Newton Developer Tools

Newton C++ Tools Programmer's Reference

May 16, 1996

Printed in the United States of America.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings vii

Preface **About This Book** ix

How to Use This Book ix
Related Books x
Conventions xi
Developer Products and Support xi

Chapter 1 **C++ Toolkit Introduction** 1-1

Using C++ With NewtonScript 1-1
  Calling NewtonScript from C++ 1-2
  Calling C++ from NewtonScript 1-3
C++ Code Restrictions 1-4
  Methods, Functions, and Name-Mangling 1-4
  Memory Allocation 1-4
  Function Arguments and Return Values 1-6
The Newton Object System 1-6
  Newton Symbols and Object Types 1-6
  Object References 1-7
  Accessing Data In a Binary Object 1-10
  NewtonScript Magic Pointers 1-11
  Path Expressions 1-12
  Newton Exceptions and C++ 1-12
NewtonScript and C++ Equivalences and Examples 1-13
  A Simple Example in NewtonScript and C++ 1-14
  An Example of Defining and Calling Several C++ Functions 1-15
  An Example of a Wrapper Function 1-16
  An Example of Converting a C++ Array into NewtonScript 1-16
  An Example of Automatic Allocation of RefArgs 1-17
  An Example of Allocating Persistent Storage 1-18
  An Example of Accessing Binary Data 1-19

Chapter 2 **C++ and NewtonScript Conversion Reference** 2-1

Constants for Using C++ With NewtonScript 2-1
Type Conversion Functions 2-1
Type Checking Functions 2-4
Value Checking Functions and Macros 2-5

Debugging Macros 2-6
Summary of C++ and NewtonScript Conversion Reference 2-7
Constants for Using C++ With NewtonScript 2-7
Type Conversion Functions and Macros 2-7
Type Checking Functions 2-7
Value Checking Functions and Macros 2-7
Debugging Functions and Macros 2-8

Chapter 3 Newton Object System Reference 3-1

Object System Classes 3-1
Iteration Macros 3-1
Object Iterator Class 3-4
Iterator Functions 3-5
C++ Object System Functions 3-5
Summary of Object System Reference 3-22
Object System Classes 3-22
Object Iterator Class 3-22
Newton Object System Functions and Macros 3-22
Iterator Functions 3-22
Iteration Macros 3-22
C++ Newton Object Functions 3-22

Chapter 4 Newton Memory Manager Reference 4-1

About the Newton Memory Manager 4-1
Memory Manager Functions 4-1
Summary of Memory Manager Reference 4-9
Memory Manager C++ Functions 4-9

Chapter 5 Newton Exceptions Reference 5-1

About Newton Exceptions 5-1
Defining Exceptions 5-1
Exception Data 5-3
Exception Blocks 5-4
Volatile Values 5-5
Newton System Software Exceptions 5-5
Exception Types 5-6
Exception Functions and Macros 5-6
Exception-Handling Macros 5-9
Summary of Exceptions Reference 5-13
Exception C++ Functions 5-13

Functions and Macros to Define and Throw Exceptions 5-13
Exception-Handling Macros 5-13

Chapter 6 NewtonScript Reference 6-1

NewtonScript Interpreter Functions 6-1
Functions for Calling NewtonScript Functions From C++ 6-1
Functions for Accessing NewtonScript Slot Values from C++ 6-14
Calling C++ Functions from NewtonScript 6-15
Summary of NewtonScript Interpreter Functions 6-17
Functions for Calling NewtonScript Functions From C++ 6-17
NSCall 6-17
NSCallGlobalFn 6-17
NSSend 6-17
NSSendIfDefined 6-18
NSSendProto 6-18
NSSendProtoIfDefined 6-19
Functions for Accessing NewtonScript Slot Values from C++ 6-19

Chapter 7 Newton Unicode Reference 7-1

.Unicode Constants and Data Types 7-1
The UniChar Type 7-1
Encoding Type Constants 7-1
Unicode Character and String Constants 7-2
Unicode Functions 7-2
Summary of Unicode Reference 7-9
Unicode Data Types 7-9
Encoding Type Constants 7-9
Unicode Character and String Constants 7-9
Unicode Functions 7-9

Chapter 8 Newton C Library Reference 8-1

C Library Constants and Data Types 8-1
C Library Constants 8-1
Standard Library Types 8-2
Math Types 8-3
Time Types 8-4
C Library Functions 8-5
Character Conversion Functions 8-5
Floating-point Math Functions 8-6
Financial Functions 8-17

Variable Argument List Macros 8-17
Standard Input and Output Functions 8-18
Standard C Library Functions 8-20
Heap Functions 8-23
Memory Block Manipulation Functions 8-25
String Manipulation Functions 8-26
Time Functions 8-30
Summary of C Library Reference 8-34
C Library Constants and Types 8-34
Standard Library Types 8-34
Math Types 8-34
Time Types 8-34
C Library Functions 8-35
Character Conversion Functions 8-35
Floating-point Math Functions 8-35
Financial Functions 8-37
Variable Argument List Macros 8-37
Standard Input and Output Functions 8-37
Standard C Library Functions 8-37
Heap Functions 8-38
Memory Block Manipulation Functions 8-38
String Manipulation Functions 8-38
Time Functions 8-39

Appendix A C++ Function Tables A-1

Functions and Macros for Using C++ With NewtonScript A-1
Newton Object System Functions A-2
C++ Toolkit Memory Manager Functions A-6
C++ Toolkit Exception-Handling Functions A-8
C++ NewtonScript Functions A-9
C++ Toolkit Unicode Functions A-10
C++ Toolkit ANSI-C Functions A-11

Figures, Tables, and Listings

Chapter 1 — C++ Toolkit Introduction 1-1

Table 1-1 Newton object types 1-7
Table 1-2 Summary of C++ Toolkit reference types 1-8
Table 1-3 Examples of object reference use 1-9
Listing 1-1 Using a locked pointer to access a binary object 1-10
Table 1-4 Path expressions 1-12
Listing 1-2 Working with a C++ object in an exception block 1-13
Table 1-5 NewtonScript expressions and their C++ equivalences 1-13
Listing 1-3 A NewtonScript `search` function 1-14
Listing 1-4 C++ version of the `search` function 1-14
Listing 1-5 Defining a C++ function in a module 1-15
Listing 1-6 Calling a C++ function from NewtonScript 1-16
Listing 1-7 A wrapper function for a C++ function callable from NewtonScript 1-16
Listing 1-8 Converting a C++ array into a NewtonScript object 1-16
Listing 1-9 An example of inefficient automatic allocation of `RefArgs` 1-17
Listing 1-10 An example of a more efficient `RefArg` loop 1-17
Listing 1-11 NewtonScript code for using a binary object as persistent storage for C++ 1-18
Listing 1-12 C++ code for using a binary object as persistent storage 1-18
Listing 1-13 Accessing binary data 1-19

Chapter 2 — C++ and NewtonScript Conversion Reference 2-1

Chapter 3 — Newton Object System Reference 3-1

Listing 3-1 An example of using the FOREACH macro 3-2
Listing 3-2 An example of using the FOREACH_WITH_TAG macro 3-3

Chapter 4 — Newton Memory Manager Reference 4-1

Chapter 5 — Newton Exceptions Reference 5-1

Table 5-1 An exception-handling hierarchy 5-3
Table 5-2 Newton system software exceptions 5-5
Listing 5-1 Using the `newton_try`, `newton_catch`, and `end_try` macros 5-9
Listing 5-2 Using the `newton_catch_all` macro 5-11
Listing 5-3 Using the `unwind_protect`, `on_unwind` , and `unwind_end` macros 5-12

Chapter 6 NewtonScript Reference 6-1

Chapter 7 Newton Unicode Reference 7-1

Table 7-1 Unicode punctuation symbols 7-6

Chapter 8 Newton C Library Reference 8-1

Appendix A C++ Function Tables A-1

Table A-1 C++ and NewtonScript conversion functions and macros A-1
Table A-2 C++ Toolkit Object System functions A-2
Table A-3 C++ Toolkit Memory Manager functions A-6
Table A-4 C++ Toolkit exception-handling functions A-8
Table A-5 C++ Toolkit NewtonScript functions A-9
Table A-6 C++ Toolkit Unicode functions A-10
Table A-7 C++ Library ANSI-C Library functions A-11

About This Book

This book describes the C++ Toolkit, which allows you to develop code in the C++ language that can be included in a NewtonScript application. This book documents the collection of C++ functions and data types that you can use to interface with the Newton.

IMPORTANT

The C++ Toolkit software allows you to use C++ code in a NewtonScript application. You must understand the Newton progamming environment before using the C++ Toolkit. If you have never written a Newton application, you need to read the *Newton Programmer's Guide: System Software* and *The NewtonScript Programming Language*. This book only explains those parts of the Newton programming environment that are unique for C++ programming. ▲

How to Use This Book

This book is a reference guide to the functions, data types, and constants that the C++ Toolkit provides. Many of the functions of the C++ Toolkit provide the same functionality as the functions of the NewtonScript programming language.

To learn about specific tools, menu choices, and options in the programming environment for including C++ code in your NewtonScript application, refer to the *Getting Started with C++ Tools* document.

The NewtonScript documentation describes the Newton programming environment and provides a wealth of how-to information for developing Newton applications. To learn more about programming the Newton, refer to the *Newton Programmer's Guide: System Software*.

This book contains eight chapters and one appendix:

- Chapter 1, "Introduction," provides an overview of how C++ programs interact with the NewtonScript world.
- Chapter 2, "C++ and NewtonScript Conversion Reference," describes the constants, data types, and functions that you can use to convert objects between NewtonScript and C++.
- Chapter 3, "Object System Reference," describes the C++ functions that you use to manipulate Newton objects.
- Chapter 4, "Memory Manager Reference," the C++ functions that you use to work with the Newton memory manager.

- Chapter 5, "Exceptions Reference," describes the C++ functions that you can use to raise and handle exceptions during the execution of your Newton applications.
- Chapter 6, "NewtonScript Reference," describes the programming interface that you can use from your C++ programs to call into the NewtonScript interpreter. It also explains how to structure your C++ functions to allow NewtonScript applications to call them.
- Chapter 7, "Unicode Reference," describes the C++ constants, data types, and classes that you use to manipulate Unicode strings.
- Chapter 8, "C Library Reference," describes the constants, data types, and functions from the C Library that you can use with your Newton programs.
- Appendix A, "C++ Function Tables," provides tables that show the location of the header and description for each function in the C++ Toolkit.

Related Books

This book is a standalone book that describes the C++ functions that you can use with your NewtonScript applications for the Newton. For more information about the Newton programming environment, refer to:

- *Newton C++ Tools for the Mac OS User Guide*. This book describes the development environment and tools that you use to implement C++ code for the Newton.
- *Newton Programmer's Guide*. This book is the definitive guide and reference for Newton programming. It explains how to write Newton programs and describes the system software routines that you can use to do so.
- *The NewtonScript Programming Language*. This book describes the NewtonScript programming language.

Conventions

This book uses the following font and syntax conventions:

`Courier`	The Courier font represents material that is typed exactly as shown. Code listings, code snippets, and special identifiers in the text such as predefined system frame names, slot names, function names, method names, symbols, and constants are shown in the Courier typeface to distinguish them from regular body text.
italics	Text in italics represents replacable elements, such as function parameters, which you must replace with your own values.
boldface	Key terms and concepts are printed in boldface where they're defined. Words defined in this book appear in the glossary in the *An Introduction to Newton Driver Development Kits.*
…	An ellipsis in a syntax description means that the preceding element can be repeated one or more times.
...	An ellipsis in a code example represents code not shown.
[ ]	Square brackets enclose optional elements in syntax descriptions.

Developer Products and Support

APDA is Apple's worldwide source for a large number of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Every four months, customers receive the *APDA Tools Catalog* featuring current versions of Apple's development tools and the most popular third-party development tools. Ordering is easy; there are no membership fees, and application forms are not required for most products. APDA offers convenient payment and shipping options including site licensing.

To order product or to request a complimentary copy of the *APDA Tools Catalog*:

APDA
Apple Computer, Inc.

P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	APDA
America Online	APDA
CompuServe	76666,2405
Internet	APDA@applelink.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

C++ Toolkit Introduction

This chapter introduces the C++ Toolkit, which allows you to use C++ code in NewtonScript applications. This chapter describes the details of interfacing your C++ code into the NewtonScript world. The remainder of this book provides reference descriptions of the data types and functions that you can use in your C++ code to interface with the NewtonScript world.

This chapter begins with an overview of using C++ and NewtonScript together. It then describes the Newton object system from the C++ developer's perspective and discusses the restrictions that you face in your C++ code when using the C++ Toolkit. Finally, this chapter presents a table of NewtonScript and C++ code equivalencies and provides a number of examples of using C++ with NewtonScript.

IMPORTANT
The C++ Toolkit software allows you to use C++ code in a NewtonScript application. You must understand the Newton progamming environment before using the C++ Toolkit. This book only explains those parts of the Newton programming environment that are unique for C++ programming. If you have never written a Newton application, you need to read the *Newton Programmer's Guide* and *The NewtonScript Programming Language*. You should also understand the development environment that you need to use for implementing your C++ code, which is described in *C++ Tools for the Mac OS User Guide*. ▲

Using C++ With NewtonScript

The purpose of the C++ Toolkit is to allow you to mix C++ code with NewtonScript code to create applications for the Newton. While you can manipulate objects and perform computations in C++, the user interface and main body of your Newton applications must be written in NewtonScript.

Writing C++ code for the Newton is the same as writing C++ for other computing devices; however, you do face the following important restrictions:

- The Newton does not have a file system, which means that your C++ code cannot make file system calls.
- Memory management capability is limited, as described in the section "C++ Code Restrictions" beginning on page 1-4.
- You cannot modify the Newton screen from your C++ code. You must use NewtonScript or call into NewtonScript from C++ to "talk" to the screen.

This book describes the C++ functions that you can use to manipulate objects in the Newton object system and mechanisms that you can use to call NewtonScript functions from C++. To use the C++ Toolkit, you need to understand the NewtonScript language, the Newton object system, and how to build Newton programs using the Newton Toolkit. To learn about Newton programming, read the *Newton Programmer's Guide*. To learn about the NewtonScript language, read the *NewtonScript Programming Language*. To learn about the Newton Toolkit, read the *Newton Toolkit User's Guide*.

To use C++ with NewtonScript, you need to utilize two mechanisms: calling C++ from NewtonScript, and calling NewtonScript from C++. You also need to understand how the representation of certain objects in NewtonScript is different from their representation in your C++ programs. This chapter describes those differences. Chapter 2, "C++ and NewtonScript Conversion Reference," describes the C++ functions that you can use to convert between these two representations.

IMPORTANT
This section provides information that you need to understand when you mix C++ code with NewtonScript code. Read this section carefully. ▲

Calling NewtonScript from C++

To call into NewtonScript from C++, you can use the `NSCall` function or one of its variants, which are described in Chapter 6, "NewtonScript Reference."

Some NewtonScript functions are implemented as C++ functions to improve their performance. You call these functions directly in C++ without using `NSCall` or its variants. All of the functions that you can call in this manner are documented in this book and are listed in the tables in Appendix A, "C++ Function Tables.".

If you want to call a NewtonScript function from your C++ code, you should first determine if a C++ implementation exists for the function, using either this book's index or the tables in Appendix A, "C++ Function Tables.". If the function is described in this book, use that function as documented. If a C++ version does not exist, call the NewtonScript function using `NSCall` or one of its variations, which are described in Chapter 6, "NewtonScript Reference."

Note
This book shows the declaration for each function implemented in the C++ Toolkit and provides descriptions for functions that are unique to the C++ Toolkit. This book does not provide descriptions for C++ Toolkit functions that are equivalent to NewtonScript functions. You will need to refer to the *Newton Programmer's Guide* for descriptions of these functions, as noted with the declaration of each. ◆

Calling C++ from NewtonScript

You can call a C++ function from NewtonScript just like you would call any other function in NewtonScript. However, the your usage of C++ functions is restricted in the following ways:

- You must preface the function name with its module designator, as described in the next section, "C++ Modules."
- The first parameter to the C++ function must be a reference to the receiver frame for the function. Note that the NewtonScript caller does not see supply this parameter.
- The C++ function can take from zero to six arguments; each argument must be delcared as type `RefArg.`
- The C++ function must return a `Ref` as its function result.
- The C++ function must be a standalone function (not a method of a class).
- The C++ function must be declared as `extern "C".`

The restrictions listed above are explained in more detail in the next section, "C++ Code Restrictions."

The following is an example of a C++ function that can be called from NewtonScript:

```
extern "C" Ref MyCplusFunction(  RefArg   receiver,
                                 RefArg   firstArg,
                                 RefArg   secondArg);
```

The following is an example of a NewtonScript expression for calling the C++ function in the above example:

```
call myModule.MyCplusFunction with (firstarg, secondArg);
```

C++ Modules

For NewtonScript to access your C++ functions, you must use a module designator when you call the C++ function. The module designator consists of the module name for the function, followed by a period.

You must define the module name in your MPW project exports (`".exp"`) file. The default name of the module, which is set by MPW when you create the project, is your project name.

For example, to call a C++ function named `myFcn` that is defined in the project `myProject`, your NewtonScript code would call `myProject.myFcn`. The section "An Example of Defining and Calling Several C++ Functions" beginning on page 1-15 shows a C++ module that defines several functions and NewtonScript code for calling those functions.

C++ Code Restrictions

This section describes the limitations that you face when developing C++ code for use with NewtonScript.

Methods, Functions, and Name-Mangling

The C++ language allows you to create classes that include methods. You can also define overloaded methods, which means that a single method name can be used for different declaration forms. For example, a single method can be declared to take different numbers or combinations of parameters or to return different value types.

Many C++ compilers implement this language feature using a technique that is commonly known as *name-mangling*. With name-mangling, the compiler builds an internal name for each declaration form of a method. The internal (mangled) name includes the method's class name and a representation of its parameter and return types. This makes it possible for a method to be called in various forms while retaining the type-checking capabilities of the C++ language.

Unfortunately, calling a C++ method whose name has been mangled by a compiler is not supported from NewtonScript. Due to this restriction, you cannot call a C++ method from NewtonScript; you can only call a standalone function. Furthermore, you must declare the function as `extern "C"`, which tells the compiler to not mangle the function name.

IMPORTANT
You can only call standalone C++ functions (not methods of a class) from NewtonScript. These functions must be declared as `extern "C"`. For example:

```
extern "C" Ref ReturnIt(RefArg rcvr)
```

▲

Memory Allocation

You are limited to a subset of the standard C library memory allocation and deallocation functions.

You need to know that the Newton system software uses two heaps: one for NewtonScript objects and another for system storage and C++ usage. Whenever you perform an allocation from C++ (by calling a function such as `malloc` or `NewPtr`), storage is allocated in the system heap. This means that if your C++ code runs out of heap space, the entire system software heap is out of space. You must be diligent about explicitly disposing of any storage that you allocate in your C++ code.

WARNING

The heap that you use to allocate storage in your C++ code is the same heap that the Newton system software uses for system-related objects. If you corrupt the heap, the Newton will need to be restarted. ▲

Storage for an object in the NewtonScript heap is automatically reclaimed by the Newton garbage collector when there are no longer any references to the object. If you are referring to a NewtonScript object from C++, the Newton garbage collector needs to know about your references. You accomplish this by using the object reference classes and types, which are described in the section "Object References" beginning on page 1-7.

For more information about using memory allocation and deallocation functions in your C++ code, refer to Chapter 4, "Newton Memory Manager Reference."

Static Variables

You cannot declare any static C variables. You cannot have any C++ static class variables.

WARNING

Although MPW generates an error if your code contains a static variable declaration, neither the C++ compiler nor the linker will tell you where in your code the problem exists. ▲

Global Data

Any global data that you reference in your C++ functions must be read-only data. You must reference this data with a constant pointer to constant data, which you can declare as follows:

```
const *const globPtr;
```

Allocating Persistent Storage

You sometimes need to allocate memory for use in your C++ code that is like global data. Since you cannot use non-constant global data in your C++ code, you need to utilize a coordinated effort between your NewtonScript and C++ code to achieve this.

The preferred method for allocating memory that you can use in this way is to create a binary object for the memory in your NewtonScript code. You then access the memory as a binary object from C++. By using this method, you don't need to concern yourself with deallocating the memory—the NewtonScript garbage collector will automatically collect the storage when there are no longer any references to the binary object.

The section "An Example of Allocating Persistent Storage" beginning on page 1-18 shows sample code in NewtonScript and C++ for using a binary object to create persistent storage for use in C++.

Function Arguments and Return Values

All arguments to your C++ functions that can be called from NewtonScript must be of type `RefArg`. The return value from each of your C++ functions that can be called from NewtonScript must be of type `Ref`. This means that the return value and each of the arguments must be NewtonScript objects.

Typically, you will need to implement a "wrapper" function for any C++ function that you want to call from NewtonScript. Your wrapper function can call the Newton conversion functions to convert data types. These functions are described in the section "Type Checking Functions" beginning on page 2-4.

The section "An Example of a Wrapper Function" beginning on page 1-16 shows sample code for creating a wrapper function for a C++ function that you want to call from NewtonScript.

If you need to convert a C++ array structure into a NewtonScript object, you can call functions to create a NewtonScript array or a NewtonScript frame. You can then add objects to the array or frame with other calls. These functions are described in Chapter 3, "Newton Object System Reference."

The section "An Example of Converting a C++ Array into NewtonScript" beginning on page 1-16 shows sample code for converting a C++ array into a NewtonScript object.

The Newton Object System

The Newton Object System is the name for the component of the Newton system software that manages the objects that Newton applications manipulate and store. The Newton Object System allows you to access objects from both NewtonScript and C++.

Newton Symbols and Object Types

Newton uses symbols as identifiers for variables, classes, messages, and frame slots. Symbol names can contain up to 254 characters, including any printable ASCII character.

Note
NewtonScript applications sometimes define symbols enclosed between vertical bars. You should never use vertical bars when defining symbols in C++ programs. ▲

The Newton Object System supports four primitive object classes, which are shown in Table 1-1.

Table 1-1 Newton object types

Object type	Description
Immediate	A constant value such as an integer or a character. Immediate values are signed, 30-bit, twos complement integers.
Binary	A sequence of bytes.
Array	An array of object references.
Frame	A collection of slots, each of which is a tag/value pair. The tag is a NewtonScript symbol.

The primitive object classes divide into two types: immediates and reference objects. Each object value is stored in 32 bits. Two of the bits are used to store class information. Immediate objects contain their values within the remaining 30 bits, and reference objects contain a pointer to the actual data in the remaining 30 bits.

Immediate objects can be integers, characters, and booleans.

Reference objects can be binaries, arrays, and frames. Object references are described in the next section, "Object References."

See *The NewtonScript Programming Language* for a full explanation and examples of Newton symbols and the Newton object classes.

Object References

Newton objects are referenced by object references, of type `Ref`. An object reference is a 32-bit value that can represent an immediate object or a pointer to a binary object, array object, or frame object.

Note that `Ref`s are similar to handles in other object-oriented programming systems. One significant implication of this is that you often need to lock `Ref`s before using them, as described later in this chapter.

The Newton garbage collector automatically collects the storage allocated for objects to which there are no longer any references. When you use Newton objects in your C++ code, you need to maintain references to those objects appropriately; otherwise, the garbage collector might collect the objects at the wrong time.

The C++ Toolkit provides four object reference types that make it safe for you to refer to NewtonScript objects in C++, as shown in Table 1-2.

Table 1-2 Summary of C++ Toolkit reference types

Type	Description
`Ref`	Use only as the return value of a function. The receiving function must immediately store the returned `Ref` value into one of the other reference types.
`RefVar`	A C++ class used to create a local (automatic) reference variable.
`RefStruct`	A C++ class used to store an object reference in a structure.
`RefArg`	A C++ typedef (`const RefVar &`) used to pass an object reference as an argument to a function.

Any reference that is not stored in a `RefVar` or `RefStruct` object can become invalid after any call to the object system (which may provoke a call to the garbage collector).

The following rules apply to the use of the object reference types:

- you can only use the `Ref` type as the return type for functions. You must never declare a variable of type `Ref` in your C++ code. If you write a function that receives a `Ref` as the return value of another function, you must immediately store that value into a protected structure. This is because `Ref`s are highly volatile and can be garbage collected at any time.
- to keep a reference in a local variable, use a `RefVar`. You can only allocate `RefVar`s on the stack; it is incorrect to allocate a `RefVar` with the `new` operator.
- to pass a reference to a function, use a `RefArg`, which is simply a typedef for `"const RefVar &"`. The effect of this is to reuse the caller's `RefVar` as a read-only value, which reduces the number of `RefVar` allocations. This means that you cannot assign a new value to a `RefArg` parameter; if you need to do so, you must copy the value into a local `RefVar`.
- when you pass the return value of a function (a `Ref`) as a function argument, the `RefArg` declaration of that parameter causes the automatic allocation of a temporary `RefVar`.
- you allocate and deallocate `RefStruct` objects like other C++ objects, which means that the `RefStruct` class constructor creates and initializes a `RefStruct` value for you, and the `RefStruct` destructor deallocates the memory used by the object.
- When an exception occurs in your application, the Newton system software will automatically clean up reference variables on the stack (`RefVar`s). The system software does not automatically clean up non-stack-based reference variables; thus, if you want a reference maintained after an exception is handled, you need to store the reference in a `RefStruct`.

IMPORTANT

Any `Ref` can become invalid after any call to the object system. Calls to construct a `RefVar` or `RefStruct` object are part of the object system and are thus subject to this warning too. ▲

Using Ref as The Function Return Type

You must use `Ref` as the return type of any C++ function that can be called from NewtonScript. There are two important issues to be aware of regarding `Ref`:

- NewtonScript object references are 32-bit values. In C++, `Ref` has been defined as a `long` value for compatibility. Since `Ref` is declared as a `long`, the compiler cannot distinguish between `long` and `Ref`. This means that you can mistakenly return an integer (`long`) value as the function result rather than returning a reference to a NewtonScript object (a `Ref`). If you want to return an integer value as the result of a function that returns an object reference, you must use the `MakeInt` function. For example, to return the value `1`, use the following statement in your C++ function:

  ```
  return(MakeInt(1));
  ```

  Listing 1-1 on page 1-10 shows an example of a function that uses `MakeInt` to return an integer value.
- Values of type `Ref` are highly volatile, which means that their location can change at any time. Because of this, the function that calls your `Ref` function must immediately store the result into a `RefVar` or `RefStruct`. You can also use the function value as a parameter. In this case, C++ automatically creates a temporary `RefArg` to hold the value.

Table of Object Reference Use

Table 1-3 shows several examples of declarations involving object references and explains which examples are valid and which could lead to erroneous results.

Table 1-3 Examples of object reference use

Example	Validity	Explanation
`void foo(Ref x)…`	Doesn’t work	Function parameters must be `RefArg`s
`void foo(RefStruct x)…`	Doesn’t work	Function parameters must be `RefArg`s
`void foo(RefVar x)…`	Could be bad	Function parameters must be `RefArg`s
`void foo(RefArg x)…`	CORRECT	
`Ref x = …`	Doesn’t work	Only use `Ref` as the return type of a function.

Table 1-3 Examples of object reference use

Example	Validity	Explanation
`RefStruct x = …`	Doesn’t work	Do not allocate `RefStruct`s on the stack.
`RefVar x = …`	CORRECT	
`Ref*`	Doesn’t work	Only use `Ref` as the return type of a function.
`RefVar*`	Could be bad	`RefVar`s are for local, stack-based references only.
`RefStruct*`	CORRECT	
`new RefVar`	Doesn’t work	`RefVar`s are for local, stack-based references only.
`new RefStruct`	CORRECT	

Accessing Data In a Binary Object

When you need to access the data in a binary object, you need to use a locked pointer. The C++ Toolkit provides two macros for using locked pointers.

You start a block of code with the `WITH_LOCKED_BINARY` macro and end that block of code with the `END_WITH_LOCKED_BINARY` macro.

The `WITH_LOCKED_BINARY` macro takes a reference to a binary object and a pointer variable; it makes the pointer variable work as a pointer to the binary object within the block. The `END_WITH_LOCKED_BINARY` macro terminates the locked pointer block and unlocks the object.

The `WITH_LOCKED_BINARY` macro declares the pointer variable (of type `void*`) for you. Note that the pointer is no longer valid once you exit the locked pointer block of code. Within the locked pointer block of code, you can access the binary object with the pointer. For example, the code segment in Listing 1-1 makes the variable `thePtr` a pointer to the binary object `binObj`.

Listing 1-1 Using a locked pointer to access a binary object

```
RefVar   binObj;

WITH_LOCKED_BINARY(binObj, thePtr)
   // use thePtr to access data in the binary object
END_WITH_LOCKED_BINARY(binObj)
```

WARNINGS

There are several key points that you must keep in mind when working with locked pointers:

- If you assign something to the binary object (`binObj` in Listing 1-1), the object could be destroyed.
- The pointer that the `WITH_LOCKED_BINARY` macro declares for you is not valid after the `END_WITH_LOCKED_BINARY` macro executes. You must not attempt to use the pointer after that.
- You must not access locations before the pointer or after the end of the object (after the location defined by `((char*) thePtr) + Length(binObj)` in Listing 1-1).
- You can use the `SetLength` function to resize the binary object within the locked code block; however, attempting to lengthen the size of the object with the code block will almost always fail.

If you do attempt to use the pointer or access memory outside of the bounds of the binary object, you can corrupt the Newton frames heap and cause your program to terminate. ▲

The section "An Example of Accessing Binary Data" beginning on page 1-19 shows sample code for accessing a NewtonScript binary object in C++.

