



N e w t o n D e v e l o p e r T o o l s

Newton Formats

Version 1.1

Apple Computer, Inc. Confidential; Use subject to Newton Formats License Agreement.



 Apple Computer, Inc.

© 1995, 1997 Apple

Computer, Inc.

All rights reserved.

This document contains confidential and proprietary information of Apple

Computer, Inc. and its use is licensed only pursuant to the express terms of the Newton Formats License Agreement.

No part of this document or software described in it may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without the prior written permission of Apple Computer, Inc. The same proprietary and copyright notices must be affixed to any permitted copies as were affixed to the original.

Under the law, copying includes translating into another language or format. Except as specifically

set forth in the Newton Formats License Agreement, No licenses,

express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document.

This document is intended to assist application developers to develop applications only for licensed Newton platforms.

Printed in the United States of America.

The Apple logo is a registered trademark of Apple Computer, Inc. Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, Macintosh, and Newton are trademarks of Apple Computer, Inc., registered in the United States and other countries.

The light bulb logo, MessagePad, NewtonScript, and Newton Toolkit are trademarks of Apple Computer, Inc.

Simultaneously published in the United States and Canada.

Apple licenses the use of this document pursuant to the Newton Formats License

Agreement on an "AS IS" basis. APPLE MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE INFORMATION CONTAINED IN THIS DOCUMENT OR ITS USE, QUALITY OR ACCURACY. APPLE MAKES NO WARRANTY OR REPRESENTATION THAT IT WILL NOT MODIFY THE INFORMATION CONTAINED IN THIS DOCUMENT OVER TIME.

In no event shall Apple be liable for special, indirect, incidental or consequential damages arising from the use of information contained in this document, or the use, sale, licensing or distribution of Licensee Programs by Licensee or any third party, whether under theory of contract, tort (including negligence), product liability or otherwise.

Contents

Preface

About This Document v

Chapter 1 Newton Package Specification 1-1

Introduction	1-1
Scope of this document	1-1
Data structure definitions	1-2
The Package-Loading Process	1-2
Loading	1-3
Activating	1-3
Deactivating	1-3
Deleting	1-4
Package Container Format	1-4
Package Directory	1-5
Fixed header	1-5
Part entries	1-6
Variable-length data area	1-8
Relocation information	1-8
Fixed header	1-9
Relocation sets	1-9
Part Data	1-10
NewtonScript Object Parts	1-10
Basic object format	1-10
Refs	1-11
Objects	1-11
Object header	1-12
Object formats	1-12
Frame maps	1-13
Other object formats	1-15
Part layout	1-16
Magic pointers	1-16

Chapter 2 NewtonScript Bytecode Interpreter Specification 2-1

Introduction	2-1
Virtual machine	2-2
Function objects	2-2

Locals frame	2-2
Function frame	2-4
Virtual machine specification	2-5
Exceptions thrown by the interpreter	2-5
Global variables and functions	2-6
Function call and message send	2-6
Function call	2-6
Message send	2-7
Inheritance	2-8
Proto Lookup	2-8
Lexical lookup	2-8
Full lookup	2-8
Assignment	2-9
Lexical assignment	2-9
Exception handling	2-10
Structure of instructions	2-11
Instruction definitions	2-11
Simple instructions	2-12
Parameterized instructions	2-14
Primitive functions	2-20
Arithmetic operations	2-20
Array/string functions	2-21
Comparison functions	2-23
Logical operations	2-24
Miscellaneous	2-24
Support objects	2-25
Iterators	2-25
Operation	2-25
Access	2-26
Reference	2-26

Chapter 3	Newton Load Package Protocol	3-1
-----------	-------------------------------------	-----

Protocol Overview	3-1
Loading a Package	3-2
Newton -> Desktop	3-2
Desktop-> Newton	3-3
Desktop-> Newton or Newton<-Desktop	3-4

Chapter 4	Newton Streamed Object Format	4-1
-----------	--------------------------------------	-----

Introduction	4-1
Streamed Object Format	4-2
Encoding	4-2

Immediate Objects	4-4
Binary Object Data	4-5
Special Case Types	4-5
Example of Newton Streamed Object Format	4-5

About This Document

This document describes various Newton formats and protocols. Using the information published in this document, it is possible to develop an environment in which Newton-compatible software can be written and then downloaded for execution on a Newton Device.

▲ WARNING

The formats and protocols described in this document are compatible with all existing Newton ROMs. However, these specifications may change without notice, and Apple may render them incompatible in future Newton systems.

The document contains four chapters

- Chapter 1, “Newton Package Specification,” specifies the format of a package file as a container (that is, a collection of parts). It also specifies the format of a frames part, including the low-level format of objects, and some of the object-level format of a form part (which is a kind of frames part).
- Chapter 2, “NewtonScript Bytecode Interpreter Specification,” specifies the format of NewtonScript bytecode function objects. It also specifies the behavior of the NewtonScript virtual machine (which is necessary to specify the function objects).
- Chapter 3, “Newton Load Package Protocol,” specifies the protocol used over a byte stream to download packages to a Newton using the Connection icon.
- Chapter 4, “Newton Streamed Object Format,” specifies the format in which Newton objects are sent and received over a byte stream by the communications subsystem in Newton (for example, `endpoint:OutputFrame`).

P R E F A C E

Newton Package Specification

This document specifies the format of Newton packages, the units of application installation and removal.

▲ **WARNING**

The format described here is compatible with all existing versions of the Newton OS. However, this specification may change without notice, and Apple may render it incompatible in future Newton systems.

Introduction

Newton software applications are delivered in the form of *packages*. A package consists of one or more *parts*—units of functionality such as a form, font, or device driver—all of which need to be installed and removed together.

Packages are intended to eliminate the scattering of software components that takes place in many operating systems, where an installer program may place parts of an application in many different locations in the system, but there is no easy way to find and remove the parts once the application is no longer desired. Although it consists of many components that affect different parts of the Newton system, a package is installed and removed as a single unit.

Scope of this document

The format described in this document is used when packages are stored or transmitted. Packages are commonly installed by downloading them to devices running the Newton OS. This document specifies the data that the Newton OS expects to receive when it is downloading a package.

Packages are often downloaded by “package installer” applications running on desktop computers. For this purpose, packages are distributed in the form of *package files*. A

Newton Package Specification

package file is simply meant to be downloaded as-is, so this document also specifies the format of a package file.

Starting in version 2.0, the Newton OS can install a package from a binary object. Again, the binary object contains package data as specified in this document.

This document first describes the *package container* format, which is the outer structure of a package—a container for various parts. This format is the same in every package.

Within the package container format are regions for *part data*, one region for each part in the package. Each part is of a certain *part type*, and each part type has its own format for part data. This document describes the basic formats of the part types. The details of particular part formats are contained in their own documents.

Data structure definitions

Data structures defined in this document are presented as C struct definitions using the following types:

ULong	Unsigned 32-bit integer. This document sometimes writes a ULong as a four-character ASCII string in single quotes. The first character is the most-significant byte of the integer, and so forth; for example, 'auto' corresponds to the integer 0x6175746F.
UShort	Unsigned 16-bit integer.
Byte	Unsigned byte (or ASCII character).
UniChar	Two-byte Unicode character.
Date	Unsigned 32-bit integer representing a date and time as the number of seconds since midnight, January 4, 1904.
InfoRef	An unsigned 16-bit offset followed by an unsigned 16-bit length. InfoRefs are used to refer to variable-length data items in the variable-length data area of the package directory. The offset is from the beginning of the data area; the length is the number of bytes in the data item.

The format is big-endian: all integers are stored most-significant-byte first. Data structure elements are stored in the order in which they appear, with no implicit padding for alignment.

The Package-Loading Process

In order to understand the purpose of many of the fields of the package format, it is necessary to understand the various processes packages go through in the Newton operating system.

The life cycle of a package has four stages:

1. The package is *loaded* into a Newton device's object store.

Newton Package Specification

2. The package is *activated*, and each part is *installed*.
3. The package is *deactivated*, and each part is *uninstalled*.
4. The package is *deleted* from the object store.

Loading

When a package is loaded into the object store, it may be compressed automatically. So-called “compressed” packages are not packages that have been compressed, but packages that *will be* compressed when a Newton device loads them. A flag in the package header signals the package loader to compress the package as it is being stored.

Activating

When a package is activated, it is mapped into virtual memory contiguously, just as it appears in this format. However, the package is not continuously resident in physical memory. The package is divided into pages (currently 1K in size) that are loaded into memory on demand.

The region of virtual memory occupied by the package is allocated each time the package is activated, so the package may be at a different location at each activation. While the package is activated, it does not move.

As part of the activation process, each part in the package is *installed*. The actual installation behavior is determined by the *part kind* and *part type*. For example, a protocol part is registered as a P-class, a form part is installed as an icon in the Extras drawer, and a font part is installed as a newly-available font.

Each part type may define its own mechanism for parameterizing installation behavior. For example, form parts may specify NewtonScript methods that are run to complete the installation. Some part, such as font parts, have no such mechanism.

Parts are normally left in virtual memory, but they may optionally be copied into permanently-resident memory by setting `kAutoCopyFlag`. This is only necessary if the part must remain accessible even if the package’s store becomes unavailable (for example, if the package is stored on a card, and the card is removed), or if the part is used to service page faults. Because this uses extra RAM, it should only be used when absolutely necessary.

Deactivating

Deactivation is similar to activation, but in reverse. Each part is uninstalled according to its part type, which may involve part-specific behavior. The memory occupied by autocopied parts is freed. Finally, the virtual memory occupied by the package is released.

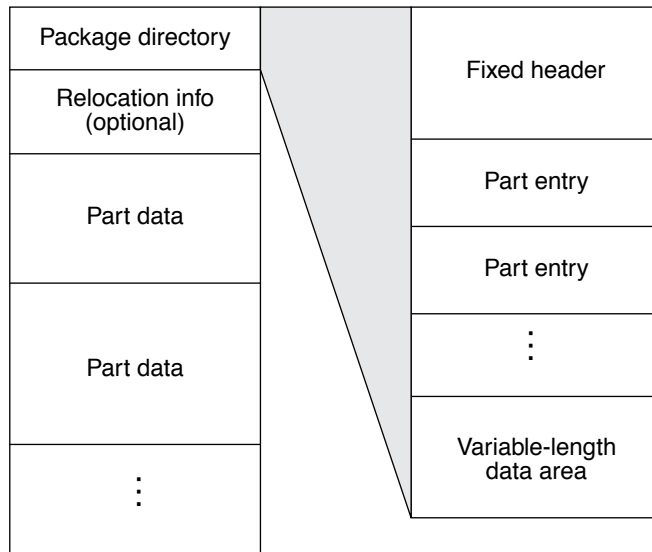
Deleting

After a package is deactivated, it may be deleted from the object store. The behavior of the system when deleting a package does not depend on the contents of the package.

