed. If a record is saved with a key, specifying *key* will get only that record. If *key* is not specified the next record will be retrieved. To use a key with the GET Statement a key must have been specified when OPEN was used for the *chan* as well.

GET uses a variable named FSTAT to indicate that the record was either read or not read. FSTAT will be set to one of three values:

| | |
|---|---|
| 0 | Record successfully read, *variable* set to read record |
| 1 | End of file reached, *variable* is set to NIL |
| 2 | *key* not found, *variable* is set to next closest record |

**Note:** You should avoid using a variable named FSTAT for your own purposes.

### EXAMPLE

```
10 REM GET Example
20 PRINT "The first 5 first names of the names
file will be displayed."
30 OPEN CH,"Names"
40 IF FSTAT <> 0 THEN STOP
50 FOR i = 1 TO 5
60 GET CH, NameData
70 IF FSTAT = 1 THEN STOP
80 PRINT NameData.Name.first
90 NEXT i
```

### OUTPUT

```
RUN
The first 5 first names of the names file will
be displayed.

John
Jane
Bob
Chris
Karen
*
(The names above will be the first 5 names of
the "Names" file on your Newton.)
```

### RELATED ITEMS

CREATE, OPEN, PUT, DEL

# GETGLOBALS Function

GETGLOBALS().*element*

**DESCRIPTION**

GETGLOBALS retrieves *element* from your Newton's global information area. The most common information that you will want to retrieve is in the element named `userConfiguration`. However, other data is also available. A list of some common fields is provided in the Accessing and Using Other Files, Data, and Applications section (4.5) of this Handbook. There are many other fields available for advanced users. Values can also be assigned to GETGLOBALS().*element*.

**Note:** Changing system values can have unexpected and undesirable consequences. Use great caution when changing system values.

**Warning:** Caution should be used when accessing and changing the `userConfiguration` element. The elements may vary for different Newton devices.

**EXAMPLE**

```
10 REM GETGLOBALS Example. Show User's name
and address
20 PRINT
GETGLOBALS().userConfiguration.company
30 PRINT
GETGLOBALS().userConfiguration.address
40 PRINT
GETGLOBALS().userConfiguration.cityzip
```

**OUTPUT**

```
NS BASIC Corporation
77 Hill Crescent
Toronto M1M 1J3
```

**RELATED ITEMS**

# GLANCE Widget

WINDOW *winNum, windowSpec,* "GLANCE"

**DESCRIPTION**

The GLANCE widget provides a display of a text message in a window for three seconds. This window is displayed when the SHOW Statement is executed for the widget. Once the window has shown and hidden itself, you must re-create it with another WINDOW Statement. In other words, you can never SHOW a GLANCE widget more than once.

The widget is controlled using the *windowSpec.* These fields are supported:

`text`: The message text

You may also use these fields in *windowSpec:* `viewBounds`, `viewFlags`, `viewFont`, `viewFormat`.

**EXAMPLE**

```
10 REM GLANCE Example
20 w1Spec = {text:"Read me quickly"}
30 WINDOW w1, w1Spec, "GLANCE"
40 SHOW w1
```

**OUTPUT**

(a window displaying Read me quickly is displayed and hidden)

**RELATED ITEMS**

SHOW, WINDOW

# GOSUB Statement

GOSUB *lineNumber*

### DESCRIPTION

GOSUB causes execution to branch to the line of code specified by *lineNumber*. A GOSUB must be paired with a RETURN Statement. When a RETURN Statement is found, execution continues from the line after the GOSUB. As with the GOTO Statement, if the *lineNumber* specified in the GOSUB Statement refers to a REM Statement, NS BASIC will also display that comment at the end of the GOSUB Statement as a line comment when the program is listed. The example shows this automatic commenting behavior of GOSUB.

### EXAMPLE

```
10 REM GOSUB Example
20 PRINT "GOSUB Routines-"
30 GOSUB 60 //Routine #2
40 PRINT "Routine #1"
50 END
60 REM Routine #2
70 PRINT "Routine #2"
80 RETURN
```

### OUTPUT

```
GOSUB Routines-
Routine #2
Routine #1
*
```

### RELATED ITEMS

REM, GOTO, LIST, RETURN

## GOTO Statement

GOTO *lineNumber*

**DESCRIPTION**

GOTO causes execution to branch to the line of code specified by *lineNumber*. As with the GOSUB Statement, if the *lineNumber* specified in the GOTO Statement refers to a REM Statement, NS BASIC will also display that comment at the end of the GOTO Statement as a line comment when the program is listed. The example shows this automatic commenting behavior of GOTO.

**EXAMPLE**

```
10 REM GOTO Example
20 PRINT "Please enter a number..."
30 INPUT x
40 IF x >100 THEN GOTO 80
50 PRINT "The number is too small"
60 PRINT "Please Re-enter..."
70 GOTO 30
80 END
```

**OUTPUT**

```
Please enter a number...
? 13
The number is too small
Please Re-enter...
? 137
•
```

**RELATED ITEMS**

REM, GOSUB, LIST

# HASSLOT Function

HASSLOT(*frame, slotName*)

**DESCRIPTION**

HASSLOT returns TRUE if the symbol in *slotName* is the name of a field in *frame*. Returns NIL otherwise.

**EXAMPLE**

```
10 REM HASSLOT Example
20 testFrame := {name: "Fred", fridge: TRUE}
30 IF hasslot(testFrame, 'name) THEN PRINT "It
has a name"
40 IF HASSLOT(testFrame, 'size) THEN PRINT "It
has a size"
50 IF HASSLOT(testFrame, 'fridge) THEN PRINT
"It has a fridge"
```

**OUTPUT**

```
It has a name
It has a fridge
*
```

**RELATED ITEMS**

ELEMENTS, REMOVESLOT

# HEXDUMP Function

HEXDUMP(*object, start, end*)

**DESCRIPTION**

HEXDUMP returns a string containing a hex dump of the string or binary *object*. The entire dump is created and placed in the return string, so you may run out of memory if you try and dump very large objects. You may use the SUBSTR() Function to dump only a portion of a string, or you may specify the *start* and *end* bytes to dump. If *start* and *end* are NIL the entire *object* is dumped.

HEXDUMP is useful for Serial and IR programming.

**EXAMPLE**

```
10 REM HEXDUMP Example
20 dumpString = "This is a String"
30 PRINT HEXDUMP(dumpString,0,20)
```

**OUTPUT**

```
0000: 00540068  00690073  00200069  00730020
.T.h.i.s. .i.s.
0016: 00610020                              .a.
•
```

**RELATED ITEMS**

SUBSTR

HIDE [*winNum*] | [*winNumlist*]

**DESCRIPTION**

HIDE removes the single window *winNum*, or the list of windows *winNumlist* from the screen. *winNum* and *winNumlist* are the numbers created by the WINDOW Statement. If HIDE is used with no arguments, all currently displayed windows are removed. Note that using HIDE without arguments means that you must re-create windows with the WINDOW Statement before showing them again.

**EXAMPLE**

```
10 REM HIDE Example
20 W1Spec = {ViewBounds: SETBOUNDS(10, 50,
100, 100)}
30 WINDOW Win1, W1Spec
40 WINDOW Win2, W2Spec
50 WPRINT Win1, "Window 1"
60 WPRINT Win2, "Window 2"
70 SHOW Win1
80 SHOW Win2
90 WAIT
100 HIDE Win2
110 SHOW Win2
120 HIDE
```

**OUTPUT**

```
(Two windows are created and then removed from
the screen.)
*
```

**RELATED ITEMS**

SHOW, WINDOW, WPRINT, CLS

HITSHAPE(*shape*, X, Y)

**DESCRIPTION**

HITSHAPE returns TRUE if the point described by X, *Y* falls within the supplied *shape*. Returns NIL if the point is outside the shape. You create *shape* using MAKELINE, MAKEOVAL, etc.

**EXAMPLE**

```
10 REM HITSHAPE Example
15 button = MAKEOVAL(10,10,40,40)
20 ws := {GOTO: 100, DRAWING: button}
30 WINDOW w1,ws
50 SHOW w1
70 WAIT 1000
80 GOTO 70
100 REM process user tap
110 IF HITSHAPE(button, ws.firstX, ws.firstY)
THEN PRINT "You tapped in the button!" ELSE
PRINT "You missed the button!"
120 HIDE
```

**OUTPUT**

```
(A window with an oval is displayed. Tap
inside the oval.)
You tapped in the button!
*
```

**RELATED ITEMS**

SHOW, WINDOW, MAKELINE, MAKEOVAL, etc.

HOURMINUTE(*Time*)

**DESCRIPTION**

HOURMINUTE returns a string giving the time as HH:MM. *Time* is the returned value from the TIME Function. To get the number of seconds, you must use the TICKS Function.

**EXAMPLE**

```
10 REM HOURMINUTE Example
20 CurTime = TIME()
30 PRINT HOURMINUTE(CurTime)
```

**OUTPUT**

```
12:45 pm
*
```

**RELATED ITEMS**

TIME, TICKS

# HWINPUT Statement

HWINPUT variable [,prompt [, popUpList]]

### DESCRIPTION

HWINPUT opens a box for hand written input. It places the result into *variable*. As with the INPUT Statement, if variable ends in a "$", the result is made into a string.

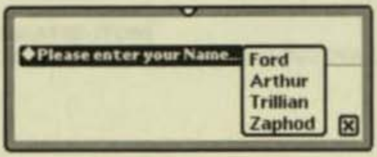
*Prompt* is an optional argument. The value of prompt is displayed in the user box. If *prompt* is not supplied, a simple box where the user may enter hand written input is displayed.

*PopUpList* is also an optional argument. It is only available if *prompt* is supplied. HWINPUT creates a pop-up list similar to the ones used in other applications on your Newton. The user may display the *popUpList* by tapping on *prompt* in the displayed box. *PopUpList* must be an array of strings. i.e. ["George", "Liz", "John"].

### EXAMPLE

```
10 REM HWINPUT Example
15 PopUp = ["Ford", "Arthur", "Trillian",
"Zaphod"]
20 HWINPUT Name,"Please enter your
Name...",PopUp
30 PRINT "Hello " ; Name
```

### OUTPUT



### RELATED ITEMS

INPUT, WINDOW, SHOW, HIDE, WPRINT

IF *expression* THEN [*statement1* [ ELSE *statement2* ]]

**DESCRIPTION**
The IF THEN ELSE Statement allows conditional execution of program code based on the evaluation of an expression. If the result of *expression* is TRUE then *statement1* is processed, otherwise *statement2* is executed.

When ELSE *statement2* is not supplied, the next statement in the program is executed if *expression* is NIL.

When no statements follow the THEN, this begins a block IF THEN ELSE END IF. You may place as many statements as you need between the IF THEN statement, and an optional ELSE statement. After the ELSE statement you may place multiple statements, followed by the END IF statement. Use this form if you need to execute more than one statement if *expression* is TRUE or NIL.

**EXAMPLE**

```
10 REM IF THEN ELSE Example
20 PRINT "Please Enter a Number."
30 INPUT Number
40 IF Number>=100 THEN PRINT "Number is
greater than or equal to 100" ELSE PRINT
"Number is less than 100"
50 IF Number=0 THEN PRINT "Number is equal to
zero"
```

**OUTPUT**

```
Please Enter a Number.
? 30
Number is less than 100.
* RUN
Please Enter a Number Between 1 & 100.
? 157
Number is greater than or equal to 100
*
```

**RELATED ITEMS**
ELSE, END IF

# INPUT Statement

INPUT *variable1* [ ,*variable2* ] ... [ ,*variableN* ]

**DESCRIPTION**

INPUT prompts the user for information. A question mark followed by a blinking insertion point is displayed. The information the user enters at the INPUT prompt is placed into *variable*. Multiple inputs to different variables may be assigned using a single INPUT Statement. The variable type is automatically assigned by NS BASIC to match the data entered by the user. If any of the variable names ends in a "**$**" then the type for that variable is string, and any data entered by the user will be converted to a string prior to storing it in the variable.

When the INPUT statement specifies a single string variable, then the user may enter commas, or an empty string (i.e., just press return) at the input prompt.

**EXAMPLE**

```
10 REM INPUT Example
20 PRINT "Please enter two things."
30 INPUT a,b
40 PRINT "Please enter one more thing."
50 INPUT c$
60 PRINT "You typed in...", a; " & "; b; " &
"; c$
```

**OUTPUT**

```
Please enter two things.
? 5 , Llamas
Please enter one more thing.
? 12.8, see the comma!
You typed in...    5 & Llamas & 12.8, see the
comma!
```

**RELATED ITEMS**

PRINT, LET

# INTERN Function

INTERN(*string*)

### DESCRIPTION

INTERN returns an internal reference to *string*. It is most commonly used to access elements within a frame through a variable. INTERN returns a symbol.

**Note:** The result must be placed within parenthesis when used in an expression that accesses a frame element.

### EXAMPLE

```
10 REM INTERN Example
20 frame:={a: 1, b:2, c:3}
30 frame_ele=INTERN("b")
40 PRINT frame.(frame_ele)
50 frame_names=ELEMENTS(frame)
60 FOR i=0 TO LENGTH(frame_names)-1
70 PRINT frame_names[i],
frame.(INTERN(frame_names[i]))
80 NEXT i
90 PRINT frame
```

### OUTPUT

```
2
a         1
b         2
c         3
```

### RELATED ITEMS

ELEMENTS

WINDOW *winNum*, *windowSpec*, "LABELINPUT"

### DESCRIPTION

The LABELINPUT widget provides a label with a text entry line. The widget may also contain a pick-list. If it does, then a small diamond is displayed in front of the label. Tapping the label displays the pick-list. Tapping an item in the list enters it into the text entry line.

The widget is controlled using the *windowSpec*. These fields are supported:

`entryFlags`: recognition flags for the entry field, as used in `viewFlags`.
`label`: The label text
`labelFont`: The label font
`text`: The initial entry field value
`entryLine.text`: The user entered or updated entry field value
`labelCommands`: The optional pick list (an array of strings)
`curLabelCommand`: The initial selection from the optional pick list
`viewValue`: The current selection from the optional pick list

**Note:** you can update the text displayed by the LABELINPUT widget using the following method:

```
SETVALUE(U.windowSpec.entryLine, 'text, "New Value")
```

This function changes the text displayed to `New Value`, and re-draws the widget. You can retrieve the value entered by the user using the following expression:

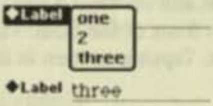
```
fieldText = windowSpec.entryLine.text
```

You may also use these fields in *windowSpec*: `viewBounds`, `viewFlags`, `viewFont`, `viewFormat`.

**EXAMPLE**

```
10 REM LABELINPUT Example
20 w1Spec = {labelCommands:["one", "2",
"three"]}
30 WINDOW w1, w1Spec, "LABELINPUT"
40 SHOW w1
```

**OUTPUT**



**RELATED ITEMS**

HIDE, SHOW, SETVALUE, WINDOW

# LABELPICKER Widget

WINDOW *winNum*, *windowSpec*, "LABELPICKER"

### DESCRIPTION

The LABELPICKER widget provides a label with a text display line. The widget also contains a pick-list. A small diamond is displayed in front of the label. Tapping the label displays the pick-list. Tapping an item in the list displays it next to the label.

The widget is controlled using the *windowSpec*. These fields are supported:

`text`: The label text

`labelCommands`: The pick list (an array of strings)

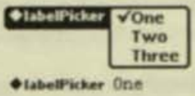
`viewValue`: The current selection from the pick list

You may also use these fields in *windowSpec*: `viewBounds`, `viewFlags`, `viewFont`, `GOTO`, `GOSUB`, `viewFormat`.

### EXAMPLE

```
10 REM LABELPICKER Example
20 w1Spec = {labelCommands:["one", "two",
"three"]}
30 WINDOW w1, w1Spec, "LABELPICKER"
40 SHOW w1
```

### OUTPUT



### RELATED ITEMS

HIDE, SHOW, WINDOW

# LENGTH Function

LENGTH(*x*)

**DESCRIPTION**

Length returns the number of elements in array *x*.