Note

You can nest an instance of the `WITH_LOCKED_BINARY` macro inside of another instance of the macro, as long as each instance has a corresponding call to the `END_WITH_LOCKED_BINARY` macro. ◆

NewtonScript Magic Pointers

NewtonScript uses special references known as magic pointers to access certain objects that are stored in Newton ROM. Magic pointer references are resolved at run time by the operating system, which substitutes the actual address of a ROM object for each magic pointer reference.

You only need to be concerned with magic pointers in your C++ code if you receive a pointer from NewtonScript and subsequently try to manipulate it as a C++ pointer. In that case, you have to know that you can't use the magic pointer like an ordinary pointer; for example, you would not want to follow the pointer when traversing a list of objects.

WARNING

If you try to use the `WITH_LOCKED_BINARY` macro with a magic pointer, disastrous results will occur. ▲

You can use the `IsMagicPtr` function, which is described on page page 2-4, to determine if a pointer is indeed a magic pointer. The `IsRealPtr` function, which is described on page 2-4, determines if a pointer is not a magic pointer.

Path Expressions

Some object functions allow you to specify a path expression as the value of a parameter. A path expression can be specified in three ways, as shown in Table 1-4.

Table 1-4 Path expressions

Path expression type	Example
symbol	`SYM(fuzzy)` `MakeSymbol("fuzzy");`
integer immediate	`MakeInt(432);`
array	`AllocateArray(SYM(pathexpr),2);`

Specifying Symbols

When an object function uses a symbol as a parameter, you need to use either the `MakeSymbol` function or the `SYM` macro to specify that symbol. The `SYM` macro is the same as the `MakeSymbol` function, except that it eliminates the need to quote the symbol name. `SYM` is defined as follows:

```
#define SYM(name) MakeSymbol(#name)
```

For example, to specify the NewtonScript symbol `'|fuzzy|`, you can use either of the following expressions in your C++ code:

```
MakeSymbol("fuzzy");
SYM(fuzzy)
```

Newton Exceptions and C++

The Newton system software supports the use of exceptions, which allow an application to break out of the normal flow of control to respond to exceptional conditions. You can read about C++ exception handling in Chapter 5, “Newton Exceptions Reference,” and you can read about NewtonScript exception handling in *The NewtonScript Programming Language*.

There is one important issue of concern to C++ developers with regard to exceptions. When an exception occurs, the Newton system software knows to automatically destroy any NewtonScript objects that were created within the block of code that is handled by the exception. However, the Newton system software cannot automatically destroy C++ objects when an exception occurs.

This means that you must be sure to call the object destructor function yourself. When you create a C++ object, you should work with that object within the context of an exception handling (`newton_try`) block and include a call to the object’s destructor function in the `cleanup` clause of the exception handler. Listing 1-2 shows the skeleton code for working with a C++ object.

Listing 1-2 Working with a C++ object in an exception block

```
TMyClass *myObj;

newton_try {
   …
   myObj = new TMyClass;
   …
}
cleanup {
   delete myObj;
}
end_try;
```

For more information about handling Newton exceptions in C++, including reference information for the `newton_try`, `cleanup`, and `end_try` calls, see Chapter 5, "Newton Exceptions Reference."

NewtonScript and C++ Equivalences and Examples

This section provides several examples of NewtonScript and C++ equivalencies as well as examples of C++ functions that illustrate some of the restrictions that you must beware of when writing code for the Newton.

Table 1-5 provides examples of C++ equivalences for common NewtonScript expressions. This table includes the page number in this book for the description of the C++ function used in the NewtonScript equivalent.

Table 1-5 NewtonScript expressions and their C++ equivalences

NewtonScript expression	C++ equivalent	location of C++ Toolkit description
`1`	`MakeInt(1)`	page 2-2
`nil`	`NILREF`	page 2-1
`true`	`TRUEREF`	page 2-1
`$x`	`MakeChar('x')`	page 2-2
`{}`	`AllocateFrame`	page 3-6
`[]`	`AllocateArray(SYM(array), 0)`	page 3-6
`Array(10,nil)`	`AllocateArray(SYM(array), 10)`	page 3-6
`value := x.y`	`GetFrameSlot(x, SYM(y))`	page 3-11
`x.y := z`	`SetFrameSlot(x, SYM(y), z)`	page 3-16

Table 1-5 NewtonScript expressions and their C++ equivalences (continued)

NewtonScript expression	C++ equivalent	location of C++ Toolkit description
`value := x.(y)`	`GetFramePath(x, y, value)`	page 3-11
`x.(y) := z`	`SetFramePath(x, y, z)`	page 3-15
`GetSlot(x,y)`	`GetFrameSlot(x,y)`	page 3-12
`HasSlot(x,y)`	`FrameHasSlot(x,y)`	page 3-10
`x[y]`	`GetArraySlot(x,y)`	page 3-11
`x[y]:=z`	`SetArraySlot(x,y,z)`	page 3-14
`X(a,b)`	`NSCallGlobalFn(SYM(x), a, b)`	page 6-4
`call x with (a,b)`	`NSCall(x, a, b)`	page 6-2
`f:msg(a,b)`	`NSSend(f, SYM(msg), a, b)`	page 6-6
`f:?msg(a,b)`	`NSSendIfDefined(f, SYM(msg), a, b)`	page 6-8

A Simple Example in NewtonScript and C++

This section presents a NewtonScript function and a C++ function that performs the same operation.

Listing 1-3 shows a NewtonScript function that searches through an array for a value and returns the index of that array entry.

Listing 1-3 A NewtonScript search function

```
{
   items: […],
   search: func(value) begin
            for i:=0 to Length(items)-1 do
               if items[i] = value then
                  return i;
            nil;
         end;
}
```

Listing 1-4 shows the C++ equivalent of the NewtonScript `search` function that is shown in Listing 1-3.

Listing 1-4 C++ version of the search function

```
extern "C" Ref Search(RefArg rcvr, RefArg value)
{
   RefVar items = GetVariable(rcvr, SYM(items));
```

```
   RefVar slotValue;

   long len = Length(items);
   for (long i = 0; i < len; i++) {
      slotValue = GetArraySlot(items, i);
      if (EQ(slotValue, value))
         return(MakeInt(i));
   }
   return NILREF;
}
```

The C++ `Search` function in Listing 1-4 can be called from NewtonScript. The `Search` function begins by retrieving a reference to the `items` array and calling the `Length` function to determine the number of entries in `items`.

The `GetVariable` function is described on page 6-14.

The `Length` function is described on page 3-14.

The `GetArraySlot` function is described on page 3-11.

The `MakeInt` function is described on page 2-2.

Note

The C++ `Search` function is slightly different than the NewtonScript `search` function because the `EQ` function does not perform exactly the same equality testing as does the NewtonScript = operator. Specifically, `EQ` tests the equality of floating point values differently than does the = operator. The testing performed by the `EQ` function is described on page 2-5. ◆

An Example of Defining and Calling Several C++ Functions

This section presents a listing of a C++ file that defines a simple function that is callable from NewtonScript, and the NewtonScript code for calling that function.

The C++ code in Listing 1-5 is part of a file `para.cp`, which is part of a project named `para`.

Listing 1-5 Defining a C++ function in a module

```
#include "objects.h"

extern "C" Ref ReturnIt(RefArg rcvr)
{
   short x;

   x = 23;
   Ref theValue = MakeInt((long) x);
```

```
   return theValue;
}
```

The NewtonScript code in Listing 1-6

Listing 1-6 Calling a C++ function from NewtonScript

```
func()
begin
   local x;
   local xtext;

   x:= call para.ReturnIt with ();
   xtext := NumberStr(x);
   staticwindow.text := Clone(xtext);
end
```

An Example of a Wrapper Function

Listing 1-7 shows an example of a wrapper function for the `EQ` function.

Listing 1-7 A wrapper function for a C++ function callable from NewtonScript

```
extern "C" Ref WEQ ( RefArg rcvr, RefArg a , RefArg b )
{
   int result;

   result = EQ( a, b);     // actual call

   return MakeBoolean(result);
}
```

An Example of Converting a C++ Array into NewtonScript

Listing 1-8 shows an example of a function that converts a C++ array into a NewtonScript array object.

Listing 1-8 Converting a C++ array into a NewtonScript object

```
extern "C" Ref CArrayToNSArray(long* myArray, long arraySize)
{
   RefVar arrayRef = AllocateArray(SYM(array), arraySize);
   for (long i = 0; i < arraySize; i++)
```

```
   SetArraySlot(arrayRef, i, MAKEINT(myArray[i]));

   return(arrayRef);
}
```

An Example of Automatic Allocation of RefArgs

C++ will automatically create a temporary `RefArg` object for you if you pass a `Ref` as a parameter value. This is convenient; however, it can be inefficient and can use up a lot of memory under certain circumstances. For example, the code segment in Listing 1-9 allocates a temporary `RefArg` for each iteration of the loop.

Listing 1-9 An example of inefficient automatic allocation of `RefArgs`

```
Ref MyFcn1(int i)
{
...
}

int MyFcn2(RefArg arg)
{
...
}

for (i=1; i<1000; i++)
   val = MyFcn2(MyFcn1(i));
```

Each call to `MyFcn2` in Listing 1-9 creates a temporary `RefArg` for the result of the call to `MyFcn1`. Since the C++ language definition does not specify that these objects have to be deallocated within the loop, you could potentially be allocating 1000 temporary `RefArg` objects. Listing 1-10 uses a temporary variable to create a more efficient version of the loop.

Listing 1-10 An example of a more efficient `RefArg` loop

```
RefVar temp;
for (i=1; i<1000; i++) {
   temp = MyFcn1(i);
   val = MyFcn2(temp);
}
```

An Example of Allocating Persistent Storage

Listing 1-11 shows the NewtonScript code and Listing 1-12 shows you the C++ code for using a binary object to allocate persistent storage for use in your C++ code, as described in the section "Allocating Persistent Storage" beginning on page 1-5.

Listing 1-11 NewtonScript code for using a binary object as persistent storage for C++

```
{
viewSetupFormScript:
   func() begin
      …
      cMemory := MakeBinary('myMemObj, 234);
      …
   end,
cMemory:nil,
foo:
   func() begin
      …
      self:DoSomeCThing();
      …
   end,

DoSomeCThing:myCmodule.DoSomeCThing,
}
```

The NewtonScript code in Listing 1-11 allocates a binary object named with the symbol `cMemory` and then calls the C++ function `DoSomeCThing`, which is defined in a module (file) named `myCmodule`.

Listing 1-12 C++ code for using a binary object as persistent storage

```
extern "C" Ref DoSomeCThing(RefArg rcvr)
{
   RefVar cMemory = GetVariable(rcvr, SYM(cMemory));
   WITH_LOCKED_BINARY(cMemory, mem)
      /* do something with mem */
   END_WITH_LOCKED_BINARY(cMemory)
}
```

The C++ function `DoSomeCThing` accesses the binary object that represents the memory area by calling the `GetVariable` function with `SYM(cMemory)`, the symbol that was used in NewtonScript to create the object. The `DoSomeCThing` function then accesses the object by using the `WITH_LOCKED_BINARY` macro, which is described in the section "Accessing Data In a Binary Object" beginning on page 1-10.

Note

If you use the method shown in Listing 1-11 and Listing 1-12 to allocate persistent storage for use in your C++ code, you do not have to be concerned with deallocating the memory. The NewtonScript garbage collector will take care of collecting the memory when it is no longer in use. ◆

An Example of Accessing Binary Data

Listing 1-13 shows an example of accessing binary data, as described in the section "Accessing Data In a Binary Object" beginning on page 1-10. In this case, the binary data is a terminated Unicode string.

Listing 1-13 Accessing binary data

```
extern "C" Ref GetStringLength(RefArg rcvr, RefArg str)
{
   long result;

   WITH_LOCKED_BINARY(str, strPtr)
      result = Ustrlen((UniChar*) strPtr);
   END_WITH_LOCKED_BINARY(str)

   return(MakeInt(result));
}
```

The `Ustrlen` function is described on page 7-8.

The `MakeInt` function is described on page 2-2.

C++ and NewtonScript Conversion Reference

This chapter describes the constants and functions that you can use in your C++ programs to convert or check the representation of objects for interfacing with NewtonScript applications. NewtonScript uses a different representation for certain value types than does the C++ language, which makes it necessary for you to convert objects of these types when using the objects in a cross-language function call.

Constants for Using C++ With NewtonScript

The C++ Toolkit defines three constants for use with NewtonScript.

```
const Ref NILREF     = 0x02;
const Ref TRUEREF    = 0x1A;
const Ref FALSEREF   = NILREF;
```

Constant descriptions

`NILREF`	A reference to the NewtonScript constant `NIL`.
`TRUEREF`	A reference to the NewtonScript constant `TRUE`.
`FALSEREF`	A reference to the NewtonScript constant `NIL`.

Type Conversion Functions

The C++ Toolkit provides a number of type conversion functions to help you pass values back and forth between C++ and NewtonScript.

MakeBoolean

Ref `MakeBoolean(int` *i*`);`

i An integer value.

The `MakeBoolean` function converts the C++ value *i* into a NewtonScript Boolean reference. If *i* is `0`, `MakeBoolean` returns `NILREF`; otherwise, `MakeBoolean` returns `TRUEREF`.

MakeChar

Ref `MakeChar(unsigned char` *c*`);`

c An unsigned character value.

The `MakeChar` function converts the C++ char value *c* into a NewtonScript immediate object with the character value and returns a reference to that object.

MakeInt

Ref `MakeInt(long` *i*`);`

i A long integer value.

The `MakeInt` function converts the C++ long integer value *i* into a NewtonScript immediate object with the integer value and returns a reference to that object.

WARNING
NewtonScript integer values are signed, 30-bit two's complement values. ▲

MakeReal

Ref `MakeReal(double` *d*`);`

d A double precision value.

The `MakeReal` function converts the C++ double precision value *d* into a NewtonScript real number object with the value of *d* and returns a reference to that object.

MakeString

Ref `MakeString(const char *`*s*`);`
Ref `MakeString(const UniChar *`*s*`);`

s A C++ string of 8-bit characters or a C++ string of 16-bit Unicode characters.

The `MakeString` function converts the C++ string value *s* into a NewtonScript string object and returns a reference to that object.

MakeSymbol

Ref `MakeSymbol(char*` *name*`);`

name A C++ string.

The `MakeSymbol` function converts the C++ string *name* into a NewtonScript object and returns a reference to that object.

WARNING
The `MakeSymbol` function is fairly slow. If you are using a symbol in a loop, you should consider caching the symbol in a local variable. ◆

RefToUniChar

UniChar `RefToUniChar(RefArg` *r*`);`

r A reference to a NewtonScript immediate object.

The `RefToUniChar` function converts the NewtonScript character immediate *r* into the equivalent Unicode character value and returns the character value.

Note
Unicode characters are 16-bit integer values (`typedef unsigned short`). Values of type *UniChar* contain a single Unicode character. ◆

RefToInt

long `RefToInt(Ref` *r*`);`

r A reference to a NewtonScript immediate object.

The `RefToInt` function converts the NewtonScript integer immediate *r* into the equivalent C++ long integer value and returns the integer value.

SYM

Ref `SYM(` *name*`);`

name A C++ string.

The `SYM` macro converts the C++ string *name* into a NewtonScript symbol and returns a reference to that symbol.

Note
The `SYM` macro is equivalent to the `MakeSymbol` function, except that you do not have to quote the name string when supplying it to `SYM`. The SYM macro is defined as follows:

```
#define SYM(name) MakeSymbol(#name)
```

◆

Type Checking Functions

The C++ Toolkit provides a number of functions that you can use in C++ to type-check NewtonScript values.

IsChar

Boolean `IsChar(Ref r);`

r A reference to a NewtonScript immediate object.

The `IsChar` function returns `TRUE` if the NewtonScript value referenced by *r* is an immediate character value, and `FALSE` if not.

IsInt

Boolean `IsInt(Ref r);`

r A reference to a NewtonScript immediate object.

The `IsInt` function returns `TRUE` if the NewtonScript value referenced by *r* is an immediate integer, and `FALSE` if not.

IsMagicPtr

Boolean `IsMagicPtr(Ref r);`

r A reference to a NewtonScript immediate object.

The `IsMagicPtr` function returns `TRUE` if the NewtonScript value referenced by *r* is a magic pointer, and `FALSE` if not.

IsPtr

Boolean `IsPtr(Ref r);`

r A reference to a NewtonScript immediate object.

The `IsPtr` function returns `TRUE` if the NewtonScript value referenced by *r* is a pointer, and `FALSE` if not.

IsRealPtr

Boolean `IsRealPtr(Ref r);`

r A reference to a NewtonScript immediate object.

The `IsRealPtr` function returns `TRUE` if the NewtonScript value referenced by *r* is a real pointer (not a magic pointer), and `FALSE` if not.

Value Checking Functions and Macros

The C++ Toolkit provides several macros that you can use to test the value of NewtonScript objects.

EQ

Boolean `EQ(RefArg` *a*`, RefArg` *b*`);`

The `EQ` function returns `TRUE` if the NewtonScript object referenced by *a* is equal to the NewtonScript object referenced by *b*; otherwise, `EQ` returns `FALSE`.

The `EQ` function tests equality as follows:

- If the objects referenced by *a* and *b* are both immediates, `EQ` returns `TRUE` if the immediate values are equal.
- If the objects referenced by *a* and *b* are not both immediates, `EQ` returns `TRUE` if the object referenced by *a* is the same object as the object referenced by *b*.
- The `EQ` function returns `FALSE` in all other circumstances.

ISNIL

Boolean `ISNIL(Ref` *r*`);`

The `ISNIL` macro returns `TRUE` if the NewtonScript value referenced by *r* is `NILREF`; otherwise, `ISNIL` returns `FALSE`.

ISFALSE

Boolean `ISFALSE(Ref` *r*`);`

The `ISFALSE` macro returns `TRUE` if the NewtonScript value referenced by *r* is `FALSEREF`; otherwise, `ISFALSE` returns `FALSE`.

ISTRUE

Boolean `ISTRUE(Ref` *r*`);`

The `ISTRUE` macro returns `TRUE` if the NewtonScript value referenced by *r* is `TRUEREF`; otherwise, `ISTRUE` returns `FALSE`.

NOTNIL

Boolean `NOTNIL(Ref` *r*`);`

The `NOTNIL` macro returns `TRUE` if the NewtonScript value referenced by *r* is not `NILREF`; otherwise, `NOTNIL` returns `FALSE`.

Debugging Macros

This section describes the macros you can use with the C++ Toolkit to interact with the debugger. Note that you should conditionally include debugging statements in your code so that they do not end up in your final versions.

Debugger

`Debugger()`

The `Debugger` macro generates a debugger trap.

DebugStr

`DebugStr(`*msg*`)`

msg — The message you want displayed by the debugger. Note that *msg* is a null-terminated C string.

The `DebugStr` macro generates a debugger trap and displays the *msg* string in a debugger window.

WARNING
The `DebugStr` function always displays its output, regardless of the default `stdout` setting. ▲

Note
The `DebugStr` and `DebugCStr` functions are equivalent on the Newton. ◆

DebugCStr

`DebugCStr(`*msg*`)`

msg — The message you want displayed by the debugger. Note that *msg* is a null-terminated C string.

The `DebugCStr` macro generates a debugger trap and displays the `msg` string in a debugger window.

WARNING
The `DebugStr` function always displays its output, regardless of the default `stdout` setting. ▲

Note
The `DebugStr` and `DebugCStr` functions are equivalent on the Newton. ◆

Summary of C++ and NewtonScript Conversion Reference

Constants for Using C++ With NewtonScript

```
const Ref NILREF     = 0x02;
const Ref TRUEREF    = 0x1A;
const Ref FALSEREF   = NILREF;
```

Type Conversion Functions and Macros

Ref	`MakeBoolean(int` *i*`);`
Ref	`MakeChar(unsigned char` *c*`);`
Ref	`MakeInt(long` *i*`);`
Ref	`MakeReal(double` *d*`);`
Ref	`MakeString(const char *`*s*`);`
Ref	`MakeString(const UniChar *`*s*`);`
Ref	`MakeSymbol(char *`*name*`);`
UniChar	`RefToUniChar(RefArg` *r*`);`
long	`RefToInt(Ref` *r*`);`
Ref	`SYM(char *`*name*`);`

Type Checking Functions

Boolean	`IsChar(Ref` *r*`);`
Boolean	`IsInt(Ref` *r*`);`
Boolean	`IsMagicPtr(Ref` *r*`);`
Boolean	`IsPtr(Ref` *r*`);`
Boolean	`IsRealPtr(Ref` *r*`);`

Value Checking Functions and Macros

Boolean	`EQ(RefArg` *a*`, RefArg` *b*`);`
Boolean	`ISNIL(Ref` *r*`);`
Boolean	`ISFALSE(Ref` *r*`);`
Boolean	`ISTRUE(Ref` *r*`);`

Boolean `NOTNIL(Ref` *r*`);`

Debugging Functions and Macros

`Debugger()`

`DebugStr(`*msg*`)`

`DebugCStr(`*msg*`)`

Newton Object System Reference

This chapter describes the data types and functions that you use to manipulate Newton objects. This chapter provides the function declaration for each of the Newton Object System functions that you can use in your C++ applications.

Many of the func tions that you use to manipulate Newton objects are C++ wrappers for NewtonScript functions. The descriptions of these functions are provided in the *Newton Programmer's Guide*. You call NewtonScript functions using the NewtonScript C++ interface functions, which is described in Chapter 6, "NewtonScript Reference."

Some Newton Object System functions are implemented directly in C++ (not as wrappers for NewtonScript functions) to improve their performance. These functions are described in this chapter.

If you want to use a NewtonScript function in your C++ program, you should first determine if a C++ implementation exists for the function. If the function is described in this book, it has a C++ implementation. An easy way to determine that is to look up the function name in the index or in Appendix A, "C++ Function Tables." If the function is described in this book, use it as documented. If a C++ version does not exist, call the NewtonScript function using one of the NewtonScript C++ interface functions, as described in Chapter 6, "NewtonScript Reference."

Object System Classes

This section describes the classes that you can use in your C++ programs to interface with the Newton object system.

Iteration Macros

This section describes the macros that you can use to iterate through slots in NewtonScript array and frame objects. The next section, "Object Iterator Class" beginning on page 3-4, describes the class these macros use.

FOREACH

`FOREACH(`*obj*`, `*value_var*`)`

obj	The array or frame object with slots through which you want to iterate.
value_var	The name of a variable into which you want the value of the current slot in the iteration assigned. The `FOREACH` macro declares this variable, a `RefVar`, for you.

You use the `FOREACH` macro when you want to iterate through the slots in a NewtonScript array or frame *obj* and perform some action using the value of each slot. The `FOREACH` macro creates an iterator for you and traverses the slots, allowing you to operate on each (the current slot), one at a time. The `FOREACH` macro assigns the value of the current slot to *value_var*, which you can use as shown in Listing 3-1.

Note

The `FOREACH` macro declares the *value_var* variable for you. ◆

Listing 3-1 An example of using the FOREACH macro

```
EXTERNC
Ref FrameScan(RefArg rcvr, RefArg obj)
{
   RefVar   result=0;
   RefVar   value;

FOREACH(obj,value)
   if (IsNumber(value))
      result = result + value;
   else if (IsFrame(value) || IsArray(value))
      result = result + FrameScan(rcvr, value);
END_FOREACH
return result;
}
```

FOREACH_WITH_TAG

`FOREACH_WITH_TAG(`*obj*`, `*tag_var*`, `*value_var*`);`

obj — The array or frame object with slot through which you want to iterate.

tag_var — The name of a variable into which you want the tag (name) of the current slot in the iteration assigned. The `FOREACH_WITH_TAG` macro declares this variable, a `RefVar`, for you.

value_var — The name of a variable into which you want the value of the current slot in the iteration assigned. The `FOREACH_WITH_TAG` macro declares this variable, a `RefVar`, for you.

You use the `FOREACH_WITH_TAG` macro when you want to iterate through the slots in a NewtonScript array or frame *obj* and perform some action using the name and value of each slot. The `FOREACH_WITH_TAG` macro creates an iterator for you and traverses the slots, allowing you to operate on each (the current slot), one at a time. The `FOREACH_WITH_TAG` macro assigns the name of the current slot to *tag_var* and the value of the current slot to *value_var*, which you can use, as shown in Listing 3-2.

Note
The `FOREACH_WITH_TAG` macro declares the *tag_var* and *value_var* variables for you. ◆

Listing 3-2 An example of using the FOREACH_WITH_TAG macro

```
RefVar obj;
RefVar myTag = SYM(foo.bar);

FOREACH_WITH_TAG(obj, tag, value)
   …
   if (SymbolCompareLex(tag, myTag) == 0)
      DoSomething(value);
   …
END_FOREACH
```

END_FOREACH

`END_FOREACH`

The `END_FOREACH` macro terminates an iteration started with either the `FOREACH` or `FOREACH_WITH_TAG` macros. The `END_FOREACH` macro deletes the iterator that was created by the other macro.

WARNING
You must call the `END_FOREACH` macro at the end of an iteration that you started by calling either the `FOREACH` or the `FOREACH_WITH_TAG` macros.

Object Iterator Class

You use objects of the `TObjectIterator` class to iterate through the slots in an array or frame.

Note
You can use the object iteration macros, which are described in the previous section, for almost all of your iteration needs. Most programs do not need to make direct use of the `TObjectIterator`class.

```
class TObjectIterator : public SingleObject {
   void     Reset(RefArg newObj);
   int      Next(void);
   int      Done(void);
   Ref      Tag(void);
   Ref      Value(void);
};
```

Reset

void `Reset(RefArg newObj);`

newObj A reference to an object with slots over which to iterate.

The `Reset` method of the `TObjectIterator` class resets the iteration to the first slot in the object `newObj`.

Next

int `Next(void);`

The `Next` method of the `TObjectIterator` class advances the iteration to the next slot in the iterator's object and returns a non-zero value. If there are no more slots in the object, `Next` returns `0`.

Done

int `Done(void);`

The `Done` method of the `TObjectIterator` class returns a non-zero value if the iteration is done (if the current slot is the last slot belonging to the object or its siblings), and returns `0` if the iteration is not done (if there are more entries)

Tag

Ref `Tag(void);`

The `Tag` method of the `TObjectIterator` class returns a reference to the tag for the current slot.

Value

Ref `Value(void);`

The `Value` method of the `TObjectIterator` class returns a reference to the value of the current slot.

Iterator Functions

The object creation and object destructor functions for the `TObjectIterator` class are private functions. If you want to use a `TObjectIterator` object, you need to use the functions described in this section to create and destory that object.

NewTObjectIterator

`TObjectIterator* NewTObjectIterator(RefArg` *obj*`);`

obj A reference to an object with slots over which to iterate.

Creates a new TObjectIterator object and returns a pointer to that object.

DeleteTObjectIterator

`DeleteTObjectIterator(TObjectIterator*` *iter*`);`

iter A pointer to a `TObjectIterator` object that was created by calling the `NewTObjectIterator` function.

Deallocates storage for and deletes the `TObjectIterator` object *iter*.

C++ Object System Functions

This section describes the C++ functions that you can call directly to work with the Newton Object System.

AddArraySlot

```
void AddArraySlot( RefArg   obj,
                   RefArg   value);
```

obj A reference to an array object.

value A reference to a value object that you want added as a new element in the array.

This function is described in the "Utility Functions" chapter of *Newton Programmer's Guide*.

AllocateArray

```
Ref AllocateArray( RefArg    theClass,
                   long      length);
```

theClass A reference to a class object. This is the class of the new array object.

length The number of slots in the array.

The `AllocateArray` function creates a new array object, with *length* slots, of class *theClass*.

The `AllocateArray` function returns a reference to the newly created array object.

Note
Calling the `AllocateArray` function in C++ is the same as using the following function call in NewtonScript:

```
SetClass(Array(length,nil), theClass)
```

◆

AllocateBinary

```
Ref AllocateBinary(RefArg    theClass,
                   long      length);
```

theClass A reference to a class object. This is the class of the new object.

length The number of bytes allocated for the object.

The `AllocateBinary` function creates a new binary object, with *length* bytes, of class *theClass*, and returns a reference to the new object.

AllocateFrame

```
Ref AllocateFrame(void);
```

The `AllocateFrame` function creates a new, empty (slotless) frame object, and returns a reference to the frame object.

Note
Calling the `AllocateFrame` function in C++ is the same as using the following expression in NewtonScript:

```
{ };
```

◆

ArrayMunger

```
void ArrayMunger(  RefArg    a1,
                   long      a1start,
                   long      a1count,
                   RefArg    a2,
                   long      a2start,
```

```
long        a2count);
```

a1 A reference to the destination array.

a1start The starting element in the destination array.

a1count The number of elements to be replaced in the destination array. If you specify `-1` as the value of *a1count*, elements are replaced to the end of the array.

a2 A reference to the source array. If you specify `NILREF` as the value of *a2*, there is no source array and elements are deleted from *a1*.

a2start The starting element in the source array.

a2count The number of elements to use from the source array. If you specify `-1` as the value of *a2count*, elements are taken to the end of the array.

This function is described in the "Utility Functions" chapter of *Newton Programmer's Guide*.