Package Container Format

A package consists of a package directory, optionally followed by a relocation table, followed by the data areas for the individual parts. The package format at this level (that is, ignoring the contents of the part data areas) is called the package container format. Figure 1-1 gives a conceptual view of the layout of a package.

Figure 1-1 Package layout



The package begins with a *package directory*, which contains information about the whole package as well as *part entries* describing each part in the package. There is one part entry per part. The part entries are followed by an area used to store variable-length data (such as strings) for the package directory.

If the package needs to be relocated, an optional area containing relocation information follows the directory. This area is present if the package signature is "package1" and the flag `kRelocationFlag` is set.

Finally, the part data regions complete the package. Each part data region must begin on a four-byte boundary. If a region's length is not divisible by four, pad bytes must be inserted as necessary to move the next region to a four-byte boundary.

2.0
ONLY

The ability to process packages with relocation information is new in OS 2.0.

Package Directory

The format of the package directory is defined as a `PackageDirectory` structure, followed by zero or more `PartEntry` structures (one per part), followed by the variable-length data area.

Fixed header

The package directory begins with a fixed set of fields represented by the `PackageDirectory` structure.

Listing 0-1 `PackageDirectory` structure

```
struct PackageDirectory {
    Byte    signature[8];
    ULong   reserved1;
    ULong   flags;
    ULong   version;
    InfoRef copyright;
    InfoRef name;
    ULong   size;
    Date    creationDate;
    ULong   reserved2;
    ULong   reserved3;
    ULong   directorySize;
    ULong   numParts;
    /* PartEntry parts[numParts]; */
    /* Byte variableLengthData[]; */
};
```

Fields `reserved1`, `reserved2`, and `reserved3` are reserved and must be set to zero.

signature An eight-byte ASCII string specifying the format of the package. The signature "package0" signifies a package without a relocation information area; "package1" signifies a package that may contain one, depending on `kRelocationFlag`.



The "package1" signature is not understood by Newton OS versions before 2.0, so it can be used in packages without any relocation information to prevent older systems from loading the package. This is useful when the package uses other 2.0-only features.

flags The following flags are defined. All other bits are reserved and must be set to zero.

`kAutoRemoveFlag = 0x80000000`

Specifies that parts in the package are to be removed immediately after installation. When an auto-remove package is activated, the

Newton Package Specification

system activates the parts and then deactivates them *without* performing part-specific deactivation behavior (such as a removal script). The only recommended constituent of an auto-remove package is a single part of type 'auto'.

`kCopyProtectFlag = 0x40000000`

Marks the package as copy-protected. This field is a convention recognized by software that copies packages; it is not an absolute lock against copying.

The Newton OS refuses to beam or email a copy-protected package. A copy-protected package can be backed up and synchronized to the desktop, so users can copy the package using selective restore.

`kNoCompressionFlag = 0x10000000`

Specifies that the Newton OS should not compress the package as it is stored. (The default is to compress the package.)

`kRelocationFlag = 0x04000000`

Specifies that the package contains a relocation information area. This flag is valid only in "package1" packages.

`kUseFasterCompressionFlag = 0x02000000`

Specifies that the package should be compressed using a faster, but less space-efficient method. This flag is valid only in "package1" packages, and effective only when `kNoCompressionFlag` is not set.

2.0
ONLY

2.0
ONLY

<code>version</code>	An arbitrary number used to identify the version of the package. The Newton OS interprets higher numbers as newer versions.
<code>copyright</code>	A Unicode string containing a copyright notice. (May be empty.)
<code>name</code>	A Unicode string naming the package. This string is assumed to uniquely identify a package. A registered developer signature is normally used as a suffix to ensure uniqueness.
<code>size</code>	The total size in bytes of the package, including the directory.
<code>creationDate</code>	The time and date the package was created.
<code>directorySize</code>	The size in bytes of the package directory, including the <code>PackageDirectory</code> structure, the part entries, and the data area.
<code>numParts</code>	The number of parts in the package.

Part entries

Following the `PackageDirectory`, each part is represented by a `PartEntry` structure. Parts are often referred to by number (part 0, part 1, and so forth); the number is determined by the order of the `PartEntry` structures. The first `PartEntry` corresponds to part 0.

Listing 0-2 PartEntry structure

```
struct PartEntry {
    ULong    offset;
```

Newton Package Specification

```

    ULong    size;
    ULong    size2;
    ULong    type;
    ULong    reserved1;
    ULong    flags;
    InfoRef  info;
    ULong    reserved2;
};

```

Fields `reserved1` and `reserved2` are reserved and must be set to zero.

<code>offset</code>	The offset in bytes of the part data from the beginning of the part data section. Must be a multiple of four.
<code>size</code>	The size in bytes of the part data.
<code>size2</code>	Must be the same value as <code>size</code> .
<code>type</code>	A code indicating the type of the part. (See individual part kind documentation for codes.)
<code>flags</code>	<p>The following flags are defined. All other bits are reserved and must be set to zero.</p> <pre> kProtocolPart = 0x00000000 kNOSPart = 0x00000001 kRawPart = 0x00000002 </pre> <p>The low-order two bits of this field signify whether the part data consists of a protocol (<code>kProtocolPart</code>), a region of NewtonScript objects (<code>kNOSPart</code>) or raw data (<code>kRawPart</code>). The correct flags are given in the specification of each part type.</p> <pre> kAutoLoadFlag = 0x00000010 </pre> <p>Should only be set for protocol parts. Specifies that the protocol should be registered automatically when the package is activated.</p> <pre> kAutoRemoveFlag = 0x00000020 </pre> <p>Should only be set for protocol parts. Specifies that the protocol should be unregistered automatically when the package is deactivated. Normally set whenever <code>kAutoLoadFlag</code> is set.</p> <pre> kNotifyFlag = 0x00000080 </pre> <p>Specifies that the system handler corresponding to the part's type should be notified. Should be set unless the part type specification says otherwise.</p> <pre> kAutoCopyFlag = 0x00000100 </pre> <p>Specifies that the part should be moved into permanently-resident memory before being activated. Should be used only for parts (normally protocol parts) that cannot tolerate the usual page faulting mechanism.</p>
<code>info</code>	A block of data that is passed to the part type handler when the part is activated. The contents of this data are specified by the part type specification.

Variable-length data area

The part entries are followed by the variable-length data area referred to by the `InfoRef` values. Most data in this area consists of ASCII or Unicode strings. Note that although the length of the string is provided in the `InfoRef`, null terminators are necessary and should be included at the end of the strings.

Relocation information

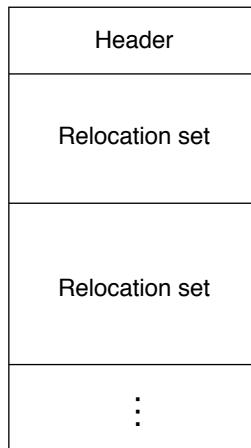
2.0
ONLY

If the package signature is "package1", and the package directory `kRelocationFlag` is set, the package contains a relocation information area after the package directory. This information is used to relocate package-relative addresses to compensate for the virtual address that gets assigned to the package when it is activated. This allows native code in the package to use absolute addresses.

Because the address of the package may change each time it is activated, the package data is left in its original form in the object store. Relocations are applied to each page of the package as it is demand-paged into physical memory. To make this process efficient, the relocation information is split into "chunks" corresponding to package pages.

The format of the relocation information is shown in Figure 1-2.

Figure 1-2 Relocation information area



The relocation information consists of a fixed header followed by a series of relocation sets. Each relocation set contains the data needed to relocate one page of the package: a list of the offsets of the words to be relocated in that page.

The system relocates each 4-byte word by adding to it the difference between the original base address of the package and the virtual address assigned to the package at activation. For example, if the package was built assuming a base address of `0x100000`, and at activation it was assigned the starting address `0x60015000`, then a word would

Newton Package Specification

be relocated by adding $0x60015000 - 0x100000$, or $0x5FF15000$. Only words beginning on four-byte boundaries can be relocated.

Fixed header

The relocation information begins with a fixed set of fields represented by the `RelocationHeader` structure.

Listing 0-3 `RelocationHeader` structure

```
struct RelocationHeader {
    ULong    reserved;
    ULong    relocationSize;
    ULong    pageSize;
    ULong    numEntries;
    ULong    baseAddress;
};
```

<code>reserved</code>	Must be zero.
<code>relocationSize</code>	The total size in bytes of the relocation information area, including the header.
<code>pageSize</code>	The size in bytes of a relocation page. Must be 1024.
<code>numEntries</code>	The number of relocation entries following the header.
<code>baseAddress</code>	The original base address of the package.

Relocation sets

The `RelocationHeader` is followed by a series of `RelocationSet` structures. Each `RelocationSet` must be padded to a multiple of four bytes.

Listing 0-4 `RelocationSet` structure

```
struct RelocationSet {
    UShort    pageNumber;
    UShort    offsetCount;
    /* Byte offsets[]; */
};
```

<code>pageNumber</code>	The zero-based index of the page in the package to which this set applies.
<code>offsetCount</code>	The number of offset bytes in <code>offsets</code> .
<code>offsets</code>	The offsets of the words to be relocated. Each byte is the zero-based index of a word in the page to be relocated. For example, the offset <code>0x10</code> means to relocate a word <code>0x40</code> bytes into the page.

Part Data

After the package directory, or the relocation information if present, the remainder of the package contains the data for the individual parts. Each part data region must begin at a four-byte boundary, but is otherwise not constrained in its position. It is customary for the part data chunks to be adjacent in this area, and to appear in the same order as in the part directory, but this is not strictly required by the format.

The format of a part data region depends on the part's type and kind. Specific part formats are described elsewhere, but all parts containing NewtonScript object parts have a common underlying format, which is described in the following section.

NewtonScript Object Parts

A part of the `kNOSPart` kind (see "Part entries" on page 1-6) contains a group of NewtonScript objects. The Newton OS has special support for relocating objects in `kNOSPart` parts, so this kind is always used for part types that contain NewtonScript objects.

The objects in the part are rooted in a single frame, called the *part frame*. All other objects in the part must be reachable through some path from the part frame. (Technically, parts can contain unreachable objects, but such objects are useless.) A part type definition specifies the contents of the part frame.

Basic object format

This is a very brief introduction to the NewtonScript object system. For more information, see *The NewtonScript Programming Language*.

The most basic element of the object format is a *Ref*, which is a 32-bit value that can either contain a small piece of data like an integer, or be a pointer to an object.

There are three types of NewtonScript objects. *Binary objects* are used to store chunks of uninterpreted (non-pointer) data, such as strings, bitmaps, and sounds. *Arrays* are zero-based arrays of Refs. *Frames* are collections of name-value pairs, where the names are pointer Refs to symbol objects (see page 1-15) and values are Refs. Binary objects and arrays also contain a Ref used to refer to the "class" of the object*, which is usually a symbol object.