**Note**: use STRLEN for strings.

**EXAMPLE**

```
10 REM LENGTH Example
20 a = [1,2,"Three", 4]
30 PRINT "a has "; LENGTH(a); " elements."
```

**OUTPUT**

```
a has 4 elements.
*
```

**RELATED ITEMS**

STRLEN

[LET] *variable* = *expression* (Normal form)

[LET] *variable* := *expression* (Special use only)

**DESCRIPTION**

The LET statement evaluates *expression* and assigns it to *variable*. NS BASIC automatically adds the word "LET" in a program listing if you do not enter it.

The variable type (e.g., integer, real, string, etc.) is determined automatically by NS BASIC depending on the contents of *expression*. If *variable* has a "$" after it the type will always be a string.

The second form, using := as the assignment operator, assigns a reference to the right hand side instead of the value. This is useful for saving memory when accessing large objects, such as getGlobals().

**EXAMPLE**

```
10 REM LET Example
20 PRINT "What is your Name?"
30 INPUT Name$
40 PRINT "What is your age?"
50 INPUT age
60 LET age = age + 10
70 PRINT Name$; "...";"In 10 years your age
will be...";age
```

**OUTPUT**

```
What is your Name?
? John
What is your age?
? 21
John...In 10 years your age will be...31
```

**RELATED ITEMS**

# LIST Command

LIST [*startline* [,*endline*[ ,*fileName*]]]

**DESCRIPTION**

The LIST Command displays the currently LOADed program's source code. The user may specify *startline* and *endline* together or separately. If a single number follows LIST, only that one line will be displayed. If no starting or ending line is given the LIST Command displays the entire program. Only a single screen of code will be displayed by NS BASIC at a time. If there is more than one screen to be listed then

```
--More--
```

will be displayed at the end of each screen. Tap the return key to continue.

To save the listing to a file, place a comma and *fileName* after the first two parameters. The resulting file can either be used by other programs or can be exported to a desktop computer. NS BASIC adds ".txt" to the end of *fileName*. The file is created on the default store. These saved files can be utilized in other programs by using the ENTER Statement.

**EXAMPLE**

```
* LIST
```

**OUTPUT**

```
0010 REM Counting Program
0020 FOR i = 1 TO 10
0030   PRINT i
0040 NEXT i
0050 PRINT "All Done"
*
```

**EXAMPLE**

```
* LIST 20
```

**OUTPUT**

```
0020 FOR i = 1 TO 10
*
```

**EXAMPLE**

```
* LIST 30,
```

**OUTPUT**

```
0030 PRINT i
0040 NEXT i
0050 PRINT "All Done"
*
```

**EXAMPLE**

```
* LIST 20,30
```

**OUTPUT**

```
0020 FOR i = 1 TO 10
0030   PRINT i
*
```

**EXAMPLE**

```
* LIST ,30
```

**OUTPUT**

```
0010 REM Counting Program
0020 FOR i = 1 TO 10
0030   PRINT i
*
```

**EXAMPLE**

```
* LIST 10,50,"LISTProgram"
```

**OUTPUT**

```
*
```

**RELATED ITEMS**

ENTER

# LOAD Command

LOAD *fileName*

**DESCRIPTION**

LOAD recalls a SAVEd program named *fileName* to the active memory. If file *fileName* does not exist an I/O error will result.

**EXAMPLE**

```
LOAD "Llamas"
```

**OUTPUT**

```
*
```

**RELATED ITEMS**

DIR, SAVE

# LOG Function

LOG(*x*)

LOGB(*x*)

LOGIP(*x*)

LOG10(*x*)

LGAMMA(*x*)

### DESCRIPTION

LOG returns the Natural (base -e) logarithm of *x*.

LOGB returns the binary exponent of *x*.

LOG1P returns LOG(1+*x*).

LOG10 returns the base 10 log of *x*.

LGAMMA returns the base e log of the absolute value of the gamma of *x*.

### EXAMPLE

```
10 REM LOG Example
20 PRINT "Please enter a number"
30 INPUT Number
40 PRINT "The LOG of the number entered is ";
LOG(Number)
```

### OUTPUT

```
Please enter a number
100
The LOG of the number entered is
4.60517018598809
•
```

### RELATED ITEMS

LOOP [WHILE *expression* |UNTIL *expression*]

**DESCRIPTION**

The LOOP statement ends a loop. The loop begins with a DO statement. You may test for the ending condition of the loop in the LOOP statement by using the WHILE *expression* or UNTIL *expression*. You can only use WHILE or UNTIL in either the DO or the LOOP statement for a loop, but not both. When you use WHILE or UNTIL in the LOOP statement, the loop will always be executed at least once.

LOOP WHILE *expression* will evaluate the Boolean *expression* each time after executing the loop. If *expression* is TRUE, then the loop is executed again. If it is NIL, the statement following the LOOP statement is executed.

LOOP UNTIL *expression* will evaluate the Boolean *expression* each time after executing the loop. If *expression* is NIL, then the loop is executed again. If it is TRUE, the statement following the LOOP statement is executed.

You can exit the loop by using the EXIT DO statement within the loop. You can create an infinite loop by omitting WHILE and UNTIL in both the DO and LOOP statements of a loop. If you do, then you must use EXIT DO or a GOTO within the loop to exit it.

**EXAMPLE**

```
10 REM LOOP Example
20 i = 0
30 DO
40   i = i + 1
50   IF i > 5 THEN EXIT DO
60 LOOP WHILE i < 10
70 PRINT i
```

**OUTPUT**

```
6
*
```

**RELATED ITEMS**

DO, NEXT, EXIT DO

MAKEBITMAP(*width, height, options*)

MAKELINE(*x1, y1, x2, y2*)

MAKEOVAL(*left, top, right, bottom*)

MAKEPOLYGON(*arrayOfPoints*)

MAKERECT(*left, top, right, bottom*)

MAKEROUNDRECT(*left, top, right, bottom, diameter*)

MAKESHAPE(*points*)

MAKETEXT(*string, left, top, right, bottom*)

MAKEWEDGE(*left, top, right, bottom, startAngle, arcAngle*)

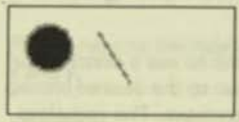
**DESCRIPTION**

The MAKE Functions create shapes that can be displayed in windows with the WDRAW Statement. They each use parameters to describe the desired shape. For MAKEBITMAP, the *width* and *height* in pixels of the blank bitmap are given. The *options* parameter should be NIL. For MAKELINE, the starting and ending X, Y coordinates are given. For MAKEOVAL and MAKERECT the coordinates of a bounding box are given. For MAKEROUNDREC, an additional parameter describes the diameter of the circle to use for the corners. MAKETEXT uses a bounding box and a string to specify the text. MAKEWEDGE uses a bounding box, the wedge angle and arc angle. MAKESHAPE is used with ARRAYTOPOINTS to create custom shapes.

**EXAMPLE**

```
10 REM WDRAW Example
20 W1Spec={viewBounds: SETBOUNDS(10, 10, 150,
75)}
30 WINDOW WinNum, W1Spec
40 SHOW WinNum
50 WDRAW WinNum, [MAKELINE(55,15,75,45),
MAKEOVAL(10,10,40,40)], {penSize:2,
penPattern:vfGray, fillPattern:vfBlack}
```

**OUTPUT**



**RELATED ITEMS**

ARRAYTOPOINTS, DRAWINTOBITMAP, SETICON, WDRAW, WINDOW

# MAKEPACKAGE Command

MAKEPACKAGE *programName*

**DESCRIPTION**

MAKEPACKAGE creates a stand-alone package in the Extras drawer. The name in the Extras drawer is the name *programName* was SAVEd as. All stand-alone packages use a default icon in the extras drawer.

You can use the SETICON Statement to use a custom icon for a stand-alone package. Set the icon to the desired bitmap before you create the stand-alone package. The complete name of the package is *programName*.pkg:NSBASIC. The name displayed in the Extras drawer is *programName*.

**EXAMPLE**

```
10 REM MAKEPACKAGE Example
20 PRINT "Enter starting principal"
30 INPUT principal
40 PRINT "Enter interest rate as % (i.e. 10)"
50 INPUT rate
60 PRINT "Enter term (i.e. 12 for monthly)"
70 INPUT term
80 PRINT "Enter number of years"
90 INPUT years
100 REM Compute final interest
110 rate = rate * 0.01 / term
120 PRINT "After ";years;" years the balance
is: "; compound(rate, years*term) * principal
130 END
* SAVE INVEST
INVEST saved.
* MAKEPACKAGE INVEST
```

**OUTPUT**

```
*
```

**RELATED ITEMS**

SETICON

MAX(x, y)

FMAX(x, y)

### DESCRIPTION

MAX returns the maximum value of the two integers *x* and *y*.

FMAX returns the maximum value of the two real numbers *x* and *y*.

### EXAMPLE

```
10 REM MAX Example
20 PRINT "Please enter a number"
30 INPUT Number1
40 PRINT "Please enter a second number"
50 INPUT Number2
60 PRINT "The largest number entered was " ;
MAX(Number1,Number2)
```

### OUTPUT

```
Please enter a number
? 12
Please enter a second number
? 108.727
The largest number entered was 108.727
*
```

### RELATED ITEMS

MIN

MIN(*x*, *y*)

FMIN(*x*, *y*)

### DESCRIPTION

MIN returns the minimum value of the two integers *x* and *y*.

FMIN returns the minimum of the two real numbers *x* and *y*.

### EXAMPLE

```
10 REM MIN Example
20 PRINT "Please enter a number"
30 INPUT Number1
40 PRINT "Please enter a second number"
50 INPUT Number2
60 PRINT "The smallest number entered was " ;
MIN(Number1,Number2)
```

### OUTPUT

```
Please enter a number
? 72.820
Please enter a second number
? 102
The smallest number entered was 72.820
*
```

### RELATED ITEMS

MAX

# MOD Function

*x* MOD *y*

*x* FMOD *y*

**DESCRIPTION**

MOD returns the modulus of the integers *x* and *y*.

FMOD returns the modulus of the reals *x* and *y*.

**Note:** MOD is not the same as REMAINDER.

**EXAMPLE**

```
10 REM MOD Example
20 REM This program takes two numbers and
computes their modulus.
30 PRINT "Please enter two numbers."
40 INPUT Number1,Number2
50 Result = Number1 MOD Number2
60 PRINT "The modulus of " ; Number1 ; " and
" ; Number2; " is " ; Result
```

**OUTPUT**

```
Please enter two numbers.
? 7,5
The modulus of 7 and 5 is 2.
*
```

**RELATED ITEMS**

REMAINDER, FMOD, DIV

WINDOW *winNum*, *windowSpec*, "MONTH"

### DESCRIPTION

The MONTH widget provides a display of a single month. The days of the month can be selected as in the Dates application.

The widget is controlled using the *windowSpec*. These fields are supported:

`selectedDates`: An array of integers (from the TIME() function) representing the selected dates. The first date determines which month is displayed. If no value is supplied, the current month is displayed

`noSelection`: TRUE if MONTH is display-only

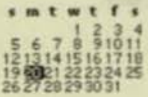
`singleDay`: TRUE if only a single day may be selected

You may also use these fields in *windowSpec*: `viewBounds`, `viewFlags`, `viewFont`, `GOTO`, `GOSUB`, `viewFormat`.

### EXAMPLE

```
10 REM MONTH Example
20 w1Spec = {}
30 WINDOW w1, w1Spec, "MONTH"
40 SHOW w1
```

### OUTPUT



```
 s  m  t  w  t  f  s
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

### RELATED ITEMS

HIDE, SHOW, TIME, DATEPICKER, WINDOW

NEW

**DESCRIPTION**

NEW clears the active memory of all program and variable information. This allows you to create a new NS BASIC program.

**EXAMPLE**

```
10 REM NEW Example
20 PRINT "Hello World!"
* NEW
* LIST
```

**OUTPUT**

```
*
```

**RELATED ITEMS**

# NEWSETCLOCK Widget

WINDOW *winNum, windowSpec*, "NEWSETCLOCK"

**DESCRIPTION**

The NEWSETCLOCK widget provides the standard Newton clock face for time display and entry. The clock face is drawn scaled to the supplied viewBounds. Whenever either clock hand is changed by the user, your GOTO or GOSUB routine will be called. You access the user's selection using:

```
hours = windowSpec.hours
minutes = windowSpec.minutes
```

The widget is controlled using the *windowSpec*. These fields are supported:

`hours`: current setting of the hour hand (or the current hour if not supplied)

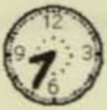
`minutes`: current setting of the minute hand (or the current minute if not supplied)

You may also use these fields in *windowSpec*: `viewBounds`, `viewFlags`, `GOTO`, `GOSUB`, `viewFormat`.

**EXAMPLE**

```
10 REM NEWSETCLOCK Example
20 w1Spec = {}
30 WINDOW w1, w1Spec, "NEWSETCLOCK"
40 SHOW w1
100 REM A selection was made
110 PRINT "Hours: "; w1Spec.hours; ", Minutes:
"; w1Spec.minutes
```

**OUTPUT**



**RELATED ITEMS**

HIDE, SHOW, SETCLOCK, WINDOW

# NEXT Statement

NEXT *variable*

**DESCRIPTION**

NEXT causes another iteration of the nearest preceding FOR Statement. The variable must match the variable used in the corresponding FOR Statement.

**EXAMPLE**

```
10 REM FOR/NEXT Example
20 FOR i = 1 TO 5
30 PRINT i
40 NEXT i
```

**OUTPUT**

```
1
2
3
4
5
*
```

**RELATED ITEMS**

FOR

NOTIFY(*header, message*)

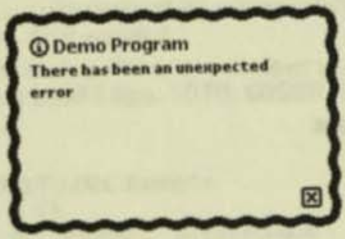
**DESCRIPTION**

NOTIFY displays a standard Newton notification box containing the header and message specified. Program execution continues after the notice is displayed. The function returns a frame. If the user closes the notification display, the `seenByUser` field of the frame is set to TRUE.

**EXAMPLE**

```
10 REM NOTIFY Example
20 NOTIFY("Demo Program","There has been an
unexpected error")
30 END
```

**OUTPUT**



ⓘ Demo Program

There has been an unexpected error

**RELATED ITEMS**

# NUMBERPICKER Widget

WINDOW *winNum*, *windowSpec*, "NUMBERPICKER"

**DESCRIPTION**

The NUMBERPICKER widget displays the standard Newton number picker. A number can be entered by tapping on the number display.

The widget is controlled using the *windowSpec*. These fields are supported:

value: An integer representing the selected number. The initial value of this field determines the initial display.

minValue: The minimum allowed value.

maxValue: The maximum allowed value. This number is used to determine how many digits to display. Seven digits are shown if maxValue is not specified.

showLeadingZeros: TRUE to display them, NIL to hide them.

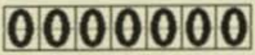
viewBounds: The width is calculated automatically based on maxValue. The left value is then calculated from the supplied right value. The height (bottom - top) should be 32.

You may also use these fields in *windowSpec*: viewFlags, GOTO, GOSUB.

**EXAMPLE**

```
10 REM NUMBERPICKER Example
20 w1Spec = {GOTO: 100, value: 0}
30 WINDOW w1, w1Spec, "NUMBERPICKER"
40 SHOW w1
100 REM value changed
110 PRINT "Value is: "; w1Spec.value
```

**OUTPUT**



**RELATED ITEMS**

HIDE, SHOW, WINDOW

NUMBERSTR(*number*)

**DESCRIPTION**

NUMBERSTR returns a string representation of *number*. *number* may be of any numerical type. You may manipulate the resulting string using the other string functions. Numbers in string format cannot be used in calculations or numeric expressions.

**EXAMPLE**