IMPORTANT
This function is the same as the NewtonScript function `ArrayMunger` with one important difference: in the NewtonScript version, you specify `NIL` as the value of *a1count* or *a2count* to indicate that elements are taken to the end of the array. In the C++ version, you specify `-1` to indicate the same thing. ▲

ArrayPosition

```
long ArrayPosition(RefArg   array,
                   RefArg   item,
                   long     start,
                   RefArg   test);
```

array A reference to an array object.

item A reference to an item that might be an element in the array.

start The starting position in the array.

test A reference to a function object used for testing. If you specify `NILREF`, the equality test is used.

This function is described as the `ArrayPos` function in the "Utility Functions" chapter of *Newton Programmer's Guide*.

ArrayRemove

```
Boolean ArrayRemove(   RefArg   array,
                       RefArg   element);
```

array A reference to an array object.

element A reference to the element to remove from the array.

The `ArrayRemove` function searches for the specified *element* in the *array*. If the element is found in the array, `ArrayRemove` removes it from the array and shifts any following elements left so that no empty elements remain.

Note

If there are two matching elements in the array, the `ArrayRemove` function only removes the first one. ◆

The `ArrayRemove` function returns `true` if `element` is found and removed, and `false` if *element* is not found in the array.

WARNING

The `ArrayRemove` function cannot remove an element that is an array or a frame. ▲

ArrayRemoveCount

```
void ArrayRemoveCount(RefArg   array,
                      FastInt  start,
                      FastInt  removeCount);
```

array A reference to an array object.

start The index of the first element to remove from the array.

removeCount The number of elements to remove from the array.

This function is described in the "Utility Functions" chapter of *Newton Programmer's Guide*.

ASCIIString

```
Ref ASCIIString(RefArg   str);
```

str A reference to a Unicode string object.

The `ASCIIString` function creates a binary object that holds an ASCII string from the Unicode string in *str*.

WARNING

Since the Unicode string str may contain non-ASCII characters, the resulting ASCII string may contain characters with the value `kNoTranslationChar`, as described in Chapter 7, "Newton Unicode Reference."

BinaryMunger

```
void BinaryMunger( RefArg   a1,
                   long     a1start,
                   long     a1count,
                   RefArg   a2,
                   long     a2start,
```

```
long        a2count);
```

a1 — A reference to the destination value bytes.

a1start — The starting position (numbering from 0) in *a1*.

a1count — The number of bytes to be replaced in the destination bytes. If you specify `-1` as the value of *a1count*, bytes are replaced to the end of *a1*.

a2 — A reference to the source bytes. If you specify `NILREF` as the value of *a2*, there is no source data and bytes are deleted from *a1*.

a2start — The starting position (numbering from 0) in *a2*.

a2count — The number of bytes to use from the source array. If you specify `-1` as the value of *a2count*, bytes are taken to the end of *a2*.

This function is described in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

IMPORTANT
This function is the same as the NewtonScript function `BinaryMunger` with one important difference: in the NewtonScript version, you specify `NIL` as the value of *a1count* or *a2count* to indicate that elements are taken to the end of the array. In the C++ version, you specify `-1` to indicate the same thing. ▲

ClassOf

Ref `ClassOf(RefArg` *obj*`);`

obj — A reference to an object.

This function is described in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

Clone

Ref `Clone(RefArg` *obj*`);`

obj — A reference to the object that you want cloned.

This function is described in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

CoerceToDouble

double `CoerceToDouble(RefArg` *r*`);`

r — A reference to a Newton real number object.

The `CoerceToDouble` function returns a double-precision value approximation of the Newton real number object referenced by *r*.

CoerceToInt

long `CoerceToInt(RefArg` *r*`);`

r A reference to a Newton real number object.

The `CoerceToInt` function returns a long-integer value approximation of the Newton real number object referenced by *r*.

DeepClone

Ref `DeepClone(RefArg` *obj*`);`

obj A reference to the object that you want cloned.

This function is described in the "Utility Functions" chapter of *Newton Programmer's Guide*.

EnsureInternal

Ref `EnsureInternal(RefArg` *obj*`);`

obj A pointer to an object.

This function is described in the "Utility Functions" chapter of *Newton Programmer's Guide*.

FrameHasPath

```
int FrameHasPath(  RefArg   obj,
                   RefArg   thePath);
```

obj A reference to a frame object.

thePath A reference to a path.

The `FrameHasPath` function determines if the frame referenced by *obj* contains the path expression referenced by *thePath*. If the path is found, `FrameHasPath` returns a non-zero value; if the path is not found, `FrameHasPath` returns `0`.

Note

Calling the `FrameHasPath` function in C++ is the same as using the following expression in NewtonScript:

```
obj.(thePath) exists
```

◆

FrameHasSlot

```
int FrameHasSlot(  RefArg   obj,
                   RefArg   slot);
```

obj A reference to a frame object.

slot A reference to a symbol naming a slot.

This function is described as the `HasSlot` function in the "Utility Functions" chapter of *Newton Programmer's Guide*.

GC

void `GC();`

The `GC` function invokes the Newton garbage collector.

Note
The Newton system software automatically invokes the garbage collector as required. You rarely, if ever, need to call this function. ◆

GetArraySlot

```
Ref GetArraySlot( RefArg   obj,
                  long     slot);
```

obj	A reference to an array object.
slot	The index of the slot in the array.

The `GetArraySlot` function returns a reference to the element at index *slot* in the array *obj*.

Note
Calling the `GetArraySlot` function in C++ is the same as using the following expression in NewtonScript:

```
obj[slot]
```

◆

GetFramePath

```
Ref GetFramePath( RefArg   obj,
                  RefArg   thePath);
```

obj	A reference to a frame object.
thePath	A reference to a path expression.

The `GetFramePath` function returns the value of the object reached by the path expression *thePath* in the frame specified by *obj*.

Note
Calling the `GetFramePath` function in C++ is the same as using the following expression in NewtonScript:

```
value := obj.(thePath)
```

◆

GetFrameSlot

```
Ref GetFrameSlot(  RefArg   obj,
                   RefArg   slot);
```

obj — A reference to a frame object.

slot — A reference to a symbol naming the slot whose value you want to get.

This function is described as the `GetSlot` function in the "Utility Functions" chapter of *Newton Programmer's Guide*.

IsArray

```
Boolean IsArray(RefArg   ref);
```

ref — A reference to an object.

The `IsArray` function returns `TRUE` if the object referenced by *ref* is a Newton array object and `FALSE` if not.

IsBinary

```
Boolean IsBinary(RefArg   ref);
```

ref — A reference to an object.

The `IsBinary` function returns `TRUE` if the object referenced by *ref* is a Newton binary object and `FALSE` if not.

IsFrame

```
Boolean IsFrame(RefArg ref);
```

ref — A reference to an object.

The `IsFrame` function returns `TRUE` if the object referenced by *ref* is a Newton frame object and `FALSE` if not.

IsFunction

```
Boolean IsFunction(RefArg  ref);
```

ref — A reference to an object.

The `IsFunction` function returns `TRUE` if the object referenced by *ref* is a Newton function object and `FALSE` if not.

IsInstance

```
Boolean IsInstance( RefArg    obj,
                    RefArg    super);
```

obj A reference to an object.

super A reference to a class symbol.

The `IsInstance` function returns `TRUE` if the object referenced by *obj* is an instance of the class *super,* and `FALSE` if not.

IsNumber

```
Boolean IsNumber(RefArg   ref);
```

ref A reference to an object.

The `IsNumber` function returns `TRUE` if the object referenced by *ref* is a Newton number object and `FALSE` if not.

IsReadOnly

```
Boolean IsReadOnly(RefArg    obj);
```

obj A reference to an object.

The `IsReadOnly` function returns `TRUE` if the object referenced by *obj* is in read-only memory and `FALSE` if not.

IsReal

```
Boolean IsReal(RefArg  r);
```

ref A reference to an object.

The `IsReal` function returns `TRUE` if the object referenced by *ref* is a Newton real number object and `FALSE` if not.

IsString

```
Boolean IsString(RefArg   ref);
```

ref A reference to an object.

The `IsString` function returns `TRUE` if the object referenced by *ref* is a Newton string object and `FALSE` if not.

IsSubclass

```
Boolea n IsSubclass( RefArg    sub,
                     RefArg    super);
```

sub A reference to a class symbol.

super A reference to a class symbol.

The `IsSubclass` function returns `TRUE` if the class referenced by *sub* is a subclass of the class referenced by *super,* and `FALSE` if not.

IsSymbol

Boolean `IsSymbol(RefArg` *obj*`);`

obj A reference to an object.

The `IsSymbol` function returns `TRUE` if the object referenced by *ref* is a Newton symbol object and `FALSE` if not.

Length

long `Length(RefArg` *obj*`);`

obj A reference to an object.

This function is described in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

RemoveSlot

```
void RemoveSlot(RefArg   frame,
                RefArg   tag);
```

frame A reference to a frame object.

tag A reference to a symbol naming a slot.

This function is described in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

IMPORTANT
The C++ version of the `RemoveSlot` function does not work with arrays. ▲

ReplaceObject

```
void ReplaceObject(RefArg   target,
                   RefArg   replacement);
```

target A reference to the original object.

replacement A reference to the object to which you want to redirect any references to *target*.

This function is described in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

SetArraySlot

```
void SetArraySlot( RefArg   obj,
                   long     slotIndex,
```

```
                    RefArg    value);
```

obj A reference to an array object.

slotIndex The index of the slot in the array.

value A reference to the new value for the slot in the array.

The `SetArraySlot` function establishes the *value* of the element at index *slot* in the array *obj*.

Note
Calling the `SetArraySlot` function in C++ is the same as using the following expression in NewtonScript:

obj[*slot*] := *value;*

◆

SetClass

```
void SetClass(   RefArg    obj,
                 RefArg    theClass);
```

obj A reference to an object.

theClass A reference to a class symbol.

This function is described in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

SetFramePath

```
Ref SetFramePath(   RefArg    obj,
                    RefArg    thePath,
                    RefArg    value);
```

obj A reference to a frame object.

thePath A reference to a path expression.

value A reference to the new value for the slot specified by the path expression.

The `SetFramePath` function sets the value of a slot to *value*. The slot whose value is set is determined by *thePath,* starting at the object *obj*.

Note
Calling the `SetFramePath` function in C++ is the same as using the following expression in NewtonScript:

obj.(*thePath*) := *value*

◆

SetFrameSlot

```
void SetFrameSlot( RefArg   obj,
                   RefArg   slot,
                   RefArg   value);
```

obj A reference to a frame object.

slot A reference to a symbol naming the slot whose value you want to change.

value A reference to an object that you want to be the value of the slot.

The `SetFrameSlot` function searches for the slot whose name matches the *slot* symbol in the frame referenced by *obj*. If the named slot is found in the frame, `SetFrameSlot` modifies the value of the slot to *value*. If the named slot is not found, `SetFrameSlot` adds a new slot with name *slot* to the frame and initializes it to *value*.

Note

Calling the `SetFrameSlot` function in C++ is the same as using the following expression in NewtonScript:

```
obj.(slot) := value
```

◆

IMPORTANT

The `SetFrameSlot` function adds a new slot to the frame referenced by *obj* if the slot does not already exist. ▲

SetLength

```
void SetLength( RefArg   obj,
                long     length);
```

obj A reference to an object.

length The new length for the object.

This function is described in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

SortArray

```
void SortArray( RefArg   array,
                RefArg   test,
                RefArg   key);
```

array A reference to the array that you want sorted.

test A reference to a function object. The function must take two parameters and return an integer value that specifies their sorting relationship.

key The sort key within each array element. .

This function is described as the `Sort` function in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

Statistics

```
void Statistics(   ULong*   freeSpace,
                   ULong*   largestFreeBlock);
```

freeSpace On return, the amount of free space, in bytes, in the task heap.

largestFreeBlock On return, the number of bytes in the largest block of free memory in the task heap.

The `Statistics` function returns the total amount of free space in the task heap and the size of the largest block of free space in the task heap.

StrBeginsWith

```
int StrBeginsWith( RefArg   str,
                   RefArg   prefix);
```

str A reference to a string object.

prefix A reference to a string object.

This function is described as the `BeginsWith` function in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

StrCapitalize

```
void StrCapitalize(RefArg   str);
```

str A reference to a string object.

This function is described as the `Capitalize` function in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

StrCapitalizeWords

```
void StrCapitalizeWords(RefArg str);
```

str A reference to a string object.

This function is described as the `CapitalizeWords` function in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

StrDowncase

```
void StrDowncase(RefArg str);
```

str A reference to a string object.

This function is described as the `Downcase` function in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

StrEndsWith

```
int StrEndsWith(RefArg   str,
                RefArg   suffix);
```

str A reference to a string object.

suffix A reference to a string object.

This function is described as the `EndsWith` function in the "Utility Functions" chapter of *Newton Programmer's Guide*.

StrMunger

```
void StrMunger( RefArg   s1,
                long     s1start,
                long     s1count,
                RefArg   s2,
                long     s2start,
                long     s2count);
```

s1 A reference to the destination string.

s1start The starting position in the destination string.

s1count The number of characters to be replaced in the destination string. If you specify `-1` as the value of *s1count*, characters are replaced to the end of the string.

s2 A reference to the source string. If you specify `NILREF` as the value of *s2*, there is no source string and characters are deleted from *s1*.

s2start The starting position in the source string.

s2count The number of characters to use from the source string. If you specify `-1` as the value of *s2count*, characters are taken to the end of the string.

This function is described in the "Utility Functions" chapter of *Newton Programmer's Guide*.

IMPORTANT

This function is the same as the NewtonScript function `StrMunger` with one important difference: in the NewtonScript version, you specify `NIL` as the value of *a1count* or *a2count* to indicate that elements are taken to the end of the array. In the C++ version, you specify `-1` to indicate the same thing. ▲

StrPosition

```
long StrPosition(RefArg   str,
                 RefArg   substr,
```

```
                    long      startPos);
```

str A reference to the string object that you want searched.

substr A reference to the string object for which you want to search.

startPos A reference to the character position in *str* at which you want the search to start.

This function is described as the `StrPos` function in the "Utility Functions" chapter of *Newton Programmer's Guide*.

Note that the `StrPosition` function returns `-1` if substr is not found in str. The NewtonScript `StrPos` function returns `NIL` instead.

StrReplace

```
long StrReplace(RefArg    str,
                RefArg    substr,
                RefArg    replacement,
                long      count);
```

str A reference to a string in which a substring replacement is to be made.

substr A reference to the substring to be replaced.

replacement A reference to the replacement string.

count The maximum number of replacements that can be made. If you specify `-1`, all occurrences will be replaced.

This function is described as in the "Utility Functions" chapter of *Newton Programmer's Guide*.

Note
The C++ `StrReplace` function uses `-1` as the value of count to indicate that all occurrences of *substr* should be replaced. The NewtonScript version uses `NIL`. ◆

StrUpcase

```
void StrUpcase(RefArgstr);
```

str A reference to a string object.

This function is described as the `Upcase` function in the "Utility Functions" chapter of *Newton Programmer's Guide*.

Substring

```
Ref Substring(RefArg  str,
              long    start,
```

```
                long    count);
```

str — A reference to a string object.

start — The starting position of the substring in the string.

count — The number of characters in the substring.

This function is described as the `SubStr` function in the "Utility Functions" chapter of *Newton Programmer's Guide*.

SymbolCompareLex

```
int SymbolCompareLex( RefArg   sym1,
                      RefArg   sym2);
```

sym1 — A reference to a symbol object to compare.

sym2 — A reference to the other symbol object to compare.

The `SymbolCompareLex` function compares the name of *sym1* to the name of *sym2*, using a case-insensitive string comparison. `SymbolCompareLex` returns a value as follows:

- if *sym1* is greater than *sym2*, return a positive integer value
- if *sym1* is equivalent to *sym2*, return 0
- if *sym1* is less than *sym2*, return a negative integer value

symcmp

```
int symcmp(   char* s1,
              char* s2);
```

s1 — The string name of a symbol to compare.

s2 — The string name of the other symbol to compare.

The symcmp function compares the two symbol names with a case-insensitive string comparison. symcmp returns a value as follows:

- if *s1* is greater than *s2*, return a positive integer value
- if *s1* is equivalent to *s2*, return 0
- if *s1* is less than *s2*, return a negative integer value

ThrowBadTypeWithFrameData

```
void ThrowBadTypeWithFrameData(NewtonErr    errorCode,
                               RefArg       value);
```

errorCode — A numeric error code.

value — A reference to the frame data object that caused the exception.

The `ThrowBadTypeWithFrameData` function raises a "bad type" exception. The exception frame contains two slots:

- a slot named `'errorcode` whose value is the integer representation of *errorCode*
- a slot named `'value` whose value is *value*.

ThrowRefException

```
void ThrowRefException(ExceptionName name,
                       RefArg        data);
```

name — An exception symbol.

data — A reference to the data object that caused the exception.

The `ThrowRefException` function raises an exception and creates an exception frame with the specified *name* and *data*.

TotalClone

```
Ref TotalClone(RefArg obj);
```

obj — A reference to an object that you want cloned.

This function is described in the "Utility Functions" chapter of *Newton Programmer's Guide*.

TrimString

```
void TrimString(RefArg   str);
```

str — A reference to a string object.

This function is described in the "Utility Functions" chapter of *Newton Programmer's Guide*.

Summary of Object System Reference

Object System Classes

Object Iterator Class

```
class TObjectIterator : public SingleObject {
   void     Reset(RefArg newObj);
   int      Next(void);
   int      Done(void);
   Ref      Tag(void);
   Ref      Value(void);
};
```

void `Reset(RefArg newObj);`

int `Next(void);`

int `Done(void);`

Ref `Tag(void);`

Ref `Value(void);`

Newton Object System Functions and Macros

Iterator Functions

*TObjectIterator** NewTObjectIterator(RefArg *obj*);

DeleteTObjectIterator(TObjectIterator* *iter*);

Iteration Macros

`FOREACH(`*obj*`, ` *value_var*`)`

`END_FOREACH`

`FOREACH_WITH_TAG(`*obj*`, ` *tag_var,* *value_var*`)`

C++ Newton Object Functions

void `AddArraySlot(RefArg` *obj*`, RefArg` *value*`);`

Ref `AllocateArray(RefArg th`*eClass*`, long` *length*`);`

```
Ref      AllocateBinary(RefArg theClass, long length);
Ref      AllocateFrame(void);
void     ArrayMunger(RefArg a1, long a1start, long a1count,
                     RefArg a2, long a2start, long a2count);
long     ArrayPosition(RefArg array, RefArg item, long start,
                     RefArg test);
Boolean  ArrayRemove(RefArg array, RefArg element);
void     ArrayRemoveCount(RefArg array, FastInt start,
                     FastInt removeCount);
Ref      ASCIIString(RefArg str);
void     BinaryMunger(RefArg a1, long a1start, long a1count,
                     RefArg a2, long a2start, long a2count);
Ref      ClassOf(RefArg obj);
Ref      Clone(RefArg obj);
double   CoerceToDouble(RefArg r);
long     CoerceToInt(RefArg r);
Ref      DeepClone(RefArg obj);
Ref      EnsureInternal(RefArg obj);
int      FrameHasPath(RefArg obj, RefArg thePath);
int      FrameHasSlot(RefArg obj, RefArg slot);
Ref      GetArraySlot(RefArg obj, long slot);
void     GetFramePath(RefArg obj, RefArg thePath);
Ref      GetFrameSlot(RefArg obj, RefArg slot);
void     GC();
Boolean  IsArray(RefArg ref);
Boolean  IsBinary(RefArg ref);
Boolean  IsFrame(RefArg ref);
Boolean  IsFunction(RefArg ref);
Boolean  IsInstance(RefArg obj, RefArg super);
Boolean  IsNumber(RefArg ref);
Boolean  IsReadOnly(RefArg obj);
Boolean  IsReal(RefArg r);
Boolean  IsString(RefArg ref);
Boolean  IsSubclass(RefArg sub, RefArg super);
Boolean  IsSymbol(RefArg obj);
long     Length(RefArg obj);
void     RemoveSlot(RefArg frame, RefArg tag);
```

```
void     ReplaceObject(RefArg target, RefArg replacement);
void     SetArraySlot(RefArg obj, long slot, RefArg value);
void     SetClass(RefArg obj, RefArg theClass);
Ref      SetFramePath(RefArg bj, RefArg thePath, RefArg value);
void     SetFrameSlot(RefArg obj, RefArg slot, RefArg value);
void     SetLength(RefArg obj, long length);
void     Statistics(ULong* freeSpace, ULong* largestFreeBlock);
void     SortArray(RefArg array, RefArg test, RefArg key);
int      StrBeginsWith(RefArg str, RefArg prefix);
void     StrCapitalize(RefArg str);
void     StrCapitalizeWords(RefArg str);
void     StrDowncase(RefArg str);
int      StrEndsWith(RefArg str, RefArg suffix);
void     StrMunger(RefArg s1,long s1start,long s1count,
                   RefArg s2,long s2start, long s2count);
long     StrPosition(RefArg str, RefArg substr, long startPos);
long     StrReplace(RefArg str, RefArg substr,
                    RefArg replacement, long count);
void     StrUpcase(RefArg str);
Ref      Substring(RefArg str, long start, long count);
int      SymbolCompareLex(RefArg sym1, RefArg sym2);
int      symcmp(char* s1, char* s2);
void     ThrowBadTypeWithFrameData(NewtonErr errorCode,RefArg value);
void     ThrowRefException(ExceptionName name, RefArg data);
Ref      TotalClone(RefArg obj);
void     TrimString(RefArg str);
```

Newton Memory Manager Reference

This chapter describes the functions that you use to work with the Newton memory manager.

About the Newton Memory Manager

The Newton Memory Manager presents a memory model that allows you to allocate and deallocate heap objects as you would in a standard C++ application programming environment.

The one thing in the Newton environment of which you must be aware is that the Newton object system maintains its own heap, the *object heap*. This is separate from the *application heap* that your C++ program uses. Although this is an important fact, it should not have any impact on your applications.

Memory Manager Functions

This section describes the C++ Toolkit Memory Manager functions.

BlockMove

```
void BlockMove( const void*    srcPtr,
                void*          destPtr,
```

```
                    Size            byteCount);
```

srcPtr A pointer to the block in memory that you want copied.

destPtr A pointer to the area into which you want the block copied.

byteCount The number of bytes to copy.

The `BlockMove` function copies *byteCount* bytes from *srcPtr* to *destPtr*.

Note
The `BlockMove` function is the same as the C library function `memmove`. ◆

CountFreeBlocks

```
unsigned long CountFreeBlocks(Heap    h=DEFAULT_NIL);
```

h A pointer to a heap object. Always use `DEFAULT_NIL` as the value of this parameter.

The `CountFreeBlocks` function returns the number of free blocks in the application heap.

IMPORTANT
The value of the *h* parameter to this function must always be `DEFAULT_NIL`. ▲

DisposePtr

```
void DisposePtr(Ptr p);
```

p A pointer to a block of memory allocated in the heap.

The `DisposePtr` function disposes of (releases) the block of memory pointed to by *p*.

Note
The `DisposePtr` function is the same as the C library function `free`. ◆

EqualBytes

```
int EqualBytes( const void*     ptr1,
                const void*     ptr2,
                Size            byteCount);
```

ptr1 A pointer to the first block of memory you want compared.

ptr2 A pointer to the second block of memory you want compared.

byteCount The number of bytes that you want compared.

The `EqualBytes` function compares bytes in memory. It first compares the byte at *ptr1* with the byte at *ptr2* and then advances each pointer by one byte and compares again. The comparison continues until the comparison fails or until *byteCount* bytes have been compared.

The `EqualBytes` function returns `1` if the two blocks are equal and `0` if not.

FillBytes

void `FillBytes(` `void*` *ptr*`,`
`Size` *length*`,`
`UChar` *pattern*`);`

ptr	A pointer to a block of memory.
length	The number of bytes in the block that you want modified. See the warning below for special considerations.
pattern	The byte value that you want assigned to each location in the block.

The `FillBytes` function fills a block of memory with the byte value specified by *pattern*. Each byte starting at *ptr* and continuing for *length* bytes is assigned the *pattern* value.

WARNING
The `FillBytes` function does not protect against negative or extremely large *length* values. It attempts to allocate the specified amount of memory, even though such values can cause disastrous results in your program. You must ensure that your calls to `FillBytes` supply appropriate *length* values. ▲

Note
The `FillBytes` function is the same as the C library function `memset`. ◆

FillLongs

void `FillLongs(` `void*` *ptr*`,`
`Size` *length*`,`
`ULong` *pattern*`);`

ptr	A pointer to a block of memory.
length	The number of bytes in the block that you want modified.
pattern	The unsigned long value that you want to fill the block.

The `FillLongs` function fills a block of memory with the unsigned long value specified by *pattern*. The *pattern* is treated as a sequential array of bytes that is repeatedly written to the block, starting with the byte pointed to by *ptr* and continuing until the byte at offset *length* from *ptr* is written.

Note
The *length* parameter indicates the number of bytes that you want modified. Remember that you are modifying those bytes by writing a long (4-byte) value. For example, if you want to overwrite twelve bytes in memory with a long value, you specify 12 as the value of *length*. The *pattern* will be written three times in this case. ◆

GetPtrName

ULong `GetPtrName(Ptr` *ptr*`);`

ptr A pointer to an object in the heap.

The `GetPtrName` function returns the 4-byte ID tag associated with *ptr*.

GetPtrSize

Size `GetPtrSize(Ptr` *p*`);`

p A pointer to a block of memory allocated in the heap.

The `GetPtrSize` function returns the number of bytes in the memory block pointed to by *p*.

LargestFreeInHeap

Size `LargestFreeInHeap(Heap` *h*`=DEFAULT_NIL);`

h A pointer to a heap object. Always use `DEFAULT_NIL` as the value of this parameter.

The `LargestFreeInHeap` function returns the size of the largest free block in the application heap.

IMPORTANT

The value of the h parameter to this function must always be `DEFAULT_NIL`. ▲

MaxHeapSize

Size `MaxHeapSize(Heap` *h*`=DEFAULT_NIL);`

h A pointer to a heap object. Always use `DEFAULT_NIL` as the value of this parameter.

The `MaxHeapSize` function returns the application heap size in bytes.

IMPORTANT

The value of the h parameter to this function must always be `DEFAULT_NIL`. ▲

MemError

NewtonErr `MemError(void);`

The `MemError` function returns the result of the most recent call by your task to the Memory Manager.

NewNamedPtr

```
Ptr NewNamedPtr( Size   byteCount,
                 ULong  name);
```

byteCount The number of bytes in the block to be allocated.

name The name to assign to the block. This is a 4-byte ID tag. See the warning below for special considerations.

The `NewNamedPtr` function allocates a non-relocatable block of memory in the heap. The size of the allocated block is indicated by *byteCount.* The `NewNamedPtr` function returns a pointer to the newly allocated block. The ID tag *name* is assigned to the pointer.

WARNING

The value of *name* is limited to valid 30-bit long integer values. If you specify a larger value, the name is set to `0x7FFFFFF`. ▲

If the allocation is successful, the Memory Manager result code (which is returned by the *MemError* function) is set to `noErr`. If the allocation is not successful, the Memory Manager result code is set to `memFullErr`.

NewPtr

```
Ptr NewPtr(Size  byteCount);
```

byteCount The number of bytes in the block to be allocated. See the warning below for special considerations.

The `NewPtr` function allocates a non-relocatable block of memory in the heap. The size of the allocated block is indicated by *byteCount.* The `NewPtr` function returns a pointer to the newly allocated block.

WARNING

The `NewPtr` function does not protect against negative or extremely large *byteCount* values. It attempts to allocate the specified amount of memory, even though such values can cause disastrous results in your program. You must ensure that your calls to `NewPtr` supply appropriate *byteCount* values. ▲

If the allocation is successful, the Memory Manager result code (which is returned by the *MemError* function) is set to `noErr`. If the allocation is not successful, the Memory Manager result code is set to `memFullErr`.

Note

The `NewPtr` function is the same as the C library function `malloc`. ◆

NewPtrClear

```
Ptr NewPtrClear(Size   byteCount);
```

byteCount The number of bytes in the block to be allocated. See the warning below for special considerations.

The `NewPtrClear` function allocates a non-relocatable block of memory in the heap. The size of the allocated block is indicated by *byteCount.* Each byte in the newly allocated block is cleared to zero. The `NewPtrClear` function returns a pointer to the newly allocated block.

WARNING
The `NewPtrClear` function does not protect against negative or extremely large *byteCount* values. It attempts to allocate the specified amount of memory, even though such values can cause disastrous results in your program. You must ensure that your calls to `NewPtrClear` supply appropriate *byteCount* values. ▲

If the allocation is successful, the Memory Manager result code (which is returned by the *MemError* function) is set to `noErr`. If the allocation is not successful, the Memory Manager result code is set to `memFullErr`.

ReallocPtr

```
Ptr ReallocPtr( Ptr     p,
                Size    newSize);
```

p A pointer to a block of memory allocated in the heap.

newSize The size, in bytes, that you want allocated for the block pointed to by *p*.

The `ReallocPtr` function modifies the size (and address) of the otherwise non-relocatable block of memory pointed to by *p*, copying the previous contents of the block as required. The `ReallocPtr` function returns a pointer to the newly allocated block.

If *p* is `NULL`, `ReallocPtr` simply calls and returns the value of the `NewPtr` function.

Note
The `ReallocPtr` function behaves differently than the standard, ANSI C library implementation in one case. If the value of *newSize* is `0`, `ReallocPtr` does not free *p*; instead, it sets the size of the buffer pointed to by *p* to `0`, which indicates that the Newton System Software can free the pointer at a later time. ◆

If the allocation is successful, the Memory Manager result code (which is returned by the *MemError* function) is set to `noErr`. If the allocation is not successful, the Memory Manager result code is set to `memFullErr`.