All objects are based on the three basic types. The formats of certain kinds of objects, such as symbols and strings, are derived from these three. For the details of some derived object formats, see "Other object formats" on page 1-15.

* This usage of the word "class" has nothing to do with object-oriented programming. The "class" of a NewtonScript object is a symbol giving a semantic type to the object, so that objects that are otherwise just aggregations of data can contain a description of their contents. For example, a picture is stored as a binary object of class "PICT". By itself, the data could only be interpreted in context, but with the class added, the object is self-describing.

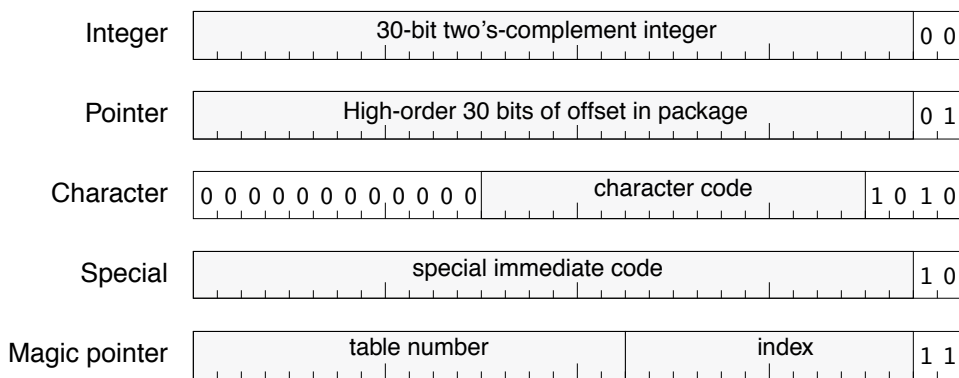
Refs

A Ref is a 32-bit value that either contains a piece of data itself (an *immediate* Ref) or points to an object (a *pointer* Ref). The low-order bits of the Ref are used as *tags* to signify the proper interpretation of the Ref.

Three of the tags are encoded in just the two low-order bits, to allow 30 bits to be used for data. The rest of the types, which have smaller data requirements, use more bits for the tag.

The tag values and their associated data types are shown in Figure 1-3.

Figure 1-3 Format of Refs



An integer Ref is a 30-bit two's complement integer shifted left two bits, with a tag of 00. For example, the integer 5 would be represented by a Ref of 0x14, and the integer -1 would be represented by a Ref of 0xFFFFF4.

A pointer Ref contains the byte offset in the package (not the part) of the object to which the Ref points. Since all NewtonScript objects in the package are aligned to a 4-byte boundary, the two low-order bits of the offset are always zero. Those bits are used for the tag of 01. Thus, subtracting 1 from a pointer Ref yields the actual offset.

Non-integer immediate Refs have the tag 10. Characters have the four-bit tag 1010, preceded by the 16-bit Unicode character code. "Special" immediate Refs are unique values: the Ref for TRUE is 0x1A, and the Ref for NIL is 0x2. Function objects use special immediates to identify function types; see "Function objects" on page 2-2.

A "magic" pointer is a Ref that points to an object, but does not contain the actual offset of the object. Instead, it contains a symbolic reference to the object, in the form of a table number and an index into the table. Magic pointers provide a means for parts to have references to objects which are not contained within the part, or even the package. See "Magic pointers" on page 1-16 for more information.

Objects

There are three fundamental types of objects:

Newton Package Specification

<i>Binary object</i>	Contains a class Ref and a region of uninterpreted (non-pointer) data.
<i>Array</i>	Contains a class Ref and a sequence of zero-indexed slots, which are Refs.
<i>Frame</i>	Contains a Ref pointing to a frame map object, and a sequence of named slots, which are Refs.

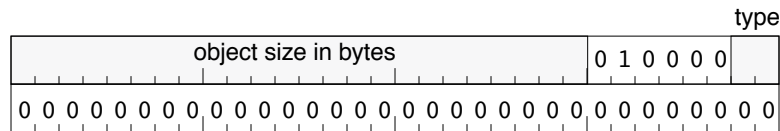
All objects are padded to a multiple of four or eight bytes. All the objects in a part must use the same padding alignment, but each part can use either four- or eight-byte alignment. The alignment is signalled by a flag bit in the part (see “Part layout” on page 1-16). The bytes used for padding may have any value.

Binary objects cannot contain any pointers other than the class Ref. The system ignores the data part of a binary object when it performs pointer relocation.

Object header

Every object in the package has an 8-byte header, as shown in Figure 1-4.

Figure 1-4 An object header



The first word contains the size of the object in the upper three bytes, and flag bits in the lower byte. The second word is always zero.

The size in the header is the logical size of the object, not including any padding that may be necessary to reach the four- or eight-byte-aligned size.

The following object flags are defined. The other flag bits must be set to 0x40, as shown above.

`kObjSlotted = 0x01`

If set, object is an array or frame; otherwise, object is a binary object.

`kObjFrame = 0x02`

If set, object is a frame. Cannot be set unless `kObjSlotted` is also set.

Object formats

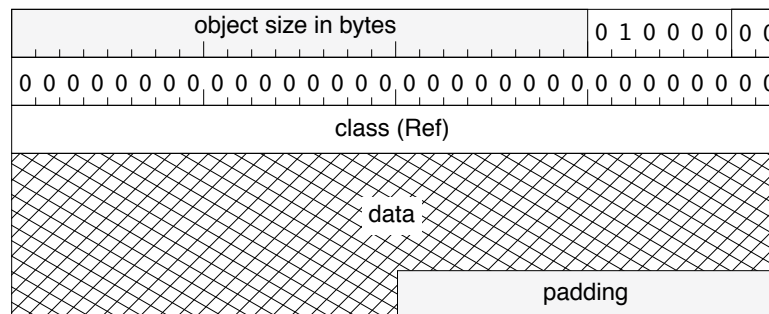
The detailed formats of the three types of object are shown in Figure 1-5. The padding is shown in the binary object case because it is more common for binary objects; arrays and frames are always a multiple of four bytes, so they are only padded in parts with eight-byte alignment.

Newton Package Specification

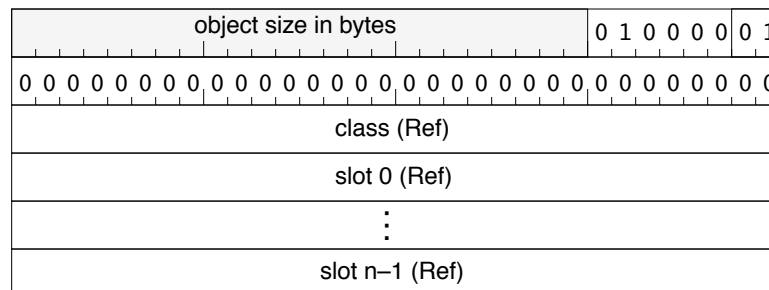
Note that binary objects and arrays have a slot for the class `Ref`, but frames do not. Frames have a slot in the same place that is used for frame maps (see next section).

Figure 1-5 Object formats

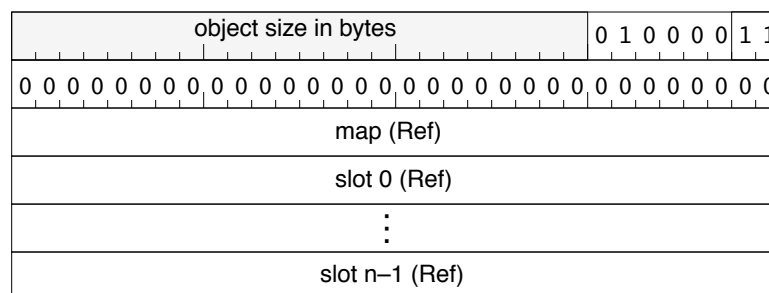
Binary object



Array



Frame



Frame maps

A frame is essentially the same as an array, but with an associated object, called a *frame map*, that associates names (symbols) with slots.

Newton Package Specification

It is possible for each frame to have its own map. To save space, however, two or more frames may share the same map if they have the same slots in the same order. As a further optimization, a frame can use a map to name a subset of its slots. Each map may refer to another map, called its *supermap*, whose slots are included by reference. For example, a frame with six slots could have a map [D, E, F] with supermap [A, B, C], with the same effect as if it had one map [A, B, C, D, E, F]. Of course, a supermap may itself have a supermap.

Supermaps allow frames to share one map for their common slots without losing the ability to have their own unique slots added. When a slot is added to a frame with a shared map, a new map containing the new slot name can be created for the frame, with the original shared map as its supermap.

A map may be sorted, which improves search speed for sufficiently-large maps. The usual threshold for sorting a map is a length of 20. Symbols in a sorted map are arranged in increasing hash value order; symbols with equal hash values are arranged in increasing case-insensitive ASCII lexicographical order. A sorted map has the `kMapSorted` bit set in its class. Note that each individual map object may be sorted or unsorted, so a frame with multiple maps may have maps of both types.

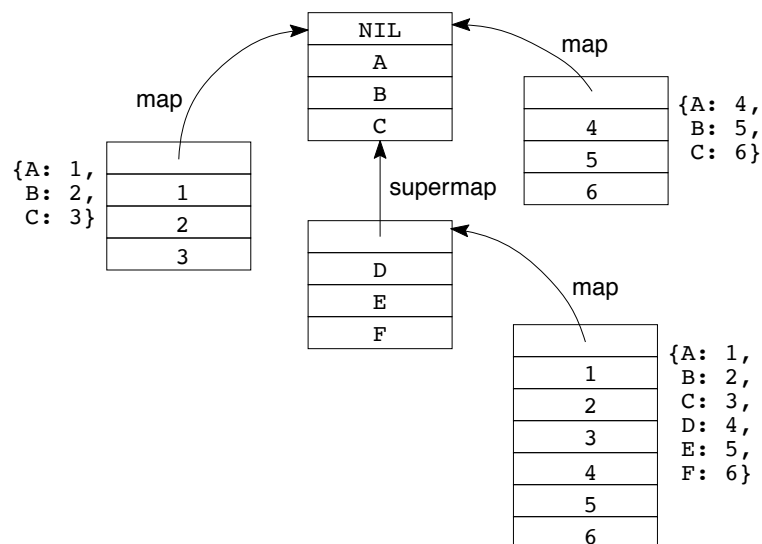
A map is an array object. Slot 0 points to the supermap, or is `NIL` if there is no supermap. The remaining slots point to symbols naming the slots in the frame that correspond to the map. The class of the array is an integer Ref containing bit flags as follows:

`kMapSorted` = 1 The map is sorted.

`kMapProto` = 4 The map, or one of its supermaps, contains the symbol `_proto`.
This bit *must* be set if this condition is true.

The example in Figure 1-6 shows three frames sharing two maps. For simplicity, object headers and pointers to symbols are not shown in the figure.

Figure 1-6 Maps and frames



Other object formats

The formats of all objects in the package are based on the three major object formats above. The formats for certain derived object types used by the system are specified here.