```
10 REM NUMBERSTR Example
20 Number = 127.924
30 PRINT "Number is " ; Number
40 PRINT "String representation is ";
NUMBERSTR(Number)
```

**OUTPUT**

```
Number is 127.924
String representation is 127.924
```

**RELATED ITEMS**

STRINGTONUMBER

ON ERROR GOTO *lineNumber*

**DESCRIPTION**

ON ERROR GOTO enables program error handling. Once error handling has been enabled, all errors detected cause NS BASIC to immediately GOTO *lineNumber*. If *lineNumber* does not exist execution stops and an error message is displayed. Program error handling may be disabled using

```
ON ERROR GOTO 0
```

This tells NS BASIC to perform standard error processing from now on. Execution stops and the error number is printed when there is an error.

**Note:** Division by Zero does not cause an error.

**EXAMPLE**

```
10 REM Error Checking Example
20 ON ERROR GOTO 60
30 x = 1+"2"
40 ON ERROR GOTO 0
50 END
60 PRINT "Error Routine"
```

**OUTPUT**

```
Error Routine
*
```

**RELATED ITEMS**

# ON GOTO/GOSUB Statement

ON *expression* GOTO *lineList*

ON *expression* GOSUB *lineList*

**DESCRIPTION**

ON GOTO performs a GOTO to one of the lines in *lineList*, depending on the value returned when *expression* is evaluated.

ON GOSUB performs a GOSUB in the same manner as ON GOTO.

*Expression* can be any numeric expression. It is evaluated and rounded to an integer, and is then used to select one line from *lineList*. *lineList* consists of a list of program line numbers separated by commas. The value of expression determines which of these lines the program will branch to. The value of *expression* is used as an index into *lineList*. The index of the first line number in *lineList* is one. If *expression* evaluates to more than the number of arguments in *lineList*, the line following the ON GOTO/GOSUB Statement is executed.

**EXAMPLE**

```
10 REM ON GOSUB/GOTO Example
20 PRINT "Please enter a value for
expression..."
30 INPUT Expression
40 ON Expression GOTO 50, 70, 90
50 PRINT "Routine #1"
60 END
70 PRINT "Routine #2"
80 END
90 PRINT "Routine #3"
```

**OUTPUT**

```
Please enter a value for expression...
? 2
Routine #2
* RUN
Please enter a value for expression...
? 1.4
Routine #1
* RUN
Please enter a value for expression...
? 0
Routine #1
*
```

**RELATED ITEMS**

GOSUB, RETURN, GOTO

OPEN *chan*, *fileName* [,*key* ]

**DESCRIPTION**

OPEN prepares file *fileName* for data storage, retrieval, and deletion. The channel number for the open file is assigned to *chan*. You must use *chan* to refer to the open file in GET, PUT, and DEL Statements.

*fileName* is a quoted string literal or string variable containing the name of the file to be opened either in your Newton's internal memory or on the storage card currently installed in the Newton. Please refer to the Memory and Storage section of your Newton Handbook regarding controlling where new information is stored.

*key* is the name of the field used for ordering and fast access. The file must have been created with the same *key* used by the CREATE Statement.

OPEN uses a variable named FSTAT to indicate that the file was either opened or not opened. FSTAT will be set to one of three values:

| | |
|---|---|
| 0 | *fileName* opened successfully |
| 1 | *fileName* not found |
| 2 | *fileName* found, but *key* not valid |

**Note**: You should avoid using a variable named FSTAT for your own purposes.

**EXAMPLE**

```
10 REM OPEN file Example
20 PRINT "A file will be opened for data
transfer."
30 OPEN CH,"Names"
40 IF FSTAT <> 0 THEN STOP
50 GET CH,FIRSTNAME
60 PRINT FIRSTNAME.Name.last
```

**OUTPUT**

```
RUN
John
*
```

(The name above will be the first surname of your "Names" record on your Newton)

**RELATED ITEMS**

CREATE, GET, PUT, DEL

ORD(*x*)

### DESCRIPTION

ORD returns the numeric representation of character *x*. You must supply a character for *x*.

### EXAMPLE

```
10 REM ORD Example
20 PRINT "Please enter a string"
30 INPUT X
60 PRINT "The ORD of the first character of X
is ";ORD(X[0])
```

### OUTPUT

```
Please enter a string
? ABC
The ORD of the first character of X is 65
*
```

### RELATED ITEMS

CHR

# PARAGRAPH

**Widget**

WINDOW *winNum*, *windowSpec*, "PARAGRAPH"

## DESCRIPTION

The PARAGRAPH widget provides a text display area that does not scroll. It is very similar to a WINDOW.

The widget is controlled using the *windowSpec*. These fields are supported:

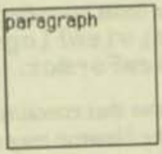
`text`: The text displayed

You may also use these fields in *windowSpec*: `viewBounds`, `viewFlags`, `viewFont`, `viewFormat`.

## EXAMPLE

```
10 REM PARAGRAPH Example
20 w1Spec = {}
30 WINDOW w1, w1Spec, "PARAGRAPH"
40 SHOW w1
```

## OUTPUT



```
paragraph
```

## RELATED ITEMS

HIDE, SCROLLER, SHOW, TEXT, WINDOW

WINDOW *winNum*, *windowSpec*, "PICKER"

**DESCRIPTION**

The PICKER widget provides a pop-up list of choices the user may select from. Once a selection is made, the widget is hidden.

The widget is controlled using the *windowSpec*. These fields are supported:

`pickItems`: The pick list, an array of strings, the symbol `'PICKSEPARATOR` to draw a line, and frames. Frames are in the form:

```
item: "item to display",
pickable: TRUE, // or NIL if not pickable
mark: CHR(8730) // the checkmark to display
```

`viewValue`: The current selection (as a number from 0 to LENGTH(pickItems-1) from the pick list

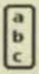
You may also use these fields in *windowSpec*: `Bound` (used like `viewBounds`), `viewFlags`, `viewFont`, `GOTO`, `GOSUB`, `viewFormat`.

**Note:** `Bound` is a frame that contains the same four fields as `viewBounds`. The Newton may actually move your pop-up to a different location on the screen if it would not fit in the location specified by `Bound`. The pickItems array can contain up to 22 items for a MessagePad sized screen. More than this will not fit.

**EXAMPLE**

```
10 REM PICKER Example
20 w1Spec = {GOTO: 200, pickItems:
["a","b","c"]}
30 WINDOW w1, w1Spec, "PICKER"
40 SHOW w1
50 WAIT 1000
60 END
200 PRINT "You picked item: ";
w1Spec.viewValue
```

**OUTPUT**



```
You picked item: 2
```

**RELATED ITEMS**

HIDE, SHOW, WINDOW

# PICTUREBUTTON

WINDOW *winNum*, *windowSpec*, "PICTUREBUTTON"

### DESCRIPTION

The NUMBERPICKER widget displays the standard Newton number picker. A number can be entered by tapping on the number display.

The widget is controlled using the *windowSpec*. These fields are supported:

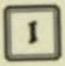
`icon`: the icon to display.

You may also use these fields in *windowSpec*:
`viewBounds`, `viewFlags`, GOTO, GOSUB.

### EXAMPLE

```
10 REM PICTUREBUTTON Example
20 shape=[MAKERECT(1,1,30,30),
MAKETEXT("I",12,10,21,21)]
30 myIcon:=MAKEBITMAP(32,32,NIL)
40 DRAWINTOBITMAP(shape, NIL, myIcon)
50 w1Spec = {icon: myIcon, GOTO: 200,
viewBounds: SETBOUNDS(101, 101, 132, 132)}
60 WINDOW w1, w1Spec, "PICTUREBUTTON"
70 SHOW w1
80 WAIT 1000
90 GOTO 80
200 HIDE
210 PRINT "Tapped."
```

### OUTPUT



### RELATED ITEMS

HIDE, SHOW, MAKEBITMAP, DRAWINTOBITMAP, WINDOW

# POINTSTOARRAY Function

POINTSTOARRAY(*points*)

### DESCRIPTION

POINTSTOARRAY returns a *shapeArray*. The format of *points* is described in the reference page for the DRAW widget, and shapeArray is described in the ARRAYTOPOINTS Function.

### EXAMPLE

```
10 REM POINTSTOARRAY Example
20 dSpec := {viewBounds: SETBOUNDS(1, 1, 239,
318), viewFlags: VSHAPESALLOWED + VCLICKABLE
+ VGESTURESALLOWED}
30 WINDOW drawWin, dSpec, "DRAW"
40 spec := {GOTO: 90}
50 WINDOW quitWin, spec, "LARGECLOSEBOX"
60 SHOW drawWin, quitWin
70 WAIT 1000
80 GOTO 0070
90 REM User Tapped Close Box
100 IF LENGTH(dSpec.windowSpec.viewChildren)
< 1 THEN GOTO 0120
110 PRINT "First Drawing: ";
POINTSTOARRAY(drawSpec.windowSpec.viewChildr
en[0].points)
120 HIDE
130 END
```

### OUTPUT

```
* run
(Draw a square and tap the Close box)
First Drawing: [11,5,0,0,0,45,60,45,60,0,0,0]
```

### RELATED ITEMS

ARRAYTOPOINTS, DRAW, WINDOWS

POW(*x*,*y*)

DESCRIPTION

POW returns the value of *x* raised to the power of *y*. *x* and *y* may be integer or real numbers.

EXAMPLE

```
10 REM POW Example
20 PRINT "Please enter a number"
30 INPUT X
40 PRINT "Please enter power to raise to"
50 INPUT Y
60 PRINT "X to the power Y is ";POW(X,Y)
```

OUTPUT

```
Please enter a number
? 16
Please enter power to raise to
? 2
X to the power Y is 256
*
```

RELATED ITEMS

SQRT

# PRINT Statement

PRINT [*expression1* [ ,*expression2* ]]

; [*expression1* [ ,*expression2* ]]

**DESCRIPTION**

PRINT evaluates each expression and outputs it to the screen. Variables, strings, and numerical expressions can all be output by NS BASIC using the PRINT Statement. If the PRINT Statement is used on its own, a blank line is output. PRINT is automatically substituted by NS BASIC when a semicolon is used as the first character in a line. A comma between arguments moves the output to the next tab. Tabs are 10 spaces apart. A semicolon between the expressions leaves no spaces.

When a comma or a semicolon is placed at the end of a PRINT Statement, the output from the next PRINT Statement will continue on the same line.

If the printed expression is longer than the screen width, it will wrap around to the next line.

**EXAMPLE**

```
10 REM PRINT Example
20 PRINT "The PRINT Command"
30 PRINT
40 ; "Can be used to separate", "text"
50 ; "Or Join Numbers and Text"
60 PRINT 10*10; " Llamas"
```

**OUTPUT**

```
The PRINT Command

Can be used to separate       text
Or Join Numbers and Text
100 Llamas
```

**RELATED ITEMS**

PUT *chan*, *variable*

**DESCRIPTION**
PUT writes data to a file. The file is specified by *chan*. *chan* is the number returned from the OPEN or CREATE Statements. *Variable* is a frame to be written.

If you wish to update a record in a file, use GET to retrieve the frame. Update the elements as needed, but do not change the key element. Use PUT to replace the updated frame.

If GET was not used to retrieve the frame, or if you change the key element of the frame, a new record is created.

The key specified on OPEN must be an element in *variable*. The key must be a string.

PUT uses a variable named FSTAT to indicate that the record was either written or not written. FSTAT will be set to one of two values:

| | |
|---|---|
| 0 | *variable* written successfully |
| 1 | *variable* not written |

**Note**: You should avoid using a variable named FSTAT for your own purposes.

**EXAMPLE(S):**

```
10 REM PUT Example
20 REM Creates a file...prompts for some
information, stores then deletes it.
40 CREATE chan, "EXAMPLEFile", keyname
45 IF FSTAT=1 THEN STOP // CREATE error
50 PRINT "Please enter some key data..."
60 INPUT FileKey
70 fileRecord = {}
80 fileRecord.keyname = FileKey
90 PUT chan, fileRecord
100 IF FSTAT=1 THEN STOP // PUT error
110 PRINT "Data now in file is..."
120 GET chan,FetchedData,FileKey
130 IF FSTAT=1 THEN STOP // GET error
140 PRINT FetchedData
150 PRINT "Deleting Record From File"
160 DEL chan,FetchedData
```

**OUTPUT**

```
Please enter some data...
? Lemons and Llamas
Data now in file is...
{KEYNAME:"Lemons and Llamas",_uniqueID:0}
Deleting Record From File
*
```

**RELATED ITEMS**

CREATE, OPEN, GET, DEL

# RANDOM Function

RANDOM (*low, high*)

### DESCRIPTION

RANDOM returns a random number between *low* and *high*.

### EXAMPLE

```
10 REM RANDOM Example
20 REM Displays 10 random numbers between 5
and 15
30 FOR i = 1 to 10
40 PRINT RANDOM(5,15)
50 NEXT i
```

### OUTPUT

```
6
8
13
7
9
9
8
12
14
6
*
```

### RELATED ITEMS

RANDOMIZE

# RANDOMIZE Statement

RANDOMIZE [*seed*]

### DESCRIPTION

RANDOMIZE seeds the random number generator with *seed*. When seeded with the same number, the RANDOM function will return the same sequence of numbers. To generate virtually random numbers do not enter *seed*. The default setting for *seed* is the number of ticks since system startup.

### EXAMPLE

```
10 REM RANDOMIZE Example
20 RANDOMIZE 34
30 FOR i = 1 to 10
40 PRINT RANDOM(1,10)
50 NEXT i
```

### OUTPUT

```
9
9
8
8
5
10
10
2
2
```

### RELATED ITEMS

RANDOM

READ *variable1* [,*variable2*]...[,*variableN*]

**DESCRIPTION**

READ reads the next value or values from the DATA Statement.

A READ Statement must always be used together with one or more DATA Statements. READ assigns DATA Statement values to variables.

A single READ Statement may access one or more DATA Statements, or several READ Statements may access the same DATA Statement.

If the number of variables in the variable list (*variable1 ... variableN*) exceed the number of elements in the DATA Statements an "End of Data" error results. If the number of variables specified is fewer than the number of elements in the DATA Statement(s), the next READ Statement will begin reading data at the next unread element. If there are no following READ statements, the extra data is ignored. To reset the list of DATA items, use the RESTORE Statement.

### EXAMPLE

```
10 REM READ Example
20 DATA 0.76,3.55,7.80,2.65,9.52
25 DATA 9.96,6.32,8.15,6.61,9.73
30 FOR i = 1 TO 10
40 READ a
50 PRINT a
60 NEXT i
```

### OUTPUT

```
0.76
3.55
7.80
2.65
9.52
9.96
6.32
8.15
6.61
9.73
*
```

### RELATED ITEMS

DATA, RESTORE

REM *remark*

**DESCRIPTION**

REM Statements are used to insert comments into a program. They are not processed when a program is executed. If a REM Statement is encountered while a program is running NS BASIC skips the line and continues with the execution of the program.

Comments may also be added to the end of any Statement (except GOTO and GOSUB) by preceding them with the characters "//".

When a REM Statement is the target line for a GOSUB or GOTO Statement, NS BASIC places the remark after a double backslash at the end of the GOSUB or GOTO Statement.

**EXAMPLE**

```
10 REM REM Example 1
15 A=1 // Set A to 1
20 PRINT "This line is printed"
30 REM But this line is not printed
40 REM Neither is this one
```

**OUTPUT**

```
This line is printed
*
```

**EXAMPLE**

```
10 REM REM Example 2
20 REM It shows how the REM Statement is used
with
30 REM GOSUB and GOTO Routines.
40 GOSUB 70
50 PRINT "Return from GOSUB"
60 END
70 REM Notice the Backslashes
80 PRINT "Here I Am!"
90 RETURN
```

**OUTPUT**