Note
The `ReallocPtr` function is the same as the C library function `realloc`. ◆

SetPtrName

void `SetPtrName(Ptr` *ptr*`,`
`ULong` *name*`);`

ptr A pointer to an object in the heap.

name A 4-byte ID tag for the object. See the warning below for special considerations.

The `SetPtrName` function associates the tag *name* with *ptr*.

WARNING
The value of *name* is limited to valid 30-bit long integer values. If you specify a larger value, the name is set to `0x7FFFFFF`. ▲

SystemRAMSize

Size `SystemRAMSize(`*void*`);`

The `SystemRamSize` function returns maximum number of bytes available for allocation before the user has stored anything. This is equivalent to all of RAM minus any user stores in RAM.

TotalFreeInHeap

Size `TotalFreeInHeap(Heap` *h*`=DEFAULT_NIL);`

h A pointer to a heap object. Always use `DEFAULT_NIL` as the value of this parameter.

The `TotalFreeInHeap` function returns the total number of bytes of free space in the application heap.

IMPORTANT
The value of the *h* parameter to this function must always be `DEFAULT_NIL`. ▲

TotalUsedInHeap

Size `TotalUsedInHeap(Heap` *h*`=DEFAULT_NIL);`

h A pointer to a heap object. Always use `DEFAULT_NIL` as the value of this parameter.

The `TotalUsedInHeap` function returns the total number of bytes that have been stored in the application heap.

IMPORTANT
The value of the *h* parameter to this function must always be `DEFAULT_NIL`. ▲

XORBytes

```
void XORBytes(   const void*     src1,
                 const void*     src2,
                 void*           dest,
                 Size            byteCount);
```

src1 A pointer to a block of memory.

src2 A pointer to a block of memory.

destPtr A pointer to a block of memory.

byteCount The number of bytes on which to perform the operation.

The `XORBytes` functions performs a byte-by-byte exclusive-or operation on two blocks of memory and writes the resulting bytes to a third block. Each byte in the block pointed to by *src1* is xor'ed with the corresponding byte in the block pointed to by *src2*; the result of that exclusive-or is written to the corresponding byte in the block pointed to by *destPtr*.

ZeroBytes

```
void ZeroBytes(  void*     ptr,
                 Size      length);
```

ptr A pointer to a block of memory.

length The number of bytes in the block that you want zeroed.

The `ZeroBytes` function clears each byte in the block of memory pointed to by *ptr* to zero. A total of *length* bytes is cleared.

Summary of Memory Manager Reference

Memory Manager C++ Functions

void	`BlockMove(const void*` *srcPtr*`, void*` *destPtr*`, Size` *byteCount*`);`
unsigned long	`CountFreeBlocks(Heap` *h*`=DEFAULT_NIL);`
void	`DisposePtr(Ptr` *p*`);`
int	`EqualBytes(const void*` *ptr1*`, const void*` *ptr2*`, Size` *byteCount*`);`
void	`FillBytes(void*` *ptr*`, Size` *length*`, UChar` *pattern*`);`
void	`FillLongs(void*` *ptr*`, Size` *length*`, ULong` *pattern*`);`
ULong	`GetPtrName(Ptr` *ptr*`);`
Size	`GetPtrSize(Ptr` *p*`);`
Size	`LargestFreeInHeap(Heap` *h*`=DEFAULT_NIL);`
Size	`MaxHeapSize(Heap` *h*`=DEFAULT_NIL);`
NewtonErr	`MemError(void);`
Ptr	`NewNamedPtr(Size` *byteCount*`, ULong` *name*`);`
Ptr	`NewPtr(Size` *byteCount*`);`
Ptr	`NewPtrClear(Size` *byteCount*`);`
Ptr	`ReallocPtr(Ptr` *p,* `Size` *newSize*`);`
void	`SetPtrName(Ptr` *ptr*`, ULong` *name*`);`
Size	`SystemRAMSize(`*void*`);`
Size	`TotalFreeInHeap(Heap` *h*`=DEFAULT_NIL);`
Size	`TotalUsedInHeap(Heap` *h*`=DEFAULT_NIL);`
void	`XORBytes(const void*` *src1*`, const void*` *src2*`, void*` *dest*`, Size` *byteCount*`);`
void	`ZeroBytes(void*` *ptr*`, Size` *length*`);`

Newton Exceptions Reference

This chapter describes the constants, data types, and classes that you use to raise and handle exceptions in your Newton C++ applications.

About Newton Exceptions

You can use exceptions and exception handling to "catch" error conditions that occur during the execution of your Newton application. Exceptions provide a mechanism for breaking out of the normal flow of control, responding to an exceptional condition, and then continuing with execution of your application.

The C++ Toolkit provides exception handling that is analagous to the exception handling provided in NewtonScript. You can read about NewtonScript exception handling in The *NewtonScript Programming Language*.

With the C++ Toolkit, you can define your own exceptions, throw exceptions, and catch exceptions. When you catch an exception, your exception-handling code is invoked. Some exceptions include data, which your exception handler can use to process the exception.

Defining Exceptions

The Newton system software defines a number of exceptions that you can catch and handle. The system software provides default handling for these exceptions, which are listed in Table 5-2 on page 5-5.

You can use the `DefineException` macro, which is described on page 5-7, to define a new exception for use in your application. Each exception is defined with a class name and a structured exception string.

IMPORTANT

The class name that you use to define the exception is the name that you use with the C++ exception functions and macros. This is, among other things, a symbolic name for the structured string name of the exception. ▲

When you define an exception, the Newton system software creates a new class for the exception. In the following call to `DefineException`, `exMyException` is the class name for the new exception:

```
DefineException(exMyException, evt.ex.myApp);
```

In subsequent calls to exception-handling functions, you would use `exMyException` to specify this exception.

Note

The C++ exception-handling macros, including `DefineException`, do not require the use of quotes around their string arguments. ◆

Exception names are structured strings that create a hierarchy of exceptions. These structured strings consist of a sequence of dot-separated prefix parts, followed by the final and most specific exception part. Exception names and exception-handling hierarchies are described more fully in *The NewtonScript Programming Language*.

WARNING

Exception name strings cannot exceed 127 characters in length. Longer exception names can cause a system crash. ▲

Your exception handlers can be structured to handle exceptions in a hierarchy of specificity: the handler for the most specific exception name catches that exception, and a less specific handler can catch any exceptions whose prefixes match it. The following are examples of exception names:

```
evt.ex
evt.ex.myApp
evt.ex.myApp.entryErr
evt.ex.myApp.entryErr.noDigit
```

Given the above exception definitions, exception handlers would catch exceptions as shown in Table 5-1.

Table 5-1 An exception-handling hierarchy

Exception handler string	Exceptions handled
`evt.ex`	Any with the prefix `evt.ex` that has not been handled by a more specific handler.
`evt.ex.myApp`	Any matching the prefix `evt.ex.myApp` that has not been handled by a more specific handler.
`evt.ex.myApp.entryErr`	An `evt.ex.myApp.EntryErr` exception or an `evt.ex.myApp.entryErr.noDigit` exception that has not been handled by a more specific handler.
`evt.ex.myApp.entryErr.noDigit`	Only the `evt.ex.myApp.entryErr.noDigit` exception.

WARNING
The Newton system software catches exceptions that begin with one of two prefixes: `evt.ex` or `type.ref`. If an exception does not begin with one of these prefixes, the system software throws the `evt.ex.fr` exception with error number `kFramesErrBadExceptionName`. ▲

Exception Data

When you throw an exception, you can optionally include data in the call to the `Throw` function. You can include a pointer to any data that you want to pass along.

Note
One important difference between NewtonScript and C++ exceptions is that the data included with a C++ exception can be any kind of data. The data included with a NewtonScript exception is always a NewtonScript object. ◆

When you include data with an exception, the exception handler needs to be able to destroy the data after using it. Since the shape of the data is arbitrary, you must tell the exception handler how to destroy it. You do so by including an `ExceptionDestructor` function specification along with the data. The `Throw` function can then call the `ExceptionDestructor` function to deallocate the data.

The `ExceptionDestructor` specification is described in the section “The Exception Destructor Type” on page 5-6. The `Throw` function is described in the section “Throw” on page 5-8.

Exception Blocks

Exceptions are handled in **exception blocks**. This is a block of code that begins with the `newton_try` macro and ends with the `end_try` macro. The exception-handling macros can only be used within an exception block. The following is an example of an exception block:

```
newton_try
{
    DoSomeStuff;
}
newton_catch(exMyException)
{
printf("Caught exception %s", CurrentException()->name);
}
end_try;
```

WARNING
You must not `return` or `break` out of an exception block, which includes `newton_try`, `newton_cleanup`, `unwind_protect`, and `on_unwind` blocks. Exiting from one of these blocks with a `return` or `break` statement will cause strange and possibly disastrous behavior in your program. ▲

Catch Blocks

The code block following the `newton_catch` clause is referred to as a **catch block**. Some exception-handling calls, including the `CurrentException` macro, are only valid within these blocks. These restrictions are described in the section "Exception-Handling Macros" beginning on page 5-9.

Other Exception-handling Blocks

Within an exception block, you can include several `newton_catch` clauses as well as `cleanup` and `unwind_protect` clauses. All of these clauses are optional and are followed by code blocks: the `newton_catch` clause, is followed by a catch block, the `cleanup` clause is followed by a **cleanup block**, and the `unwind_protect` clause is followed by an **unwind block**.

The `cleanup` clause, if present, must appear after any `newton_catch` blocks. If none of the `newton_catch` clauses catch the exception, the code in the `cleanup` block is executed before the next exception handler in the hierarchy is invoked.

The `unwind_protect` clause introduces a block of code that must be run whether or not an exception occurs. This code is known as protected code. Within the `unwind_protect` block, you can include `on_unwind` clause to specify the code that closes out the protected code block.

The macros mentioned in this section are described in the section “Exception-Handling Macros” beginning on page 5-9.

Volatile Values

You need to declare some local variables as volatile to work around a subtle problem that occurs with exception usage. The problem occurs as follows:

- The C++ compiler assigns a local variable to a register.
- You modify that variable inside of a **`try` block** that precedes code that can raise an exception.
- You need to access the local variable after exiting the `try` block. In other words, the value that you assigned to that variable inside of the `try` block is used outside of the `try` block.

The problem is that the local variable can be kept in a register if you do not declare it volatile. If this is the case and an exception is raised, the state of the register is restored to the value that it had when the `try` block was entered and the value that you assigned to the variable in the `try` block is lost.

IMPORTANT
You must use the `volatile` keyword when declaring a local variable that you modify inside of a `try` block. ▲

Newton System Software Exceptions

Table 5-2 lists the exceptions that the Newton system software generates.

Table 5-2 Newton system software exceptions

Exception name	Data type	Description
`exAbort`		generic abort
`exAlignment`		alignment error
`exBusError`		bus error
`exDivideByZero`		divide by zero error
`exIllegalInstr`		illegal instruction
`exMsgException`		exception with message
`exOutOfStack`		out of stack space error
`exPermissionViolation`		permission error
`exRootException`		the mother of all exceptions
`exSkia`		skia error
`exWriteProtected`		write-protection error

Exception Types

This section describes the data types that you use to work with exceptions in your C++ applications.

The Exception Structure Type

When you define an exception, the Newton system software creates a new object class for that exception. The name of the class is the name that you specify as the first parameter to the `DefineException` macro.

The `CurrentException` macro returns a pointer to an object of this class. You can access the name of an exception by using the name field of the object. For example, the structured string name of the current exception can be accessed with the following statement:

```
CurrentException()->name;
```

The Exception Destructor Type

The Newton system software uses the exception destructor type, of type `ExceptionDestructor`, to define the function type used to destroy the data associated with an exception. Some exceptions are raised with a pointer to data that is in the heap; the destructor function is used to deallocate that data. Each destructor function must be declared with the form defined by the `ExceptionDestructor` type:

```
typedef void (*ExceptionDestructor)(void*);
```

Exception Functions and Macros

This section describes the functions that you can use in your C++ applications to work with Newton exceptions.

CurrentException

*void** `CurrentException();`

The `CurrentException` macro returns a pointer to the exception object for the exception that is currently being handled. This is a pointer to an object whose class is the class of the current exception. See the section "The Exception Structure Type" beginning on page 5-6.

IMPORTANT
The `CurrentException` macro is only valid from within a `newton_catch` or `cleanup` block of an exception handler. ▲

DefineException

`DefineException(`*excptClass*`,` *string*`);`

excptClass The string name of the exception class. See the warning below for special considerations.

string A "structured string" that becomes the string name of the exception.

The `DefineException` macro declares a new exception class using the name *excptClass*. The exception name given by string is a structured string that defines the exception name in a manner that allows for hierarchical exception handling.

An exception name can be structured by separating its component parts with the period (`'.'`) character. Each part that follows a period adds specificity to the exception name. You can then structure your exception handlers to handle increasingly specific exceptions. For example, you could define three exception names:

```
evt.ex.myApp
evt.ex.myApp.str
evt.ex.myApp.str.null
```

You could then define three exception handlers: one to handle only the `'str.null'` exceptions in your application, another to handle any other `'str'` exceptions in your application, and another to handle any other exceptions in your application.

WARNING
Exception name strings cannot exceed 127 characters in length. Longer exception names can cause a system crash. ▲

The following is an example of using the `DefineException` macro:

```
DefineException(exMyException, evt.ex.myException);
```

rethrow

`rethrow();`

The rethrow macro reraises the current exception to allow the next handler (the next enclosing `Try` block) the opportunity to handle it. The data and destructor function associated with the current exception (if any) are passed along to the next handler.

Subexception

```
int Subexception(  ExceptionName  sub,
                   ExceptionName  super);
```

sub The name of an exception. This string can contain a number of semicolon-separated parts

super The name of an exception.

The `Subexception` function determines if the exception named by *super* is equivalent to or a prefix of any of the parts of the exception named by *sub*.

The `Subexception` function returns `1` if *super* is a prefix of any part of *sub* and `0` if not.

Throw

```
void Throw(  ExceptionName        name,
             void*                data = NULL,
             ExceptionDestructor  destructor = NULL);
```

name A string that is the class name of the exception.

data A pointer to the data that you want associated with the exception. This is an optional parameter

destructor The function that you want used to destroy the data associated with the exception. This is an optional parameter.

The `Throw` function raises an exception. You can optionally associate *data* and a a data *destructor* function with the exception.

If you pass heap *data* into the `Throw` function, you can provide a *destructor* function to deallocate that data. The `Throw` function uses the *destructor* function to deallocate the *data* after the exception has been handled.

The following are examples of using the `Throw` function:

```
Throw(exMyException);
Throw(exMyException, (void*) 1234);
Throw(exMyException, (void*) 1234, 0);
```

ThrowMsg

```
void ThrowMsg(char* msg);
```

msg A message string.

The `ThrowMsg` function raises an exception with the name `exMsgException`. The exception uses the string *msg* as its data.

You can use the `ThrowMsg` function to generate debugging messages. For example,

```
ThrowMsg("You are here");
```

Exception-Handling Macros

This section describes the macros that you can use to control exception handling in your C++ programs.

To handle exceptions in your C++ applications, you need to catch the exception. You can only catch exceptions within a `Try` block, which is also known as an exception handler. An exception handler is a block of code that you begin with a call to the `newton_try` macro and end with a call to the `end_try` macro.

You catch exceptions within an exception handler by calling the `newton_catch` macro. Exception handlers can be nested within other exception handlers, which allows you to handle a hierarchy of exception conditions.

Listing 5-1 shows an example of using the `newton_try`, `newton_catch`, and `end_try` macros:

Listing 5-1 Using the newton_try, newton_catch, and end_try macros

```
newton_try
{
   DoMySetup();
   DoMyFcn();
}
newton_catch(exMyException);
{
   printf("Exception raised: %s", CurrentException()->name);
}
end_try
```

In Listing 5-1, the `newton_catch` clause will handle any exceptions named `exMyException` that are raised during the processing of the `DoMySetup` and `DoMyFcn` functions. Any other exceptions will be handled by the next enclosing exception handler (oftentimes the system software).

cleanup

```
cleanup
```

You can add a `cleanup` clause after any newton_catch clauses in your exception handler. If no newton_catch clause handles the exceptions, the cleanup clause will execute before the exception is passed onto the next handler.

Note
The `cleanup` clause operates in exactly the same manner as does the `newton_catch_all` clause, except that the `cleanup` clause implicitly (automatically) rethrows the current exception. ▲

WARNING
You must not `return` or `break` out of a `newton_cleanup` code block. Doing so will cause strange and possibly disastrous behavior in your program. ▲

end_unwind

`end_unwind`

The `end_unwind` clause ends a block of protected code.

end_try

`end_try`

The `end_try` macro marks the end of a block of code within which exceptions can be caught and handled.

An example of using the `end_try` macro is shown in Listing 5-1.

newton_catch

`newton_catch(`*excptName*`)`

excptName A string that is the class name of the exception. This is the same string that is used in the call to the `Throw` function.

The `newton_catch` macro catches and handles the exception named by *excptName*. The macro is followed by a block of code that handles the exception. Within that block of code, you can reraise the exception by calling the rethrow macro, which is described in the section “rethrow” on page 5-7.

An example of using the `newton_catch` macro is shown in Listing 5-1.

WARNING
You must not `return` or `break` out of a `newton_catch` code block. Doing so will cause strange and possibly disastrous behavior in your program. ▲

newton_catch_all

`newton_catch_all`

The `newton_catch_all` macro catches any exceptions that have not been caught by any preceding newton_catch clauses. The `newton_catch_all` clause must follow any `newton_catch` clauses.

Listing 5-2 shows an example of using the `newton_catch_all` macro.

Listing 5-2 Using the `newton_catch_all` macro

```
newton_try
{
   DoMySetup();
   DoMyFcn();
}
newton_catch(exMyException);
{
   printf("Exception raised: %s", CurrentException()->name);
}
newton_catch_all
{
   exception_occurred = true;
}
end_try
```

WARNING
You must not `return` or `break` out of a `newton_catch_all` code block. Doing so will cause strange and possibly disastrous behavior in your program. ▲

newton_try

`newton_try`

The `newton_try` macro marks the beginning of a block of code within which exceptions can be caught and handled.

An example of using the `newton_try` macro is shown in Listing 5-1.

WARNING
You must not `return` or `break` out of a `newton_try` code block. Doing so will cause strange and possibly disastrous behavior in your program. ▲

on_unwind

`on_unwind`

The `on_unwind` clause closes out a block of protected code. You can call the `unwind_failed` macro from within this clause to determine if an exception occurred during the processing of the protected code block.

WARNING
You must not `return` or `break` out of a `on_unwind` code block. Doing so will cause strange and possibly disastrous behavior in your program. ▲

unwind_failed

```
unwind_failed()
```

You can call the `unwind_failed` macro from within the `on_unwind` clause of a protected block of code to determine if an exception occurred during the execution of the block of code. If an exception did occur, it will automatically be rethrown at the end of the `on_unwind` clause.

unwind_protect

```
unwind_protect
```

You can use the `unwind_protect` construct to specify code in an exception handler that must be run whether or not an exception occurs. The `unwind_protect` construct consists of an `unwind_protect` clause, an `on_unwind` clause, and an `end_unwind` macro.

Listing 5-3 shows an example of using the `unwind_protect` clause. Note that you can use the `unwind_protect` construct within an exception handler, although you need not do so.

Listing 5-3 Using the `unwind_protect`, on_unwind , and unwind_end macros

```
unwind_protect
{
   OpenAFile();
   DoSomethingWithFile();
}
on_unwind
{
   CloseTheFile();
}
end_unwind
```

WARNING
You must not `return` or `break` out of a `unwind_protect` code block. Doing so will cause strange and possibly disastrous behavior in your program. ▲

Summary of Exceptions Reference

Exception C++ Functions

Functions and Macros to Define and Throw Exceptions

*void**	`CurrentException();`
	`DefineException(`*excptClass*`, `*string*`);`
	`rethrow();`
int	`Subexception(ExceptionName `*sub*`, ExceptionName `*super*`);`
void	`Throw(ExceptionName `*name*`, void* `*data*` = NULL,` `ExceptionDestructor `*destructor*` = NULL);`
void	`ThrowMsg(char* `*msg*`);`

Exception-Handling Macros

```
cleanup
end_onwind
end_try
newton_catch(excptName)
newton_catch_all
newton_try
on_unwind
unwind_failed()
unwind_protect
```

NewtonScript Reference

This chapter describes the programming interface that you can use from your C++ programs to call into the NewtonScript interpreter. It also explains how to structure your C++ functions to allow NewtonScript applications to call them.

NewtonScript Interpreter Functions

You can use the NewtonScript interpreter functions in your C++ programs to call NewtonScript functions.

Some NewtonScript functions are implemented directly as C++ functions to improve their performance. Those functions are described in Chapter 3, "Newton Object System Reference."

If you want to use a NewtonScript function in your C++ program, you should first determine if a C++ implementation exists for the function. If so, use that function, as documented in Chapter 3, "Newton Object System Reference.". If a C++ version does not exist, use the functions described in this chapter to call the NewtonScript function.

Note
The NewtonScript Interpreter functions use object references (`Ref`), object reference parameters (`RefArg`) and symbols, all of which are described in the section "The Newton Object System" beginning on page 1-6 in Chapter 1, "C++ Toolkit Introduction." ◆

Functions for Calling NewtonScript Functions From C++

This section describes the NewtonScript Interpreter functions that you can call in your C++ programs. These functions allow you to execute NewtonScript function objects directly from C++.

Note

Each of these functions is overloaded, which means that they are supplied in different variations that allow you to supply different numbers of arguments to the functions. There is also a version of each function that you can call with an argument array. ◆

NSCall

```
Ref NSCall(  RefArg   fcn);

Ref NSCall(  RefArg   fcn,
             RefArg   arg0);

Ref NSCall(  RefArg   fcn,
             RefArg   arg0,
             RefArg   arg1);

Ref NSCall(  RefArg   fcn,
             RefArg   arg0,
             RefArg   arg1,
             RefArg   arg2);

Ref NSCall(  RefArg   fcn,
             RefArg   arg0,
             RefArg   arg1,
             RefArg   arg2,
             RefArg   arg3);

Ref NSCall(  RefArg   fcn,
             RefArg   arg0,
             RefArg   arg1,
             RefArg   arg2,
             RefArg   arg3,
             RefArg   arg4,

Ref NSCall(  RefArg   fcn,
             RefArg   arg0,
             RefArg   arg1,
             RefArg   arg2,
             RefArg   arg3,
             RefArg   arg4,
```

```
RefArg    arg5);
```

fcn	The function object that you want to call.
arg0	The value of the first argument to supply as a parameter value to the function you are calling.
arg1	The value of the second argument to supply as a parameter value to the function you are calling.
arg2	The value of the third argument to supply as a parameter value to the function you are calling.
arg3	The value of the fourth argument to supply as a parameter value to the function you are calling.
arg4	The value of the fifth argument to supply as a parameter value to the function you are calling.
arg5	The value of the sixth argument to supply as a parameter value to the function you are calling.

The `NSCall` function calls the NewtonScript function named by *fcn* and passes it any supplied parameter values. The provided variations of `NSCall` allow you to call functions that require any number of parameter values from zero to six.

The following is an example of using the `NSCall` function to call a NewtonScript function named `MyFcn` that requires two parameter values:

```
NSCall(MyFcn, x, y);
```

The C++ statement above has the same semantics as using the following NewtonScript expression:

```
call MyFcn with (x, y);
```

The `NSCall` function returns an object reference to the returned value of the function that you called. If the named function is not defined, `NSCall` throws an "is not defined as a function" exception.

NSCallWithArgArray

```
Ref NSCallWithArgArray(  RefArg    fcn,
                         RefArg    argArray);
```

fcn	The function object that you want to call.
argArray	A reference to an array that contains the function parameter values.

The `NSCallWithArgArray` function is a variant of the `NSCall` function that allows you to provide an array of parameter values. You can use this form to call a NewtonScript function with more than six arguments.

NSCallGlobalFn

```
Ref NSCallGlobalFn(RefArg   sym);

Ref NSCallGlobalFn(RefArg   sym,
                   RefArg   arg0);

Ref NSCallGlobalFn(RefArg   sym,
                   RefArg   arg0,
                   RefArg   arg1);

Ref NSCallGlobalFn(RefArg   sym,
                   RefArg   arg0,
                   RefArg   arg1,
                   RefArg   arg2);

Ref NSCallGlobalFn(RefArg   sym,
                   RefArg   arg0,
                   RefArg   arg1,
                   RefArg   arg2,
                   RefArg   arg3);

Ref NSCallGlobalFn(RefArg   sym,
                   RefArg   arg0,
                   RefArg   arg1,
                   RefArg   arg2,
                   RefArg   arg3,
                   RefArg   arg4);

Ref NSCallGlobalFn(RefArg   sym,
                   RefArg   arg0,
                   RefArg   arg1,
                   RefArg   arg2,
                   RefArg   arg3,
                   RefArg   arg4,
                   RefArg   arg5);
```

sym A symbol representing the name of the function that you want to call.

arg0 The value of the first argument to supply as a parameter value to the function you are calling.

arg1 The value of the second argument to supply as a parameter value to the function you are calling.

arg2 The value of the third argument to supply as a parameter value to the function you are calling.

arg3 The value of the fourth argument to supply as a parameter value to the function you are calling.

arg4 The value of the fifth argument to supply as a parameter value to the function you are calling.

arg5 The value of the sixth argument to supply as a parameter value to the function you are calling.

The `NSCallGlobalFn` function calls the NewtonScript global function named by *sym* and passes it any supplied parameter values. The provided variations of `NSCallGlobalFn` allow you to call functions that require any number of parameter values from zero to six.

The following is an example of using the `NSCallGlobalFn` function to call a NewtonScript function named `MyGlobalFcn` that requires two parameter values:

```
NSCallGlobalFn(SYM(MyGlobalFcn), x, y);
```

The C++ statement above has the same semantics as using the following NewtonScript expression:

```
MyGlobalFcn(x, y);
```

The `NSCallGlobalFn` function returns an object reference to the returned value of the function that you called. If the named function is not defined as a global function, `NSCallGlobalFn` throws an “is not defined as a function” exception.

NSCallGlobalFnWithArgArray

```
Ref NSCallGlobalFnWithArgArray(RefArg   sym,
                               RefArg   argArray);
```

sym A symbol representing the name of the function that you want to call.

argArray A reference to an array that contains the function parameter values.

The `NSCallGlobalFnWithArgArray` function is a variant of the `NSCallGlobalFn` function that allows you to provide an array of parameter values. You can use this form to call a NewtonScript function with more than six arguments.

NSSend

```
Ref NSSend( RefArg   receiver,
            RefArg   sym);

Ref NSSend( RefArg   receiver,
            RefArg   sym,
            RefArg   arg0);

Ref NSSend( RefArg   receiver,
            RefArg   sym,
            RefArg   arg0,
            RefArg   arg1);

Ref NSSend( RefArg   receiver,
            RefArg   sym,
            RefArg   arg0,
            RefArg   arg1,
            RefArg   arg2);

Ref NSSend( RefArg   receiver,
            RefArg   sym,
            RefArg   arg0,
            RefArg   arg1,
            RefArg   arg2,
            RefArg   arg3);

Ref NSSend( RefArg   receiver,
            RefArg   sym,
            RefArg   arg0,
            RefArg   arg1,
            RefArg   arg2,
            RefArg   arg3,
            RefArg   arg4);

Ref NSSend( RefArg   receiver,
            RefArg   sym,
            RefArg   arg0,
            RefArg   arg1,
            RefArg   arg2,
            RefArg   arg3,
            RefArg   arg4,
            RefArg   arg5);
```

receiver	A symbol representing the frame to which the function message is sent (the message receiver).
sym	A symbol representing the name of the function that you want to call.
arg0	The value of the first argument to supply as a parameter value to the function you are calling.
arg1	The value of the second argument to supply as a parameter value to the function you are calling.
arg2	The value of the third argument to supply as a parameter value to the function you are calling.
arg3	The value of the fourth argument to supply as a parameter value to the function you are calling.
arg4	The value of the fifth argument to supply as a parameter value to the function you are calling.
arg5	The value of the sixth argument to supply as a parameter value to the function you are calling.

The `NSSend` function sends the message named by *sym* to *receiver* with any supplied parameter values.

The provided variations of `NSSend` allow you to call methods that require any number of parameter values from zero to six.

The following is an example of using the `NSSend` function to call a NewtonScript method named `MySMthd` that requires two parameter values:

```
NSSend(x, SYM(MySMthd), y, z);
```

The C++ statement above has the same semantics as using the following NewtonScript expression:

```
x:MySMthd(y, z);
```

The `NSSend` function returns an object reference to the returned value of the method that was invoked. If the named method is not defined in the receiver frame, the parent chain, or the proto chain, `NSSend` throws an “undefined method” exception.

NSSendWithArgArray

```
Ref NSSendWithArgArray( RefArg   receiver,
                        RefArg   sym,
```

```
                    RefArg    argArray);
```

receiver A symbol representing the frame to which the function message is sent (the message receiver).

sym A symbol representing the name of the function that you want to call.

argArray A reference to an array that contains the function parameter values.

The `NSSendWithArgArray` function is a variant of the `NSSend` function that allows you to provide an array of parameter values. You can use this form to call a NewtonScript function with more than six arguments.