Symbol

Symbol objects are used as classes and as names for frame slots. They are represented as binary objects.

The class of a symbol is the special immediate 0x55552 (`kSymbolClass`). The data for a symbol consists of a 4-byte hash value followed by a null-terminated ASCII string giving the symbol's name. Only character codes 32–127 are allowed in the symbol name.

The hash value for a symbol is calculated as follows:

Listing 0-5 Symbol hash function

```
/* Assumes ASCII character set and 32-bit longs */
unsigned long SymbolHashFunction(char* name)
{
    unsigned long result = 0;
    char c;
    while (*name) {
        c = *name;
        if (c >= 'a' && c <= 'z')
            result = result + c - ('a' - 'A');
        else
            result = result + c;
    }
    return result * 2654435769;
}
```

String

String objects represent Unicode strings. They are represented as binary objects whose class is the symbol `'string'` (or, in OS 2.0, a subclass thereof), and whose data consists of a null-terminated Unicode string.

Real

Reals are represented as binary objects whose class is the symbol `'real'`, and whose data consists of an IEEE standard double-precision floating point number.

Part layout

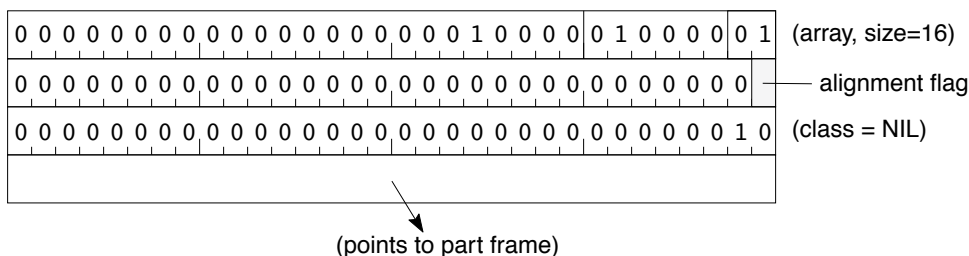
The part consists of objects concatenated together. There must be no space before, after, or between the objects in the part, other than the pad bytes following the last object in the part.

The first object in the part is used to locate the part frame. (See Figure 1-7.) It is required to be an array of class NIL with one slot, which points to the part frame.

2.0
ONLY

In OS 2.0, the low-order bit of the second long of this array—normally set to zero in all objects—is used as an alignment flag. If the bit is set, the objects in the part are padded to four-byte boundaries. Otherwise, the objects are padded to eight-byte boundaries. Only eight-byte-aligned parts can be used on Newton OS versions prior to 2.0.

Figure 1-7 First object's format



Magic pointers

Magic pointers contain a table number and an index. The table number is used to select a particular set of objects, and the index is used to select one object from the set.

Table 0 is used to refer to objects within a Newton ROM. The index numbers for the objects in a particular version of the Newton OS may be found in the *Newton Programmers Guide* covering that version.

Table 1 is undefined, and should not be used in packages.

2.0
ONLY

In OS 2.0, higher-numbered tables are used for the unit import mechanism. Tables 2 and above refer, in order, to units which are imported by the package in which the magic pointer appears. That is, objects from the first imported unit are accessed by magic pointers for table number 2, objects from the next imported unit by magic pointers for table number 3, and so on.

NewtonScript Bytecode Interpreter Specification

This chapter specifies the format of NewtonScript bytecode function objects, and the virtual machine that interprets them. Only those parts of the format that are necessary to generate Newton applications are described. Other kinds of function objects that may be interpretable by a Newton device are not documented here.

Some of the information in this document actually belongs in the *NewtonScript Language Specification*, which does not yet exist. It will move there in the future.

This specification assumes familiarity with NewtonScript and the Newton object system. Some knowledge of language interpreter implementation is also useful.

▲ WARNING

The format described here is compatible with all existing Newton ROMs. However, this specification may change without notice, and Apple may render it incompatible in future Newton systems.

Introduction

Functions are the executable objects in NewtonScript: the targets of function calls and message sends. The purpose of a function is to calculate a single value, given zero or more arguments and a runtime environment. In the process, it may cause side-effects to the system state (often this is more useful than the value returned).

There are several kinds of function objects. This document specifies the format of only one of them, the *CodeBlock*, which contains a bytecode representation of a function. This is by far the most common kind of NewtonScript function; until NTK 1.5, it was the only kind NTK could generate.

NewtonScript Bytecode Interpreter Specification

This document is descriptive, not prescriptive. It describes what the instructions do, not when they should be generated. It does give the conditions under which the behavior of an instruction is defined, and which values in a data structure have defined meanings.

▲ WARNING

When the effect of an operation under certain circumstances is described as “undefined”, it is allowed to have any effect at all, including crashing the interpreter. All other circumstances should produce a legal result or throw an exception.

Virtual machine

In the same way that native code consists of instructions for a hardware machine, *bytecodes* are instructions for a *virtual machine*. Rather than being executed directly by the processor, bytecodes are executed by a software implementation of the virtual machine, called the *interpreter*. Thus, a function implemented as bytecodes is independent of the specific hardware or operating system it is executing on.

The operations represented by NewtonScript bytecodes are at a much higher level than processor instructions; they are defined in terms of NewtonScript semantics. This makes them much denser than the equivalent native code, which serves a more general purpose.

This document defines the virtual machine’s characteristics, and the virtual machine operations that are represented by the bytecodes. It also defines the format of the function objects themselves.

Function objects

Each function object has three objects associated with it that are referred to implicitly by the bytecodes:

- The *instructions object* is a binary object of class `instructions` that contains the bytecode instructions for the function.
- The *locals frame* is a frame containing the arguments and locals of the function, plus some extra slots. It is explained in more detail below.
- The *literals array* contains objects that are referred to in the function. For example, the literals array of the function `func() "hello"` would contain the "hello" string. The literals array is optional, since not all functions require it.

Locals frame

Each function has a *lexical environment* that defines a set of named variables. The function itself defines a “local” environment containing names for zero or more arguments and zero or more local variables. If the function is nested within another function, its lexical environment includes the lexical environments of the outer functions. The combination of these environments is called the “outer” lexical environment of the function.

NewtonScript Bytecode Interpreter Specification

Consider the following two functions:

```
f := func (a, b) begin
    local p;
    local g := func (d) begin
        local r, s;
        ...
    end;
    ...
end;
```

The lexical environment of function *f* consists only of its local environment: the arguments *a* and *b*, and the local variables *p* and *g*. The lexical environment of function *g* consists of a local environment (*d*, *r*, and *s*) plus an outer environment (the local environment of *f*).

The lexical environment of a function is represented as a frame called the *locals frame*. The locals frame contains one slot for each variable in the local environment of the function, tagged with the name of the variable. It also contains a `_nextArgFrame` slot that refers to the locals frame of the enclosing function, if any, and two slots (`_parent` and `_implementor`) that are placeholders for the interpreter's use.

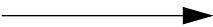
The locals frame is unusual in that the order of slots is important (normally, it is incorrect to rely on the order of the slots in a frame). The slot order has to be as follows:

- `_nextArgFrame`
- `_parent`
- `_implementor`
- arguments, from left to right
- local variables

The order of the local variables is not defined, but it determines the variable indices used by certain bytecodes.

All slots of the locals frame must contain `nil`, except for the `_nextArgFrame` slot, which must be a reference to the locals frame of the enclosing function, or `nil` if the function is at the top level.

The locals frames for the two example functions would look like this:

function g { <code>_nextArgFrame</code> :  , <code>_parent</code> : <code>nil</code> , <code>_implementor</code> : <code>nil</code> , <code>d</code> : <code>nil</code> , <code>r</code> : <code>nil</code> , <code>s</code> : <code>nil</code> }	function f { <code>_nextArgFrame</code> : <code>nil</code> , <code>_parent</code> : <code>nil</code> , <code>_implementor</code> : <code>nil</code> , <code>a</code> : <code>nil</code> , <code>b</code> : <code>nil</code> , <code>p</code> : <code>nil</code> , <code>g</code> : <code>nil</code> }
---	---

Note that `g`'s locals frame's `_nextArgFrame` slot points to `f`'s locals frame.

Function frame

The function object itself is a frame. Again, the ordering of the slots is critical and must be as follows:

<code>class</code>	Must be the symbol <code>CodeBlock</code> .
<code>instructions</code>	A binary object of class <code>instructions</code> containing the bytecodes.
<code>literals</code>	An array of class <code>literals</code> containing the literals, or <code>nil</code> if no literals are used.
<code>argFrame</code>	The locals frame for the function.
<code>numArgs</code>	The number of arguments the function takes.

The complete function frames for the example functions would be as follows:

function `f`

```
{class: 'CodeBlock,
  instructions: <instructions, length ?>,
  literals: [literals: ...],
  argFrame: {_nextArgFrame: nil,
             _parent: nil,
             _implementor: nil,
             a: nil,
             b: nil,
             p: nil,
             g: nil},
  numArgs: 2}
```

function `g`

```
{class: 'CodeBlock,
  instructions: <instructions, length ?>,
  literals: [literals: ...],
  argFrame: {_nextArgFrame: f.argFrame,
             _parent: nil,
             _implementor: nil,
             d: nil,
             r: nil,
             s: nil},
  numArgs: 1}
```

Virtual machine specification

The virtual machine (VM) consists of six registers and a stack of values. The virtual machine registers are:

Name	Contents
FUNC	Current function object
PC	Zero-based byte index in the instructions object of the current instruction
SP	Index of the most-recently-pushed value on the stack
LOCALS	Current locals frame
RCVR	Current receiver (for message sends)
IMPL	Current implementor (for message sends)

The values on the stack are referred to as `STACK[N]` where *N* is an integer index. Values pushed more recently have larger indexes, and indexes are assigned consecutively. The “top” of the stack is the most-recently-pushed value; that is, `STACK[SP]`.

The VM has two other stacks that are not explicitly represented in this specification. One is used implicitly by various instructions to save and restore the `FUNC`, `PC`, `LOCALS`, `RCVR`, and `IMPL` registers on function call and return. The other is used for exception handling (see “Exception handling” on page 2-10).

`RCVR`, the current receiver, is the value of the pseudo-variable `self`. `IMPL`, the current implementor, is a reference to the frame in which the current method was found (used by the lookup for inherited message send.) When a function is executed via a message send, these registers are set by the message send operation; when executed via a function call, these registers are set from the function being called.

Execution proceeds by repeatedly performing the operation at offset `PC` of the instructions object of `FUNC`. All instructions alter `PC`, explicitly or implicitly. The effect of a `PC` value outside the bounds of the instructions object (that is, less than zero or greater than the length of the instructions object minus one) is undefined.

Exceptions thrown by the interpreter

When the specification requires the interpreter to throw a particular exception, it is described as follows:

“Error is thrown” means an exception with name `|evt.ex.fr|` is thrown with error code `kFramesError`.