```
* LIST
0010 REM REM Example 3
0020 REM It shows how the REM Statement is used
with
0030 REM GOSUB and GOTO Routines.
0040 GOSUB 0070 //Notice the Backslashes
0050 PRINT "Return from GOSUB"
0060 END
0070 REM Notice the Backslashes
0080 PRINT "Here I Am!"
0090 RETURN
*
```

**RELATED ITEMS**

GOSUB, GOTO

# REMAINDER Function

REMAINDER(x,y)

**DESCRIPTION**

REMAINDER returns the remainder of x divided by y.

The result my be surprising: REMAINDER(12,7) is -2 (12 is 2 short of 14, a number that is evenly divisible by 7.) The MOD function will return the modulo of two numbers. MOD(12,7) is 5.

**EXAMPLE**

```
10 REM REMAINDER Example
20 REM This program takes two numbers and
computes the remainders of their division.
30 PRINT "Please enter two numbers."
40 INPUT Number1,Number2
50 PRINT "The Remainder of " ; Number1 ; "
divided by " ; Number2; " is " ;
REMAINDER(Number1, Number2)
```

**OUTPUT**

```
Please enter two numbers.
? 7,5
The Remainder of 7 divided by 5 is 2.
•
```

**RELATED ITEMS**

MOD, FMOD, DIV

REMOVESLOT(*frame, slotName*)

**DESCRIPTION**

REMOVESLOT deletes the field specified by the symbol in *slotName*. Returns NIL.

**EXAMPLE**

```
10 REM REMOVESLOT Example
20 aFrame = {name: "Fred", fridge: NIL}
30 REMOVEslot(aFrame, 'fridge)
40 PRINT aFrame
```

**OUTPUT**

```
{name: "Fred"}
*
```

**RELATED ITEMS**

ELEMENTS, HASSLOT

# RENUM Command

RENUM [ *startline* [ *,endline* [ *,increment* [ *,base* ]]]]

### DESCRIPTION

RENUM renumbers the lines of the currently LOADed program. *startline* and *endline* mark the range of line numbers in the program to be renumbered. *increment* is the numbering difference to use between each line. *base* is the first line number to use.

If a line already exists where a renumbered line is supposed to be placed, error 8 – Renum overlap is signaled and the program is left unchanged.

If *base* is not specified NS BASIC starts numbering from line 10. RENUM will also correct references in GOTO and GOSUB Statements which change as a result of the RENUMbering.

### EXAMPLE

```
10 REM RENUM Program
20 PRINT "This is line 0020"
30 PRINT "This is line 0030"
40 PRINT "This is line 0040"
50 PRINT "This is line 0050"
```

### OUTPUT

```
* RENUM 20,40,20,60
0010 REM RENUM Program
0050 PRINT "This is line 0050"
0060 PRINT "This is line 0020"
0080 PRINT "This is line 0030"
0100 PRINT "This is line 0040"
*
```

### RELATED ITEMS

REPLACE *fileName*

**DESCRIPTION**

REPLACE overwrites a previously SAVEd program. Quotation marks are required for *fileName*. If there is no file named *fileName*, REPLACE simply creates a new file. If there is a file named *fileName* REPLACE overwrites the file with the program in active memory.

REPLACE with no *fileName* is not valid if the current program has not been SAVEd yet.

**EXAMPLE**

```
* REPLACE "Llamas"
```

**OUTPUT**

```
Llamas saved
*
```

**RELATED ITEMS**

SAVE, LOAD, DELETE, RM, DIR

# RESTORE Statement

RESTORE [*lineNumber*]

**DESCRIPTION**

RESTORE allows DATA Statements to be re-read from line *lineNumber*.

When a RESTORE Statement is executed with *lineNumber*, the next READ Statement will access the first element in the specified DATA Statement. When *lineNumber* is not given, the next READ Statement will access the first element of the first DATA Statement.

**EXAMPLE**

```
110 REM RESTORE Example
20 DATA 0.76,3.55,7.80,2.65,9.52
25 DATA 9.96,6.32,8.15,6.61,9.73
30 FOR i = 1 TO 4
40 READ a
50 PRINT a
60 NEXT i
70 RESTORE 20
80 FOR j = 1 TO 4
90 READ b
100 PRINT b
110 NEXT j
```

**OUTPUT**

```
0.76
3.55
7.80
2.65
0.76
3.55
7.80
2.65
```

**RELATED ITEMS**

DATA, READ

RETURN

**DESCRIPTION**

RETURN causes NS BASIC to return from a previous GOSUB Statement.

A GOSUB causes NS BASIC to branch to a subroutine. RETURN makes NS BASIC return from a GOSUB. Program execution begins again at the line following the original GOSUB.

**EXAMPLE**

```
10 REM RETURN Example
20 PRINT "Beginning of Program"
30 GOSUB 0060 // Subroutine # 1
40 PRINT "End of Program"
50 END
60 REM Subroutine #1
70 PRINT "Here I am!"
80 RETURN
```

**OUTPUT**

```
Beginning of Program
Here I am!
End of Program
*
```

**RELATED ITEMS**

GOSUB, REM

# REVUP

### DESCRIPTION

REVUP converts all the programs on the default store of your Newton to the current revision of NS BASIC. Enter REVUP by itself after you install a new version of NS BASIC. This command can take a while to complete, depending on how many and how long your programs are. You can use REVUP in version 2.04 or later of NS BASIC, so if you'd like to install one of the older versions just enter REVUP again after installing the application.

### EXAMPLE

```
* REVUP
```

### OUTPUT

```
*
```

### RELATED ITEMS

# ROUND Function

ROUND(*x*)

**DESCRIPTION**

ROUND returns a real number that contains the rounded integral value. *X* is rounded upwards if it is greater or equal to 0.5, otherwise it is rounded downward.

**EXAMPLE**

```
10 REM ROUND Example
20 REM ROUNDS three numbers and adds them
together.
30 PRINT "Please enter three numbers"
40 INPUT Number1,Number2,Number3
50 Total = ROUND(Number1) + ROUND(Number2) +
ROUND(Number3)
60 PRINT "The Total is = " ; Total
```

**OUTPUT**

```
Please enter three numbers
? 12,17.32,1.997
The Total is = 31
*
```

**RELATED ITEMS**

RUN [ *fileName* | *lineNumber* ]

**DESCRIPTION**

RUN begins execution of a program.

If RUN is entered without arguments, NS BASIC executes the entire program in active memory. If you provide *fileName*, a NEW is performed and *fileName* is then LOADed and executed.

If you provide *lineNumber*, NS BASIC starts execution of the current program at *lineNumber*. Variables are not reset in programs that are executed from a line.

**EXAMPLE**

```
10 REM Run Example
20 INPUT a
30 PRINT a
* RUN
```

**OUTPUT**

```
? Llamas
Llamas
* RUN 30
Llamas
*
```

**RELATED ITEMS**

CON

# SAVE Command

SAVE *fileName*

**DESCRIPTION**

Save writes the active program to the internal memory or storage card. You may include quotation marks around *fileName*. NS BASIC automatically adds ".bas" to the end of *fileName*. If *fileName* already exists an I/O error will result. To replace an existing program, use the REPLACE Command.

**EXAMPLE**

```
* SAVE "Llamas"
```

**OUTPUT**

```
Llamas saved
*
```

**RELATED ITEMS**

DIR, ENTER, LOAD, REPLACE

WINDOW *winNum*, *windowSpec*, "SCROLLER"

**DESCRIPTION**

The SCROLLER widget provides a text entry area that scrolls. When the user wishes to enter new text, they tap on the mountain icon. The widget will expand to fill the entire Newton screen, and the user can enter text. Tapping the mountain icon again shrinks the widget back to its original size. The scroll arrows scroll the widget in either view. You extract the text entered by the user with this expression:

```
enteredText = windowSpec.notes.text
```

The widget is controlled using the *windowSpec*. These fields are supported:

`text`: The initial value

`notes.text`: The user entered or updated value

`boxTitle`: The title on the edit box

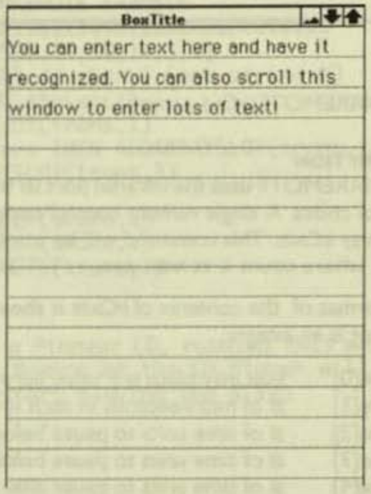
`editOK`: TRUE if the user can edit the text

You may also use these fields in *windowSpec*: `viewBounds`, `viewFlags`.

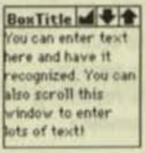
**EXAMPLE**

```
10 REM SCROLLER Example
20 w1Spec = {text: "You can..."}
30 WINDOW w1, w1Spec, "SCROLLER"
40 SHOW w1
```

## OUTPUT



You can enter text here and have it recognized. You can also scroll this window to enter lots of text!



You can enter text here and have it recognized. You can also scroll this window to enter lots of text!

## RELATED ITEMS

HIDE, PARAGRAPH, SHOW, WINDOW

# SENDIRREMOTE Function

SENDIRREMOTE(*irCode*, *count*)

**DESCRIPTION**

SENDIRREMOTE uses the infrared port to transmit remote control codes. A single remote control code is encoded in the array *irCode*. This command will be transmitted *count* times, where count is at least one.

The format of  the contents of *irCode* is shown below. Each element is an integer.

| | |
|---|---|
| irCode[0] | you may place any value here |
| irCode[1] | # of microseconds in each time unit |
| irCode[2] | # of time units to pause before sending |
| irCode[3] | # of time units to pause before repeating |
| irCode[4] | # of time units to pause after sending |
| irCode[5] | must be zero |
| irCode[6..N] | sequence of numbers representing the number of time units to remain in each state, starting with OFF |

Refer to the Technical Notes file on the disk for a detailed description of infrared remote control programming.

**EXAMPLE**

```
10 REM SENDIRREMOTE Example
20 t="01000101101110101110100000010111"
30 trans=[0,500,14,50,14,0,8]
35 zero="0"[0] // char 0 (not string 0)
40 FOR i=0 TO strLen(t)-1
50 ADDARRAYSLOT(trans,1)
60 IF t[i]=zero THEN ADDARRAYSLOT(trans,1)
ELSE ADDARRAYSLOT(trans,3)
70 NEXT i
80 ADDARRAYSLOT(trans,1)
90 ADDARRAYSLOT(trans,1)
100 SENDIRREMOTE(trans,1)
```

**OUTPUT**

```
(If you have a Pioneer CD, running this while
pointing the Newton at the CD Player will
cause it to start playing the disk)
```

**RELATED ITEMS**

WINDOW *winNum*, *windowSpec*, "SETCLOCK"

**DESCRIPTION**

The SETCLOCK widget provides a clock face for time display and entry. The clock face is always drawn such that it uses a 64x64 pixel area. You must be sure that your supplied viewBounds provides an area of this size. Whenever either clock hand is changed by the user, your GOTO or GOSUB routine will be called. You access the user's selection using:

```
hours = windowSpec.hours
minutes = windowSpec.minutes
```

The widget is controlled using the *windowSpec*. These fields are supported:

`hours`: current setting of the hour hand (or the current hour if not supplied)

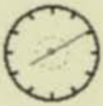
`minutes`: current setting of the minute hand (or the current minute if not supplied)

You may also use these fields in *windowSpec*: `viewBounds`, `viewFlags`, `viewFont`, `GOTO`, `GOSUB`, `viewFormat`.

**EXAMPLE**

```
10 REM SETCLOCK Example
20 w1Spec = {}
30 WINDOW w1, w1Spec, "SETCLOCK"
40 SHOW w1
```

**OUTPUT**



**RELATED ITEMS**

HIDE, SHOW, NEWSETCLOCK, WINDOW

SETBOUNDS(*left, top, right, bottom*)

**DESCRIPTION**

SETBOUNDS returns a viewBounds frame for use in a windowSpec. When you use SETBOUNDS, you reduce the amount of memory needed to store viewBounds frames. If you create a large number of windows and widgets, the memory savings can be significant.

**EXAMPLE**

```
10 REM SETBOUNDS Example
20 W1Spec={viewBounds: SETBOUNDS(10, 50, 200,
80)}
30 WINDOW Win1, W1Spec
40 SHOW Win1
```

**OUTPUT**

```
(a window with viewBounds: {left:10,
top:50, right:200, bottom:80} is
displayed)
*
```

**RELATED ITEMS**

WINDOW

# SETICON Statement

SETICON *program, icon*

**DESCRIPTION**

SETICON sets the icon that is displayed in the Extras drawer for a program. Use this Statement to supply a custom icon when creating a stand-alone package with the MACKPACKAGE Statement. the *program* paramater is a string value specifing the name of a previously SAVEd program. The *icon* parameter is a value returned from the MAKEBITMAP function.

**EXAMPLE**

```
10 REM SETICON Example
20 ws := {viewBounds: SETBOUNDS(100, 132, 100,
132)}
30 shape := [MAKERECT(1,1,30,30),
MAKETEXT("$",12,10,21,21)]
40 icon:=MAKEBITMAP(32,32,NIL)
50 DRAWINTOBITMAP(shape,NIL,icon)
60 SETICON "INVEST",icon
```

**OUTPUT**

```
(An icon of $ is used by the INVEST program,
when a stand-alone package is created from it)
*
```

**RELATED ITEMS**

DRAWINTOBITMAP, MAKEBITMAP, MAKEPACKAGE, WINDOW

# SETVALUE Function

SETVALUE(*windowSpec*, *fieldName*, *value*)

**DESCRIPTION**

SETVALUE updates a value of a field in a *windowSpec* for a widget. The widget is re-displayed to reflect the new value. NIL is always returned. If you just change the field value in *windowSpec* without using SETVALUE, the Newton display is not updated.

**EXAMPLE**

```
10 REM SETVALUE Example
20 W1Spec={viewBounds:
SETBOUNDS(10,50,200,80)}
30 WINDOW Win1, W1Spec, "LabelInput"
40 SHOW Win1
50 FOR i = 1 TO 10
60 SETVALUE(W1Spec.entryline, 'text, "Number:
" &i)
70 WAIT 100
80 NEXT i
90 HIDE Win1
```

**OUTPUT**

```
(A label input widget is displayed and the
value in the entry line counts up from
Number:1 to Number:10.)
•
```

**RELATED ITEMS**

WINDOW

# SHOW Statement

SHOW *winNum* | *winNumlist*

### DESCRIPTION

SHOW displays the previously declared single window *winNum* or list of windows *winNumlist* on the screen. *winNum* and *winNumlist* use the number returned by the WINDOW Statement. To hide windows use the HIDE Statement.

### EXAMPLE

```
110 REM SHOW Example
20 W1Spec={ViewBounds: SETBOUNDS(10, 50, 100,
100)}
30 WINDOW Win1, W1Spec
40 WINDOW Win2, W2Spec
50 WPRINT Win1, "Window 1"
60 WPRINT Win2, "Window 2"
70 SHOW Win1, Win2
80 WAIT
90 HIDE Win2
100 SHOW Win2
110 HIDE
```

### OUTPUT

```
(Two windows are created and then removed from
the screen.)
•
```

### RELATED ITEMS

HIDE, WINDOW, WPRINT

SIGNUM(*x*)

**DESCRIPTION**

SIGNUM returns the sign of *x*. It returns 1 if *x* is positive, 0 if *x* is zero, and -1 if *x* is negative.

**EXAMPLE**

```
10 REM SIGNUM Example
20 PRINT "Please enter a number"
30 INPUT X
40 PRINT "SIGNUM of x is = " ; SIGNUM(X)
```

**OUTPUT**

```
Please enter a number
? -4
SIGNUM of x is = -1
*
```

**RELATED ITEMS**

# SIN Function

SIN(*x*)

SINH(*x*)

ASIN(*x*)

ASINH(*x*)

### DESCRIPTION

SIN returns the sine of the angle *x* in radians.

SINH returns the hyperbolic sine of *x*.

ASIN returns the arc sine of *x*.

ASINH returns the arc-hyperbolic sine of *x*.

### EXAMPLE