NSSendIfDefined

```
Ref NSSendIfDefined(RefArg  receiver,
                    RefArg  sym);

Ref NSSendIfDefined(RefArg  receiver,
                    RefArg  sym,
                    RefArg  arg0);

Ref NSSendIfDefined(RefArg  receiver,
                    RefArg  sym,
                    RefArg  arg0,
                    RefArg  arg1);

Ref NSSendIfDefined(RefArg  receiver,
                    RefArg  sym,
                    RefArg  arg0,
                    RefArg  arg1,
                    RefArg  arg2);

Ref NSSendIfDefined(RefArg  receiver,
                    RefArg  sym,
                    RefArg  arg0,
                    RefArg  arg1,
                    RefArg  arg2,
                    RefArg  arg3);

Ref NSSendIfDefined(RefArg  receiver,
                    RefArg  sym,
                    RefArg  arg0,
                    RefArg  arg1,
                    RefArg  arg2,
                    RefArg  arg3,
```

```
                        RefArg    arg4);

Ref NSSendIfDefined(RefArg    receiver,
                        RefArg    sym,
                        RefArg    arg0,
                        RefArg    arg1,
                        RefArg    arg2,
                        RefArg    arg3,
                        RefArg    arg4,
                        RefArg    arg5);
```

receiver — A symbol representing the frame to which the function message is sent (the message receiver).

sym — A symbol representing the name of the function that you want to call.

arg0 — The value of the first argument to supply as a parameter value to the function you are calling.

arg1 — The value of the second argument to supply as a parameter value to the function you are calling.

arg2 — The value of the third argument to supply as a parameter value to the function you are calling.

arg3 — The value of the fourth argument to supply as a parameter value to the function you are calling.

arg4 — The value of the fifth argument to supply as a parameter value to the function you are calling.

arg5 — The value of the sixth argument to supply as a parameter value to the function you are calling.

The `NSSendIfDefined` function sends the message named by *sym* to *receiver,* if and only if the method is defined.

If the method is defined, it is called with any supplied parameter values. The provided variations of `NSSendIfDefined` allow you to call methods that require any number of parameter values from zero to six.

The following is an example of using the `NSSendIfDefined` function to call a NewtonScript method named `MyIfMthd` that requires two parameter values:

```
NSSendIfDefined(x, SYM(MyIfMthd), y, z);
```

The C++ statement above has the same semantics as using the following NewtonScript expression:

```
x:?MyIfMthd(y, z);
```

The `NSSendIfDefined` function returns an object reference to the returned value of the method that was invoked. If the method is not defined, `NSSendIfDefined` returns the constant `NILREF`.

NSSendIfDefinedWithArgArray

```
Ref NSSendIfDefinedWithArgArray( RefArg    receiver,
                                 RefArg    sym,
                                 RefArg    argArray);
```

receiver A symbol representing the frame to which the function message is sent (the message receiver).

sym A symbol representing the name of the function that you want to call.

argArray A reference to an array that contains the function parameter values.

The `NSSendIfDefinedWithArgArray` function is a variant of the `NSSendIfDefined` function that allows you to provide an array of parameter values. You can use this form to call a NewtonScript function with more than six arguments.

NSSendProto

```
Ref NSSendProto(RefArg    receiver,
                RefArg    sym);

Ref NSSendProto(RefArg    receiver,
                RefArg    sym,
                RefArg    arg0);

Ref NSSendProto(RefArg    receiver,
                RefArg    sym,
                RefArg    arg0,
                RefArg    arg1);

Ref NSSendProto(RefArg    receiver,
                RefArg    sym,
                RefArg    arg0,
                RefArg    arg1,
                RefArg    arg2);

Ref NSSendProto(RefArg    receiver,
                RefArg    sym,
                RefArg    arg0,
                RefArg    arg1,
                RefArg    arg2,
```

```
                RefArg    arg3);

Ref NSSendProto(RefArg    receiver,
                RefArg    sym,
                RefArg    arg0,
                RefArg    arg1,
                RefArg    arg2,
                RefArg    arg3,
                RefArg    arg4);

Ref NSSendProto(RefArg    receiver,
                RefArg    sym,
                RefArg    arg0,
                RefArg    arg1,
                RefArg    arg2,
                RefArg    arg3,
                RefArg    arg4,
                RefArg    arg5);
```

receiver	A symbol representing the frame to which the function message is sent (the message receiver).
sym	A symbol representing the name of the function that you want to call.
arg0	The value of the first argument to supply as a parameter value to the function you are calling.
arg1	The value of the second argument to supply as a parameter value to the function you are calling.
arg2	The value of the third argument to supply as a parameter value to the function you are calling.
arg3	The value of the fourth argument to supply as a parameter value to the function you are calling.
arg4	The value of the fifth argument to supply as a parameter value to the function you are calling.
arg5	The value of the sixth argument to supply as a parameter value to the function you are calling.

The `NSSendProto` function sends the message named by *sym* to *receiver* and passes it any supplied parameter values. The `NSSendProto` function only looks in the proto chain for the method.

If the method is defined, it is called with any supplied parameter values. The provided variations of `NSSendProto` allow you to call methods that require any number of parameter values from zero to six.

The following is an example of using the `NSSendProto` function to call a NewtonScript method named `MyProtoMthd` that requires two parameter values:

```
NSSendProto(x, SYM(MyProtoMthd), y, z);
```

The C++ statement above has the same semantics as using the following NewtonScript expression:

```
if x.MyProtoMthd exists then x:MyProtoMthd(y, z)
else Throw(<undef method>);
```

The `NSSendProto` function returns an object reference to the returned value of the method that was invoked. If the named method is not defined, `NSSendProto` throws an "undefined function" exception.

NSSendProtoWithArgArray

```
Ref NSSendProtoWithArgArray(RefArg    receiver,
                            RefArg    sym,
                            RefArg    argArray);
```

receiver A symbol representing the frame to which the function message is sent (the message receiver).

sym A symbol representing the name of the function that you want to call.

argArray A reference to an array that contains the function parameter values.

The `NSSendProtoWithArgArray` function is a variant of the `NSSendProto` function that allows you to provide an array of parameter values. You can use this form to call a NewtonScript function with more than six arguments.

NSSendProtoIfDefined

```
Ref NSSendProtoIfDefined(RefArg    receiver,
                         RefArg    sym);

Ref NSSendProtoIfDefined(RefArg    receiver,
                         RefArg    sym,
                         RefArg    arg0);

Ref NSSendProtoIfDefined(RefArg    receiver,
                         RefArg    sym,
                         RefArg    arg0,
                         RefArg    arg1);

Ref NSSendProtoIfDefined(RefArg    receiver,
                         RefArg    sym,
                         RefArg    arg0,
```

```
                                 RefArg    arg1,
                                 RefArg    arg2);

Ref NSSendProtoIfDefined(RefArg    receiver,
                                 RefArg    sym,
                                 RefArg    arg0,
                                 RefArg    arg1,
                                 RefArg    arg2,
                                 RefArg    arg3);

Ref NSSendProtoIfDefined(RefArg    receiver,
                                 RefArg    sym,
                                 RefArg    arg0,
                                 RefArg    arg1,
                                 RefArg    arg2,
                                 RefArg    arg3,
                                 RefArg    arg4);

Ref NSSendProtoIfDefined(RefArg    receiver,
                                 RefArg    sym,
                                 RefArg    arg0,
                                 RefArg    arg1,
                                 RefArg    arg2,
                                 RefArg    arg3,
                                 RefArg    arg4,
                                 RefArg    arg5);
```

receiver — A symbol representing the frame to which the function message is sent (the message receiver).

sym — A symbol representing the name of the function that you want to call.

arg0 — The value of the first argument to supply as a parameter value to the function you are calling.

arg1 — The value of the second argument to supply as a parameter value to the function you are calling.

arg2 — The value of the third argument to supply as a parameter value to the function you are calling.

arg3 — The value of the fourth argument to supply as a parameter value to the function you are calling.

arg4 — The value of the fifth argument to supply as a parameter value to the function you are calling.

arg5 — The value of the sixth argument to supply as a parameter value to the function you are calling.

The `NSSendProtoIfDefined` function sends the message named by *sym* to *receiver,* if and only if the method is defined. The `NSSendProtoIfDefined` function only looks in the proto chain for the method.

If the method is defined, it is called with any supplied parameter values. The provided variations of `NSSendProtoIfDefined` allow you to call methods that require any number of parameter values from zero to six.

The following is an example of using the `NSSendProtoIfDefined` function to call a NewtonScript method named `MyProtoIfMthd` that requires two parameter values:

```
NSSendProtoIfDefined(x, SYM(MyProtoIfMthd), y, z);
```

The C++ statement above has the same semantics as using the following NewtonScript expression:

```
if x.MyProtoIfMthd exists then x:MyProtoIfMthd(y, z)
else nil;
```

The `NSSendProtoIfDefined` function returns an object reference to the returned value of the method that was invoked. If the method is not defined, `NSSendProtoIfDefined` returns the constant `NILREF`.

NSSendProtoIfDefinedWithArgArray

Ref `NSSendProtoIfDefinedWithArgArray(RefArg` *receiver*`,`
`RefArg` *sym*`,`
`RefArg` *argArray*`);`

receiver — A symbol representing the frame to which the function message is sent (the message receiver).

sym — A symbol representing the name of the function that you want to call.

argArray — A reference to an array that contains the function parameter values.

The `NSSendProtoIfDefinedWithArgArray` function is a variant of the `NSSendProtoIfDefined` function that allows you to provide an array of parameter values. You can use this form to call a NewtonScript function with more than six arguments.

Functions for Accessing NewtonScript Slot Values from C++

This section describes several functions that you can use from your C++ programs to set the value or retrieve the value of NewtonScript variables.

GetVariable

Ref `GetVariable(` `RefArg` *frame*`,`
`RefArg` *varName*`,`

```
long*     found = 0);
```

frame A reference to the frame in which to start searching for the slot.

varName A symbol representing the name of the slot that you want to find.

found A Boolean value. On exit, this is `true` if the variable was found and `false` if not.

The `GetVariable` function searches for the slot with name *varName* and returns its value. The named slot is searched for using the combined prototype and parent inheritance lookup, as described in *The NewtonScript Programmer's Language*.

If the variable was not found, the function returns `NILREF` as its result.

SetVariable

```
void SetVariable(RefArg    contextFrame,
                 RefArg    varName,
                 RefArg    value);
```

contextFrame A reference to the frame in which to start searching for the slot.

varName A symbol representing the name of the slot that you want to find.

value A reference to the new value that you want assigned to the slot.

The `SetVariable` function searches for the slot with name *varName* and modifies the value of that slot to *value*. The named slot is searched for using the combined prototype and parent inheritance lookup, as described in *The NewtonScript Programmer's Language*.

If the `SetVariable` function does not find a slot with name *varName*, it adds a new slot to *contextFrame*, using *varName* and *value* for the new slot.

Calling C++ Functions from NewtonScript

This section explains how you can call C++ functions from a NewtonScript application. Each NewtonScript-callable C++ function must use the following format:

```
Ref MyCplusFunction(RefArg  receiver,
                    RefArg  arg0,
                    RefArg  arg1,
                    ...
                    RefArg  argn);
```

receiver A reference to the receiver frame for the C++ function.

arg0 The first argument to your function.

arg1 The second argument to your function.

...

argn The final argument to your function.

Assuming that the above function is declared in a C++ module named "`mymodule`," you would use the following NewtonScript expression to call the function (with two arguments):

```
call myModule.MyCplusFunction with (arg1, arg2);
```

Note

The NewtonScript caller does not supply a value for the *receiver* parameter. The Newton system software manages this automatically. ◆

Your C++ function must always include the receiver as its first parameter. You can define your function with anywhere from zero to five additional parameters. The Newton system software automatically fills this value in when a NewtonScript application calls your C++ function.

Your C++ function always returns a reference as its return value.

Summary of NewtonScript Interpreter Functions

Functions for Calling NewtonScript Functions From C++

NSCall

```
Ref NSCall(RefArg fcn);
Ref NSCall(RefArg fcn, RefArg arg0);
Ref NSCall(RefArg fcn, RefArg arg0, RefArg arg1);
Ref NSCall(RefArg fcn, RefArg arg0, RefArg arg1, RefArg arg2);
Ref NSCall(RefArg fcn, RefArg arg0, RefArg arg1, RefArg arg2,
                                         RefArg arg3);
Ref NSCall(RefArg fcn, RefArg arg0, RefArg arg1, RefArg arg2,
                               RefArg arg3, RefArg arg4);
Ref NSCall(RefArg fcn, RefArg arg0, RefArg arg1, RefArg arg2,
                   RefArg arg3, RefArg arg4, RefArg arg5);
Ref NSCallWithArgArray(RefArg fcn, RefArg argArray);
```

NSCallGlobalFn

```
Ref NSCallGlobalFn(RefArg sym);
Ref NSCallGlobalFn(RefArg sym, RefArg arg0);
Ref NSCallGlobalFn(RefArg sym, RefArg arg0, RefArg arg1);
Ref NSCallGlobalFn(RefArg sym, RefArg arg0, RefArg arg1, RefArg arg2);
Ref NSCallGlobalFn(RefArg sym, RefArg arg0, RefArg arg1, RefArg arg2,
                                         RefArg arg3);
Ref NSCallGlobalFn(RefArg sym, RefArg arg0, RefArg arg1, RefArg arg2,
                               RefArg arg3, RefArg arg4);
Ref NSCallGlobalFn(RefArg sym, RefArg arg0, RefArg arg1, RefArg arg2,
                   RefArg arg3, RefArg arg4, RefArg arg5);
Ref NSCallGlobalFnWithArgArray(RefArg sym, RefArg argArray);
```

NSSend

```
Ref NSSend(RefArg receiver, RefArg sym);
Ref NSSend(RefArg receiver, RefArg sym, RefArg arg0);
Ref NSSend(RefArg receiver, RefArg sym, RefArg arg0, RefArg arg1);
```

```
Ref NSSend(RefArg receiver, RefArg sym, RefArg arg0, RefArg arg1,
                                                   RefArg arg2);
Ref NSSend(RefArg receiver, RefArg sym, RefArg arg0, RefArg arg1,
                                       RefArg arg2, RefArg arg3);
Ref NSSend(RefArg receiver, RefArg sym, RefArg arg0, RefArg arg1,
                              RefArg arg2, RefArg arg3, RefArg arg4);
Ref NSSend(RefArg receiver, RefArg sym, RefArg arg0, RefArg arg1,
                  RefArg arg2, RefArg arg3, RefArg arg4, RefArg arg5);
Ref NSSendWithArgArray(RefArg receiver, RefArg sym, RefArg argArray);
```

NSSendIfDefined

```
Ref NSSendIfDefined(RefArg receiver, RefArg sym);
Ref NSSendIfDefined(RefArg receiver, RefArg sym, RefArg arg0);
Ref NSSendIfDefined(RefArg receiver, RefArg sym, RefArg arg0,
                                                   RefArg arg1);
Ref NSSendIfDefined(RefArg receiver, RefArg sym, RefArg arg0,
                                       RefArg arg1, RefArg arg2);
Ref NSSendIfDefined(RefArg receiver, RefArg sym, RefArg arg0,
                          RefArg arg1, RefArg arg2, RefArg arg3);
Ref NSSendIfDefined(RefArg receiver, RefArg sym, RefArg arg0,
                  RefArg arg1, RefArg arg2, RefArg arg3, RefArg arg4);
Ref NSSendIfDefined(RefArg receiver, RefArg sym, RefArg arg0,
      RefArg arg1, RefArg arg2, RefArg arg3, RefArg arg4, RefArg arg5);
Ref NSSendIfDefinedWithArgArray(RefArg receiver, RefArg sym,
                                                RefArg argArray);
```

NSSendProto

```
Ref NSSendProto(RefArg receiver, RefArg sym);
Ref NSSendProto(RefArg receiver, RefArg sym, RefArg arg0);
Ref NSSendProto(RefArg receiver, RefArg sym, RefArg arg0, RefArg arg1);
Ref NSSendProto(RefArg receiver, RefArg sym, RefArg arg0, RefArg arg1,
                                                   RefArg arg2);
Ref NSSendProto(RefArg receiver, RefArg sym, RefArg arg0, RefArg arg1,
                                       RefArg arg2, RefArg arg3);
Ref NSSendProto(RefArg receiver, RefArg sym, RefArg arg0, RefArg arg1,
                              RefArg arg2, RefArg arg3, RefArg arg4);
Ref NSSendProto(RefArg receiver, RefArg sym, RefArg arg0, RefArg arg1,
                  RefArg arg2, RefArg arg3, RefArg arg4, RefArg arg5);
```

```
Ref NSSendProtoWithArgArray(RefArg receiver, RefArg sym,
                                          RefArg argArray);
```

NSSendProtoIfDefined

```
Ref NSSendProtoIfDefined(RefArg receiver, RefArg sym);
Ref NSSendProtoIfDefined(RefArg receiver, RefArg sym, RefArg arg0);
Ref NSSendProtoIfDefined(RefArg receiver, RefArg sym, RefArg arg0,
                                                    RefArg arg1);
Ref NSSendProtoIfDefined(RefArg receiver, RefArg sym, RefArg arg0,
                                      RefArg arg1, RefArg arg2);
Ref NSSendProtoIfDefined(RefArg receiver, RefArg sym, RefArg arg0,
                       RefArg arg1, RefArg arg2, RefArg arg3);
Ref NSSendProtoIfDefined(RefArg receiver, RefArg sym, RefArg arg0,
            RefArg arg1, RefArg arg2, RefArg arg3, RefArg arg4);
Ref NSSendProtoIfDefined(RefArg receiver, RefArg sym, RefArg arg0,
    RefArg arg1, RefArg arg2, RefArg arg3, RefArg arg4, RefArg arg5);
Ref NSSendProtoIfDefinedWithArgArray(RefArg receiver, RefArg sym,
                                          RefArg argArray);
```

Functions for Accessing NewtonScript Slot Values from C++

```
Ref GetVariable(RefArg frame, RefArg varName, long* found = 0);
void SetVariable(RefArg contextFrame, RefArg varName, RefArg value);
```

Newton Unicode Reference

This chapter describes the constants, data types, and classes that you use to manipulate Unicode strings.

.Unicode Constants and Data Types

This section describes the constants and data types that you use with the Unicode functions that are described in this chapter.

The UniChar Type

The Newton System Software defines the `UniChar` type for Unicode characters.

```
typedef unsigned short  UniChar;
```

Encoding Type Constants

You use the encoding type constants to specify the kind of encoding to use for the various Unicode conversion functions.

```
#define kMacRomanEncoding  1
#define kASCIIEncoding     2
#define kWizardEncoding    4
#define kShiftJISEncoding  5
#define kMacKanjiEncoding  6
```

Constant Descriptions

`kMacRomanEncoding`	Macintosh Roman character encoding.
`kASCIIEncoding`	7-bit ASCII character encoding.
`kWizardEncoding`	English Wizards character encoding.
`kShiftJISEncoding`	Japanese character encoding.
`kMacKanjiEncoding`	Macintosh KanjiTalk 7 character encoding.

Note

The system software automatically sets the value of the constant `kDefaultEncoding` to the character set of the current desktop platform.

Unicode Character and String Constants

The Unicode functions use these constants to indicate values in strings.

```
const UChar    kNoTranslationChar      1
const UChar    kEndOfCharString        2
const UniChar  kEndOfUnicodeString     2
```

Constant Descriptions

`kNoTranslationChar`	Stored in the string destination string to indicate that there was no translation for the character in the source string.
`kEndOfCharString`	The string termination character for a string resulting from the conversion of a Unicode string.
`kEndOfUnicodeString`	The string termination character for a string resulting from the conversion of a string to a Unicode string.

Unicode Functions

This section describes the functions that you can use with Unicode strings.

ConvertFromUnicode

```
void ConvertFromUnicode( const UniChar* source,
                         void*          dest,
                         FastInt        destEncoding = kDefaultEncoding,
```

```
                    long         length = 0x7FFFFFFF);
```

source The Unicode character string to be converted.

dest A pointer to the destination string.

destEncoding The encoding method to use for the conversion. Use one of the constants shown in the section “Encoding Type Constants” on page 7-1.

length The maximum number of characters to convert.

The `ConvertFromUnicode` function converts the characters in the *source* string from Unicode character encoding into another encoding. The output of the conversion is written to the destination string, which is pointed to by *dest*.

The destination string is always terminated by the `kEndOfCharString` character constant.

The `ConvertFromUnicode` function converts up to *length* characters or until the string termination character is encountered in the source string. You must ensure that adequate memory has been allocated for *dest* to contain all of the converted characters.

ConvertToUnicode

```
void ConvertToUnicode(const void*   source,
                      UniChar*      dest,
                      FastInt       srcEncoding = kDefaultEncoding,
                      long          length = 0x7FFFFFFF);
```

source The character string to be converted.

dest A pointer to the destination (Unicode) string.

destEncoding The encoding method to use for the conversion. Use one of the constants shown in the section “Encoding Type Constants” on page 7-1.

length The maximum number of characters to convert.

The `ConvertToUnicode` function converts the characters in the *source* string into Unicode characters. The output of the conversion (the Unicode characters) is written to the destination string, which is pointed to by *dest*.

The destination string is always terminated by the `kEndOfUnicodeString` character constant.

The `ConvertToUnicode` function converts up to *length* characters or until the string termination character is encountered in the source string. You must ensure that adequate memory has been allocated for *dest* to contain all of the converted characters.

ConvertUnicodeChar

```
long ConvertUnicodeChar( UniChar*   c,
                         Ptr        b,
```

```
                    FastInt       conversionType);
```

c — A pointer to a Unicode character.

b — A pointer to the destination string.

conversionType — The encoding method to use for the conversion. Use one of the constants shown in the section "Encoding Type Constants" on page 7-1.

The `ConvertUnicodeChar` function converts the Unicode character pointed to by c and stores the output of the conversion into the buffer *b*. The `ConvertUnicodeChar` function is a convenience function that makes the following call:

```
ConvertFromUnicode(c, b, conversionType, 1);
```

The `ConvertUnicodeChar` function returns the length, in bytes, of the character (*c*) that was converted.

ConvertUnicodeCharacters

```
void ConvertUnicodeCharacters( UniChar*    array,
                               Ptr         buffer,
                               FastInt     conversionType,
                               long        len);
```

array — The source array of characters.

buffer — The destination buffer.

conversionType — The encoding method to use for the conversion. Use one of the constants shown in the section "Encoding Type Constants" on page 7-1.

len — The number of characters to convert.

The `ConvertUnicodeCharacters` function converts the characters in the source *array* from Unicode character encoding into another encoding. The output of the conversion is written to the destination buffer, which is pointed to by *buffer*.

The `ConvertUnicodeCharacters` function converts *len* characters. The `ConvertUnicodeCharacters` function does apply any special handling to string terminators, which are treated just like any other character.

You must ensure that *array* contains at least the specified number (*len*) of characters. You must also ensure that adequate memory has been allocated for *buffer* to contain all of the converted characters.

HasChars

```
Boolean HasChars(UniChar* c);
```

c — A pointer to a Unicode string.

The `HasChars` function examines the string referenced by *c* to determine if it contains any alphabetic characters. An alphabetic character is any lowercase character between `'a'`and `'z'`, inclusively, or any uppercase character between `'A'` and `'Z'`, inclusively.

If the string referenced by *c* contains at least one alphabetic character, `HasChars` returns `true`; otherwise, `HasChars` returns `false`.

HasDigits

Boolean `HasDigits(UniChar*` *c*`);`

c — A pointer to a Unicode string.

The `HasDigits` function examines the string referenced by *c* to determine if it contains any numeric characters. A numeric character is any character between `'0'`and `'9'`, inclusively.

If the string referenced by *c* contains at least one numeric character, `HasDigits` returns `true`; otherwise, `HasDigits` returns `false`.

HasSpaces

Boolean `HasSpaces(UniChar*` *c*`);`

c — A pointer to a Unicode string.

The `HasSpaces` function examines the string referenced by *c* to determine if it contains any space (`' '`) characters.

If the string referenced by *c* contains at least one space character, `HasSpaces` returns `true`; otherwise, `HasSpaces` returns `false`.

IsPunctSymbol

```
Boolean IsPunctSymbol( UniChar     *word,
                       FastInt     index);
```

word — A pointer to a Unicode string.

index — The index in *word* of the character to be tested.

The `IsPunctSymbol` function examines the character specified by *word*[*index*] to determine if it is a punctuation symbol. The `IsPunctSymbol` function returns `true` if the specified character is a punctuation symbol and `false` if not.

The character is not considered a punctuation symbol if it is preceded by `'s’'` or `'S’'`.

The `IsPunctSymbol` function considers the characters shown in Table 7-1 on page 7-6 to be punctuation symbols.

StripPunctSymbols

void `StripPunctSymbols(UniChar*` *word*`);`

word — A pointer to a Unicode string.

The `StripPunctSymbols` function strips any leading and trailing punctuation symbols from the string *word*. Any of the characters shown in Table 7-1 are considered to be punctuation symbols.

Table 7-1 Unicode punctuation symbols

Character	Character Name	Hexadecimal Value
!	exclamation mark	0x21L
"	quotation mark	0x22L
'	single quote	0x27L
(	left parenthesis	0x28L
)	right parenthesis	0x29L
,	comma	0x2CL
.	period	0x2EL
:	colon	0x3AL
;	semicolon	0x3BL
?	question mark	0x3FL
“	left double quotation	0x2018L
”	right double quotation	0x2019L
‘	left single quote	0x201CL
’	right single quote	0x201DL

WARNING
The `StripPunctSymbols` function modifies the string *word*. ▲

Umemset

```
void* Umemset(   void*        str,
                 UniChar      ch,
                 ULong        numChars);
```

str A pointer to a buffer in memory.

ch A character.

numChars The number of characters to set.

The `Umemset` function initializes the block of memory pointed to by *str*. It copies the value of the character *ch* into each of the first *numChars* characters of *str*.

The `Umemset` function returns a pointer to *str*.

Ustrcat

```
UniChar* Ustrcat(   UniChar*          destStr,
                    const UniChar*    sourceStr );
```

destStr TheUnicode string on which to concatenate characters.

sourceStr The Unicode string to be copied.

The `Ustrcat` function concatenates the Unicode string *sourceStr* onto the end of the Unicode string *destStr*. This is done by copying each character of *sourceStr* to the end of the *destStr*. You must ensure that adequate memory has been allocated for *destStr* to contain the additional characters from *sourceStr*.

Ustrchr

```
UniChar* Ustrchr(   const UniChar     *str,
                    UniChar           ch );
```

str A pointer to a Unicode string.

ch A character.

The `Ustrchr` function finds the first occurrence of the character *ch* in the string *str* and returns a pointer to that character. If the character it not found in the string, `Ustrchr` returns `0`.

Ustrcmp

```
FastInt Ustrcmp( const UniChar*    str1,
                 const UniChar*    str2 );
```

str1 A pointer to a Unicode string.

str2 A pointer to a Unicode string.

The `Ustrcmp` function compares two strings according to the Unicode collating sequence.

The `Ustrcmp` function returns `0` if the two strings are equal.

The `Ustrcmp` function returns a negative number if *str1* is less than *str2*.

The `Ustrcmp` function returns a positive number if *str1* is greater than *str2*.

Ustrcpy

```
UniChar* Ustrcpy(   UniChar*          destStr,
                    const UniChar*    sourceStr );
```

destStr The Unicode string into which to copy. On exit, this is the string copy.

sourceStr The Unicode string to be copied.

The `Ustrcpy` function copies the Unicode string *sourceStr* to *destStr*. You must ensure that adequate memory has been allocated for *destStr* to hold all of the characters in *sourceStr*.

Ustrlen

```
ULong Ustrlen( const UniChar* str );
```

str	A Unicode string.

The `Ustrlen` function returns the length of the Unicode string *str*.

Ustrncat

```
UniChar* Ustrncat( UniChar*         destStr,
                   const UniChar*   sourceStr,
                   ULong            n );
```

destStr	The Unicode string on which to concatenate characters.
sourceStr	The Unicode string to be copied.
n	The number of characters to copy.

The `Ustrncat` function concatenates *n* characters of the Unicode string *sourceStr* onto the end of the Unicode string *destStr*. This is done by copying each character of *sourceStr* to the end of the *destStr*. You must ensure that adequate memory has been allocated for *destStr* to contain the additional characters from *sourceStr*. The `Ustrncat` function stops copying (concatenating) if it encounters the string termination character in *sourceStr*.

Ustrncpy

```
UniChar* Ustrncpy( UniChar*         destStr,
                   const UniChar*   sourceStr,
                   ULong            n );
```

destStr	The Unicode string into which to copy. On exit, this is the string copy.
sourceStr	The Unicode string to be copied.
n	The number of characters to copy.

The `Ustrncpy` function copies *n* characters of the Unicode string *sourceStr* to *destStr*. You must be certain that adequate memory has been allocated for *destStr* to hold *n* characters. The `Ustrncpy` function stops copying if it encounters the string termination character in *sourceStr*.

The `Ustrncpy` function always writes a string termination character at the end of *destStr*.