“Interpreter error *error* is thrown” means an exception with name `|evt.ex.fr.intrp|` is thrown with error code `kFramesError`.

NewtonScript Bytecode Interpreter Specification

Any of the above “with *data*” means “;type.ref.frame” is appended to the exception name, and the exception data is a frame with slots `errorcode`, containing *error*, and *data*, containing the value *data*.

“Bad type exception *error* is thrown” means an exception with name `|evt.ex.fr.type;type.ref.frame|` is thrown with error code `kFramesError`, and the exception data is a frame with slots `errorcode`, containing *error*, and *data*, containing the offending value.

Global variables and functions

The interpreter looks up global variables and global functions by name. The mechanism by which this lookup occurs is not defined. However, there are two global variables that affect it.

The global variable `vars` is required to contain a frame, each of whose slots contains the value of a global variable whose name is the slot tag. Changing one of these slots is required to change the value of the corresponding global variable. Creating a new slot is required to create a corresponding global variable. However, there may be valid global variables that do not correspond to slots in this frame; that is, the frame may contain only a subset of the global variables. The result of an assignment to the variable `vars` itself is undefined.

The global variable `functions` is required to contain a frame, each of whose slots contains the value of a global function whose name is the slot tag. Creating a new slot is required to create a corresponding global function. There may be valid global functions that do not correspond to slots in this frame. The result of an assignment to a slot of `functions`, or an assignment to the variable `functions` itself, is undefined.

Function call and message send

Because several bytecodes share the same call and send operations with only minor differences, the common features are documented here.

When a function call or message send occurs, there are arguments and other values parameterizing the operation (receiver, function name, etc.) on the stack. The function itself should leave a result on the stack; that is, the net result of the function should be a “push”. However, the stack contents when the function is entered are not defined. The values used by the call or send operation itself may be removed from the stack before or after the function executes. Thus, a function cannot rely on the stack contents when it is entered.

Function call

Calling a function works as follows:

1. If the target is not a function object, an exception is thrown. (The exception may vary and is undefined.)

NewtonScript Bytecode Interpreter Specification

2. If the number of arguments required by the function is not equal to the number of arguments on the stack, interpreter error `WrongNumberOfArgs` is thrown.
3. The VM registers are saved (including the updated PC).
4. `FUNC` is set to the new function object.
5. `PC` is set to zero.
6. The function's locals frame (`FUNC.argFrame`) is cloned, and `LOCALS` is set to the cloned frame.
7. The argument slots of `LOCALS` are filled with the arguments from the stack. That is, the arguments are put into the slots of `LOCALS` in left to right order, beginning with the fourth slot of `LOCALS`.
8. `RCVR` is set to the value of the `_parent` slot of `LOCALS`.
9. `IMPL` is set to the value of the `_implementor` slot of `LOCALS`.
10. Execution resumes.

Message send

A message is sent to a frame (the *receiver*). The receiver and its inheritance paths are searched for a slot matching the message name, and a slot is found in some frame (the *implementor*, which is not necessarily the same object as the receiver). The object in that slot is the *method*. The message send itself proceeds as follows:

1. If the method is not a function object, an exception is thrown. (The exception may vary and is undefined.)
2. If the number of arguments required by the method is not equal to the number of arguments on the stack, interpreter error `WrongNumberOfArgs` is thrown.
3. The VM registers are saved (including the updated PC).
4. `FUNC` is set to the method.
5. `PC` is set to zero.
6. `RCVR` is set to the receiver.
7. `IMPL` is set to the implementor.
8. The method's locals frame (`FUNC.argFrame`) is cloned, and `LOCALS` is set to the cloned frame.
9. `LOCALS._parent` is set to `RCVR`.
10. `LOCALS._implementor` is set to `IMPL`.
11. The argument slots of `LOCALS` are filled with the arguments from the stack. That is, the arguments are put into the slots of `LOCALS` in left to right order, beginning with the fourth slot of `LOCALS`.
12. Execution resumes.

Inheritance

The interpreter implements inheritance by looking up variables, slots, and messages along the NewtonScript inheritance paths. There are three lookup algorithms and two assignment algorithms, which are presented here in NewtonScript form. Three frame access functions are used:

- `HasSlot(frame, name)` returns `true` if `frame` contains a slot with the tag `name`, or `nil` if it does not.
- `GetSlot(frame, name)` returns the value of the slot with tag `name` in `frame`. If there is no such slot, it returns `nil`.
- `SetSlot(frame, name, value)` sets the value of the slot with tag `name` in `frame` to `value`. If there is no such slot, it creates one with that value.

The use of these functions is for pedagogical purposes only. There is no requirement that these algorithms actually be implemented by calls to such functions.

Each of these algorithms may succeed (by reaching the point marked “success”) or fail (by reaching the point marked “failure”). Operations that use the algorithms will refer to their “success” or “failure”. If the algorithm fails, its return value is not important.

Proto Lookup

A “proto lookup” follows only the `_proto` inheritance path to find a value.

```
func ProtoLookup(start, name) begin
    local current := start;
    while current <> nil do begin
        if HasSlot(current, name) then
            return GetSlot(current, name); // success
        current := GetSlot(current, '_proto');
    end;
    // failure
end;
```

Lexical lookup

A “lexical lookup” is exactly the same as a “proto lookup”, except substituting the symbol `_nextArgFrame` for the symbol `_proto`. It is referred to as the pseudo-function `LexicalLookup(start, name)`.

Full lookup

A “full lookup” follows both the `_proto` and `_parent` inheritance paths to find a value.

```
func FullLookup(start, name) begin
    local left := start;
    while left <> nil do begin
        local current := left;
```


NewtonScript Bytecode Interpreter Specification

```

    while current <> nil do begin
        if HasSlot(current, name) then
            return GetSlot(current, name); // success
        current := GetSlot(current, '_proto');
    end;
    left := GetSlot(left, '_parent');
end;
// failure
end;

```

Assignment

An “assignment” finds the location of a variable by searching the `_proto` and `_parent` inheritance paths. If a slot with the given name is found in a frame in the `_parent` chain, the value of that slot is changed. If the slot is found in the `_proto` chain of a frame in the `_parent` chain, a slot with the given name and value is created in the `_parent` frame.

```

func Assignment(start, name, value) begin
    local left := start;
    while left <> nil do begin
        local current := left;
        while current <> nil do begin
            if HasSlot(current, name) then begin
                SetSlot(left, name, value); // success
                return;
            end;
            current := GetSlot(current, '_proto');
        end;
        left := GetSlot(left, '_parent');
    end;
    // failure
end;

```

Lexical assignment

A “lexical assignment” find the location of a lexical variable by searching the `_nextArgFrame` chain and setting the value of the slot found.

```

func LexicalAssignment(start, name, value) begin
    local current := start;
    while current <> nil do begin
        if HasSlot(current, name) then begin
            SetSlot(current, name, value); // success
            return;
        end;
    end;
end;

```

```

        end;
        current := GetSlot(current, '_nextArgFrame');
    end;
    // failure
end;

```

Exception handling

The interpreter maintains a stack of exception handler contexts, each of which represents the dynamic scope of a `try/onexception` statement. A context contains the saved state of the virtual machine registers (except for PC) and the function call stack. It also contains a mapping from exception names to PC values.

A context may be “used” or “unused”. Contexts are created in the “unused” state. When a context is selected to catch an exception, it is set to the “used” state, and the exception name and data are stored in the context.

The `new-handlers` instruction (see page 2-20) creates an exception handler context containing the current VM state and pushes it on the handler stack. It takes exception names and PC values from the value stack to produce the mapping saved in the exception handler context.

The `pop-handlers` instruction (see page 2-13) removes the most recent context from the handler stack.

When an exception is thrown, the handler stack is searched for unused contexts with matching exception names.

If such a context is found, the following actions occur:

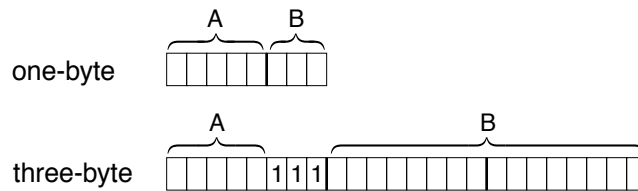
- All more recent contexts are popped from the handler stack (that is, the matching context becomes the top of the stack).
- The context is marked “used”, and the exception name and data are stored in it.
- The VM state is restored from the context
- PC is set to the PC value in the context corresponding to the exception name.
- Execution continues.

If no matching context is found, the effect is undefined.

The name and data of the “current” exception may be obtained by calling the global function `CurrentException`. If there are no used contexts on the handler stack, its result is `nil`. Otherwise, it returns the name and data of the most recent used context on the stack.

Structure of instructions

Each instruction in the bytecode stream is either one or three bytes long. An instruction contains two fields, A and B.



- The A field is the most significant five bits of the first byte of the instruction.
- The B field has two formats, depending on the size of the value:
 - If the value can be represented in three bits, it is stored in the least significant three bits of the first byte. The instruction is one byte long.
 - Otherwise, the least significant three bits of the first byte are set to all ones, and the value is stored in the following two bytes, with the most-significant byte of the value first. The instruction is three bytes long.

Depending on the instruction, the B field may be signed or unsigned. That is, a B field of 0xFFFF may be interpreted as -1 or as 65535.

▲ WARNING

The interpreter in Newton **in ROM versions prior to 2.0** incorrectly interprets the B field values of all three-byte instructions as signed.

There are two kinds of instructions:

- In *simple* instructions, the A field is fixed, and the B field gives the operation to be performed.
- In *parameterized* instructions the A-field gives the operation to be performed, and the B-field is a parameter to the operation.

NOTE

A parameterized instruction may use the three-byte format even with B-field values that would fit into three bits. This makes generating backpatched branch instructions easier. Simple instructions must use the one-byte format when possible.

Each instruction has a mnemonic name. The name is used only for documentation purposes; it has no significance to the format.

Instruction definitions

Each instruction definition includes a line describing the values expected on the stack when it is encountered and the values it leaves on the stack after it executes. For example:

```
arg1 arg2 -- arg1 ret1
```

NewtonScript Bytecode Interpreter Specification

This indicates that the instruction expects `arg1` as `STACK[SP-1]` and `arg2` as `STACK[SP]`. It will leave `arg1` on the stack and push `ret1`. That is, on exit, `STACK[SP-1]` will be `arg1` and `STACK[SP]` will be `ret1`. If the stack contents on entry are not relevant, the left side will be empty. If all the items on the left side are popped from the stack, the right side will be empty.

In an instruction description, “pushing `X` onto the stack” means, in pseudocode:

```
SP := SP + 1; STACK[SP] := X
```

“Popping `X` from the stack” means, in pseudocode:

```
X := STACK[SP]; SP := SP - 1
```

All instructions except `branch`, `branch-if-true`, and `branch-if-false` implicitly increment `PC` by the size of the instruction.