```
10 REM SIN Example
20 PRINT "Please enter an angle"
30 INPUT Angle
40 PRINT "The Sine of the angle is = " ;
SIN(Angle) ; " radians"
```

### OUTPUT

```
Please enter an angle
? 63.7
The Sine of the angle is = 0.763132715516785
radians
*
```

### RELATED ITEMS

TAN, COS

# SLIDER Widget

WINDOW *winNum*, *windowSpec*, "SLIDER"

### DESCRIPTION

The SLIDER widget provides a gauge that the user can set. The value of the widget is a number from 0 (slider all the way to the left) to 100 (slider all the way to the right). Whenever the slider is changed by the user, your GOTO or GOSUB routine will be called. You access the user's selection using:

`setting = `*`windowSpec`*`.viewValue`

The widget is controlled using the *windowSpec*. These fields are supported:

`viewValue`: current setting of the slider from 0 to 100

You may also use these fields in *windowSpec*: `viewBounds`, `viewFlags`, `viewFont`, `GOTO`, `GOSUB`, `viewFormat`.

### EXAMPLE

```
10 REM SLIDER Example
20 w1Spec = {}
30 WINDOW w1, w1Spec, "SLIDER"
40 SHOW w1
```

### OUTPUT

### RELATED ITEMS

GAUGE, HIDE, SHOW, WINDOW

# SORT

Function

SORT(*array*, *test*, *key*)

**DESCRIPTION**

SORT returns *array* sorted by *test* applied to the element *key*. Values for test are as follows.

| | |
|---|---|
| '\|<\| | Sort in ascending numerical order |
| '\|>\| | Sort in descending numerical order |
| '\|str<\| | Sort in ascending string order |
| '\|str>\| | Sort in descending string order |

If *key* is NIL, the items of *array* are sorted directly by their values. To sort an *array* where each element is a frame, put the name of the element to be sorted by as the third parameter, preceded by a ' sign.

**EXAMPLE 1**

```
10 REM SORT an array Example
20 DIM A[3]
30 A[0]=23
40 A[1]=5
50 A[2]=54
60 A=SORT(A,'|<|,NIL)
70 PRINT A[0],A[1],A[2]
```

**OUTPUT**

```
5      23         54
•
```

### EXAMPLE 2

```
10 REM SORT of array of frames Example
15 DIM a[4]
20 a[0]={name: "Arthur", seq: 2}
30 a[1]={name: "Ford", seq: 3}
40 a[2]={name: "Trill", seq: 1}
50 a[3]={name: "Zaphod", seq: 4}
60 a=SORT(a,'|<|, 'seq)
70 FOR i=0 TO 3
80 PRINT a[i].name
90 NEXT i
```

### OUTPUT

```
Trill
Arthur
Ford
Zaphod
*
```

### RELATED ITEMS

# SQRT Function

SQRT(x)

**DESCRIPTION**

SQRT returns the square root of the number x.

**EXAMPLE**

```
10 REM SQRT Example
20 REM This program returns the square root of
the number entered at the prompt.
30 PRINT "Please enter a number"
40 INPUT Number
50 PRINT "Square root = " ; SQRT(Number)
```

**OUTPUT**

```
Please enter a number
? 2
Square root = 1.14121356237309
*
```

**RELATED ITEMS**

POW

# STATS Command

STATS

### DESCRIPTION

STATS shows information on memory usage for the current program.

Under the name of the currently loaded program are three lines. The first line displays the number of lines of code for the program, and how much active memory it uses.

The second line displays the memory used for code space.

The third line displays the remaining available memory.

**Note:** There is no direct correlation between the program size and how much memory remains.

The remaining lines show the program build time and the Environment Variables.

### EXAMPLE

```
* STATS
```

### OUTPUT

```
CurrentProgram: SCRATCH
   410 bytes used for 5 statements
   510 bytes used for code space
 45232 bytes free.

Build:8/19/95 7:10 am

ENV: {tag: "BASIC:NSBASIC", programName:
"SCRATCH.BAS:NSBASIC", serialNumber: xxxxxx,
useScratch: TRUE, io: "S0", inputPrompt: "? ",
s0:{ data:[ 9600 2 1 "no"] connect:
"Connected.
* " unpend:
}, Store: 1}
```

### RELATED ITEMS

VARS

STOP

**DESCRIPTION**

STOP halts execution of the program, and plays a BEEP on the Newton. The program may then be continued from the line after STOP by using the CON Command. The STOP Command can be used during debugging to STOP the program at a certain line.

**EXAMPLE**

```
10 REM STOP Example
20 PRINT "First Program Section"
30 STOP
40 PRINT "Second Program Section"
```

**OUTPUT**

```
First Program Section
Stop at 0030
* CON
Second Program Section
*
```

**RELATED ITEMS**

END, CON

# STRCOMPARE **Function**

STRCOMPARE(*string1*, *string2*)

### DESCRIPTION

STRCOMPARE returns a negative number if *string1* is less than *string2*. It returns zero if *string1* and *string2* are equal. It returns a positive number if *string1* is greater than *string2* This function is not case sensitive. The strings are compared based on all the ASCII codes of the characters within them.

### EXAMPLE

```
10 REM STRCOMPARE Example
20 REM User enters two items which are forced
into strings. Computer compares them.
30 PRINT "Please enter item 1"
40 INPUT String1$
50 PRINT "Please enter item 2"
60 INPUT String2$
70 Result = STRCOMPARE(String1$, String2$)
80 IF Result = 0 THEN PRINT "Strings are Equal"
90 IF Result > 0 THEN PRINT "Second string is
larger"
100 IF Result < 0 THEN PRINT "First string is
larger"
```

### OUTPUT

```
Please enter item 1
? Hello World
Please enter item 2
? Llamas
First string is larger
*
```

### RELATED ITEMS

STREQUAL

# STREQUAL Function

STREQUAL(*string1*, *string2*)

**DESCRIPTION**

STREQUAL returns TRUE if *string1* and *string2* are equal. It returns NIL for all other cases. This function is not case sensitive. The strings are compared based on all the ASCII codes of the characters within them.

**EXAMPLE**

```
10 REM STREQUAL Example
20 REM User enters two items which are forced
into strings. Computer compares them.
30 PRINT "Please enter item 1"
40 INPUT String1$
50 PRINT "Please enter item 2"
60 INPUT String2$
70 Result = STREQUAL(String1$, String2$)
80 IF STREQUAL(String1$, String2$) THEN PRINT
"Strings are Equal" ELSE PRINT "Strings are
not Equal"
```

**OUTPUT**

```
Please enter item 1
? Hello World
Please enter item 2
? Goodbye World
Strings are not Equal
*
```

**RELATED ITEMS**

STRCOMPARE

# STRINGER Function

STRINGER(*array*)

### DESCRIPTION

STRINGER returns a string containing all the elements in *array* concatenated together. Numbers, characters, and symbols are all converted to their string representation. Elements that are frames, arrays or Booleans are converted to an empty string.

### EXAMPLE

```
10 REM STRINGER Example
20 REM Concatenates 3 array elements
30 DIM Array[3]
40 FOR i = 0 TO 2
50 PRINT "Please enter something"
60 INPUT Element
70 Array[i] = Element
80 NEXT i
90 PRINT "The result is..."
100 PRINT STRINGER(Array)
```

### OUTPUT

```
Please enter something
? Hello
Please enter something
? World
Please enter something
? 17.9
The result is...
HelloWorld17.9
•
```

### RELATED ITEMS

# STRINGTOTIME Function

STRINGTOTIME(*string*)

### DESCRIPTION

STRINGTOTIME returns the TIME() value of *string*. *string* must contain a string representation of a time, such as `"3:40 pm"`.

### EXAMPLE

```
10 REM STRINGTOTIME Example
20 theTime = STRINGTOTIME("3:40 pm")
30 PRINT theTime
```

### OUTPUT

```
48371980
*
```

### RELATED ITEMS

DIGITALCLOCK, TIME, TIMESTR

# STRINGTONUMBER Function

STRINGTONUMBER(*string*)

**DESCRIPTION**

STRINGTONUMBER returns the real number value of *string*. *string* must contain a string representation of a number, such as "46".

**EXAMPLE**

```
10 REM STRINGTONUMBER Example
20 REM Places two "string" numbers together
and adds 5 to that number.
30 PRINT "Please enter 2 numbers"
40 INPUT Number1$,Number2$
50 NewNumber = Number1$ & Number2$
60 PRINT "The numbers concatenated are... " ;
NewNumber
70 PRINT "The Numbers with 5 added are... " ;
STRINGTONUMBER(NewNumber)+5
```

**OUTPUT**

```
Please enter 2 numbers
? 5,7
The numbers concatenated are... 57
The Numbers with 5 added are... 62
*
```

**RELATED ITEMS**

NUMBERSTRING

STRLEN(*string*)

**DESCRIPTION**

STRLEN returns the number of characters in *string*.

**EXAMPLE**

```
10 REM STRLEN Example
20 PRINT "Enter a String"
30 INPUT string$
40 PRINT "There are " ; STRLEN(String$) ; "
characters in the string"
```

**OUTPUT**

```
Enter a string
? Hello World
There are 11 characters in the string
*
```

**RELATED ITEMS**

# STRPOS Function

STRPOS(*string*, *substring*, *start*)

### DESCRIPTION

STRPOS returns the position of *substring* in *string*, or NIL if *substring* is not found. The search begins at character position *start* (the first character position is zero.) This function is not case sensitive. The position returned is also numbered from zero.

### EXAMPLE

```
10 REM STRPOS Example
20 REM Looks for a substring in a user defined
string.
30 PRINT "Please enter a string"
40 INPUT String
50 PRINT "Please enter a string to look for"
60 INPUT Substring
70 Result = STRPOS(String,Substring,0)
80 IF Result = NIL THEN PRINT "Substring not
found" ELSE PRINT "Substring is at character
" ; Result
```

### OUTPUT

```
Please enter a string
? This is a simple string
Please enter a string to look for
? Simple
Substring is at character 10
*
```

### RELATED ITEMS

SUBSTR, STRLEN

# SUBSTR Function

SUBSTR(*string*, *start*, *count*)

### DESCRIPTION

SUBSTR returns a new string containing *count* characters from *string*, starting at character *start*. Character positions begin with zero for the first character. If count is NIL, all characters from *start* to the end of *string* are returned.

### EXAMPLE

```
10 REM SUBSTR Example
20 REM Creates a substring from the first 5
characters of a string.
30 PRINT "Please enter a string"
40 INPUT String
50 Result = SUBSTR(String, 0, 4)
60 PRINT "The new substring is " ; Result
```

### OUTPUT

```
Please enter a string
? Sample string
The new substring is "Samp"
*
```

### RELATED ITEMS

STRPOS, STRLEN

# TAN — Function

TAN(*x*)

ATAN(*x*)

ATAN2(*x*,*y*)

TANH(*x*)

ATANH(*x*)

**DESCRIPTION**

TAN returns the tangent of the angle *x* in radians.

ATAN returns the arc tangent of *x*.

ATAN2 returns the arc tangent of *x*/*y* in [-π,π].

TANH returns the hyperbolic tangent of *x*.

ATANH returns the arc-hyperbolic tangent of *x*.

**EXAMPLE**

```
10 REM TAN Example
20 PRINT "Please enter an angle"
30 INPUT Angle
40 PRINT "The tangent of the angle is = " ;
TAN(Angle) ; " radians"
```

**OUTPUT**

```
Please enter an angle
? 72
The tangent of the angle is =
-0.262417377501932 radians
*
```

**RELATED ITEMS**

COS, SIN

# TEXT Widget

WINDOW *winNum*, *windowSpec*, "TEXT"

**DESCRIPTION**

The TEXT widget provides a text entry area that does not scroll. Hand written entry in this area will be recognized and converted into text. The `viewFlags` field of the *windowSpec* can be used to indicate which recognition should be attempted. You extract the text entered by the user with this expression:

```
enteredText = windowSpec.text
```

The widget is controlled using the *windowSpec*. These fields are supported:

`text`: The text displayed and entered by the user

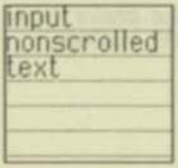
`viewLineSpacing`: spacing of the lines, in pixels

You may also use these fields in *windowSpec*: `viewBounds`, `viewFlags`, `viewFont`, `viewFormat`.

**EXAMPLE**

```
10 REM TEXT Example
20 w1Spec = {text: "Input..."}
30 WINDOW w1, w1Spec, "TEXT"
40 SHOW w1
```

**OUTPUT**



```
input
nonscrolled
text
```

**RELATED ITEMS**

HIDE, PARAGRAPH, SCROLLER, SHOW, WINDOW

TICKS()

**DESCRIPTION**

TICKS returns the number of ticks of the system clock. A tick is 1/60th of a second. There is no defined starting time for ticks. TICKS are used to measure intervals and durations of time.

**EXAMPLE**

```
10 REM TICKS Example
20 Oldtime = TICKS()
30 PRINT "Tap any key, then the enter key when
ready"
40 INPUT A$
50 Newtime = TICKS()
60 PRINT (Newtime-Oldtime) / 60 ; " Seconds
passed"
```

**OUTPUT**

```
Tap the enter key when ready
?
12.83333333 Seconds passed
*
```

**RELATED ITEMS**

TIME, HOURMINUTE, DATENTIME

TIME()

**DESCRIPTION**

TIME returns the current time in minutes as an integer. This is the number of minutes passed since midnight, January 1, 1904. Use the HOURMINUTE and DATENTIME Functions to process the number returned by TIME.

**EXAMPLE**

```
10 REM TIME Example
20 PRINT "The Number of Minutes passed since
01/01/04 is..." ; TIME()
30 PRINT "The Current Date and Time is " ;
DATENTIME(TIME())
```

**OUTPUT**

```
The Number of Minutes passed since 01/01/04
is... 47526491
The Current Date and Time is 5/20/94 2:05 AM
*
```

**RELATED ITEMS**

DATENTIME, HOURMINUTE

# TIMESTR Function

TIMESTR(*timeValue*, *option*)

### DESCRIPTION

TIMESTR returns a string representation of the *timeValue*.

The *option* paramaters controls how the string is formatted.

| | |
|---|---|
| 0 | "HH:MM:SS AM/PM" |
| 1 | Hours |
| 2 | Minues |
| 3 | Seconds |
| 4 | AM/PM |

The DIGITALCLOCK widget returns a timeValue, as does the TIME() Function.

### EXAMPLE

```
10 REM TIMESTR Example
20 theTime = TIME()
30 PRINT TIMESTR(theTime, 0)
40 PRINT TIMESTR(theTime, 1)
50 PRINT TIMESTR(theTime, 2)
60 PRINT TIMESTR(theTime, 3)
70 PRINT TIMESTR(theTime, 4)
```

### OUTPUT

```
* run
2:18:00 pm
2
18
00
 pm
*
```

### RELATED ITEMS

DATENTIME, DIGITALCLOCK, HOURMINUTE, TIME

TRACE ON

TRACE OFF

### DESCRIPTION

TRACE ON enables the tracing of line numbers during program execution. TRACE OFF disables it.

After processing the TRACE ON Statement, NS BASIC will display each line number as that line is executed.

The TRACE Statement is useful in debugging programs where it can show you exactly where a problem happened.

If a program is executed from a point other than the beginning, the condition (ON or OFF) of the TRACE Statement is not reset. RUNning a program from the beginning always turns off tracing.

**EXAMPLE**

```
10 REM TRACE Example
20 PRINT "This is an EXAMPLE"
30 PRINT "Llamas"
40 TRACE ON
50 FOR i = 1 TO 3
60 PRINT i
70 NEXT i
80 TRACE OFF
90 PRINT "End of program reached."
```

**OUTPUT**