Summary of Unicode Reference

Unicode Data Types

```
typedef unsigned short  UniChar;
```

Encoding Type Constants

```
#define kMacRomanEncoding  1
#define kASCIIEncoding     2
#define kWizardEncoding    4
#define kShiftJISEncoding  5
#define kMacKanjiEncoding  6
```

Unicode Character and String Constants

```
const UChar    kNoTranslationChar      1
const UChar    kEndOfCharString        2
const UniChar  kEndOfUnicodeString     2
```

Unicode Functions

```
void      ConvertFromUnicode(const UniChar* source,void* dest,
                             FastInt destEncoding = kDefaultEncoding,
                             long length = 0x7FFFFFFF);
void      ConvertToUnicode(const UniChar* source, void* dest,
                             FastInt srcEncoding = kDefaultEncoding,
                             long length = 0x7FFFFFFF);
long      ConvertUnicodeChar(UniChar*    c,
                             Ptr         b,
                             FastInt     conversionType);
void      ConvertUnicodeCharacters(
                             UniChar*    array,
                             Ptr         buffer,
                             FastInt     conversionType,
                             long        len);
Boolean   HasChars(UniChar* c);
Boolean   HasDigits(UniChar* c);
Boolean   HasSpaces(UniChar* c);
```

```
Boolean     IsPunctSymbol(UniChar* word, FastInt index);
void        StripPunctSymbols(UniChar* word);
void*       Umemset(void* str, UniChar ch, ULong numChars);
UniChar*    Ustrcat(UniChar* destStr, const UniChar* sourceStr );
UniChar*    Ustrchr(const UniChar* str, UniChar ch );
FastInt     Ustrcmp(const UniChar* str1, const UniChar* str2 );
UniChar*    Ustrcpy(UniChar* destStr, const UniChar* sourceStr );
ULong       Ustrlen(const UniChar* str );
UniChar*    Ustrncat(UniChar* destStr, const UniChar* sourceStr,
                                                    ULong n );
UniChar*    Ustrncpy(UniChar* destStr, const UniChar* sourceStr,
                                                    ULong n );
```

Newton C Library Reference

This chapter describes the constants, data types, and functions from the C Library that you can use with your Newton programs.

IMPORTANT

With a few exceptions, all of the functions described in this chapter are part of the C Library that is supplied with most C and C++ compilers.

The description for many of these functions states that "this function is part of the standard ANSI-C library." This means that you need to read about the function in the C library documentation that accompanied your compiler.

The description for some of these functions states that "the Newton implementation of this function is described in the Utility Functions chapter of *Newton Programmer's Guide*". This means that you need to read about the function in the *Newton Programmer's Guide*.

Finally, a few of the functions are only found in the Newton C++ Toolkit. These functions—`asctime_newton`, `ctime_newton`, and `localtime_newton`—are variations of their analogs in the C library and are described in this chapter. ▲

C Library Constants and Data Types

This section describes the data types that you use with the Newton C Library functions.

C Library Constants

This section describes the constants that you can use with the C Library functions.

The NULL Pointer

The `NULL` pointer is used as the value of a pointer that does not point to anything.

```
#define NULL 0
```

The HUGE_VAL Constant

The `HUGE_VAL` constant is used to approximate infinity. This value is returned by several of the math functions when certain conditions exist.

```
#define HUGE_VAL  _inf();
```

The Maximum Random Number Value

The maximum random number value constant, `RAND_MAX`, defines the largest number that the `rand` function can return.

```
#define RAND_MAX 0x7fffffff
```

Standard Library Types

This section describes the data types that you use with the standard C Library functions.

The Size Type

You use the size type, of type `size_t`, to define the sizes of objects used in various of the C Library functions.

```
typedef unsigned int size_t;
```

The Wide Char Type

You use the wide character type, of type wchar_t, for characters that require more than one byte.

```
typedef int wchar_t;
```

The Division Result Type

You use the division result type, of type div_t, to hold the results of the `div` function.

```
typedef struct div_t {
   int   quot;
   int   rem;
} div_t;
```

Field descriptions

`quot`	The quotient for the division.
`rem`	The remainder for the division.

The Long Division Result Type

You use the long division result type, of type `ldiv_t`, to hold the results of the ldiv function.

```
typedef struct ldiv_t {
   long int quot;
   long int rem;
} ldiv_t;
```

Field descriptions

`quot`	The quotient for the division.
`rem`	The remainder for the division.

Math Types

This section describes the data types that you use with the math functions.

Double-precision Value Type

The double-precision value type, of type `double_t`, is used for double-precision values. It is exactly equivalent to the C++ `double` type.

Relational Operator Type

The relational operator type, of type `relop`, describes the relationship between two numbers.

```
typedef short relop;
enum {
      GREATERTHAN = ( ( relop ) ( 0 ) ),
      LESSTHAN,
      EQUALTO,
      UNORDERED
      };
```

Constant descriptions

`GREATERTHAN`	The first operand is greater than the second operand.
`LESSTHAN`	The first operand is less than the second operand.
`EQUALTO`	The first operand is equal to the second operand.
`UNORDERED`	At least one of the two operands is not a number.

Time Types

This section describes the data types that you use with the time functions.

Clock Time Type

The C Library functions use the clock time type, of type `clock_t`, to represent the cpu time type, which is the number of ticks per second in the value that is returned by the clock function. The clock function is described on page 8-31.

```
typedef unsigned int clock_t;
```

Calendar Time Type

The C Library functions use the calendar time type, of type `time_t`, to represent the current calendar time in a single, integer value. The internal representation of this value is not specified.

```
typedef unsigned int time_t;
```

Calendar Clock Time Structure

The C Library time functions use the calendar clock time structure, of type `tm`, to hold the components of a calendar clock reading.

```
struct tm {
   int tm_sec;
   int tm_min;
   int tm_hour;
   int tm_mday;
   int tm_mon;
   int tm_year;
   int tm_wday;
   int tm_yday;
   int tm_isdst;
};
```

Field descriptions

`tm_sec`	The number of seconds after the minute. This is a value between 0 and 59.
`tm_min`	The number of minutes after the hour. This is a value between 0 and 59.
`tm_hour`	The number of hours since midnight. This is a value between 0 and 23.
`tm_mday`	The day of the month. This is a value between 1and 31.
`tm_mon`	The number of months since January. This is a value between 0 and 11.
`tm_year`	The number of years since 1900.
`tm_wday`	The number of days since Sunday. This is a value between 0 and 6.
`tm_yday`	The number of days since January 1. This is a value between 0 and 365.
`tm_isdst`	A flag indicating whether Daylight Savings Time is in effect. This value is positive if Daylight Savings Time is in effect, zero if Daylight savings time is not in effect, and negative if the information is not available.

C Library Functions

This section describes the C Library functions that you can use in your Newton programs.

Character Conversion Functions

This section describes the C Library functions that convert a single character.

tolower

int `tolower(int` *c*`);`

c — A single character.

The `tolower` function is part of the standard ANSI-C library.

toupper

int `toupper(int` *c*`);`

c — A single character.

The `toupper` function is part of the standard ANSI-C library.

Floating-point Math Functions

This section describes the C Library functions for working with floating-point math values.

WARNING

The functions in this section, which are declared in the fp.h include file, cannot be used in p-classes. This might be of concern to you if you are using the C++ library functions to develop a Newton driver; however, this is not a concern for developers who are using C++ code with a NewtonScript application. ▲

acos

double_t `acos(double_t` *x*`);`

x A double-precision value.

The `acos` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

acosh

double_t `acosh(double_t` *x*`);`

x A double-precision value.

The `acosh` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

asin

double_t `asin(double_t` *x*`);`

x A double-precision value.

The `asin` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

asinh

double_t `asinh(double_t` *x*`);`

x A double-precision value.

The `asinh` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

atan

```
double_t atan(double_t    x);
```

x A double-precision value.

The `atan` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

atan2

```
double_t atan2(  double_t x,
                 double_t y);
```

x A double-precision value.

y A double-precision value.

The `atan2` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

atanh

```
double_t atanh(double_t   x);
```

x A double-precision value.

The `atanh` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

ceil

```
double_t ceil(double_t    x);
```

x A double-precision value.

The `ceil` function is part of the standard ANSI-C library.

copysign

```
double_t copysign(  double_t x,
                    double_t y);
```

x A double-precision value.

y A double-precision value.

The `copysign` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

copysignf

```
float copysign(   float    x,
                  float    y);
```

x A floating-point value.

y A floating-point value.

The `copysignf` function is part of the standard ANSI-C library. This function is the same as the copysign function, except that `copysignf` takes floating-point values for arguments and returns a floating-point value. The Newton implementation of this function is documented as the `copysign` function in the "Utility Functions" chapter of *Newton Programmer's Guide*.

cos

```
double_t cos(double_t     x);
```

x A double-precision value.

The `cos` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

cosh

```
double_t cosh(double_t    x);
```

x A double-precision value.

The `cosh` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

erf

```
double_t erf(double_t     x);
```

x A double-precision value.

The `erf` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

erfc

```
double_t erfc(double_t x);
```

x A double-precision value.

The `erfc` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

exp

```
double_t exp(double_t x);
```

x A double-precision value.

The `exp` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

exp2

```
double_t exp2(double_t x);
```

x A double-precision value.

The `exp2` function is part of the standard ANSI-C library.

expm1

```
double_t expm1(double_t x);
```

x A double-precision value.

The `expm1` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

fabs

```
double_t fabs(double_t x);
```

x A double-precision value.

The `fabs` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

fdim

```
double_t fdim( double_t x,
               double_t y);
```

x A double-precision value.

y A double-precision value.

The `fdim` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

floor

```
double_t floor(double_t x);
```

x A double-precision value.

The `floor` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

fmax

```
double_t fmax( double_t    x,
               double_t    y);
```

x A double-precision value.

y A double-precision value.

The `fmax` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

fmin

```
double_t fmin( double_t    x,
               double_t    y);
```

x A double-precision value.

y A double-precision value.

The `fmax` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

fmod

```
double_t fmod( double_t    x,
               double_t    y);
```

x A double-precision value to be divided (the dividend).

y The double-precision divisor.

The `fmod` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

frexp

```
double_t frexp(   double_t     x,
                  int          *exponent);
```

x A double-precision value.

exponent On exit, the exponent of *x*.

The `frexp` function is part of the standard ANSI-C library.

hypot

```
double_t hypot(   double_t x,
                  double_t y );
```

x A double-precision value.

y A double-precision value.

The `hypot` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

isfinite

```
int isfinite(long double x);
```

x A long double-precision value.

The `isfinite` function is documented in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

isnan

```
int isnan(long double x);
```

x A long double-precision value.

The `isnan` function is documented in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

isnormal

```
int isnormal(long double x);
```

x A long double-precision value.

The `isnormal` function is documented in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

ldexp

```
double_t ldexp(   double_t    x,
                  int         n);
```

x The double-precision mantissa value.

n The exponent value.

The `ldexp` function is part of the standard ANSI-C library.

log

```
double_t log(double_t     x);
```

x A double-precision value.

The `log` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

logb

```
double_t logb(double_t    x);
```

x A double-precision value.

The `logb` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

log1p

```
double_t log1p(double_t   x);
```

x A double-precision value.

The `log1p` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

log10

```
double_t log10(double_t   x);
```

x A double-precision value.

The `log10` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the “Utility Functions” chapter of *Newton Programmer’s Guide*.

log2

```
double_t log2(double_t    x);
```

x A double-precision value.

The `log` function is part of the standard ANSI-C library.

modf

```
double modf(  double      x,
              double      *iptr );
```

x A double-precision value.

iptr On exit, the integral part of *x*.

The `modf` function is part of the standard ANSI-C library.

modff

```
float modff(float      x,
            float      *iptrf);
```

x — A floating-point value.

iptr — On exit, the integral part of *x,* stored as a floating point value.

The `modff` function is part of the standard ANSI-C library.

nearbyint

```
double_t nearbyint(double_t  x);
```

x — A double-precision value.

The `nearbyint` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

nextafterd

```
double nextafterd( double   x,
                   double   y);
```

x — A double-precision value.

y — A double-precision value.

The `nextafterd` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

nextafterf

```
float nextafterf(float    x,
                 float    y);
```

x — A double-precision value.

y — A double-precision value.

The `nextafterf` function is part of the standard ANSI-C library.

pow

```
double_t pow( double_t    x,
              double_t    y);
```

x — A double-precision value.

y — A double-precision number representing the power.

The `pow` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

randomx

```
double_t randomx(double_t*   x);
```

x On entry, the seed value. On exit, the new seed value.

The `randomx` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

relation

```
relop relation(double_t   x,
               double_t   y);
```

x A double-precision value.

y A double-precision value.

The `relation` function is part of the standard ANSI-C library.

remainder

```
double_t remainder( double_t x,
                    double_t y);
```

x A double-precision value to be divided (the dividend).

y The double-precision divisor.

The `remainder` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

remquo

```
double_t remquo( double_t    x,
                 double_t    y,
                 int*        quo );
```

x A double-precision value to be divided (the dividend).

y The double-precision divisor.

quo On exit, the seven low-order bits of *x* divided by *y* as a value between `-127` and `127`.

The `remquo` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

rint

double_t `rint(double_t` *x*`);`

x A double-precision value.

The `rint` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

rinttol

long int `rinttol(double_t` *x*`);`

x A double-precision value.

The `rinttol` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

round

double_t `round(double_t` *x*`);`

x A double-precision value.

The `round` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

roundtol

long int `roundtol(double_t` *round*`);`

x A double-precision value.

The `roundtol` function is part of the standard ANSI-C library.

scalb

```
double_t scalb(  double_t   x,
                 long int   n);
```

x A double-precision value.

n A double-precision value.

The `scalb` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

signbit

int `signbit(long double` *x*`);`

x A long double-precision value.

The `signbit` function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

sin

double_t `sin(double_t` *x*`);`

x A double-precision value.

The `sin` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

sinh

double_t `sinh(double_t` *x*`);`

x A double-precision value.

The `sinh` function returns is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

sqrt

double_t `sqrt(double_t` *x*`);`

x A double-precision value.

The `sqrt` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

tan

double_t `tan(double_t` *x*`);`

x A double-precision value. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

The `tan` function is part of the standard ANSI-C library.

tanh

double_t `tanh(double_t` *x*`);`

x A double-precision value.

The `tanh` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

trunc

```
double_t trunc(double_t    x);
```

x A double-precision value.

The `trunc` function is part of the standard ANSI-C library. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

Financial Functions

This section describes the C Library functions that you can use to compute financial values.

annuity

```
double_t annuity( double_t rate,
                  double_t periods);
```

rate The interest rate per period.

periods The number of periods for which to compound interest.

The `annuity` function calculates the present value factor of an annuity at the specified interest *rate* over the specified number of *periods*. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

compound

```
double_t compound(   double_t rate,
                     double_t periods);
```

rate The interest rate per period.

periods The number of periods for which to compound interest.

The `compound` function calculates the compounded interest factor for the specified interest *rate* over the specified number of *periods*. The Newton implementation of this function is documented in the "Utility Functions" chapter of *Newton Programmer's Guide*.

Variable Argument List Macros

This section describes the C Library macros you can use to define functions that take a variable number of arguments..

va_start

```
void va_start(  va_list  ap,
                         parmN);
```

ap A va_list object that you have declared.

parmN A buffer in which to save the current program environment.

The `va_start` macro is part of the standard ANSI-C library.

va_arg

```
type va_arg( va_list  ap,
                      type);
```

ap A `va_list` object that has been initialized by the `va_start` macro.

type The type of the object that you are retrieving from the variable argument list.

The `va_arg` macro is part of the standard ANSI-C library.

va_end

```
void va_end(va_list   ap);
```

ap A `va_list` object that has been initialized by the `va_start` macro.

The `va_end` macro is part of the standard ANSI-C library.

Standard Input and Output Functions

This section describes the C Library functions for standard input and output processing.

sprintf

```
int sprintf(char*        s,
            const char*  format,
            ...);
```

s The string into which you want to write. See the warning below for special considerations regarding floating-point strings.

format The format specification string, which tells sprintf how to convert its arguments and write them into *s*.

... The data arguments. A variable number of objects, each of which is a pointer to an object that is to be converted into a string. The first argument points to the first value to convert, the second pointer points to the second value to convert, and so on.

The `sprintf` function is part of the standard ANSI-C library.

WARNING

The Newton implementation of `sprintf` has problems with conversion of floating-point values that have out-or-range exponents. If you supply a string representation of a float that is too large (one that evaluates to `INF`), `sprintf` hangs up, forcing you to reboot the system. You can get around this problem by following one of these rules:

1. Convert floating-point strings from ASCII to double before calling sprintf, and avoid using single-precision floating-point values.
2. Perform your own range checking on doubles to ensure that `INF` values do not get passed to `sprintf`.

▲

IMPORTANT

The Newton implementation of `sprintf` adds the `'%U'` directive for Unicode strings. The `'%U'` directive converts the Unicode string to the Macintosh Roman character set and prints it. ▲

sscanf

```
int sscanf(char*        s,
           const char*  format,
           ...);
```

s	The string that you want converted.
format	The format specification string, which tells `sscanf` how to convert the contents of *s*.
...	The data arguments. A variable number of objects, each of which is a pointer to the object that is to receive the converted value. The first pointer receives the first converted value, the second pointer receives the second converted value, and so on.

The `sscanf` function is part of the standard ANSI-C library.

vsprintf

```
int vsprintf(char*        s,
             const char*  format,
             _va_list     arg);
```

s	The string into which you want to write.
format	The format specification string, which tells vsprintf how to convert its arguments and write them into *s*.
arg	A pointer to an argument list, which contains the objects that you want converted into strings.

The `vsprintf` function is part of the standard ANSI-C library.

The `vsprintf` function is the same as the `sprintf` function, except that `vsprintf` takes a pointer to an argument list as its third and final parameter. The `sprintf` function is described in the section "sprintf" beginning on page 8-18.

IMPORTANT
The Newton implementation of `vsprintf` adds the `'%U'` directive for Unicode strings. The `'%U'` directive converts the Unicode string to the Macintosh Roman character set and prints it. ▲

Standard C Library Functions

This section describes the C Library functions that are part of the standard ANSI C Library definition.

_ANSI_rand

int `_ANSI_rand(void);`

The `_ANSI_rand` function is part of the standard ANSI-C library.

_ANSI_srand

void `_ANSI_srand(unsigned int` *seed*`);`

seed An unsigned integer value.

abs

int `abs(int` *j*`);`

j An integer value.

The `abs` function is part of the standard ANSI-C library.

atof

double `atof(const char*` *nptr*`);`

nptr A character string.

The `atof` function is part of the standard ANSI-C library.

atoi

int `atoi(const char*` *nptr*`);`

nptr A character string.

The `atoi` function is part of the standard ANSI-C library

atol

```
long int atol(const char* nptr);
```

nptr — A character string.

The `atol` function is part of the standard ANSI-C library

bsearch

```
void* bsearch(  const void* key,
                const void* base,
                size_t      nmemb,
                size_t      size,
                int         (* compar)(  const void *key,
                                         const void *data));
```

key — A pointer to the object that you want to be matched in the array.

base — A pointer to the initial element in the array to be searched.

nmemb — The number of objects in the array.

size — The size, in bytes, of each array element.

compar — A pointer to a comparison function. This is a function that compares a *key* value with a *data* value (from the array) and returns a value that describes the comparison.

The `bsearch` function is part of the standard ANSI-C library.

div

```
div_t div( int numer,
           int denom);
```

numer — The numerator value.

denom — The denominator value.

The `div` function is part of the standard ANSI-C library. It fills in the fields of a `div_t` structure with the results. The `div_t` structure is described in the section “The Division Result Type” on page 2.

Note

The Newton implementation of the `div` function does not generate an exception if the value of *denom* is `0`. The `div` function returns `0` as its result without generating an exception. ◆

labs

```
long int labs(long int j);
```

j — A long integer value.

The `labs` function is part of the standard ANSI-C library.

ldiv

```
ldiv_t ldiv(   long int numer,
               long int denom);
```

numer The numerator value.

denom The denominator value.

The `ldiv` function is part of the standard ANSI-C library. It fills in the fields of an `ldiv_t` structure with the results. The `ldiv_t` structure is described in the section "The Long Division Result Type" on page 3.

Note
The Newton implementation of the `ldiv` function does not generate an exception if the value of *denom* is `0`. The `ldiv` function returns `0` as its result without generating an exception. ◆

qsort

```
void* qsort(  const void* base,
              size_t      nmemb,
              size_t      size,
              int         (* compar)(  const void* e1,
                                       const void* e2));
```

base A pointer to the initial element in the array to be sorted.

nmemb The number of objects in the array.

size The size, in bytes, of each array element.

compar A pointer to a comparison function. This is a function that compares two elements of the array and returns a value that describes the comparison.

The `qsort` function is part of the standard ANSI-C library.

rand

```
int rand(void);
```

The `rand` function is part of the standard ANSI-C library.

srand

```
void srand(unsigned int seed);
```

seed An unsigned integer value.

The `srand` function is part of the standard ANSI-C library.

strtod

```
double strtod(const char* nptr,
              char**      endptr);
```

nptr A pointer to a string.

endptr On exit, a pointer to the remainder of the string.

The `strtod` function is part of the standard ANSI-C library

strtol

```
long int strtol( const char* nptr,
                 char**      endptr,
                 int         base);
```

nptr A pointer to a string.

endptr On exit, a pointer to the remainder of the string.

base The number base of the value.

The `strtol` function is part of the standard ANSI-C library.

strtoul

```
unsigned long int strtoul(const char* nptr,
                          char**      endptr,
                          int         base);
```

nptr A pointer to a string.

endptr On exit, a pointer to the remainder of the string.

base The number base of the value.

The `strtoul` function is part of the standard ANSI-C library.

Heap Functions

This section describes the C Library functions that you can use to allocate and free memory in the heap.

calloc

```
void* calloc( size_t nmemb,
              size_t size);
```

nmemb The number of array members in the block that you want allocated.

size The size, in bytes, of each array member.

The `calloc` function is part of the standard ANSI-C library.

free

void `free(void* ptr);`

ptr A pointer to a block of memory in the heap.

The `free` function is part of the standard ANSI-C library.

Note

The `free` function is the same as the Newton Memory Manager function `DisposePtr`. ◆

malloc

*void** `malloc(size_t size);`

size The size, in bytes, of the block of memory that you want allocated.

The `malloc` function is part of the standard ANSI-C library.

WARNING

The Newton implementation of the `malloc` function does not protect against negative or extremely large *size* values. It attempts to allocate the specified amount of memory, even though such values can cause disastrous results in your program. You must ensure that your calls to `malloc` supply appropriate *size* values. ▲

Note

The `malloc` function is the same as the Newton Memory Manager function `NewPtr`. ◆

realloc

```
void* realloc(  void*    ptr,
                size_t   size);
```

ptr A pointer to a block of memory in the heap.

size The new size for the object, in bytes.

The `realloc` function is part of the standard ANSI-C library.

Note

The `realloc` function behaves differently than the standard, ANSI C library implementation in one case. If the value of *size* is `0`, `realloc` does not free *ptr*; instead, it sets the size of the buffer pointed to by *ptr* to `0`, which indicates that the Newton System Software can free the pointer at a later time. ◆

Note

The `realloc` function is the same as the Newton Memory Manager function `ReallocPtr`. ◆

Memory Block Manipulation Functions

This section describes the C Library functions that you can use to work with memory blocks.

memchr

```
void* memchr( const void*    s,
              int            c,
              size_t         n);
```

s A pointer to the string to be searched.

c A character to search for in *s*.

n The number of characters to search in *s*.

The `memchr` function is part of the standard ANSI-C library.

memcmp

```
int memcmp(   const void* s1,
              const void* s2,
              size_t      n);
```

s1 A pointer to a block of memory.

s2 A pointer to a block of memory.

n The number of characters to compare.

The `memcmp` function is part of the standard ANSI-C library.

memcpy

```
void* memcpy( void*       s1,
              const void* s2,
              size_t      n);
```

s1 A pointer to a block of memory.

s2 A pointer to a block of memory.

n The number of characters to copy.

The `memcpy` function is part of the standard ANSI-C library.

memmove

```
void* memmove(   void*       s1,
                 const void* s2,
```

```
                size_t      n);
```

s1 A pointer to a block of memory.

s2 A pointer to a block of memory.

n The number of characters to copy.

The `memmove` function is part of the standard ANSI-C library.

Note
The `memmove` function is the same as the Newton Memory Manager function `BlockMove`. ◆

memset

```
void* memset( void*    s,
              int      c,
              size_t   n);
```

s A pointer to a block of memory.

c A character.

n The number of characters to initialize.

The `memset` function is part of the standard ANSI-C library.

WARNING
The `memset` function does not protect against negative or extremely large *n* values. It attempts to allocate the specified amount of memory, even though such values can cause disastrous results in your program. You must ensure that your calls to `FillBytes` supply appropriate *n* values. ▲

Note
The `memset` function is the same as the Newton Memory Manager function `FillBytes`. ◆

String Manipulation Functions

This section describes the C Library functions that you can use to work with strings.

strcat

```
char* strcat( char*       s1,
              const char* s2);
```

s1 A pointer to a null-terminated string.

s2 A pointer to a null-terminated string.

The `strcat` function is part of the standard ANSI-C library.

strchr

```
char* strchr( const char* s,
              int         c);
```

s A pointer to a null-terminated string.

c A character.

The `strchr` function is part of the standard ANSI-C library

strcmp

```
int strcmp(  const char* s1,
             const char* s2);
```

s1 A pointer to a null-terminated string.

s2 A pointer to a null-terminated string.

The `strcmp` function is part of the standard ANSI-C library.

strcoll

```
int strcoll( const char* s1,
             const char* s2);
```

s1 A pointer to a null-terminated string.

s2 A pointer to a null-terminated string.

The `strcoll` function is part of the standard ANSI-C library.

strcpy

```
char* strcpy( char*       s1,
              const char* s2);
```

s1 A pointer to a null-terminated string.

s2 A pointer to a null-terminated string.

The `strcpy` function is part of the standard ANSI-C library.

strcspn

```
size_t strcspn(  const char* s1,
                 const char* s2);
```

s1 A pointer to a null-terminated string.

s2 A pointer to a null-terminated string.

The `strcspn` function is part of the standard ANSI-C library.

strlen

size_t `strlen(const char* s);`

s A pointer to a null-terminated string.

The `strlen` function is part of the standard ANSI-C library.

strncat

```
char* strncat(   char*       s1,
                 const char* s2,
                 size_t      n);
```

s1 A pointer to a null-terminated string.

s2 A pointer to a null-terminated string.

n The maximum number of characters to copy.

The `strncat` functionis part of the standard ANSI-C library.

strncmp

```
int strncmp( const char* s1,
             const char* s2,
             size_t      n);
```

s1 A pointer to a null-terminated string.

s2 A pointer to a null-terminated string.

n The number of characters to compare.

The `strncmp` function is part of the standard ANSI-C library.

strncpy

```
char* strncpy(   char*       s1,
                 const char* s2,
                 size_t      n);
```

s1 A pointer to a null-terminated string.

s2 A pointer to a null-terminated string.

n The maximum number of characters to copy.

The `strncpy` function is part of the standard ANSI-C library.

strpbrk

```
char* strpbrk(   const char* s1,
                 const char* s2);
```

s1 A pointer to a null-terminated string.

s2 A pointer to a null-terminated string.

The `strpbrk` function is part of the standard ANSI-C library.

strrchr

```
char* strrchr(   const char* s,
                 int         c);
```

s A pointer to a null-terminated string.

c A character.

The `strrchr` function is part of the standard ANSI-C library.

strspn

```
size_t strspn(const char* s1,
              const char* s2);
```

s1 A pointer to a null-terminated string.

s2 A pointer to a null-terminated string.

The `strspn` function is part of the standard ANSI-C library.

strstr

```
char* strstr( const char* s1,
              const char* s2);
```

s1 A pointer to a null-terminated string.

s2 A pointer to a null-terminated string.

The `strstr` function is part of the standard ANSI-C library.

strtok

```
char* strtok( char*       s1,
              const char* s2);
```

s1 A pointer to null-terminated string.

s2 A pointer to a null-terminated string.

The `strtok` function is part of the standard ANSI-C library.

strxfrm

```
size_t strxfrm(  char*        s1,
                 const char*  s2,
                 size_t       n);
```

s1	A pointer to a string into which characters are copied. This string must be long enough to contain n+1 characters.
s2	A pointer to a string to be copied.
n	The number of characters to copy.

The `strxfrm` function is part of the standard ANSI-C library.

Time Functions

This section describes the C Library functions that you can use to work with clock and processor time values.

asctime

```
char* asctime(const struct tm*  timeptr);
```

The `asctime` function is not available for use on the Newton. Use the `asctime_newton` function instead. The `asctime_newton` function is described in the next section.

asctime_newton

```
char* asctime_newton(  const struct tm*  timeptr,
                       char*             timebuf);
```

timeptr	A pointer to a calendar clock time structure. The calendar clock time structure is described on page 8-4.
timebuf	A character buffer. You must allocate at least 70 bytes for this buffer.

The `asctime` function is a Newton C++ Toolkit variation of the standard C library function `asctime`.

The `asctime_newton` function differs from the `asctime` function in that you must preallocate the output buffer *timebuf*.

The `asctime_newton` function returns *timebuf* as its function value.

WARNING
You must allocate *timebuf* by calling either the `NewPtr` function or the `malloc` function, or you can declare *timebuf* as a local variable within a function. You cannot declare *timebuf* as a static global variable. ▲

clock

```
clock_t clock(void);
```

The `clock` function is part of the standard ANSI-C library.

WARNING
You cannot use the `clock` function on the Newton in the same way as you can on many other computing devices. This is because your application is sharing task space with other applications, which means that the concept of "CPU task time" is distorted on the Newton. You thus cannot use the difference between two calls to `clock` to determine how long it took your application to perform an operation. ▲

ctime

```
char* ctime(const time_t* timer);
```

The `ctime` function is not available for use on the Newton. Use the `ctime_newton` function instead. The `ctime_newton` function is described in the next section.

ctime_newton

```
char* ctime_newton( const time_t*  timer,
                    char*          timebuf);
```

timer A pointer to a time_t value. The `time_t` type is described on page 8-4.

timebuf A character buffer. You must allocate at least 70 bytes for this buffer.