▲ WARNING

Any instruction not defined here is reserved for use by Apple and must not occur. The behavior of all such instructions is undefined.

Simple instructions

All of the simple instructions have an `A`-field of zero. They are distinguished by the `B`-field. All but one (`pop-handlers`) is a one-byte instruction.

pop (B = 0)

```
x --
```

Removes the top element from the stack.

The effect when the stack is empty is undefined.

dup (B = 1)

```
x -- x x
```

Duplicates the top element of the stack.

The effect when the stack is empty is undefined.

return (B = 2)

```
--
```

Returns from the current function. The VM registers are restored from the most recently saved state, and execution continues. By convention, the value left in `STACK[SP]` is the return value of the function.

The effect when no state has been saved is undefined.

push-self (B = 3)

`-- RCVR`

Pushes RCVR onto the stack.

set-lex-scope (B = 4)

`func -- closure`

Pops a function object from the stack, constructs a closure from the function object, and pushes the closure onto the stack.

A closure is a function object with saved values for the outer lexical environment, the current receiver, and the current implementor. These values are saved in the locals frame of the function object in the `_nextArgFrame`, `_parent`, and `_implementor` slots. The original function object and its locals frame are cloned, and the values are filled in. In pseudocode, the operation looks like:

```
fn := Clone(Pop());
af := Clone(fn.argFrame);
af._nextArgFrame := LOCALS;
af._parent := RCVR;
af._implementor := IMPL;
fn.argFrame := af;
Push(fn);
```

If `func` is not a `CodeBlock`, the behavior is undefined.

iter-next (B = 5)

`iterator --`

Pops a reference to an iterator from the stack and advances it to the next slot. See “Iterators” on page 2-25.

If `iterator` is not an iterator object, the behavior is undefined.

iter-done (B = 6)

`iterator -- done?`

Pops a reference to an iterator from the stack. If `iterator` is exhausted, pushes `true` onto the stack; otherwise, pushes `nil` onto the stack.

If `iterator` is not an iterator object, the behavior is undefined.

pop-handlers (B = 7)

`--`

Removes the most recent exception handler context.

See “Exception handling” on page 2-10 for more information on exception handlers.

If no exception handler context is in effect, the behavior is undefined.

Parameterized instructions

Unless otherwise specified, the B field is interpreted as an unsigned integer.

push (A = 3)

-- literal

Pushes an element of the literals array (`FUNC.literals`). The B field is the zero-based index in the literals array of the element to push.

If `FUNC.literals` is nil, the behavior is undefined. The effect of a B field value outside the bounds of the literals array is undefined.

push-constant (A = 4, B is signed)

-- value

Pushes the B field onto the stack as a literal value.

The B field must be an immediate value (that is, the low-order bits must be 00 or 10), or a magic pointer whose index is between 0 and 4095, inclusive. Otherwise, the effect is undefined.

NOTE

The B field is interpreted as a signed value in this instruction.

call (A = 5)

arg1 arg2 ... argN name -- result

Calls a global function. The function arguments (if any) are on the stack in left-to-right order, followed by a symbol giving the name of the function to call. The B field contains the number of arguments on the stack.

The point at which the arguments and function name are removed from the stack is undefined, but when control returns to the instruction following the call, they are gone.

1. The global function named by `name` is looked up. If there is no such function, interpreter error `UndefinedGlobalFunction` is thrown with the unresolved name.
2. Function call proceeds as in “Function call” on page 2-6.

invoke (A = 6)

arg1 arg2 ... argN func -- result

Performs a function call. The function arguments (if any) are on the stack in left-to-right order, followed by the function object to call. The B field contains the number of arguments on the stack.

The point at which the arguments and function object are removed from the stack is undefined, but when control returns to the instruction following the call, they are gone.

The function call itself is described in “Function call” on page 2-6.

send (A = 7)

`arg1 arg2 ... argN name receiver -- result`

Performs a message send. The function arguments (if any) are on the stack in left-to-right order, followed by the message name (a symbol), followed by the receiver. The B field contains the number of arguments on the stack.

The point at which the arguments, message name, and receiver are removed from the stack is undefined, but when control returns to the instruction following the send, they are gone.

1. The method is looked up using `FullLookup(receiver, name)`. If the lookup fails, interpreter error `UndefinedMethod` is thrown with the unresolved name.
2. Message send proceeds as in “Message send” on page 2-7.

send-if-defined (A = 8)

`arg1 arg2 ... argN name receiver -- result`

Performs a conditional message send. The function arguments (if any) are on the stack in left-to-right order, followed by the message name (a symbol), followed by the receiver. The B field contains the number of arguments on the stack.

The point at which the arguments, message name, and receiver are removed from the stack is undefined, but when control returns to the instruction following the send, they are gone.

1. The method is looked up using `FullLookup(receiver, name)`. If the lookup fails, `nil` is pushed onto the stack (as `result`), and execution continues with the next instruction.
2. If the lookup succeeds, message send proceeds as in “Message send” on page 2-7.

resend (A = 9)

`arg1 arg2 ... argN name -- result`

Performs an inherited message send. The function arguments (if any) are on the stack in left-to-right order, followed by the message name (a symbol). The B field contains the number of arguments on the stack.

The point at which the arguments and receiver are removed from the stack is undefined, but when control returns to the instruction following the send, they are gone.

1. If `IMPL` has no `_proto` slot, interpreter error `UndefinedMethod` is thrown with the unresolved name.
2. The method is looked up using `ProtoLookup(IMPL._proto, name)`. If the lookup fails, interpreter error `UndefinedMethod` is thrown with the unresolved name.
3. Message send proceeds as in “Message send” on page 2-7 using `RCVR` as the receiver.

resend-if-defined (A = 10)

Performs a conditional inherited message send. The function arguments (if any) are on the stack in left-to-right order, followed by the message name (a symbol). The B field contains the number of arguments on the stack.

The point at which the arguments and receiver are removed from the stack is undefined, but when control returns to the instruction following the send, they are gone.

1. If IMPL has no `_proto` slot, `nil` is pushed onto the stack (as `result`), and execution continues with the next instruction.
2. The method is looked up using `ProtoLookup(IMPL._proto, name)`. If the lookup fails, `nil` is pushed onto the stack (as `result`), and execution continues with the next instruction.
3. Message send proceeds as in “Message send” on page 2-7 using `RCVR` as the receiver.

branch (A = 11)

--

PC is set to the B field value.

branch-if-true (A = 12)

value --

A value is popped from the stack. If it is `nil`, execution continues with the next instruction. Otherwise, PC is set to the B field value.

The effect when the stack is empty is undefined.

branch-if-false (A = 13)

value --

A value is popped from the stack. If it is `nil`, PC is set to the B field value. Otherwise, execution continues with the next instruction.

The effect when the stack is empty is undefined.

find-var (A = 14)

-- value

Performs a variable lookup. The B field is the zero-based index in the literals array of a symbol (here called `name`) naming the variable.

1. The symbol is looked up in the lexical environment using `LexicalLookup(LOCALS, name)`. If the lookup succeeds, the result is pushed onto the stack and execution continues with the next instruction.
2. Otherwise, the symbol is looked up using `FullLookup(RCVR, name)`. If the lookup succeeds, the result is pushed onto the stack and execution continues with the next instruction.
3. Otherwise, if there is a global variable called `name`, its value is returned.

4. Otherwise, interpreter error `UndefinedVariable` is thrown with the name of the unresolved variable.

If `LITERALS` is `nil`, or the `B` field is not a valid index into the literals array, or the element of the literals array it references is not a symbol, the effect is undefined.

get-var (A = 15)

-- value

Gets a value from `LOCALS`. The `B` field is the zero-based index of a slot in `LOCALS`, whose value is pushed on the stack.

The effect of a `B` field value outside the bounds of `LOCALS` is undefined.

make-frame (A = 16)

val1 val2 ... valN map -- frame

Makes a frame and fills in its slots using values from the stack. The `B` field contains the number of slot values on the stack. The slot values are on the stack in index order, followed by the map to use for the frame. The slot values and map are removed from the stack, and a reference to the newly-allocated frame is pushed onto the stack.

The `B` field may contain a number less than the number of slots in the frame, in which case the remaining slots at the end of the frame are set to `nil`.

For more information about frame maps, see

.

The effect is undefined if any of the following occur:

- The `B` field value is greater than the number of slots in the frame.
- There are fewer slot values on the stack than the `B` field specifies.
- Map is not a valid frame map.

make-array (A = 17)

B = 0xFFFF: size class -- array

B < 0xFFFF: val1 val2 ... valN class -- array

Makes an array, and optionally fills in its slots using values from the stack.

The class for the array is popped from the stack. The rest of the operation depends on the `B` field:

- If the `B` field contains the value 0xFFFF, an integer `size` is popped from the stack. An array of length `size` and class `class` is allocated, all of whose slots are `nil`, and a reference to it is pushed onto the stack.
- Otherwise, the `B` field contains the size of the array. An array of that size and class `class` is allocated. The values for the array slots, on the stack in index order, are copied into the slots of the array. The values are removed from the stack, and a reference to the array is pushed onto the stack.

If there are not enough values on the stack, the effect is undefined.

get-path (A = 18)

`object pathExpr -- value`

Retrieves the value corresponding to `pathExpr` in the frame or array object. The B field may be zero or one.

1. The object and path are popped from the stack.
2. If object is `nil`, the operation depends on the B field.
 - If the B field is zero, `nil` is pushed onto the stack, and execution continues with the next instruction.
 - If the B field is one, error `PathFailed` is thrown.
3. Otherwise, the the value corresponding to `pathExpr` in the frame or array object is pushed onto the stack.

The value corresponding to a path expression is the value that would be found by doing array and frame accesses corresponding to each element of the path expression. Integers represent array accesses to the given array index; symbols represent frame accesses to the given frame slot.

If the B field is neither zero nor one, the effect is undefined.

set-path (A = 19)

`B = 0: object pathExpr value --`

`B = 1: object pathExpr value -- value`

Sets the value corresponding to `pathExpr` in the frame or array object to `value`. The B field may be zero or one.

The object, path, and value are popped from the stack. If the B field is one, the value is pushed back onto the stack.

If the B field is neither zero nor one, the effect is undefined.

set-var (A = 20)

`value --`

Sets a slot in `LOCALS`. The B field is the zero-based index of a slot in `LOCALS`, whose value is set to a value popped from the stack.

If the stack is empty, or the B field value is outside the bounds of `LOCALS`, the effect is undefined.

find-and-set-var (A = 21)

`value --`

Performs a variable assignment. The B field is the zero-based index in the literals array of a symbol (here called `name`) naming the variable.

1. The assignment is attempted in the lexical environment using `LexicalAssignment(LOCALS, name, value)`. If the lookup succeeds, execution continues with the next instruction.

2. Otherwise, the assignment is attempted using `Assignment(RCVR, name, value)`. If the lookup succeeds, execution continues with the next instruction.
3. Otherwise, if there is a global variable called `name`, its value is set to `value`.
4. Otherwise, a slot is added to `LOCALS` with the tag `name` and the value `value`.