```
This is an EXAMPLE
Llamas
[X0050]
[X0060]
1
[X0050]
[X0060]
2
[X0050]
[X0060]
3
[X0050]
[X0060]
End of program reached.
*
```

**RELATED ITEMS**

RUN, STOP, CON

# VARS Command

VARS

### DESCRIPTION

VARS displays a listing of all variables and their current values.

VARS displays the elements of arrays created with the DIM Statement, and the fields of frames.

The GOSUB stack is shown after all variables. This is a list of the line numbers for each GOSUB statement executed that has not yet reached a RETURN statement.

### EXAMPLE

```
10 X = 100
20 Y = 200
30 DIM Z[2]
40 I = { Name:"John", Age: 12}
* RUN
* VARS
```

### OUTPUT

```
x: 100
y: 200
Z:[0,0]
i:{name:"John",AGE:12}
Gosub Stack:
*
```

### RELATED ITEMS

LET, RUN, STATS

WAIT [*ticks*]

**DESCRIPTION**

WAIT stops the program for *ticks* thousandths of a second. If *ticks* is not supplied 500 (1/2 second) is used. The largest value for *ticks* is 858993, around 14 minutes 20 seconds.

Once the number of specified *ticks* have passed the next statement is executed. If the user taps on a WINDOW with a `windowspec.GOTO` value defined while the program is WAITing, the program will branch to the line number specified in the value.

This feature can be used to program buttons. Define a WINDOW with whatever size and boundaries you need, with a GOTO line number defined for it.

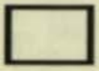
**Note:** Since windows remain on the display even after the program has stopped, the buttons remain active as well.

**EXAMPLE**

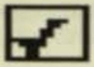
```
10 REM WAIT Example
11 f={GOTO:1000, viewBounds: SETBOUNDS(100,
100, 110, 110)}
15 CLS
20 WINDOW w1,f
30 SHOW w1
40 FOR i=1 TO 3
45 PRINT i
50 WAIT
70 NEXT i
80 STOP
1000 REM toggle checkbox
1010 cbox = NOT cbox
1020 IF cbox THEN WPRINT w1, CHR(8730) ELSE
WPRINT w1,""
```

**OUTPUT**

```
1
2
3
Stop at 0080
•
```



(Before Tap)



(After Tap)

Note that in the above example CHR(8730) is the character number that prints out as a checkmark. Refer to Appendix C of this Handbook for a list of special character codes.

**RELATED ITEMS**
GOTO, WINDOW

WDRAW *windowNum*, *shapes* [, *styleFrame*]

**DESCRIPTION**

WDRAW draws *shapes* in the window *windowNum*. *windowNum* is the number returned by the WINDOW Statement. Shapes may be a single shape or an array of shapes. The style used to display the shapes can be defined using *styleFrame*. There are several elements in the frame which can be set. If they are not set, defaults are used.

penSize:

PenSize specifies the size of the pen in pixels. An array can be used to specify [width, height]. The default is 1.

penPattern: `vfNone, vfWhite,`
`vfLtgray, vfGray,`
`vfDkgray, vfBlack`

PenPattern defines the pattern drawn by the pen. The default is vfBlack.

fillPattern: `vfNone, vfWhite,`
`vfLtgray, vfGray,`
`vfDkgray, vfBlack`

FillPattern defines the pattern inside of closed shapes. The default is vfNone.

font: `{family: `*fontName*`, face: `*fontFace*`, size: `*fontSize*`}`

font defines the font for any text shapes displayed. See the WINDOW statement for a complete list. The default is the user's default font.

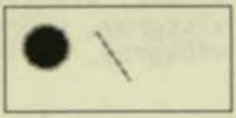
justification: `'left, 'right, 'center`

Justification defines the alignment of any text shapes displayed. The default is `'left`.

**EXAMPLE**

```
10 REM WDRAW Example
20 W1Spec={viewBounds: SETBOUNDS(10, 10, 150,
75)}
30 WINDOW WinNum, W1Spec
40 SHOW WinNum
50 WDRAW WinNum, [MAKELINE(55,15,75,45),
MAKEOVAL(10,10,40,40)], {penSize:2,
penPattern:vfGray, fillPattern:vfBlack}
```

**OUTPUT**



**RELATED ITEMS**

SHOW, HIDE, MAKELINE, WINDOW

# WINDOW Statement

WINDOW *winNum*, *windowSpec*, [, *widget*]

**DESCRIPTION**

WINDOW creates a graphic window on the Newton. Each window is given an unique number by NS BASIC. The WINDOW Command returns this number in *winNum*. Use this number for subsequent SHOW, HIDE, WPRINT, and WDRAW Statements.

*windowSpec* is a frame containing information about the window. There are several elements in the frame which can be set. If they are not set, defaults are used.

**Note:** Never use the same *windowSpec* variable in multiple WINDOW Statements without first assigning a new frame value to the variable.

viewBounds: `{top:` *position1*`, left:` *position2*`, bottom:` *position3*`, right:` *position4*`}`

ViewBounds defines the bounds of the window. *Position1* to *position4* is the location of the window on the screen. A Newton MessagePad's screen is approximately 240 pixels wide by 320 pixels high.

viewFlags: `vVisible + vFloating + vClickable + vGesturesallowed + vSingleunit + vCharsallowed + vLettersallowed + vPunctuationallowed + vShapesallowed + vStrokesallowed + vCapsrequired + vNumbersallowed + vNamefield + vPhonefield + vDatefield + vTimefield`

`viewFlags` defines the special characteristics of the window. Not all combinations are valid. Each characteristic is described below.

| | |
|---|---|
| vVisible | TRUE to make window visible, NIL to hide |
| vFloating | TRUE to make window float over all others, NIL for normal window stacking |
| vClickable | TRUE if the window accepts pen taps |
| vGesturesallowed | TRUE to accept Newton gestures such as scrub. |
| vSingleunit | TRUE to accept only one word |
| vCharsallowed | TRUE to use word recognition |
| vLettersallowed | TRUE to use letter-by-letter recognition |
| vPunctuationallowed | TRUE to accept punctuation |
| vShapesallowed | TRUE to recognize Boxes, Lines, and Circles |
| vStrokesallowed | TRUE to accept digital ink |
| vCapsrequired | TRUE to capitalize first letter of each word as entered |
| vNumbersallowed | TRUE to accept numbers |
| vNamefield | TRUE if this is a name field |
| vPhonefield | TRUE if this is a phone field |
| vDatefield | TRUE if this is a date field |
| vTimefield | TRUE if this is a time field |

**Note:** Not all combinations of these values are valid. If you try a combination that does not look as expected, then you've found an invalid combination.

viewFont: {family: *fontName*, face: *fontFace*, size: *fontSize*}

viewFont defines the font to be displayed in the WINDOW. *fontName* is the name of the font you wish to be used in the window. Possible fonts on the Newton are 'espy, 'geneva, 'newyork, or 'handwriting. **Note:** The ' sign is required.

*fontFace* is the style of the font: 0 for plain, 1 for bold, 2 for italics, 4 for underline, 8 for outline, 128 for superscript, 256 for subscript. *fontSize* may be 9,10,12,14 or 18.

**Note:** Not all combinations of fontName, fontFace and fontSize are valid. If you try a combination that does not look as expected, then you've found an invalid combination.

viewFormat: *frameColor* + *fillColor* + *x**vfpen + *y**vfshadow + *z**vfround

viewFormat defines the visual format of the WINDOW. If viewFormat is 0, then the window is transparent. *frameColor* is the color (pattern) of the window border. *frameColor* may be one of:

vfFramewhite

vfFrameGray

vfFrameBlack

vfFrameltgray

vfFrameDkgray

vfFrameMatte (thick gray bordered by black)

*fillColor* is the color of the contents of the window. *fillColor* may be one of:

vfFillWhite

vfFillGray

vfFillBlack

vfFillLtgray

vfFillDkGray

*x**vfPen sets the width of the border in pixels. X should be between 0 and 15. *Y**vfShadow sets the width of the

shadow in pixels. *Y* should be between 0 and 3.
Z*vfRound is the corner radius, in pixels. *Z* should be between 0 and 15.

viewJustify:      *justifyCode*

viewJustify defines the type of justification used for the text displayed in WINDOW. *justifyCode* is 0 for text left, 1 for text right, 2 for text centered and 3 for text stretched across the entire width of WINDOW.

GOTO:      *lineNumber*

GOTO defines tap processing. *lineNumber* is the line of code the program should GOTO if the WINDOW is tapped. A click sound is played on the Newton when the user taps on a WINDOW that has a GOTO defined for it.

GOSUB:      *lineNumber*

GOSUB defines tap processing. *lineNumber* is the line of code the program should GOSUB if the WINDOW is tapped. A click sound is played on the Newton when the user taps on a WINDOW that has a GOSUB defined for it. A RETURN will return execution to the line following the WAIT that was executing when the window was tapped.

You may examine the windowSpec frame after a tap has been processed and a GOTO or GOSUB is performed. The *windowSpec* will contain these four additional fields:

FIRSTX: The X coordinate of the first point on the Newton Screen where the user placed the pen down.

FIRSTY: The Y coordinate of the first point on the Newton Screen where the user placed the pen down.

LASTX: The X coordinate of the point on the Newton Screen where the user lifted the pen.

LASTY: The X coordinate of the point on the Newton Screen where the user lifted the pen.

Whenever NS BASIC performs a GOTO or GOSUB in response to a pen tap, the variable WSTAT is set to the *winNum* of the window that was tapped.

**Note**: You should avoid using a variable named WSTAT for your own purposes.

text:

text contains the current text displayed in the window using WPRINT.

drawing:

drawing contains the current graphic displayed in the window using WDRAW.

This code fragment creates a valid *windowSpec* with many of these elements:

```
10 windowSpec:={viewBounds: SETBOUNDS(10, 10,
40, 30), viewFont: {family: 'handwriting,
face: 0, size: 12}, viewFormat: vfFrameBlack
+ vfFillWhite + 2*vfPen + 3*vfShadow + 6*vfRound,
viewJustify: 0, GOTO: 2000, text: "Yo!"}
20 WINDOW w1, windowSpec
25 SHOW w1
30 WAIT 1000
40 GOTO 30
2000 REM Call me when tapped!
2010 END // just stop
```

*widget* is a string. If included, it must be one of the values shown below. Each widget is described separately in the Reference section.

| | |
|---|---|
| APP | an application background |
| AZTAB | an alphabet picker |
| AZVERTTAB | an alphabet picker in a verical orientation |
| CLOSEBOX | a small Newton close box |
| CHECKBOX | a checkbox followed by a label |
| DATEPICKER | a calendar display and date picker |
| DRAW | a box that accepts pen drawings |
| GAUGE | a linear display of a value |
| GLANCE | a text window that appears for 3 seconds |
| LABELINPUT | a labeled text entry field |
| LABELPICKER | a labeled field with a pick list |
| LARGECLOSEBOX | a large Newton close box. |

| | |
|---|---|
| MONTH | a month display that accepts date selections |
| NEWSETCLOCK | a clock that can be set |
| NUMBERPICKER | a number display and picker |
| PARAGRAPH | a window that displays styled text |
| PICKER | a pick list, showing the current selection |
| PICTUREBUTTON | a button that displays an icon |
| RCHECKBOX | a label followed by a checkbox |
| SCROLLER | a text entry field that expands and scrolls |
| SETCLOCK | a clock that can be set |
| SLIDER | linear display and entry of values |
| TEXT | a plain text entry window |

widgets may be hilighted (shown in inverse) once they are displayed using the *windowSpec* for the widget. For example:

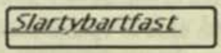
```
U.windowSpec:HILITE(TRUE)
U.windowSpec:HILITE(NIL)
```

The first line will invert the widget associated with windowSpec, the second will revert it to a normal display.

### EXAMPLE

```
10 REM WINDOW Example
20 W1Spec := {viewbounds: SETBOUNDS(10, 50,
150, 75), viewFont: {family: 'espy, face: 7,
size:14}, viewFormat: 4*vfRound +2*vfPen
+vfFrameBlack+vfFillWhite, viewJustify: 2}
30 WINDOW WinNum, W1Spec
40 SHOW WinNum
50 WPRINT WinNum, "Slartybartfast"
```

### OUTPUT



Slartybartfast

### RELATED ITEMS

HIDE, HWINPUT, SHOW, WAIT, WDRAW, WPRINT

See WAIT for an example of using the GOTO element in a *windowSpec.*

WPRINT *windowNum*, *expression*

**DESCRIPTION**

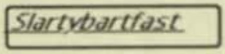
WPRINT displays the contents of *expression* in window *windowNum*. *windowNum* is the number returned by the WINDOW Statement. The font and style used to display the text will be those defined for window *windowNum*.

WPRINT can also be used to update the display of a GAUGE widget once the `viewValue` has been changed.

**EXAMPLE**

```
10 REM WPRINT Example
20 W1Spec := {viewbounds: SETBOUNDS(10, 50,
150, 75), viewFont: {family: 'espy, face: 7,
size:14}, viewFormat: 4*vfRound +2*vfPen
+vfFrameBlack+vfFillWhite, viewJustify: 2}
30 WINDOW WinNum, W1Spec
40 SHOW WinNum
50 WPRINT WinNum, "Slartybartfast"
```

**OUTPUT**



Slartybartfast

**RELATED ITEMS**

SHOW, GAUGE, HIDE, WINDOW, HWINPUT

CHAPTER

4

# 4. Advanced Topics

This chapter provides detailed examples showing the use of the advanced and Newton-specific features of NS BASIC. We'll present each topic, walk through a detailed example, and give you advice on why you'd want to use these features in your own programs. This section of the Handbook will be much more informal. Curl up with your Newton, NS BASIC, and the Handbook, and follow along!

## 4.1 Windows

The window capabilities of NS BASIC go far beyond what you may be used to in other BASICs. You will need to experiment with the use of windows to discover the many ways they can be used. You will find graph paper or a drawing program very helpful when creating the "layout" of your windows. You'll need to know the exact positions of the upper left and lower right corners of every window you want to create. Don't worry, it's not as bad as it sounds! Here are just a few examples to get you started.

### Buttons

A window can act like a button. You need to use the GOTO or GOSUB element of *windowSpec* to branch to the code that processes a user tap on the button. The following example displays a button that changes its label when tapped. The program ends after the button has been tapped 4 times:

```
10 REM Button Example
20 f = {GOTO: 130, viewBounds: SETBOUNDS(100,
100, 180, 120)}
30 buttonLabels = ["Hello", "World"]
40 taps = 0 // no taps yet
50 CLS
60 WINDOW w1,f
70 SHOW w1
80 WPRINT w1, buttonLabels[taps mod 2]
90 WAIT 1000
100 GOTO 90 // Wait Loop
110 HIDE w1
120 END
130 REM toggle button
```

```
140 taps=taps+1
150 WPRINT w1, buttonLabels[taps mod 2]
160 IF taps < 4 THEN GOTO 90 ELSE GOTO 110
```

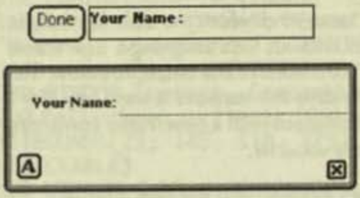
This code uses a style of window programming known as "Event loop" programming. Using this technique, you create the windows that the user can interact with, and then you enter an infinite loop. The only thing the loop does is WAIT. This allows NS BASIC to process the user taps in windows by branching to the lines specified in the GOTO or GOSUB element of each *windowSpec*.

### Hand Written Input

This next example shows how to allow user entry via HWINPUT and display the result in a window:

```
10 REM Button/HWINPUT Example
20 f={GOTO:0200, viewBounds: SETBOUNDS(50,
100, 200, 120)}
30 buttonLabel = "Your Name: "
40 nameValue = "" // no name entered
50 CLS
60 WINDOW w1,f
70 g={GOTO:0170, viewbounds: SETBOUNDS(10,
100, 45, 120)}
80 g.viewFont={family: 'espy, face: 0,
size:10}
90 g.viewJustify=2 // centered
100 g.viewFormat = 6*vfRound
+2*vfPen+vfFrameBlack+vfFillWhite
110 WINDOW w2,g
120 SHOW w1, w2
130 WPRINT w2, "Done"
140 WPRINT w1, buttonLabel & nameValue
150 WAIT 1000
160 GOTO 150
170 HIDE w1, w2
180 END
190 REM use HWINPUT
200 BEEP 5 // drawer open sound
210 IF STRLEN(nameValue) > 0 THEN currName =
[nameValue] ELSE currName = NIL
220 HWINPUT nameValue, buttonLabel, currName
230 WPRINT w1, buttonLabel & nameValue
240 GOTO 150
```

OUTPUT



This program displays two windows. The Done window acts like a button. Tapping it ends the program. Tapping in the other window opens a user input field that accepts hand written input. The entered information is placed back into the window when the user closes the input box.

## 4.2 Widgets

A widget is a Newton user interface element. In the NewtonScript language, widgets are called protos and view templates. You can create programs that use many of the visual elements found in Newton applications. Refer to the Reference section under WINDOW for a complete list of widgets.

When you want to create an application that uses widgets, you combine windows and widgets to create a single screen that the user interacts with. This screen may consists of many widgets. Once the widgets are displayed, the user can interact with the screen using pen taps, hand-written input, and the keyboard. Your screen must be designed so that the user can indicate when they are finished entering information. A window that looks like a Newton button can be used. You can create a main window for your application using the APP widget. This widget includes a standard close box. You can add one or more additional windows that look like Newton buttons, such  as an "OK" button. These windows will use a GOTO field in their *windowSpec* to allow your program to process the user entry in the form.

The following sample program creates a simple database with a form for user entry. It supports searching for records and adding new records. It uses several widgets for the form.

The three sections of the program you should examine are the creation of the widgets (lines 130-330), the extraction of the user entered values from the widgets (lines 500-510), and the displaying of new values in widgets (lines 430-470).

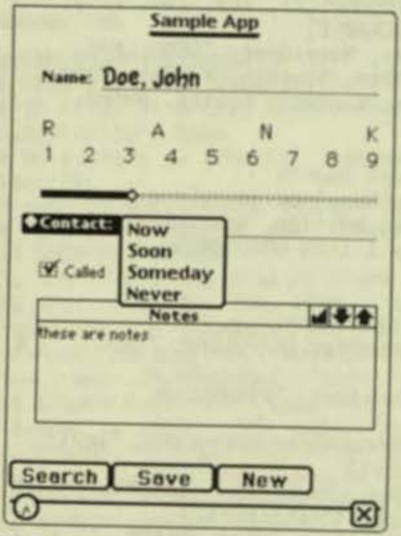
**Note:** Never use the same *windowSpec* variable in multiple WINDOW Statements without first assigning a new frame value to the variable. This is done in the example below with the spec variable. Each time this variable is used in a WINDOW Statement, it is initialized with a new frame containing the desired fields for the window.

The next section covers Frames, and the one after it discusses Files.

```
10 REM WIDGET Example
20 OPEN example, "ExampleFile", Name
30 IF FSTAT <> 0 THEN CREATE example,
"ExampleFile", Name
40 buttonFont := {family: 'geneva, face: 1,
size:12}
70 spec := {GOTO: 610, title: "Sample App"}
80 WINDOW MainWindow, spec, "APP"
110 SHOW MainWindow
130 REM create and display blank form
150 spec := {GOTO: 360, text: "Search",
viewBounds: SETBOUNDS(6, 280, 66, 294),
viewFont: buttonFont, viewJustify: 2,
viewFormat: 4*vfRound+2*vfPen+vfFrameBlack}
160 WINDOW SearchBtn, spec
170 spec := {GOTO: 490, text: "Save",
viewBounds: SETBOUNDS(70, 280, 130, 294),
viewFont: buttonFont, viewJustify: 2,
viewFormat: 4*vfRound+2*vfPen+vfFrameBlack}
180 WINDOW SaveBtn, spec
190 spec := {GOTO: 540, text: "New",
viewBounds: SETBOUNDS(134, 280, 191, 294),
viewFont: buttonFont, viewJustify: 2,
viewFormat: 4*vfRound+2*vfPen+vfFrameBlack}
200 WINDOW NewBtn, spec
210 NameSpec := {label: "Name:", viewBounds:
SETBOUNDS(9, 30, 230, 64), viewFlags:
VCLICKABLE}
220 WINDOW Name, NameSpec, "LABELINPUT"
230 RankLabelSpec := {text:"RANK\n1 2 3 4 5 6
7 8 9", viewJustify:3, viewFormat:
vfPen+vfFrameWhite, viewBounds: SETBOUNDS(21,
65, 230, 109), viewFlags: VCLICKABLE}
240 WINDOW RankLabel, RankLabelSpec,
"PARAGRAPH"
250 RankSpec := {viewValue:0, viewBounds:
SETBOUNDS(21, 110, 230, 119), viewFlags:
VCLICKABLE}
```

```
260 WINDOW Rank, RankSpec, "SLIDER"
270 ContactSpec := {label: "Contact:",
labelCommands: ["Now", "Soon", "Someday",
"Never"], viewBounds: SETBOUNDS(9, 120, 230,
144), viewFlags: VCLICKABLE}
280 WINDOW Contact, ContactSpec, "LABELINPUT"
290 CalledSpec := {text: "Called", viewBounds:
SETBOUNDS(21, 145, 110, 179), viewFlags:
VCLICKABLE}
300 WINDOW Called, CalledSpec, "CHECKBOX"
310 NotesSpec := {boxTitle:"Notes", text: "",
viewBounds: SETBOUNDS(21, 180, 230, 254),
viewFlags: VCLICKABLE}
320 WINDOW Notes, NotesSpec, "SCROLLER"
330 SHOW SearchBtn, SaveBtn, NewBtn, Name,
RankLabel, Rank, Contact, Called, Notes
340 Wait 1000
350 GOTO 340
360 REM user taps Search
370 searchKey = NameSpec.entryLine.text
380 GET example, editRec, searchKey
390 IF FSTAT <> 1 THEN GOTO 0420
400 BEEP 7
410 GOTO 340
420 REM found something!
430 SETVALUE(NameSpec.entryLine, 'text, "" &
editRec.name)
440 SETVALUE(RankSpec, 'viewValue,
editRec.rank)
450 SETVALUE(ContactSpec.entryLine, 'text, ""
& editRec.contact)
460 IF CalledSpec.viewValue <> editRec.called
THEN U.CalledSpec:TOGGLECHECK()
470 SETVALUE(NotesSpec.notes, 'text, "" &
editRec.notes)
480 GOTO 340
490 REM user taps Save
500 newRecord =
{name:NameSpec.entryLine.text,
rank:RankSpec.viewValue,
contact:ContactSpec.entryLine.text, called:
CalledSpec.viewValue, notes:
NotesSpec.notes.text}
510 PUT example, newRecord
520 IF FSTAT <> 0 THEN BEEP 7 ELSE BEEP 4
530 GOTO 0340
540 REM user taps New
550 SETVALUE(NameSpec.entryLine, 'text, "")
560 SETVALUE(RankSpec, 'viewValue, 0)
570 SETVALUE(ContactSpec.entryLine, 'text,
"")
580 IF CalledSpec.viewValue THEN x =
U.CalledSpec:TOGGLECHECK()
590 SETVALUE(NotesSpec.notes, 'text, "")
600 GOTO 340
610 REM user taps close box
620 HIDE
630 END
```

OUTPUT



## 4.3 Frames

The frame data structure is required for files and windows. It can be used for many other purposes as well. You can think of a frame as a container. You can add as many named items to the container as you'd like, and retrieve them by name in any order.

Our example shows creating a frame, adding several values to it, and then accessing those values:

```
10 REM frame Example
20 REM myUser is a variable holding
30 REM all the info for a user
40 myUser = {} // an empty container
50 PRINT "Enter your first name:"
60 INPUT name$
```

```
70 myUser.firstName = name$
80 PRINT myUser // see elements added
90 PRINT "Enter your last name:"
100 INPUT name$
110 myUser.lastName = name$
120 PRINT myUser // see another element!
130 PRINT "Enter your age, or S to Skip:"
140 INPUT age$
150 IF age$ = "S" THEN GOTO 170
160 myUser.age = STRINGTONUMBER(age$)
170 PRINT myUser // final form
180 PRINT "First Name: "; myUser.firstName
190 PRINT "Last Name: "; myUser.lastName
200 IF myUser.age = nil THEN GOTO 220
210 PRINT "Age: "; myUser.age
220 PRINT "Try again? (Y/N):"
230 INPUT ans$
240 IF ans$ = "Y" THEN GOTO 30
```

**OUTPUT**

```
Enter your first name:
? Jane
{firstname:"Jane"}
Enter your last name:
? Doe
{firstname:"Jane",lastname:"Doe"}
Enter your age, or Q to Quit:
? q
{firstname:"Jane",lastname:"Doe"}
First Name: Jane
Last Name: Doe
Try again? (Y/N):
? y
Enter your first name:
? John
{firstname:"John"}
Enter your last name:
? Doe
{firstname:"John",lastname:"Doe"}
Enter your age, or Q to Quit:
? 24
{firstname:"John",lastname:"Doe",AGE:24}
First Name: John
Last Name: Doe
Age: 24
Try again? (Y/N):
? n
```

We were able to add the named items firstName, `lastName`, and `age` to the frame simply by assigning a new named container inside the frame variable.

We also can test to see if a frame has a named item by testing to see if that item is NIL. Line 200 checks to see if there is an age item for the frame, and if not, skips the print statement on the next line. If you are testing a named item that contains a Boolean, you can use the HASSLOT Function.

You must use frames in order to write information to a file. The frames you write to a file may have different items stored in them.

### 4.4 Files

This section discusses the use of indices to quickly locate a particular entry in a file. We'll also use the techniques we just learned in the previous section to create a file of records with different elements in them.

The program below is an expanded version of a program that was first shown in the Reference Chapter for the CREATE Statement. We've expanded it in lines 40-140 to support entry of multiple records and both a key and a data field.

The retrieval section (lines 150-230) also retrieves as many records as you want.

```
10 REM File/Key retrieval Example
20 REM OPEN or CREATE a file...prompts for
some information, stores it, then allows
fetch.
30 OPEN chan, "EXAMPLEFile", keyname
40 IF FSTAT = 1 THEN CREATE chan,
"EXAMPLEFile", keyname
50 IF FSTAT =1 THEN GOTO 300
60 PRINT "Please enter a Key, Q to finish"
70 INPUT FileKey
80 IF FileKey = "Q" THEN GOTO 210
90 fileRecord = {}
100 fileRecord.keyname = FileKey
110 PRINT "Please enter some data for this
Key"
120 INPUT FileData
130 fileRecord.info = FileData
140 PRINT "Enter a number, or S to Skip:"
150 INPUT num$
160 IF num$ = "S" THEN GOTO 180
170 fileRecord.num = STRINGTONUMBER(num$)
```

```
180 PUT chan, fileRecord
190 IF FSTAT=1 THEN STOP
200 GOTO 60
210 PRINT "Please enter a Key to find, Q to
end"
220 INPUT FileKey
230 IF FileKey = "Q" THEN GOTO 290
240 GET chan,FetchedData,FileKey
250 IF FSTAT=1 THEN STOP
260 IF FSTAT=2 THEN PRINT "Not found! Close
Record is..." else PRINT "Data is..."
270 PRINT FetchedData
280 GOTO 210
290 END
300 REM error, cannot OPEN or CREATE file!
310 PRINT "Error! Cannot OPEN or CREATE
EXAMPLEfile."
320 END
```

Enter the following data into the program. We're not showing the prompts in the Handbook.

```
* RUN
? test
? mydata
? s
? abracadabera
? this IS data
? 100
? Zippy
? The Smallhead
? 47
? OK
? Middle of the DB
? -12.5
? q
Please enter a Key to find, Q to end
? abr
Not found! Close Record is...
{KEYNAME:"abracadabera",info:"this IS
data",num:100,_uniqueID:1}
Please enter a Key to find, Q to end
? abracadabera
Data is...
{KEYNAME:"abracadabera",info:"this IS
data",num:100,_uniqueID:1}
Please enter a Key to find, Q to end
? p
Not found! Close Record is...
```

```
{KEYNAME:"test",info:"mydata",_uniqueID:0}
Please enter a Key to find, Q to end
? q
* RUN
Please enter a Key, Q to finish
? Stimpy
Please enter some data for this Key
? Happy! Happy! Joy! Joy!
Enter a number, or S to Skip:
? s
Please enter a Key, Q to finish
? q
Please enter a Key to find, Q to end
? stimpy
Data is...
{KEYNAME:"Stimpy",info:"Happy! Happy! Joy!
Joy!",_uniqueID:4}
Please enter a Key to find, Q to end
? a
Not found! Close Record is...
{KEYNAME:"abracadabera",info:"this IS
data",num:100,_uniqueID:1}
Please enter a Key to find, Q to end
? q
*
```

There are several interesting things to look at in this program:

Lines 30-50 attempt to OPEN or CREATE a data file. If the OPEN fails, we try a CREATE. If that fails, we GOTO the end of the program and give up.

Lines 60-200 let the user input as many records as they want. We build our frame in the same way as shown in the previous section.

Lines 210-280 let the user enter a key to find. We check FSTAT after the GET Statement. If it is 2, then an exact match was not found.

If you look at the output, you'll note that some of the frames printed out (test and Stimpy) don't have an entry for num. This shows that you can store different kinds of frames in the same file, as long as they all have the required key entry.

We ran the program twice, and the data entered in the file was still there in the second run.

**Note:** NS BASIC adds the item __uniqueID to every frame that is PUT into a file. You should avoid using a field with this name in any frame you want to PUT in a file. Never change the value of this item in a frame you GET from a file.

### 4.5 Serial Input/Output

You can send and receive data via the serial port of your Newton. NS BASIC can PRINT to the serial port and INPUT from it. There are four steps to using the serial port:

**1** Initialize the port to the desired settings.

You control the settings for the serial port using the environment variable s0.

**2** Tell NS BASIC to use the serial port.

You control the input and output device (screen, serial port, or infrared port) using the environment variable IO.

**3** Exchange data or output information as desired using the serial port.

You control the prompt displayed by NS BASIC using the environment variable inputPrompt. You send data using PRINT and receive data using INPUT.

**4** Tell NS BASIC to stop using the serial port.

You control the input and output device (screen, serial port, or infrared port) using the environment variable IO.

It is very important that you perform these steps in this order. If your program quits unexpectedly and leaves the serial port open, other Newton applications may not be able to use the port. If this happens, reset your Newton.

The first example program accepts input via the serial port. This program leaves the serial port in its default configuration.

You will need to connect your Newton to a desktop computer to run this program. See the Using NS BASIC With a Computer or Terminal section of this Handbook for more information.

```
10 REM Serial Port Example
20 f={viewBounds:SETBOUNDS(100,100,130,110)}
30 tr = 0
40 translist = ""
50 CLS
60 WINDOW w1,f
```

```
70 SHOW w1
80 ENVIRON io="s0"
90 REM get a transaction
100 INPUT trans
110 tr = tr+1
120 translist = translist & trans & CHR(13)
130 WPRINT w1, tr
140 IF trans <> "BYE" THEN GOTO 90
150 ENVIRON io="screen"
160 PRINT translist
170 HIDE w1
```

The next example program dumps all of your appointments from the Calendar file to the serial port. The port is set to 4800 bps first. See the next section for more information on accessing Calendar and other built-in applications data.