The `ctime` function is a Newton C++ Toolkit variation of the standard C library function `ctime`.

The `ctime_newton` function differs from the `ctime` function in that you must preallocate the output buffer *timebuf*.

The `ctime_newton` function returns *timebuf* as its function value.

WARNING
You must allocate *timebuf* by calling either the `NewPtr` function or the `malloc` function, or you can declare *timebuf* as a local variable within a function. You cannot declare *timebuf* as a static global variable. ▲

difftime

```
double difftime( time_t   time1,
                 time_t   time0);
```

time1 The second calendar clock time reading value.

time0 The first calendar clock time reading value.

The `difftime` function is part of the standard ANSI-C library.

gmtime

*struct tm** `gmtime(const time_t* timer);`

timer — A pointer to a `time_t` value. The `time_t` type is described on page 8-4.

The `gmtime` function is part of the standard ANSI-C library. The Newton implementation of this function does not perform any computation and returns `NIL`.

WARNING
The Newton implementation of `gmtime` simply returns `NIL`. ▲

localtime

*struct tm** `localtime(const time_t* timer);`

The `localtime` function is not available for use on the Newton. Use the `localtime_newton` function instead. The `localtime_newton` function is described in the next section.

localtime_newton

```
struct tm* localtime_newton(  const time_t*  timer,
                              tm*            tms);
```

timer — A pointer to a `time_t` value. The `time_t` type is described on page 8-4.

tms — A pointer to a calendar clock time structure that you have allocated in your application. The calendar clock time structure is described on page 8-4.

The `localtime` function is a Newton C++ Toolkit variation of the standard C library function `localtime`.

The `localtime_newton` function differs from the `localtime` function in that you must preallocate the output calendar clock time structure.

The `localtime_newton` function returns *tms* as its function value.

WARNING
You must allocate the output calendar clock structure by calling either the `NewPtr` function or the `malloc` function, or you can declare a `tm` structure within a function in your application and pass in a pointer to that structure as the value of *tms*. You cannot declare the structure as a static global variable. ▲

mktime

time_t `mktime(struct tm* timeptr);`

timeptr — A pointer to a calendar clock time structure. The calendar clock time structure is described on page 8-4.

The `mktime` function is part of the standard ANSI-C library.

strftime

```
size_t strftime( char*              s,
                 size_t             maxsize,
                 const char*        format,
                 const struct tm*   timeptr);
```

s — A pointer to a string. On exit, this is the formatted, string representation of the time.

maxsize — The maximum number of characters to store into *s*.

format — A format specification.

timeptr — A pointer to a calendar clock time structure that contains the time you want formatted.

The `strftime` function is part of the standard ANSI-C library.

time

```
time_t time(time_t* timer);
```

timer — A pointer to a time structure that you want filled in with the current time. On exit, this is filled in with the current time. You can specify `NULL` as the value of *timer* if you don't want a structure to be filled in.

The `time` function is part of the standard ANSI-C library.

Summary of C Library Reference

C Library Constants and Types

```
#define NULL 0
#define HUGE_VAL  _inf();
#define RAND_MAX 0x7fffffff
```

Standard Library Types

```
typedef unsigned int size_t;

typedef int wchar_t;

typedef struct div_t {
   int   quot;
   int   rem;
} div_t;

typedef struct ldiv_t {
   long int quot;
   long int rem;
} ldiv_t;
```

Math Types

```
typedef short relop;
enum {
      GREATERTHAN = ( ( relop ) ( 0 ) ),
      LESSTHAN,
      EQUALTO,
      UNORDERED
      };
```

Time Types

```
typedef unsigned int clock_t;
typedef unsigned int time_t;
```

```
struct tm {
   int tm_sec;
   int tm_min;
   int tm_hour;
   int tm_mday;
   int tm_mon;
   int tm_year;
   int tm_wday;
   int tm_yday;
   int tm_isdst;
};
```

C Library Functions

Character Conversion Functions

int `tolower(int` *c*`);`

int `toupper(int` *c*`);`

Floating-point Math Functions

double_t `acos(double_t` *x*`);`

double_t `acosh(double_t` *x*`);`

double_t `asin(double_t` *x*`);`

double_t `asinh(double_t` *x*`);`

double_t `atan(double_t` *x*`);`

double_t `atan2( double_t` *x*`, double_t` *y*`);`

double_t `atanh(double_t` *x*`);`

double_t `ceil(double_t` *x*`);`

double_t `copysign(double_t` *x*`, double_t` *y*`);`

float `copysignf(float` *x*`, float` *y*`);`

double_t `cos(double_t` *x*`);`

double_t `cosh(double_t` *x*`);`

double_t `erf(double_t` *x*`);`

double_t `erfc(double_t` *x*`);`

long double
`erfcl(long double` *x*`);`

long double
`erfl(long double` *x*`);`

double_t `exp(double_t` *x*`);`

```
double_t   exp2(double_t x);
double_t   expm1(double_t x);
double_t   fabs(double_t x);
double_t   fdim(double_t x, double_t y);
double_t   floor(double_t x);
double_t   fmax(double_t x, double_t y);
double_t   fmin(double_t x, double_t y);
double_t   fmod(double_t x, double_t y);
double_t   frexp(double_t x, int* exponent,
double_t   hypot(double_t x, double_t y);
int        isfinite(long double x);
int        isnormal(long double x);
int        isnan(long double x);
double_t   ldexp(double_t x, int n);
double_t   lgamma(double_t x);
double_t   log(double_t x);
double_t   logb(double_t x);
double_t   log1p(double_t x);
double_t   log10(double_t x);
double_t   log2(double_t x);
double     modf(double x, double* iptr);
float      modff(float x, float* iptr);
double_t   nearbyint(double_t x);
double     nextafterd(double x, double y);
float      nextafterf(float x, float y);
double_t   pow(double_t x, double_t y);
double_t   randomx(double_t* x);
relop      relation(double_t x, double_t y);
double_t   remainder(double_t x, double_t y);
double_t   remquo(double_t x, double_t y, int* quo);
double_t   rint(double_t x);
long int   rinttol(double_t x);
double_t   round(double_t x);
long int   roundtol(double_t round);
double_t   scalb(double_t x, long int n);
int        signbit(long double x);
```

```
double_t    sin(double_t x);
double_t    sinh(double_t x);
double_t    sqrt(double_t x);
double_t    tan(double_t x);
double_t    tanh(double_t x);
double_t    trunc(double_t x);
```

Financial Functions

```
double_t    annuity(double_t rate, double_t periods);
double_t    compound(double_t rate, double_t periods);
```

Variable Argument List Macros

```
void        va_start(va_list ap, parmN);
type        va_arg(va_list ap, type);
void        va_end(va_list ap);
```

Standard Input and Output Functions

```
int         sprintf(char* s, const char* format, ...);
int         sscanf(char* s, const char* format, ...);
int         vsprintf(char* s, const char* format, _va_list arg);
```

Standard C Library Functions

```
int         _ANSI_rand(void);
void        _ANSI_srand(unsigned int seed);
int         abs(int j);
double      atof(const char* nptr);
int         atoi(const char* nptr);
long int    atol(const char* nptr);
void*       bsearch(const void* key, const void* base, size_t nmemb,
                  size_t size,
                  int(* compar)(const void *key, const void *data));
div_t       div(int numer, int denom);
long int    labs(long int j);
ldiv_t      ldiv(long int numer, long int denom);
void*       qsort(const void* base, size_t nmemb, size_t size,
                  int(* compar)(const void *e1, const void *e2));
int         rand(void);
```

void `srand(unsigned int` *seed*`);`

double `strtod(const char*` *nptr*`, char**` *endptr*`);`

long int `strtol(const char*` *nptr*`, char**` *endptr*`, int` *base*`);`

unsigned long int
`strtoul(const char*` *nptr*`, char**` *endptr*`, int` *base*`);`

Heap Functions

*void** `calloc(size_t nmemb, size_t size);`

void `free(void* ptr);`

*void** `malloc(size_t size);`

*void** `realloc(void* ptr, size_t size);`

Memory Block Manipulation Functions

*void** `memchr(const void* s, int c, size_t n);`

int `memcmp(const void* s1, const void* s2, size_t n);`

*void** `memcpy(void* s1, const void* s2, size_t n);`

*void** `memmove(void* s1, const void* s2, size_t n);`

*void** `memset(void* s, int c, size_t n);`

String Manipulation Functions

*char** `strcat(char* s1, const char* s2);`

*char** `strchr(const char* s, int c);`

int `strcmp(const char* s1, const char* s2);`

int `strcoll(const char* s1, const char* s2);`

*char** `strcpy(char* s1, const char* s2);`

size_t `strcspn(const char* s1, const char* s2);`

size_t `strlen(const char* s);`

*char** `strncat(char* s1, const char* s2, size_t n);`

int `strncmp(const char* s1, const char* s2, size_t n);`

*char** `strncpy(char* s1, const char* s2, size_t n);`

*char** `strpbrk(const char* s1, const char* s2);`

*char** `strrchr(const char* s, int c);`

size_t `strspn(const char* s1, const char* s2);`

*char** `strstr(const char* s1, const char* s2);`

*char** `strtok(char* s1, const char* s2);`

size_t `strxfrm(char* s1, const char* s2, size_t n);`

Time Functions

*char**	`asctime_newton(const struct tm* timeptr, char* timebuf);`
clock_t	`clock(void);`
*char**	`ctime_newton(const time_t* timer, tm* tms);`
double	`difftime(time_t time1, time_t time0);`
*struct tm**	`gmtime(const time_t* timer);`
*struct tm**	`localtime_newton(const time_t* timer, char* timebuf);`
time_t	`mktime(struct tm* timeptr);`
size_t	`strftime(char* s, size_t maxsize, const char* format, const struct tm* timeptr);`
time_t	`time(time_t* timer);`

C++ Function Tables

This appendix presents the name of each function in the C++ Toolkit and specifies where to find the description of that function. Some of the function descriptions are provided in this book, while others are located in other books.

The declaration (function header and parameter descriptions) for each function is given in this book.

Functions and Macros for Using C++ With NewtonScript

Table A-1 summarizes the functions and macros described in Chapter 2, "C++ and NewtonScript Conversion Reference.".

Table A-1 C++ and NewtonScript conversion functions and macros

Function Name	Page number
`Debugger`	2-6
`DebugStr`	2-6
`DebugCStr`	2-6
`EQ`	2-5
`IsChar`	2-4
`ISFALSE`	2-5
`IsInt`	2-4
`IsMagicPtr`	2-4
`IsNIL`	2-5
`IsPtr`	2-4
`IsRealPtr`	2-4
`ISTRUE`	2-5
`MakeBoolean`	2-2
`MakeChar`	2-2
`MakeInt`	2-2
`MakeReal`	2-2
`MakeString`	2-2

Table A-1 C++ and NewtonScript conversion functions and macros (continued)

Function Name	Page number
`MakeSymbol`	2-3
`NOTNIL`	2-5
`RefToInt`	2-3
`RefToUniChar`	2-3
`SYM`	2-3

Newton Object System Functions

Table A-2 shows the location of the description for each of the Newton Object System functions in the C++ Toolkit. The declaration for each of these functions is provided in Chapter 3, "Newton Object System Reference."

Table A-2 C++ Toolkit Object System functions

Function Name	Location of function description	Function header page in C++ book
`AddArraySlot`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-5
`AllocateArray`	In Chapter 3, "Newton Object System Reference."	3-6
`AllocateBinary`	In Chapter 3, "Newton Object System Reference."	3-6
`AllocateFrame`	In Chapter 3, "Newton Object System Reference."	3-6
`ArrayMunger`	The "Utility Functions" chapter of *Newton Programmer's Guide*.†	3-6
`ArrayPosition`	As `ArrayPos` in the "Utility Functions" chapter of *Newton Programmer's Guide*.	3-7
`ArrayRemove`	In Chapter 3, "Newton Object System Reference."	3-7
`ArrayRemoveCount`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-8
`ASCIIString`	In Chapter 3, "Newton Object System Reference."	3-8

Table A-2 C++ Toolkit Object System functions (continued)

Function Name	Location of function description	Function header page in C++ book
`BinaryMunger`	The "Utility Functions" chapter of *Newton Programmer's Guide*.†	3-8
`ClassOf`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-9
`Clone`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-9
`CoerceToDouble`	In Chapter 3, "Newton Object System Reference."	3-9
`CoerceToInt`	In Chapter 3, "Newton Object System Reference."	3-10
`DeepClone`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-10
`DeleteTObjectIterator`	In Chapter 3, "Newton Object System Reference."	3-5
`Done`	In Chapter 3, "Newton Object System Reference."	3-4
`EnsureInternal`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-10
`FrameHasPath`	The `HasPath` function in The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-10
`FrameHasSlot`	The `HasSlot` function in The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-10
`GC`	In Chapter 3, "Newton Object System Reference."	3-11
`GetArraySlot`	In Chapter 3, "Newton Object System Reference."	3-11
`GetFramePath`	In Chapter 3, "Newton Object System Reference."	3-11
`GetFrameSlot`	The `GetSlot` method in The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-12
`IsArray`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-12
`IsBinary`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-12
`IsFrame`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-12

Table A-2 C++ Toolkit Object System functions (continued)

Function Name	Location of function description	Function header page in C++ book
`IsFunction`	In Chapter 3, "Newton Object System Reference."	3-12
`IsInstance`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-13
`IsNumber`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-13
`IsReadOnly`	In Chapter 3, "Newton Object System Reference."	3-13
`IsReal`	In Chapter 3, "Newton Object System Reference."	3-13
`IsString`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-13
`IsSubclass`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-13
`IsSymbol`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-14
`Length`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-14
`NewTObjectIterator`	In Chapter 3, "Newton Object System Reference."	3-5
`Next`	In Chapter 3, "Newton Object System Reference."	3-4
`RemoveSlot`	The "Utility Functions" chapter of *Newton Programmer's Guide*.†	3-14
`ReplaceObject`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-14
`Reset`	In Chapter 3, "Newton Object System Reference."	3-4
`SetArraySlot`	In Chapter 3, "Newton Object System Reference."	3-14
`SetClass`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-15
`SetFramePath`	In Chapter 3, "Newton Object System Reference."	3-15
`SetFrameSlot`	In Chapter 3, "Newton Object System Reference."	3-16
`SetLength`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-16

Table A-2 C++ Toolkit Object System functions (continued)

Function Name	Location of function description	Function header page in C++ book
`SortArray`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-16
`Statistics`	In Chapter 3, "Newton Object System Reference."	3-17
`StrBeginsWith`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-17
`StrCapitalize`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-17
`StrCapitalizeWords`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-17
`StrDowncase`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-17
`StrEndsWith`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-18
`StrMunger`	The "Utility Functions" chapter of *Newton Programmer's Guide*.†	3-18
`StrPosition`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-18
`StrReplace`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-19
`StrUpcase`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-19
`Substring`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-19
`SymbolCompareLex`	In Chapter 3, "Newton Object System Reference."	3-20
`symcmp`	In Chapter 3, "Newton Object System Reference."	3-20
`Tag`	In Chapter 3, "Newton Object System Reference."	3-4
`ThrowBadTypeWithFrameData`	In Chapter 3, "Newton Object System Reference."	3-20
`ThrowRefException`	In Chapter 3, "Newton Object System Reference."	3-21
`TotalClone`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-21

Table A-2 C++ Toolkit Object System functions (continued)

Function Name	Location of function description	Function header page in C++ book
`TrimString`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	3-21
`Value`	In Chapter 3, "Newton Object System Reference."	3-5

† Although this C++ function is a wrapper for a NewtonScript method, there are some slight differences in parameter usage and/or return value semantics. These differences are described with the function declaration in this book.

C++ Toolkit Memory Manager Functions

Table A-3 shows the location of the description for each of the Newton Memory Manager functions in the C++ Toolkit. The declaration for each of these functions is provided in Chapter 4, "Newton Memory Manager Reference."

Table A-3 C++ Toolkit Memory Manager functions

Function Name	Location of function description	Function header page in C++ book
`BlockMove`	In Chapter 4, "Newton Memory Manager Reference."	4-1
`CountFreeBlocks`	In Chapter 4, "Newton Memory Manager Reference."	4-2
`DisposPtr`	In Chapter 4, "Newton Memory Manager Reference."	4-2
`EqualBytes`	In Chapter 4, "Newton Memory Manager Reference."	4-2
`FillBytes`	In Chapter 4, "Newton Memory Manager Reference."	4-3
`FillLongs`	In Chapter 4, "Newton Memory Manager Reference."	4-3
`GetPtrName`	In Chapter 4, "Newton Memory Manager Reference."	4-4
`GetPtrSize`	In Chapter 4, "Newton Memory Manager Reference."	4-4
`LargestFreeInHeap`	In Chapter 4, "Newton Memory Manager Reference."	4-4

Table A-3 C++ Toolkit Memory Manager functions (continued)

Function Name	Location of function description	Function header page in C++ book
`MaxHeapSize`	In Chapter 4, “Newton Memory Manager Reference.”	4-4
`MemError`	In Chapter 4, “Newton Memory Manager Reference.”	4-4
`NewNamedPtr`	In Chapter 4, “Newton Memory Manager Reference.”	4-5
`NewPtr`	In Chapter 4, “Newton Memory Manager Reference.”	4-5
`NewPtrClear`	In Chapter 4, “Newton Memory Manager Reference.”	4-5
`ReallocPtr`	In Chapter 4, “Newton Memory Manager Reference.”	4-6
`SetPtrName`	In Chapter 4, “Newton Memory Manager Reference.”	4-7
`SystemRAMSize`	In Chapter 4, “Newton Memory Manager Reference.”	4-7
`TotalFreeInHeap`	In Chapter 4, “Newton Memory Manager Reference.”	4-7
`TotalUsedInHeap`	In Chapter 4, “Newton Memory Manager Reference.”	4-7
`XORBytes`	In Chapter 4, “Newton Memory Manager Reference.”	4-8
`ZeroBytes`	In Chapter 4, “Newton Memory Manager Reference.”	4-8

C++ Toolkit Exception-Handling Functions

Table A-4 shows the location of the description of the Newton exception-handling functions in the C++ Toolkit. The declaration for each of these functions is provided in Chapter 5, “Newton Exceptions Reference.”

Table A-4 C++ Toolkit exception-handling functions

Function Name	Location of function description	Function header page in C++ book
`cleanup`	In Chapter 5, “Newton Exceptions Reference.”	5-9
`CurrentException`	In Chapter 5, “Newton Exceptions Reference.”	5-6
`end_try`	In Chapter 5, “Newton Exceptions Reference.”	5-10
`end_unwind`	In Chapter 5, “Newton Exceptions Reference.”	5-10
`newton_catch`	In Chapter 5, “Newton Exceptions Reference.”	5-10
`newton_catch_all`	In Chapter 5, “Newton Exceptions Reference.”	5-10
`newton_try`	In Chapter 5, “Newton Exceptions Reference.”	5-11
`on_unwind`	In Chapter 5, “Newton Exceptions Reference.”	5-11
`rethrow`	In Chapter 5, “Newton Exceptions Reference.”	5-7
`Subexception`	In Chapter 5, “Newton Exceptions Reference.”	5-8
`Throw`	In Chapter 5, “Newton Exceptions Reference.”	5-8
`ThrowMsg`	In Chapter 5, “Newton Exceptions Reference.”	5-8
`unwind_failed`	In Chapter 5, “Newton Exceptions Reference.”	5-12
`unwind_protect`	In Chapter 5, “Newton Exceptions Reference.”	5-12

C++ NewtonScript Functions

Table A-5 shows the location of the description of the NewtonScript functions in the C++ Toolkit. The declaration for each of these functions is provided in Chapter 6, “NewtonScript Reference.”

Table A-5 C++ Toolkit NewtonScript functions

Function Name	Location of function description	Function header page in C++ book
`GetVariable`	In Chapter 6, “NewtonScript Reference.”	6-14
`NSCall`	In Chapter 6, “NewtonScript Reference.”	6-2
`NSCallWithArgArray`	In Chapter 6, “NewtonScript Reference.”	6-3
`NSCallGlobalFn`	In Chapter 6, “NewtonScript Reference.”	6-4
`NSCallGlobalFnWithArgArray`	In Chapter 6, “NewtonScript Reference.”	6-5
`NSSend`	In Chapter 6, “NewtonScript Reference.”	6-6
`NSSendWithArgArray`	In Chapter 6, “NewtonScript Reference.”	6-7
`NSSendIfDefined`	In Chapter 6, “NewtonScript Reference.”	6-8
`NSSendIfDefinedWithArgArray`	In Chapter 6, “NewtonScript Reference.”	6-10
`NSSendProto`	In Chapter 6, “NewtonScript Reference.”	6-10
`NSSendProtoWithArgArray`	In Chapter 6, “NewtonScript Reference.”	6-12
`NSSendProtoIfDefined`	In Chapter 6, “NewtonScript Reference.”	6-12
`NSSendProtoIfDefinedWithArgArray`	In Chapter 6, “NewtonScript Reference.”	6-14
`SetVariable`	In Chapter 6, “NewtonScript Reference.”	6-15

C++ Toolkit Unicode Functions

Table A-6 shows the location of the description of the Unicode functions in the C++ Toolkit. The declaration for each of these functions is provided in Chapter 7, "Newton Unicode Reference."

Table A-6 C++ Toolkit Unicode functions

Function Name	Location of function description	Function header page in C++ book
`ConvertFromUnicode`	In Chapter 7, "Newton Unicode Reference."	7-2
`ConvertUnicodeChar`	In Chapter 7, "Newton Unicode Reference."	7-3
`ConvertUnicodeCharacters`	In Chapter 7, "Newton Unicode Reference."	7-4
`ConvertToUnicode`	In Chapter 7, "Newton Unicode Reference."	7-3
`HasChars`	In Chapter 7, "Newton Unicode Reference."	7-4
`HasDigits`	In Chapter 7, "Newton Unicode Reference."	7-5
`HasSpaces`	In Chapter 7, "Newton Unicode Reference."	7-5
`IsPunctSymbol`	In Chapter 7, "Newton Unicode Reference."	7-5
`StripPunctSymbols`	In Chapter 7, "Newton Unicode Reference."	7-5
`Umemset`	In Chapter 7, "Newton Unicode Reference."	7-6
`Ustrcat`	In Chapter 7, "Newton Unicode Reference."	7-7
`Ustrchr`	In Chapter 7, "Newton Unicode Reference."	7-7
`Ustrcmp`	In Chapter 7, "Newton Unicode Reference."	7-7
`Ustrcpy`	In Chapter 7, "Newton Unicode Reference."	7-7

Table A-6 C++ Toolkit Unicode functions (continued)

Function Name	Location of function description	Function header page in C++ book
`Ustrlen`	In Chapter 7, "Newton Unicode Reference."	7-8
`Ustrncat`	In Chapter 7, "Newton Unicode Reference."	7-8
`Ustrncpy`	In Chapter 7, "Newton Unicode Reference."	7-8

C++ Toolkit ANSI-C Functions

Table A-7 shows the location of the description of the ANSI-C Library functions in the C++ Toolkit. The declaration for each of these functions is provided in Chapter 8, "Newton C Library Reference."

Note

Many of the C Library functions are described in the *Newton Programmer's Guide*; however, the NewtonScript function names are capitalized. You need to take this into consideration when reading the description of the Newton implementation of a C Library function. For example, to read about the C Library function `acos`, you need to look up the `Acos` function in the *Newton Programmer's Guide*. ◆

Table A-7 C++ Library ANSI-C Library functions

Function Name	Location of function description	Function header page in C++ book
`abs`	Refer to ANSI-C library documentation..	8-20
`acos`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-6
`acosh`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-6
`annuity`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-17
`asctime`	Not available. Use `asctime_newton` instead.	
`asctime_newton`	In Chapter 8, "Newton C Library Reference."	8-30

Table A-7 C++ Library ANSI-C Library functions (continued)

Function Name	Location of function description	Function header page in C++ book
`asin`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-6
`asinh`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-6
`atan`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-7
`atan2`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-7
`atanh`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-7
`atof`	Refer to ANSI-C library documentation.	8-20
`atoi`	Refer to ANSI-C library documentation.	8-20
`atol`	Refer to ANSI-C library documentation.	8-21
`bsearch`	Refer to ANSI-C library documentation.	8-21
`calloc`	Refer to ANSI-C library documentation.	8-23
`ceil`	Refer to ANSI-C library documentation.	8-7
`clock`	Refer to ANSI-C library documentation.	8-31
`compound`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-17
`copysign`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-7
`copysignf`	As the `copysign` function in the "Utility Functions" chapter of *Newton Programmer's Guide*.	8-8
`cos`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-8
`cosh`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-8
`ctime`	Not available. Use `ctime_newton` instead.	
`ctime_newton`	In Chapter 8, "Newton C Library Reference."	8-31
`difftime`	Refer to ANSI-C library documentation.	8-31
`div`	Refer to ANSI-C library documentation.	8-21
`erf`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-8

Table A-7 C++ Library ANSI-C Library functions (continued)

Function Name	Location of function description	Function header page in C++ book
`erfc`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-8
`exp`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-9
`exp2`	Refer to ANSI-C library documentation.	8-9
`expm1`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-9
`fabs`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-9
`fdim`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-9
`floor`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-9
`fmax`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-10
`fmin`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-10
`fmod`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-10
`free`	Refer to ANSI-C library documentation.	8-24
`frexp`	Refer to ANSI-C library documentation.	8-10
`gmtime`	Refer to ANSI-C library documentation.	8-32
`hypot`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-11
`isfinite`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-11
`isnan`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-11
`isnormal`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-11
`labs`	Refer to ANSI-C library documentation.	8-21
`ldexp`	Refer to ANSI-C library documentation.	8-11
`ldiv`	Refer to ANSI-C library documentation.	8-22
`localtime`	Not available. Use `localtime_newton` instead.	

Table A-7 C++ Library ANSI-C Library functions (continued)

Function Name	Location of function description	Function header page in C++ book
`localtime_newton`	In Chapter 8, “Newton C Library Reference.”	8-32
`log`	The “Utility Functions” chapter of *Newton Programmer’s Guide*.	8-11
`log10`	The “Utility Functions” chapter of *Newton Programmer’s Guide*.	8-12
`log1p`	The “Utility Functions” chapter of *Newton Programmer’s Guide*.	8-12
`log2`	Refer to ANSI-C library documentation.	8-12
`logb`	The “Utility Functions” chapter of *Newton Programmer’s Guide*.	8-12
`malloc`	Refer to ANSI-C library documentation.	8-24
`memchr`	Refer to ANSI-C library documentation.	8-25
`memcmp`	Refer to ANSI-C library documentation.	8-25
`memcpy`	Refer to ANSI-C library documentation.	8-25
`memmove`	Refer to ANSI-C library documentation.	8-25
`memset`	Refer to ANSI-C library documentation.	8-26
`mktime`	Refer to ANSI-C library documentation.	8-32
`modf`	Refer to ANSI-C library documentation.	8-12
`modff`	Refer to ANSI-C library documentation.	8-13
`nearbyint`	The “Utility Functions” chapter of *Newton Programmer’s Guide*.	8-13
`nextafterd`	The “Utility Functions” chapter of *Newton Programmer’s Guide*.	8-13
`nextafterf`	Refer to ANSI-C library documentation.	8-13
`pow`	The “Utility Functions” chapter of *Newton Programmer’s Guide*.	8-13
`qsort`	Refer to ANSI-C library documentation.	8-22
`rand`	Refer to ANSI-C library documentation.	8-22
`randomx`	The “Utility Functions” chapter of *Newton Programmer’s Guide*.	8-14
`realloc`	Refer to ANSI-C library documentation.	8-24
`relation`	Refer to ANSI-C library documentation.	8-14

Table A-7 C++ Library ANSI-C Library functions (continued)

Function Name	Location of function description	Function header page in C++ book
`remainder`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-14
`remquo`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-14
`rint`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-15
`rinttol`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-15
`round`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-15
`roundtol`	Refer to ANSI-C library documentation.	8-15
`scalb`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-15
`signbit`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-15
`sin`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-16
`sinh`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-16
`sprintf`	Refer to ANSI-C library documentation.†	8-18
`sqrt`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-16
`srand`	Refer to ANSI-C library documentation.	8-22
`sscanf`	Refer to ANSI-C library documentation.	8-19
`strcat`	Refer to ANSI-C library documentation.	8-26
`strchr`	Refer to ANSI-C library documentation.	8-27
`strcmp`	Refer to ANSI-C library documentation.	8-27
`strcoll`	Refer to ANSI-C library documentation.	8-27
`strcpy`	Refer to ANSI-C library documentation.	8-27
`strcspn`	Refer to ANSI-C library documentation.	8-27
`strftime`	Refer to ANSI-C library documentation.	8-33
`strlen`	Refer to ANSI-C library documentation.	8-28
`strncat`	Refer to ANSI-C library documentation.	8-28
`strncmp`	Refer to ANSI-C library documentation.	8-28

Table A-7 C++ Library ANSI-C Library functions (continued)

Function Name	Location of function description	Function header page in C++ book
`strncpy`	Refer to ANSI-C library documentation.	8-28
`strpbrk`	Refer to ANSI-C library documentation.	8-29
`strrchr`	Refer to ANSI-C library documentation.	8-29
`strspn`	Refer to ANSI-C library documentation.	8-29
`strstr`	Refer to ANSI-C library documentation.	8-29
`strtod`	Refer to ANSI-C library documentation.	8-23
`strtok`	Refer to ANSI-C library documentation.	8-29
`strtol`	Refer to ANSI-C library documentation.	8-23
`strtoul`	Refer to ANSI-C library documentation.	8-23
`strxfrm`	Refer to ANSI-C library documentation.	8-30
`tan`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-16
`tanh`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-16
`time`	Refer to ANSI-C library documentation.	8-33
`tolower`	Refer to ANSI-C library documentation.	8-5
`toupper`	Refer to ANSI-C library documentation.	8-5
`trunc`	The "Utility Functions" chapter of *Newton Programmer's Guide*.	8-17
`va_arg`	Refer to ANSI-C library documentation.	8-18
`va_end`	Refer to ANSI-C library documentation.	8-18
`va_start`	Refer to ANSI-C library documentation.	8-18
`vsprintf`	Refer to ANSI-C library documentation.†	8-19
`_ANSI_rand`	Refer to ANSI-C library documentation.	8-20
`_ANSI_srand`	Refer to ANSI-C library documentation.	8-20

† This implementation of the C library function may be slightly different than the standard implementation. Any variances are described with the function declaration in this book.