If the `B` field is not a valid index into the literals array, or the element of the literals array it references is not a symbol, or `LITERALS` is `nil`, the effect is undefined.

incr-var (A = 22)

`addend -- addend value`

Increments a slot of Locals. The `B` field is the zero-based index of a slot in `LOCALS`. The value in this slot is added to the value in `STACK[SP]` and the result is stored into this slot. The result is also pushed onto the stack.

If either the `LOCALS` slot value or `addend` is not an integer, bad type error `NotAnInteger` is thrown with the bad value. If neither value is an integer, the value which will be used is not defined.

If the stack is empty, or the `B` field value is outside the bounds of `LOCALS`, the effect is undefined.

branch-if-loop-not-done (A = 23)

`incr index limit --`

Branch if a for loop is not done. The `B` field contains the `PC` of the loop start.

1. The three values are popped from the stack.
2. If any of `incr`, `index`, or `limit` is not an integer value, bad type error `NotAnInteger` is thrown with the bad value. If multiple values are not integers, the value which will be used is not defined.
3. If `incr` is zero, interpreter error `ZeroForLoopIncr` is thrown.
4. If `incr > 0` and `index <= limit`, set `PC` to the `B` field value.
5. If `incr < 0` and `index >= limit`, set `PC` to the `B` field value.
6. Otherwise, execution continues with the next instruction.

freq-func (A = 24)

`arg1 arg2 ... argN -- result`

Calls a primitive function. The `B` field contains the index of the primitive function to be called. The primitive functions are described in the section “Primitive functions” on page 2-20.

If the `B` field does not contain a valid primitive function index, or if the number of arguments on the stack is not that expected by the primitive function, the effect is undefined.

new-handlers (A = 25)

```
sym1 pc1 sym2 pc2 ... symN pcN --
```

Sets up an exception handler context. The B field contains the number of exception names matched by the handler context. Each name (a symbol) and the offset of the first instruction of its handler (an integer) is on the stack. All these values are popped from the stack, and the handler context is set up.

See “Exception handling” on page 2-10 for more information on exception handlers.

If there are fewer slot values on the stack than the B field specifies, or the values are invalid, the effect is undefined.

Primitive functions

Some NewtonScript operations are not implemented directly as bytecode instructions. Instead, they are defined as *primitive functions*.

A primitive function is an operation that is performed like a function call: its arguments are pushed on the stack, and it pops them and pushes a result onto the stack. However, it is not required to be implemented as a function call.

Some primitive functions have corresponding global functions that perform the same operations. Redefining the global function that corresponds to a primitive function is not required to alter the behavior of the primitive function.

Primitive functions are invoked by the `freq-func` instruction (see page 2-19). They are selected by an index number in the B field of the instruction. They are presented here grouped by functionality, not by index number.

Table 1-2 on page 2-27 lists the primitive functions by index number and gives their corresponding global function names.

Arithmetic operations

The primitive arithmetic operations operate only on “numbers”; that is, integers or reals. An argument of any other type causes the function to throw bad type error `kFramesErrNotANumber`.

Unless otherwise specified, operations with integer arguments produce integer results. Operations with real or mixed arguments produce real results.

add (index = 0)

```
num1 num2 -- result
```

Adds `num1` and `num2`. If both arguments are integers, the result is an integer. Otherwise, any integer argument is converted to a real, and the result is a real.

If both arguments are integers and their sum cannot be represented as an integer, the result is an undefined but valid number.

subtract (index = 1)

```
num1 num2 -- result
```

Subtracts num2 from num1. If both arguments are integers, the result is an integer. Otherwise, any integer argument is converted to a real, and the result is a real.

If both arguments are integers and their difference cannot be represented as an integer, the result is an undefined but valid number.

multiply (index = 7)

```
num1 num2 -- result
```

Multiplies num1 by num2. If both arguments are integers, the result is an integer. Otherwise, any integer argument is converted to a real, and the result is a real.

If both arguments are integers and their product cannot be represented as an integer, the result is an undefined but valid number.

divide (index = 8)

```
num1 num2 -- result
```

Divides num1 by num2.

- If num2 is the integer zero, the exception |`evt.ex.div0`| is thrown.
- If both arguments are integers, and num2 divides evenly into num1, the result is an integer.
- Otherwise, any integer argument is converted to a real, and the result is a real.

NOTE

If num2 is the real number 0.0, a normal floating-point division occurs.

div (index = 9)

```
int1 int2 -- result
```

Divides int1 by int2. If either argument is not an integer, bad type error `NotAnInteger` is thrown with one of the bad arguments. The result is the integer quotient, rounded towards zero.

Array/string functions

Some of the following functions may throw “out of bounds” errors. This means an exception with name |`evt.ex.fr;type.ref.frame`| is thrown with error code `kFramesErrOutOfBounds`, and the exception data is a frame with slots `errorcode`, containing *error*, `data`, containing the array/string object, and `index`, containing the offending index value.

aref (index = 2)

`object index -- element`

Gets an element from an array or string.

- If `object` is an array, the result is the value of the slot at zero-based offset `index` in the array. If `index` is less than zero, or greater than or equal to the number of slots in the array, an out of bounds error is thrown.
- If `object` is a string, the result is the character at zero-based offset `index` in the string. If `index` is less than zero, or greater than or equal to the number of characters in the string, an out of bounds error is thrown.
- If `array` is neither an array nor a string, bad type error `ArrayOrString` is thrown.

set-aref (index = 3)

`object index element -- element`

Sets an element of an array or string.

- If `object` is an array, the slot at zero-based offset `index` is set to `element`. If `index` is less than zero, or greater than or equal to the number of slots in the array, an out of bounds error is thrown.
- If `object` is a string, the character at zero-based offset `index` is set to `element`. If `index` is less than zero, or greater than or equal to the number of characters in the string, an out of bounds error is thrown. If `element` is not a character, `kFramesErrNotACharacter` is thrown. Note that `element` must not be `$\u0000` or a character whose code is in the range `0xF700` to `0xF7FF`, inclusive.
- If `array` is neither an array nor a string, bad type error `NotAnArrayOrString` is thrown.

new-iterator (index = 17)

`object deeply -- iterator`

Creates an iterator (see “Iterators” on page 2-25) for `object`. If `object` is a frame and `deeply` is non-nil, the iterator will follow `_proto` links in `object`.

If `object` is not a frame or array, bad type error `NotAFrameOrArray` is thrown.

length (index = 18)

`object -- length`

Gets the length of the array, frame, or binary object `object`. This is the same as the built-in function `Length` (see *The NewtonScript Programming Language*).

add-array-slot (index = 21)

`array object -- object`

Adds one slot, containing `object`, to the end of `array`. This is the same as the built-in function `AddArraySlot` (see *The NewtonScript Programming Language*).

Comparison functions

The comparison functions operate on integers, reals, characters, and strings. The arguments to `equals` and `not-equals` may be of any type. The types of the arguments to the ordered comparisons must be the same, except that integers and reals may be mixed in any combination; in other cases an exception is thrown.

The result of all comparisons is either `nil` or `true`.

equals (index = 4)

`obj1 obj2 -- result`

Compare objects for identity or numeric equality.

- If the arguments are numbers, compare their numeric values. (An integer-valued real is equal to the corresponding integer.)
- If the arguments are immediates, compare their values directly.
- If the arguments are pointers, compare object identities.

not-equals (index = 6)

`obj1 obj2 -- result`

The opposite of `equals`. The result is `nil` when `equals` would be `true`, and vice versa.

less-than (index = 10)

`obj1 obj2 -- result`

greater-than (index = 11)

`obj1 obj2 -- result`

less-or-equal (index = 13)

`obj1 obj2 -- result`

greater-or-equal (index = 12)

`obj1 obj2 -- result`

Compare two values.

- If both values are numbers, compare their numeric values.
- If both values are characters, compare their character codes numerically,
- If both values are strings, use a case- and diacritical-insensitive lexicographic ordering to compare. Character codes between 32 and 126, inclusive, are compared numerically by character code, after mapping lowercase letters (97–122) to their uppercase equivalents.

NOTE

Comparison of strings containing characters outside the range 32–126 is implementation-dependent.

Logical operations

not (index = 5)

`value -- result`

Logical negation. If `value` is `nil`, `result` is `true`. Otherwise, `result` is `nil`.

bit-and (index = 14)

`int1 int2 -- result`

Bitwise logical AND of two integers. The result is an integer whose two's-complement representation is the bitwise logical AND of the two's-complement representations of `int1` and `int2`.

If either argument is not an integer, the bad type exception `NotAnInteger` is thrown.

bit-or (index = 15)

`int1 int2 -- result`

Bitwise logical OR of two integers. The result is an integer whose two's-complement representation is the bitwise logical OR of the two's-complement representations of `int1` and `int2`.

If either argument is not an integer, the bad type exception `NotAnInteger` is thrown.

bit-not (index = 16)

`int -- result`

Bitwise complement of an integer. The result is an integer whose two's-complement representation is the bitwise logical complement of the two's-complement representation of `int`.

If the argument is not an integer, the bad type exception `NotAnInteger` is thrown.

Miscellaneous

set-class (index = 20)

`object class -- object`

Sets the class of `object` to `class`. This is the same as the built-in function `SetClass` (see *The NewtonScript Programming Language*).

class-of (index = 24)

`object -- class`

Gets the class of `object`. This is the same as the built-in function `ClassOf` (see *The NewtonScript Programming Language*).

clone (index = 19)

`object -- clone`

Clones `object`. This is the same as the built-in function `Clone` (see *The NewtonScript Programming Language*).

stringer (index = 22)

`array -- string`

Converts each element of `array` to a string and returns the concatenation of those strings. This is the same as the NewtonScript infix `&` operator (see *The NewtonScript Programming Language*).

If the argument is not an array, an exception is thrown.

has-path (index = 23)

`object pathExpr -- result`

Returns true if there is a slot corresponding to the path expression `pathExpr` beginning at `object`. See the `get-path` instruction on page 2-18.

Support objects

The virtual machine definition assumes the existence of certain support objects that must be provided by the interpreter.

Iterators

NewtonScript's `foreach` construct iterates through the slots of an array or frame. In the interpreter, the state of a `foreach` loop is maintained by an *iterator* object.

Operation

An iterator is created on an array or frame (the “target”) by the `new-iterator` primitive function (see page 2-22). It is advanced by the `iter-next` instruction (see page 2-13), and tested for completion by the `iter-done` instruction (see page 2-13).

At any given time, an iterator is either “done” (finished iterating), or has a “current” slot. The `iter-next` instruction changes the current slot to the next slot in the object. An

NewtonScript Bytecode Interpreter Specification

iterator on an array visits the slots in order, starting at slot zero. An iterator on a frame visits the slots in an arbitrary order. In either case, each slot is visited exactly once.