```
10 REM Serial Dump Calendar Example
20 REM set serial params
30 currPort = env("s0")
40 currPort.data = [4800,8,1,"no"]
50 ENVIRON s0=currPort
60 ON ERROR GOTO 120
70 ENVIRON IO="s0" // switch to serial
80 OPEN ch,"calendar",mtgstartdate
90 GET ch, n
100 PRINT n // dump record
110 GOTO 90
120 REM EOF or other problem
130 ENVIRON IO="screen"
140 END
```

To reliably accept larger amounts of data from the serial port, a few things should be kept in mind. Keep the loop that accepts the data as short as possible, avoid doing PRINT and WAIT statements, and turn off the inputPrompt. For larger amounts of data, a delay between lines or a baud rate lower than 9600 may be necessary. Here an example that works well:

```
10 dim a[300]
15 environ inputPrompt=""
16 window w,ws
17 show w
20 for i=0 to 299
25 wprint w,i
30 input a[i]
35 if a[i]="END" then stop
```

```
40 next i
```

### 4.6 Infrared Input/Output

You can send and receive data via the infrared port of your Newton. NS BASIC can PRINT to the infrared port and INPUT from it. There are four steps to using the infrared port:

**1** Initialize the port to the desired settings.

You control the settings for the infrared port using the environment variable IR.

**2** Tell NS BASIC to use the infrared port.

You control the input and output device (screen, serial port, or infrared port) using the environment variable IO.

**3** Exchange data or output information as desired using the infrared port.

You control the prompt displayed by NS BASIC using the environment variable inputPrompt. You send data using PRINT and receive data using INPUT.

**4** Tell NS BASIC to stop using the infrared port.

You control the input and output device (screen, serial port, or infrared port) using the environment variable IO.

It is very important that you perform these steps in this order. If your program quits unexpectedly and leaves the infrared port open, other Newton applications may not be able to use the port. If this happens, reset your Newton.

### 4.7 Accessing and Using Other Files, Data, and Applications

As shown in the previous section, you can examine the contents of any file in your Newton. The table below lists the names of the files used with the built-in applications.

**Note:** You can read these files. If you write to or delete entries in these files you may lose data. Please be sure you have backed up your Newton prior to deleting or writing records to these files.

### Dates

| File Name | Key Name | Contents |
|---|---|---|
| calendar | mtgStartDate | daily meetings that don't repeat |
| repeat meetings | mtgStopDate | daily meetings that repeat |
| calendar notes | mtgStartDate | day notes (written to the left of the calendar) that don't repeat |
| repeat notes | mtgStopDate | day notes that repeat |
| to do | | to-do entries |

### Note Pad

| File Name | Key Name | Contents |
|---|---|---|
| notes | timestamp | notepad data |

### Names

| File Name | Key Name | Contents |
|---|---|---|
| names | sortOn | name data |

You can write a small program that OPENs these files, GETs an entry, and PRINTs it. You can examine the output to learn the item names within these files, and their contents. This program displays an entry from the calendar file:

```
10 OPEN ch,"calendar",mtgstartdate
20 GET ch, n
30 PRINT n // dump record
```

OUTPUT

```
{viewStationery:Meeting,mtgStartDate:4770198
0,mtgDuration:150,mtgText:"Some Meeting
Text!",mtgAlarm:47701970,_uniqueID:35,_modti
me:47567291}
```

You can also examine and modify data within other applications installed on your Newton.

**Note**: As with files, extreme care should be taken when accessing, changing, or calling other applications. Data loss is possible. Backup your data.

The GETROOT() Function returns the root frame of your Newton. Every application currently installed on your Newton is accessible via this frame. Because of their large size, you should always use the special := assignment operator to point to this frame.

The first example program will display all of the items stored in the root frame. The output of this list is quite large, and will not be reproduced in this Handbook. This program is completely safe. You can enter and run it on your Newton with no risk of data loss.

```
10 REM getroot() Example
20 rootFrame := getroot()
30 rootElements := elements(rootFrame)
40 numElements = length(rootElements)-1
50 FOR i = 0 TO numElements
60 PRINT rootElements[i]
70 NEXT i
```

The next example will open the Names application, and is also completely safe:

```
10 getroot().cardFile:open()
```

The :open() in the above program tells the Names application to open as if you had tapped Names. The built-in applications respond to :open() and :close(). The names of the built-in applications are paperRoll (Note Pad), calendar (Dates), cardFile (Names), extrasDrawer (Extras), and assistant (Assist).

The last example program will display all of the fields in your user configuration frame. The output of this list is quite large, and will not be reproduced in this Handbook. This program is completely safe. You can enter and run it on your Newton with no risk of data loss.

```
10 REM GETROOT().userConfiguration Example
20 rootFrame :=
GETGLOBALS().userConfiguration
30 rootElements := ELEMENTS(rootFrame)
40 numElements = LENGTH(rootElements)-1
50 FOR i = 0 TO numElements
```

```
60 PRINT rootElements[i]
70 NEXT i
```

There are several built-in functions that are not documented in this handbook that my be used in NS BASIC.

**Warning:** Many of the built-in functions are not documented. Others may freeze your Newton. Do not try to use any function you do not understand. Be aware that the use of these functions may cause data loss. Not all Newton devices may have the same built-in functions.

There are a number of sources for detailed information about the Newton. If you have access to an on-line service, check and see if there is a Newton programming section.

### 4.8 Handling Errors

In a perfect world, there are never any errors. Our programs are seldom perfect worlds! We protect our users (often ourselves) from many errors using two techniques: defensive programming and ON ERROR.

The basic idea with error handling is to anticipate which parts of your program could have run-time errors. and to set up special program code to deal with it.

### Defensive programming

You can use the CLASSOF Function to verify that a variable contains the expected data type following an INPUT or READ Statement.

You can check the value of FSTAT after each file input and output statement.

You can verify that numeric values are within the valid range before using them in numeric expressions. For example, you can check that a variable is not zero before using it as the divisor.

### Using ON ERROR

If you are prompting the user to enter a numeric value, the user may well enter a string. Your program can do one of two things at that point: spit out the standard NS BASIC error message and halt, or print a message that gently reminds the user to enter a number and re-prompt for input again.

Let's look at both and see how they work.

**NO ERROR CHECKING**

```
10 REM simple user entry of a number
20 REM without error checking
30 PRINT "enter your age"
40 INPUT age
50 dogAge = age * 7
60 PRINT "You are "; dogAge; " in Dog Years!"
* RUN
enter your age
? 33
You are 231 in Dog Years!
* RUN
enter your age
? fred
0050 :Error 29 - Expression
```

Everything looked fine until the user entered **fred**. Then we got this cryptic error message. Add the following code to catch the error and deal with it in a more user-friendly way:

```
* 42 ON ERROR GOTO 100
* 52 ON ERROR GOTO 0
* 70 END // don't run into error handler
* 100 REM This error handler is for invalid
entry
* 105 BEEP 1 // error feedback
* 110 PRINT age; " is not a valid age, try
again."
* 120 GOTO 30
```

Now that we are prepared for incorrect input, the program behaves well. Let's RUN it and see:

```
* RUN
enter your age
? fred
fred is not a valid age, try again.
enter your age
? 100
You are 700 in Dog Years!
*
```

You could also use defensive programming to avoid this error. One technique is to use the CLASSOF Function to test the class of the value INPUT by the user:

```
42 IF CLASSOF(age) <> 'int and CLASSOF(age) <>
'real THEN GOTO 100
```

Alternatively, you can accept the INPUT data as a string by using a variable ending in a $ sign. Use the STRINGTONUMBER Function to convert that string into a number. If the result is NIL then the user did not enter a valid number.

Either of these approaches avoids the need for ON ERROR handlers.

The technique shown above can handle every run-time error in NS BASIC. The strategy is simple. Just before you do an operation that may fail, use an ON ERROR Statement to set up a branch to a specific error handler. Just as soon as you have passed the section that may fail, reset the error handler to the default. Always end your program with an END Statement to avoid running into your error handler code.

Your error handler code may:

- Display a helpful message and retry the operation, using GOTO to return to the section of code (like our example above),
- Correct the error by setting one or more variables to some default value (i.e., you can limit an input to a maximum value) and then return to the section that failed via GOTO, or
- Display an error message of your own design, perform some clean up (perhaps update a file) and then end the program.

## 4.9 Calling NS BASIC from NewtonScript

Several Newton Applications (the spread sheet program QuickFigure, for example) allow you to enter NewtonScript code fragments. You may call NS BASIC and run a program from these programs. Your NS BASIC program can return a value using the BYE statement. You can pass a value into your program from the NewtonScript code fragment as well. This value is placed into a variable named CHAINPARAM.

**Note:** You should avoid using a variable named CHAINPARAM for your own purposes.

This example program in NS BASIC simply increments the value passed in via CHAINPARAM, and returns it in the BYE Statement: Save it with the name `calltest`.

```
10 REM calltest
20 BYE chainParam+1
```

If you want to call this program from QuickFigure, you would insert the following NewtonScript code fragment into a cell:

```
=GETROOT().|basic:NSBASIC|:chain("calltest",
123);
```

When you press return, the NS BASIC program is executed and the new value (124) is placed into the cell.

The general form of the NewtonScript expression to call NS BASIC is:

GETROOT().|basic:NSBASIC|:chain(*programName*, *chainValue*)

Where *programName* is a string that contains the name of the program to run, without the .BAS extension, and *chainValue* is the value to place in CHAINPARAM. NS BASIC runs silently without changing the screen, unless you execute a PRINT or SHOW statement.

You can execute any kind of NS BASIC program, with one proviso: if the NS BASIC program halts at any point, to get input from the user for example, the calling NewtonScript code fragment will receive a return value of NIL. In other words, as long as you are computing values, accessing files, or creating displays, the calling code fragment will not receive a return value until the BYE Statement is reached. If your program executes an INPUT, HWINPUT, or WAIT Statement, the calling code fragment will receive a NIL return value at that point in your NS BASIC program.

APPENDIX

# A

## A. Error Messages

### Compile and Run-Time

Error 1 - Incorrect Data Type

The Statement or Function expects data of a different type. Refer to the Reference Chapter of this Handbook for the expected data type.

Error 2 - Statement or syntax invalid

NS BASIC cannot understand the Statement.

Error 4 - Invalid Checksum on Runtime

The RunTime program file is damaged.

Error 5 - Statement Number

An invalid line number was used.

Error 8 - Renumber overlap

When RENUMing a partial range of Statements, the new number overlaps an existing program Statement.

Error 11 - Parenthesis

Mismatched parenthesis.

Error 13 - Line Number

Invalid line number was used.

Error 14 - Out of Memory

You have run out of memory. Reset your Newton to free more memory.

**Note:** All variable values are cleared after this error.

Error 15 - End of DATA

A READ Statement attempted to read past the last element of your DATA Statements.

Error 16 - Arithmetic

Numeric overflow or underflow.

**Note:** Divide by Zero does not cause an error on Newton 2.0 units.

Error 19 - RETURN - No GOSUB

There is a RETURN that is missing the GOSUB.

Error 22 - NEXT - No FOR

There is a NEXT without a FOR, or the program has branched to inside of a FOR NEXT loop.

Error 29 - Expression

NS BASIC cannot understand the Expression. Try to break complex Expressions into multiple Statements.

Error 30 - Object is read only

An attempt to change a value of or add an item to a system frame. Frames retrieved using GETROOT() are often read only.

Error 31 - Subscript or Frame error

Access to an array element that is larger than the array, or a frame item that does not exist.

Error 46 - Input Error

The user entered more items (separated by commas) than were expected.

Error 48 - Incorrect SAVE version

NS BASIC may change the internal form of SAVEd programs. If you get this error, use ENTER to load the program and then SAVE the new version.

Error 59 - Zero step

A FOR loop has a Zero step.

Error 63 - Incorrect number of args

The Statement or Function expects a different number of arguments. Refer to the Reference Chapter of this Handbook for the expected number of arguments.

**File**

I/O Error 1 - Illegal file name

The file name used is not valid. File names cannot have spaces, but may contain both upper and lower case letters, as well as special characters like underscore (_) and hyphen (-).

I/O Error 2 - Illegal key

The data type of the key is not correct for the index of a file.

I/O Error 3 - Opened without keys

A GET with a key was attempted on a file that was OPENed without a key.

I/O Error 4 - Incorrect key type

A GET or PUT was attempted that specified a key of the wrong type.

I/O Error 5 - File already Exists

SAVE specified a name already used for an existing program.

I/O Error 6 - End of file

A GET was attempted after the last record was read. Use an ON ERROR handler to detect and handle the end of file when reading every record in a file.

I/O Error 10 - File not found

A file name was specified for a file or program and it does not exist.

I/O Error 12 - no key on OPEN

A PUT with a key was attempted on a file that was OPENed without a key.

I/O Error 13 - Channel not open

A GET or PUT was attempted using a channel that was not returned from CREATE or OPEN

I/O Error 14 - Error creating file

A problem (most often out of space, or card read-only) occurred while attempting to CREATE a file.

# APPENDIX B

## B. Keywords

The following list of keywords are reserved for the use of NS BASIC and should not be used as variable names.

AND
BEEP
BYE
CHAIN
CHAINPARAM
CLOSE
CLS
CON
CREATE
DATA
DEF
DEL
DELETE
DIM
DIR
EDIT
ELSE
END
ENTER
ENVIRON
ERASE
ERROR
FOR
FUNCTION
GET
GOSUB
GOTO
HIDE
HWINPUT
IF
INPUT
LET
LIST
LOAD
MAKEPACKAGE
NEXT
NIL
NOT
OFF
ON
OPEN
OR
PRINT
PUT
RANDOMIZE
READ
REM
REPLACE
RESTORE
RETURN
REVUP
RUN
RM
SAVE
SHOW
STATS
STOP
THEN
TRUE
TRACE
VARS
WAIT
WDRAW
WINDOW
WPRINT
WSTAT

## C. Special Character Codes

These special characters may be generated using the CHR Function.

| | | | | | |
|---|---|---|---|---|---|
| 160 | umlaut | 201 | É | 238 | î |
| 161 | ¡ | 202 | Ê | 239 | ï |
| 162 | ¢ | 203 | Ë | 241 | ñ |
| 163 | £ | 204 | Ì | 242 | ò |
| 164 | ¤ | 205 | Í | 243 | ó |
| 165 | ¥ | 206 | Î | 244 | ô |
| 167 | ❖ | 207 | Ï | 245 | õ |
| 168 | ¨ | 209 | Ñ | 246 | ö |
| 169 | © | 210 | Ò | 247 | ÷ |
| 170 | ª | 211 | Ó | 248 | ø |
| 171 | « | 212 | Ô | 249 | ù |
| 172 | ¬ | 213 | Õ | 250 | ú |
| 174 | ® | 214 | Ö | 251 | û |
| 175 | ¯ | 216 | Ø | 252 | ü |
| 176 | ° | 217 | Ù | 255 | ÿ |
| 177 | ± | 218 | Ú | 305 | ı |
| 180 | ´ | 219 | Û | 338 | Œ |
| 181 | µ | 220 | Ü | 339 | œ |
| 182 | ¶ | 223 | ß | 376 | Ÿ |
| 183 | · | 224 | à | 402 | ƒ |
| 184 | ¸ | 225 | á | 8706 | ∂ |
| 186 | º | 226 | â | 8710 | ∆ |
| 187 | » | 227 | ã | 8719 | ∏ |
| 191 | ¿ | 228 | ä | 8721 | ∑ |
| 192 | À | 229 | å | 8730 | √ |
| 193 | Á | 230 | æ | 8734 | ∞ |
| 194 | Â | 231 | ç | 8747 | ∫ |
| 195 | Ã | 232 | è | 8776 | ≈ |
| 196 | Ä | 233 | é | 8800 | ≠ |
| 197 | Å | 234 | ê | 8804 | ≤ |
| 198 | Æ | 235 | ë | 8805 | ≥ |
| 199 | Ç | 236 | ì | | |
| 200 | È | 237 | í | | |

# INDEX

# USER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications. Please let us know if you would like a reply. Return to:

NS BASIC Corporation
77 Hill Crescent
Toronto, Canada MIM IJ3
fax (416) 264-5888

| Page | Comments |
|---|---|
| | |

Release 3.0

1086182

REM
GOSUB
LET

*NS* BASIC

*NS* BASIC Corporation
77 Hill Crescent, Toronto, Canada M1M 1J3
Telephone: (416) 264-5999 Fax: (416) 264-5888
gh@hookup.net

ISBN 0 - 969 5844 - 1 - 5