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro 630 printer. Final page negatives were output directly from the text and graphics files. Line art was created using Adobe™ Illustrator. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

WRITER
Gary Hillerson

ILLUSTRATOR
Peggy Kunz

EDITOR
David Schneider

PRODUCTION EDITOR
Rex Wolf

PROJECT MANAGER
Gerry Kane

Special thanks to Tom Waits and Bob Dylan

.

Index

Symbols

_ANSI_RAND 8-20
_ANSI_srand 8-20

A

abs 8-20
acos 8-6
acosh 8-6
AddArraySlot 3-5
AllocateArray 3-6
AllocateBinary 3-6
AllocateFrame 3-6
allocating memory 1-18
annuity 8-17
ArrayMunger 3-6
array objects 1-7
ArrayPosition 3-7
ArrayRemove 3-7
ArrayRemoveCount 3-8
ASCIIString 3-8
asctime 8-30
asctime_newton 8-30
asin 8-6
asinh 8-6
atan 8-7
atan2 8-7
atanh 8-7
atof 8-20
atoi 8-20
atol 8-21

B

BinaryMunger 3-8
binary objects 1-7
  accessing data in 1-10 to 1-11, 1-18
  access warnings 1-11
BlockMove 4-1
bsearch 8-21

C

C++ arrays 1-16
C++ functions
  accessing slot values from 6-14 to 6-15
  and memory management 1-2
  and the Newton screen 1-2
  arguments to 1-6
  as wrapper functions 1-6
  calling from NewtonScript 1-3, 6-15
  calling NewtonScript functions from 6-1 to 6-14
  code restrictions 1-2, 1-3 to 1-6
  memory allocation and deallocation 1-4
  name-mangling 1-4
  return values 1-6
  static variables 1-5
calendar clock time type 8-4
calendar time type 8-4
calling C++ from NewtonScript 1-3
calling NewtonScript from C++ 1-2
calloc 8-23
catch blocks 5-4
ceil 8-7
ClassOf 3-9
cleanup 5-9
C library constants
  HUGE_VAL 8-2
  NULL 8-2
  RAND_MAX 8-2
C library constants and data types 8-1 to 8-5
C library functions and macros 8-5 to 8-33
  _ANSI_RAND 8-20
  _ANSI_srand 8-20
  abs 8-20
  acos 8-6
  acosh 8-6
  annuity 8-17
  asctime 8-30
  asctime_newton 8-30
  asin 8-6
  asinh 8-6
  atan 8-7
  atan2 8-7
  atanh 8-7
  atof 8-20
  atoi 8-20
  atol 8-21
  bsearch 8-21
  calloc 8-23

ceil 8-7
character conversion functions 8-5
clock 8-31
compound 8-17
copysign 8-7
copysignf 8-8
cos 8-8
cosh 8-8
ctime 8-31
ctime_newton 8-31
difftime 8-31
div 8-21
erf 8-8
erfc 8-8
exp 8-9
exp2 8-9
expm1 8-9
fabs 8-9
fdim 8-9
financial functions 8-17
floating-point math functions 8-6 to 8-17
floor 8-9
fmax 8-10
fmin 8-10
fmod 8-10
free 8-24
frexp 8-10
gmtime 8-32
heap functions 8-23 to 8-24
hypot 8-11
isfinite 8-11
isnan 8-11
isnormal 8-11
labs 8-21
ldexp 8-11
ldiv 8-22
localtime 8-32
localtime_newton 8-32
log 8-11
log10 8-12
log1p 8-12
log2 8-12
logb 8-12
malloc 8-24
memchr 8-25
memcmp 8-25
memcpy 8-25
memmove 8-25
memory block manipulation functions 8-25 to 8-26
memset 8-26
mktime 8-32
modf 8-12
modff 8-13
nearbyint 8-13
nextafterd 8-13
nextafterf 8-13
pow 8-13
qsort 8-22
rand 8-22
randomx 8-14
realloc 8-24
relation 8-14
remainder 8-14
remquo 8-14
rint 8-15
rinttol 8-15
round 8-15
roundtol 8-15
scalb 8-15
signbit 8-15
sin 8-16
sinh 8-16
sprintf 8-18
sqrt 8-16
srand 8-22
sscanf 8-19
standard C functions 8-20 to 8-23
standard input and output functions 8-18 to 8-20
strcat 8-26
strchr 8-27
strcmp 8-27
strcoll 8-27
strcpy 8-27
strcspn 8-27
strftime 8-33
string manipulation functions 8-26 to 8-30
strlen 8-28
strncat 8-28
strncmp 8-28
strncpy 8-28
strpbrk 8-29
strrchr 8-29
strspn 8-29
strstr 8-29
strtod 8-23
strtok 8-29
strtol 8-23
strtoul 8-23
strxfrm 8-30
summary of 8-34 to 8-39
tan 8-16
tanh 8-16
time 8-33
time functions 8-30 to 8-33
tolower 8-5
toupper 8-5
trunc 8-17
va_arg 8-18
va_end 8-18
va_start 8-18

variable argument list functions 8-17 to 8-18
vsprintf 8-19
C library types
clock_t 8-4
div_t 8-2
double_t 8-3
ldiv_t 8-3
math types 8-3
relop 8-3
size_t 8-2
standard types 8-2 to 8-3
time_t 8-4
time types 8-4 to 8-5
tm 8-4
wchar_t 8-2
clock 8-31
clock time type 8-4
Clone 3-9
code restrictions 1-2, 1-3 to 1-6
CoerceToDouble 3-9
CoerceToInt 3-10
compound 8-17
constants
FALSEREF 2-1
HUGE_VAL 8-2
kASCIIEncoding 7-1
kEndOfCharString 7-2, 7-9
kEndOfUnicodeString 7-2, 7-9
kMacKanjiEncoding 7-1
kMacRomanEncoding 7-1
kNoTranslationChar 7-2, 7-9
kShiftJISEncoding 7-1
kWizardEncoding 7-1
NILREF 2-1
NULL 8-2
RAND_MAX 8-2
TRUEREF 2-1
ConvertFromUnicode 7-2
ConvertToUnicode 7-3
ConvertUnicodeChar 7-3
ConvertUnicodeCharacters 7-4
copysign 8-7
copysignf 8-8
cos 8-8
cosh 8-8
CountFreeBlocks 4-2
ctime 8-31
ctime_newton 8-31
CurrentException 5-6

D

data access functions and macros
END_WITH_LOCKED_BINARY 1-10
warnings about use 1-11
WITH_LOCKED_BINARY 1-10
DebugCStr 2-6
Debugger 2-6
DebugStr 2-6
DeepClone 3-10
DefineException 5-7
DeleteTObjectIterator 3-5
difftime 8-31
DisposPtr 4-2
div 8-21
division result type 8-2
Done 3-4
double precision value type 8-3

E

END_FOREACH 3-3
end_try 5-10
end_unwind 5-10
END_WITH_LOCKED_BINARY 1-10
EnsureInternal 3-10
EQ 2-5
EqualBytes 4-2
erf 8-8
erfc 8-8
exAbort exception 5-5
exAlignment exception 5-5
exBusError exception 5-5
exception blocks 5-4 to 5-5
exception data 5-3
exception destructor type 5-6
exception functions and macros 5-6 to 5-12
cleanup 5-9
CurrentException 5-6
DefineException 5-7
end_try 5-10
end_unwind 5-10
newton_catch 5-10
newton_catch_all 5-10
newton_try 5-11
on_unwind 5-11
rethrow 5-7
Subexception 5-8
summary of 5-13
Throw 5-8
ThrowMsg 5-8
unwind_failed 5-12
unwind_protect 5-12
exceptions
about 5-1 to 5-6
blocks 5-4 to 5-5

catch blocks 5-4
class of 5-6
data 5-3
defining 5-1 to 5-3
destructor 5-6
exAbort 5-5
exAlignment 5-5
exBusError 5-5
exDivideByZero 5-5
exIllegalInstr 5-5
exMsgException 5-5
exOutOfStack 5-5
exPermissionViolation 5-5
exRootException 5-5
exSkia 5-5
exWriteProtected 5-5
functions and macros 5-6 to 5-12
Newton system 5-5
types 5-6
volatile values in 5-5
exception structure type 5-6
exception types 5-6
exDivideByZero exception 5-5
exIllegalInstr exception 5-5
exMsgException exception 5-5
exOutOfStack exception 5-5
exp 8-9
exp2 8-9
exPermissionViolation exception 5-5
expm1 8-9
exRootException exception 5-5
exSkia exception 5-5
exWriteProtected exception 5-5

F

fabs 8-9
FALSEREF 2-1
fdim 8-9
FillBytes 4-3
FillLongs 4-3
floor 8-9
fmax 8-10
fmin 8-10
fmod 8-10
FOREACH 3-2
FOREACH_WITH_TAG 3-3
FrameHasPath 3-10
FrameHasSlot 3-10
frames 1-7
free 8-24
frexp 8-10
functions and macros
_ANSI_RAND 8-20
_ANSI_srand 8-20
abs 8-20
acos 8-6
acosh 8-6
AddArraySlot 3-5
AllocateArray 3-6
AllocateBinary 3-6
AllocateFrame 3-6
annuity 8-17
ArrayMunger 3-6
ArrayPosition 3-7
ArrayRemove 3-7
ArrayRemoveCount 3-8
ASCIIString 3-8
asctime 8-30
asctime_newton 8-30
asin 8-6
asinh 8-6
atan 8-7
atan2 8-7
atanh 8-7
atof 8-20
atoi 8-20
atol 8-21
BinaryMunger 3-8
BlockMove 4-1
bsearch 8-21
calloc 8-23
ceil 8-7
ClassOf 3-9
cleanup 5-9
clock 8-31
Clone 3-9
CoerceToDouble 3-9
CoerceToInt 3-10
compound 8-17
ConvertFromUnicode 7-2
ConvertToUnicode 7-3
ConvertUnicodeChar 7-3
ConvertUnicodeCharacters 7-4
copysign 8-7
copysignf 8-8
cos 8-8
cosh 8-8
CountFreeBlocks 4-2
ctime 8-31
ctime_newton 8-31
CurrentException 5-6
DebugCStr 2-6
Debugger 2-6
DebugStr 2-6
DeepClone 3-10
DefineException 5-7
DeleteTObjectIterator 3-5

difftime 8-31
DisposPtr 4-2
div 8-21
Done 3-4
END_FOREACH 3-3
end_try 5-10
end_unwind 5-10
END_WITH_LOCKED_BINARY 1-10
EnsureInternal 3-10
EQ 2-5
EqualBytes 4-2
erf 8-8
erfc 8-8
exp 8-9
exp2 8-9
expm1 8-9
fabs 8-9
fdim 8-9
FillBytes 4-3
FillLongs 4-3
floor 8-9
fmax 8-10
fmin 8-10
fmod 8-10
FOREACH 3-2
FOREACH_WITH_TAG 3-3
FrameHasPath 3-10
FrameHasSlot 3-10
free 8-24
frexp 8-10
GC 3-11
GetArraySlot 3-11
GetFramePath 3-11
GetFrameSlot 3-12
GetPtrName 4-4
GetPtrSize 4-4
GetVariable 6-14
gmtime 8-32
HasChars 7-4
HasDigits 7-5
HasSpaces 7-5
hypot 8-11
IsArray 3-12
IsBinary 3-12
IsChar 2-4
ISFALSE 2-5
isfinite 8-11
IsFrame 3-12
IsFunction 3-12
IsInstance 3-13
IsInt 2-4
IsMagicPtr 1-11, 2-4
isnan 8-11
ISNIL 2-5
isnormal 8-11
IsNumber 3-13
IsPtr 2-4
IsPunctSymbol 7-5
IsReadOnly 3-13
IsReal 3-13
IsRealPtr 1-11, 2-4
IsString 3-13
IsSubclass 3-13
IsSymbol 3-14
ISTRUE 2-5
labs 8-21
LargestFreeInHeap 4-4
ldexp 8-11
ldiv 8-22
Length 3-14
localtime 8-32
localtime_newton 8-32
log 8-11
log10 8-12
log1p 8-12
log2 8-12
logb 8-12
MakeBoolean 2-2
MakeChar 2-2
MakeInt 2-2
MakeReal 2-2
MakeString 2-2
MakeSymbol 2-3
malloc 8-24
MaxHeapSize 4-4
memchr 8-25
memcmp 8-25
memcpy 8-25
MemError 4-4
memmove 8-25
memset 8-26
mktime 8-32
modf 8-12
modff 8-13
nearbyint 8-13
NewNamedPtr 4-5
NewPtr 4-5
NewPtrClear 4-5
NewTObjectIterator 3-5
newton_catch 5-10
newton_catch_all 5-10
newton_try 5-11
Next 3-4
nextafterd 8-13
nextafterf 8-13
NOTNIL 2-5
NSCall 6-2
NSCallGlobalFn 6-4
NSCallGlobalFnWithArgArray 6-5
NSCallWithArgArray 6-3

NSSend 6-6
NSSendIfDefined 6-8
NSSendIfDefinedWithArgArray 6-10
NSSendProto 6-10
NSSendProtoIfDefined 6-12
NSSendProtoIfDefinedWithArgArray 6-14
NSSendProtoWithArgArray 6-12
NSSendWithArgArray 6-7
on_unwind 5-11
pow 8-13
qsort 8-22
rand 8-22
randomx 8-14
realloc 8-24
ReallocPtr 4-6
RefToInt 2-3
RefToUniChar 2-3
relation 8-14
remainder 8-14
RemoveSlot 3-14
remquo 8-14
ReplaceObject 3-14
Reset 3-4
rethrow 5-7
rint 8-15
rinttol 8-15
round 8-15
roundtol 8-15
scalb 8-15
SetArraySlot 3-14
SetClass 3-15
SetFramePath 3-15
SetFrameSlot 3-16
SetLength 3-16
SetPtrName 4-7
SetVariable 6-15
signbit 8-15
sin 8-16
sinh 8-16
SortArray 3-16
sprintf 8-18
sqrt 8-16
srand 8-22
sscanf 8-19
Statistics 3-17
StrBeginsWith 3-17
StrCapitalize 3-17
StrCapitalizeWords 3-17
strcat 8-26
strchr 8-27
strcmp 8-27
strcoll 8-27
strcpy 8-27
strcspn 8-27
StrDowncase 3-17
StrEndsWith 3-18
strftime 8-33
StripPunctSymbols 7-5
strlen 8-28
StrMunger 3-18
strncat 8-28
strncmp 8-28
strncpy 8-28
strpbrk 8-29
StrPosition 3-18
strrchr 8-29
StrReplace 3-19
strspn 8-29
strstr 8-29
strtod 8-23
strtok 8-29
strtol 8-23
strtoul 8-23
StrUpcase 3-19
strxfrm 8-30
Subexception 5-8
Substring 3-19
SYM 2-3
SymbolCompareLex 3-20
syncmp 3-20
SystemRAMSize 4-7
Tag 3-4
tan 8-16
tanh 8-16
Throw 5-8
ThrowBadTypeWithFrameData 3-20
ThrowMsg 5-8
ThrowRefException 3-21
time 8-33
tolower 8-5
TotalClone 3-21
TotalFreeInHeap 4-7
TotalUsedInHeap 4-7
toupper 8-5
TrimString 3-21
trunc 8-17
Umemset 7-6
unwind_failed 5-12
unwind_protect 5-12
Ustrcat 7-7
Ustrchr 7-7
Ustrcmp 7-7
Ustrcpy 7-7
Ustrlen 7-8
Ustrncat 7-8
Ustrncpy 7-8
va_arg 8-18
va_end 8-18
va_start 8-18
Value 3-5

vsprintf 8-19
WITH_LOCKED_BINARY 1-10
XORBytes 4-8
ZeroBytes 4-8
functions and macros for using C++ with NewtonScript
DebugCStr 2-6
Debugger 2-6
DebugStr 2-6
EQ 2-5
IsChar 2-4
ISFALSE 2-5
IsInt 2-4
IsMagicPtr 2-4
ISNIL 2-5
IsPtr 2-4
IsRealPtr 2-4
ISTRUE 2-5
MakeBoolean 2-2
MakeChar 2-2
MakeInt 2-2
MakeReal 2-2
MakeString 2-2
MakeSymbol 2-3
NOTNIL 2-5
RefToInt 2-3
RefToUniChar 2-3
summary of 2-7 to 2-8
SYM 2-3

G

GC 3-11
GetArraySlot 3-11
GetFramePath 3-11
GetFrameSlot 3-12
GetPtrName 4-4
GetPtrSize 4-4
GetVariable 6-14
global data 1-5
gmtime 8-32

H

HasChars 7-4
HasDigits 7-5
HasSpaces 7-5
HUGE_VAL 8-2
hypot 8-11

I

immediate objects 1-7
IsArray 3-12
IsBinary 3-12
IsChar 2-4
ISFALSE 2-5
isfinite 8-11
IsFrame 3-12
IsFunction 3-12
IsInstance 3-13
IsInt 2-4
IsMagicPtr 1-11, 2-4
isnan 8-11
ISNIL 2-5
isnormal 8-11
IsNumber 3-13
IsPtr 2-4
IsPunctSymbol 7-5
IsReadOnly 3-13
IsReal 3-13
IsRealPtr 1-11, 2-4
IsString 3-13
IsSubclass 3-13
IsSymbol 3-14
ISTRUE 2-5

L

labs 8-21
LargestFreeInHeap 4-4
ldexp 8-11
ldiv 8-22
Length 3-14
localtime 8-32
localtime_newton 8-32
log 8-11
log10 8-12
log1p 8-12
log2 8-12
logb 8-12
long division result type 8-3

M

magic pointers 1-11
MakeBoolean 2-2
MakeChar 2-2
MakeInt 2-2
MakeReal 2-2
MakeString 2-2

`MakeSymbol` 2-3
`malloc` 8-24
`MaxHeapSize` 4-4
`memchr` 8-25
`memcmp` 8-25
`memcpy` 8-25
`MemError` 4-4
`memmove` 8-25
memory allocation and deallocation 1-4
memory management functions and macros 4-1 to 4-8
  `BlockMove` 4-1
  `CountFreeBlocks` 4-2
  `DisposPtr` 4-2
  `EqualBytes` 4-2
  `FillBytes` 4-3
  `FillLongs` 4-3
  `GetPtrName` 4-4
  `GetPtrSize` 4-4
  `LargestFreeInHeap` 4-4
  `MaxHeapSize` 4-4
  `MemError` 4-4
  `NewNamedPtr` 4-5
  `NewPtr` 4-5
  `NewPtrClear` 4-5
  `ReallocPtr` 4-6
  `SetPtrName` 4-7
  summary of 4-9
  `SystemRAMSize` 4-7
  `TotalFreeInHeap` 4-7
  `TotalUsedInHeap` 4-7
  `XORBytes` 4-8
  `ZeroBytes` 4-8
`memset` 8-26
`mktime` 8-32
`modf` 8-12
`modff` 8-13

N

name-mangling 1-4
`nearbyint` 8-13
`NewNamedPtr` 4-5
`NewPtr` 4-5
`NewPtrClear` 4-5
`NewTObjectIterator` 3-5
`newton_catch` 5-10
`newton_catch_all` 5-10
`newton_try` 5-11
Newton object system
  about 1-6 to ??
  array objects 1-7
  binary objects 1-7, 1-10 to 1-11, 1-18
  frames 1-7
  immediate objects 1-7
  object classes 3-1 to 3-5
  object system functions 3-5 to 3-21
  object types 1-6 to 1-7
  path expressions 1-12
  primitive object classes 1-7
  reference objects 1-7
  reference types 1-7 to 1-8
    `Ref` 1-7
    `RefStruct` 1-8
    `RefVar` 1-8
  symbols 1-6 to 1-7, 1-12
NewtonScript
  accessing slot values 6-14 to 6-15
  calling C++ functions from 6-15
  calling from C++ 1-2, 6-1 to 6-14
  magic pointers 1-11
  object types 1-6 to 1-7
  symbols 1-6 to 1-7, 1-12
NewtonScript interpreter functions and macros 6-1 to 6-16
  `GetVariable` 6-14
  `NSCall` 6-2
  `NSCallGlobalFn` 6-4
  `NSCallGlobalFnWithArgArray` 6-5
  `NSCallWithArgArray` 6-3
  `NSSend` 6-6
  `NSSendIfDefined` 6-8
  `NSSendIfDefinedWithArgArray` 6-10
  `NSSendProto` 6-10
  `NSSendProtoIfDefined` 6-12
  `NSSendProtoIfDefinedWithArgArray` 6-14
  `NSSendProtoWithArgArray` 6-12
  `NSSendWithArgArray` 6-7
  `SetVariable` 6-15
  summary of 6-17 to 6-19
Newton system exceptions 5-5
`Next` 3-4
`nextafterd` 8-13
`nextafterf` 8-13
`NILREF` 2-1
`NOTNIL` 2-5
`NSCall` 6-2
`NSCallGlobalFn` 6-4
`NSCallGlobalFnWithArgArray` 6-5
`NSCallWithArgArray` 6-3
`NSSend` 6-6
`NSSendIfDefined` 6-8
`NSSendIfDefinedWithArgArray` 6-10
`NSSendProto` 6-10
`NSSendProtoIfDefined` 6-12
`NSSendProtoIfDefinedWithArgArray` 6-14
`NSSendProtoWithArgArray` 6-12
`NSSendWithArgArray` 6-7
`NULL` 8-2

O

object iterator class 3-4 to 3-5
object iterator class functions and macros
  DeleteTObjectIterator 3-5
  Done 3-4
  END_FOREACH 3-3
  FOREACH 3-2
  FOREACH_WITH_TAG 3-3
  NewTObjectIterator 3-5
  Next 3-4
  Reset 3-4
  Tag 3-4
  Value 3-5
object references 1-7 to 1-8
object system functions and macros 3-5 to 3-21
  AddArraySlot 3-5
  AllocateArray 3-6
  AllocateBinary 3-6
  AllocateFrame 3-6
  ArrayMunger 3-6
  ArrayPosition 3-7
  ArrayRemove 3-7
  ArrayRemoveCount 3-8
  ASCIIString 3-8
  BinaryMunger 3-8
  ClassOf 3-9
  Clone 3-9
  CoerceToDouble 3-9
  CoerceToInt 3-10
  DeepClone 3-10
  EnsureInternal 3-10
  FrameHasPath 3-10
  FrameHasSlot 3-10
  GC 3-11
  GetArraySlot 3-11
  GetFramePath 3-11
  GetFrameSlot 3-12
  IsArray 3-12
  IsBinary 3-12
  IsFrame 3-12
  IsFunction 3-12
  IsInstance 3-13
  IsNumber 3-13
  IsReadOnly 3-13
  IsReal 3-13
  IsString 3-13
  IsSubclass 3-13
  IsSymbol 3-14
  Length 3-14
  RemoveSlot 3-14
  ReplaceObject 3-14
  SetArraySlot 3-14
  SetClass 3-15
  SetFramePath 3-15
  SetFrameSlot 3-16
  SetLength 3-16
  SortArray 3-16
  Statistics 3-17
  StrBeginsWith 3-17
  StrCapitalize 3-17
  StrCapitalizeWords 3-17
  StrDowncase 3-17
  StrEndsWith 3-18
  StrMunger 3-18
  StrPosition 3-18
  StrReplace 3-19
  StrUpcase 3-19
  Substring 3-19
  summary of 3-22 to 3-24
  SymbolCompareLex 3-20
  syncmp 3-20
  ThrowBadTypeWithFrameData 3-20
  ThrowRefException 3-21
  TotalClone 3-21
  TrimString 3-21
object types 1-6 to 1-7
on_unwind 5-11

P

path expressions 1-12
persistent storage 1-18
pow 8-13
primitive object classes 1-7

Q

qsort 8-22

R

rand 8-22
RAND_MAX 8-2
randomx 8-14
realloc 8-24
ReallocPtr 4-6
Ref 1-7
RefArg type 1-8
reference objects 1-7
reference types 1-7 to 1-8
  Ref 1-7
  RefStruct 1-8
  RefVar 1-8

RefStruct 1-8
RefStruct type 1-8
RefToInt 2-3
RefToUniChar 2-3
Ref type 1-8
RefVar 1-8
RefVar type 1-8
relation 8-14
relational operator type 8-3
remainder 8-14
RemoveSlot 3-14
remquo 8-14
ReplaceObject 3-14
Reset 3-4
rethrow 5-7
rint 8-15
rinttol 8-15
round 8-15
roundtol 8-15

S

scalb 8-15
SetArraySlot 3-14
SetClass 3-15
SetFramePath 3-15
SetFrameSlot 3-16
SetLength 3-16
SetPtrName 4-7
SetVariable 6-15
signbit 8-15
sin 8-16
sinh 8-16
size type 8-2
SortArray 3-16
sprintf 8-18
sqrt 8-16
srand 8-22
sscanf 8-19
static variables 1-5
Statistics 3-17
StrBeginsWith 3-17
StrCapitalize 3-17
StrCapitalizeWords 3-17
strcat 8-26
strchr 8-27
strcmp 8-27
strcoll 8-27
strcpy 8-27
strcspn 8-27
StrDowncase 3-17
StrEndsWith 3-18
strftime 8-33
StripPunctSymbols 7-5
strlen 8-28
StrMunger 3-18
strncat 8-28
strncmp 8-28
strncpy 8-28
strpbrk 8-29
StrPosition 3-18
strrchr 8-29
StrReplace 3-19
strspn 8-29
strstr 8-29
strtod 8-23
strtok 8-29
strtol 8-23
strtoul 8-23
StrUpcase 3-19
strxfrm 8-30
Subexception 5-8
Substring 3-19
SYM 2-3
SymbolCompareLex 3-20
symbols 1-6 to 1-7, 1-12
syncmp 3-20
system exceptions
  exAbort 5-5
  exAlignment 5-5
  exBusError 5-5
  exDivideByZero 5-5
  exIllegalInstr 5-5
  exMsgException 5-5
  exOutOfStack 5-5
  exPermissionViolation 5-5
  exRootException 5-5
  exSkia 5-5
  exWriteProtected 5-5
SystemRAMSize 4-7

T

Tag 3-4
tan 8-16
tanh 8-16
Throw 5-8
ThrowBadTypeWithFrameData 3-20
ThrowMsg 5-8
ThrowRefException 3-21
time 8-33
TObjectIterator class 3-4
tolower 8-5
TotalClone 3-21
TotalFreeInHeap 4-7
TotalUsedInHeap 4-7

toupper 8-5
TrimString 3-21
TRUEREF 2-1
trunc 8-17
type-checking functions and macros
  IsMagicPtr 1-11
  IsRealPtr 1-11
types
  clock_t 8-4
  div_t 8-2
  double_t 8-3
  ldiv_t 8-3
  relop 8-3
  size_t 8-2
  time_t 8-4
  tm 8-4
  UniChar 7-1
  wchar_t 8-2

U

Umemset 7-6
UniChar type 7-1
Unicode constants and data type 7-1 to 7-2
Unicode encoding types 7-1
Unicode functions and macros 7-2 to 7-8
  ConvertFromUnicode 7-2
  ConvertToUnicode 7-3
  ConvertUnicodeChar 7-3
  ConvertUnicodeCharacters 7-4
  HasChars 7-4
  HasDigits 7-5
  HasSpaces 7-5
  IsPunctSymbol 7-5
  StripPunctSymbols 7-5
  summary of 7-9 to 7-10
  Umemset 7-6
  Ustrcat 7-7
  Ustrchr 7-7
  Ustrcmp 7-7
  Ustrcpy 7-7
  Ustrlen 7-8
  Ustrncat 7-8
  Ustrncpy 7-8
unwind_failed 5-12
unwind_protect 5-12
using C++ with NewtonScript
  constants 2-1
  debugging functions and macros 2-6 to ??
  functions and macros 2-1 to ??
  overview 1-1 to 1-6
  type-checking functions and macros 2-4
  type-conversion functions and macros 2-1 to 2-3
  value-checking functions and macros 2-5
Ustrcat 7-7
Ustrchr 7-7
Ustrcmp 7-7
Ustrcpy 7-7
Ustrlen 7-8
Ustrncat 7-8
Ustrncpy 7-8

V

va_arg 8-18
va_end 8-18
va_start 8-18
Value 3-5
vsprintf 8-19

W

wide char type 8-2
WITH_LOCKED_BINARY 1-10
wrapper functions 1-16

X

XORBytes 4-8

Z

ZeroBytes 4-8