If the second argument to `new-iterator` is not `nil`, the iterator will follow `_proto` links. All of the slots in each frame, including the `_proto` slot, appear together in the iteration sequence, followed by all the slots of the frame referred to by the `_proto` slot, and so forth. If `object` is not a frame, or the value of a `_proto` slot is not a frame, an exception is thrown by `new-iterator`.

When all slots have been visited, `iter-done` returns `true`, and the result of `iter-next` on that iterator is undefined.

Within a `foreach` loop (that is, while `iter-next` and `iter-done` may still be called on the iterator), the current slot may be removed. Adding or removing any other slot will cause an exception to be thrown. Modifying the value of any slot is permitted, however.

Access

The state of an iterator object is accessed using the `aref` bytecode. The following `aref` indexes may be used:

- 0 The tag of the current slot
- 1 The value of the current slot
- 3 If the second argument to `new-iterator` is true, the total number of slots that will be visited by the iterator
- 5 The number of slots in `object`

NOTE

Although `aref` is used to access the state of the iterator, it is not required be an actual array. No operations may be performed on an iterator other than `iter-next`, `iter-done`, and `aref` with one of the above indexes.

Reference

Table 1-1 Bytecodes by encoding

Bytecode encodings are given in octal for convenience. An “x” as the last digit indicates that either a one- or three-byte encoding may be used, depending on the B field. Note that the B field of some bytecodes is restricted; see the detailed descriptions.

Encoding	Name
000	pop
001	dup
002	return
003	push-self

NewtonScript Bytecode Interpreter Specification

Encoding	Name
004	set-lex-scope
005	iter-next
006	iter-done
007 000 001	pop-handlers
03x	push
04x (<i>B signed</i>)	push-constant
05x	call
06x	invoke
07x	send
10x	send-if-defined
11x	resend
12x	resend-if-defined
13x	branch
14x	branch-if-true
15x	branch-if-false
16x	find-var
17x	get-var
20x	make-frame
21x	make-array
220/221	get-path
230/231	set-path
24x	set-var
25x	find-and-set-var
26x	incr-var
27x	branch-if-loop-not-done
30x	freq-func
31x	new-handlers

Table 1-2 Bytecodes by name

Name	Encoding
branch	13x
branch-if-false	15x
branch-if-loop-not-done	27x
branch-if-true	14x
call	05x

NewtonScript Bytecode Interpreter Specification

Name	Encoding
dup	001
find-and-set-var	25x
find-var	16x
freq-func	30x
get-path	220/221
get-var	17x
incr-var	26x
invoke	06x
iter-done	006
iter-next	005
make-array	21x
make-frame	20x
new-handlers	31x
pop	000
pop-handlers	007 000 001
push	03x
push-constant	04x (<i>B signed</i>)
push-self	003
resend	11x
resend-if-defined	12x
return	002
send	07x
send-if-defined	10x
set-lex-scope	004
set-path	230/231
set-var	24x

Table 1-3 Primitive functions by index

Index	Name	Global function name
0	add	+
1	subtract	-
2	aref	aref
3	set-aref	setAref
4	equals	=
5	not	not

NewtonScript Bytecode Interpreter Specification

Index	Name	Global function name
6	not-equals	<>
7	multiply	*
8	divide	/
9	div	div
10	less-than	<
11	greater-than	>
12	greater-or-equal	>=
13	less-or-equal	<=
14	bit-and	BAnd
15	bit-or	BOr
16	bit-not	BNot
17	new-iterator	newIterator
18	length	Length
19	clone	Clone
20	set-class	SetClass
21	add-array-slot	AddArraySlot
22	stringer	Stringer
23	has-path	<i>none</i>
24	class-of	ClassOf

Table 1-4 Primitive functions by name

Name	Index
add	0
add-array-slot	21
aref	2
bit-and	14
bit-not	16
bit-or	15
class-of	24
clone	19
div	9
divide	8
equals	4
greater-or-equal	12
greater-than	11

NewtonScript Bytecode Interpreter Specification

Name	Index
has-path	23
length	18
less-or-equal	13
less-than	10
multiply	7
new-iterator	17
not	5
not-equals	6
set-aref	3
set-class	20
stringer	22
subtract	1

Newton Load Package Protocol

This chapter describes the protocol used to transfer packages from a desktop computer to a Newton device.

▲ WARNING

The protocol described here is compatible with all existing Newton ROMs. However, this specification may change without notice, and Apple may render it incompatible in future Newton systems.

Protocol Overview

Newton communicates with the desktop machine by exchanging Newton event commands. The general structure of these event commands is:

```
'newt'
'dock'
'aaaa'    // The specific command
length    // the length of the following command
data      // data, if any
```

In the commands described in this chapter, all data is padded with nulls to 4 byte boundaries. The length associated with each command is the length (in bytes) of the data following the length field. The length does not include any padding that might be added to the end of the command.

Loading a Package

The protocol necessary to load a package is very simple and is illustrated below.

Desktop		Newton
	<-	kDRequestToDock
kDInitiateDocking	->	
	<-	kDNewtonName
kDSetTimeout	->	
	<-	kDResult
kDLoadPackage	->	
	<-	kDResult
kDDisconnect	->	

The following is a summary of all the commands that can be used and their four-letter definition:

kDRequestToDock	'rtdk'
kDNewtonName	'name'
kDInitiateDocking	'dock'
kDSetTimeout	'stim'
kDResult	'dres'
kDLoadPackage	'lpkg'
kDDisconnect	'disc'

All commands begin with the sequence -- 'newt', 'dock' -- as shown in the general form on the preceding page. For simplicity, that's not shown in the descriptions that follows.

Newton -> Desktop

```
kDRequestToDock
    'rtdk'
    length = 4
    protocol version = 9
```

This command is sent to the desktop after the connection is established using AppleTalk, serial, etc. (when the user taps the connect button). The protocol version is the version of the messaging protocol that's being used and should always be set to the number 9 for the version of the protocol defined here.

Newton Load Package Protocol

```

KDNewtonName
    'name'
    length
    version info
    name

```

This command is sent in response to a correct `kDInitiateDocking` command from the desktop. The version info includes things like machine type (for example, J1), ROM version, and so on. There is no requirement to process this information: it can simply be skipped when loading a package. To skip over this information, just read `length` bytes from the connection and proceed to the next command. Here's a full description of what's in the version info (each is a long):

```

length of version info in bytes
newtonUniqueID - a number uniquely identifying the newton
manufacturer id
machine type
rom version
rom stage
ram size
screen height
screen width
system update version
Newton object system version
signature of internal store
vertical screen resolution
horizontal screen resolution
screen depth

```

The version info is followed by the Newton owner's name sent as a Unicode string including the terminating zeros at the end. The string is padded to an even 4 bytes by adding zeros as necessary (the padding bytes are not included in the length sent as part of the command header).

Desktop-> Newton

```

kDInitiateDocking
    'dock'
    length = 4
    session type = 4

```

Newton Load Package Protocol

This command should be sent to the Newton in response to a `kDRequestToDock` command. Session type should be 4 to load a package.

```
kDSetTimeout
    'stim'
    length = 4
    timeout in seconds = 30
```

This command sets the timeout for the connection (the time the Newton will wait to receive data before it disconnects). This time is usually set to 30 seconds.

```
kDLoadPackage
    'lpkg'
    length
    package data
```

This command will load a package into the Newton's default store. The package data should be padded to an even multiple of 4 by adding zero bytes to the end of the package data. The `length` should be the file length of the package file. The package data itself should be the contents of the data fork of the package file.

```
kDDisconnect
    'disc'
    length = 0
```

This command is sent to the Newton when the load package operation is complete. Be sure to give the Newton time to read the command before disconnecting to avoid an error appearing on the Newton.

Desktop-> Newton or Newton<-Desktop

```
kDResult
    'dres'
    length = 4
    error code
```

This command can be sent for two reasons: to indicate that the Newton or desktop can proceed or to indicate an error. In the first case a 0 error code is sent to indicate that the protocol can proceed. In the second case, the connection is dropped after the error is processed.

Newton Streamed Object Format

This chapter describes the format of streamed Newton objects—the format used when streaming a DAG (Directed Acyclic Graph)* of objects over a communications link. A streamed object is simply a stream of byte values.

You probably do not need to know anything about Newton Streamed Object Format unless you need to write your own encoder or decoder.

This chapter assumes you understand the basics of the Newton object system. See the NewtonScript documentation for more information.

▲ **WARNING**

The format described here is compatible with all existing Newton ROMs. However, this specification may change without notice, and Apple may render it incompatible in future Newton systems.

Introduction

Newton Streamed Object Format (NSOF) can describe a set of objects that are all reachable from a “root” object. In practice, this generally means that NSOF describes a NewtonScript frame along with everything that it contains.

Encoding is done by an **encoder**. An encoder processes an object and emits a streamed object.

Decoding is done by a **decoder**. A decoder processes a streamed object and emits a copy of the original object. In general, the format insures that the decoded set of objects is identical to the original, including circular pointer relationships.

* “Directed Acyclic Graph” is a computer science term for a generalized data structure. It is used here to make clear that the objects encoded in Newton Streamed Object Format do not have to be simple tree structures—they can be more generalized structures that may contain internal circular references.

Streamed Object Format

An encoder processes an object and emits a streamed object.

The first byte of a coded object is a version byte that refers to the NSOF version. The version number of the format described here is 2. (Future versions may not be backward compatible.)

The rest of the coded object is a recursive description of the DAG of objects, beginning with the root object.

The beginning of each object's description is a tag byte that specifies the encoding type used for the object.

The tag byte is followed an ID, called a **precedent ID**. The IDs are assigned consecutively, starting with 0 for the root object, and are used by the `kPrecedent` tag to generate backward pointer references to objects that have already been introduced. Note that no object may be traversed more than once; any pointers to previously traversed objects must be represented with `kPrecedent`. Immediate objects cannot be precedents; all precedents are heap objects (binary objects, arrays, and frames).

Encoding

Table 3-2 shows how objects of different types are encoded in NOSF. Data types used in Table 3-2 are as follows:

Table 3-1 Data Types Used in Table 3-2

Data Type	Meaning
byte	Unsigned byte
halfword	Two bytes
long	Signed long
xlong	$0 \leq \text{value} \leq 254$: unsigned byte else: byte 0xFF followed by signed long
object	This definition (recursive). Must be a <code>kPrecedent</code> clause if the object has been assigned a precedent ID.

Table 3-2 shows a type in the left-hand column. The right hand column shows the sequence of values used to represent an object of that type. The first value is a tag that identifies the type. Following that tag is whatever is needed to specify the value or values held in the object.

Newton Streamed Object Format

Data types that are assigned precedent IDs are marked with asterisks (*).

Table 3-2 Encoding Table

Type	Encoded Version
immediate	kImmediate=0 (byte) Immediate Ref (xlong) See “Immediate Objects” on page 4-4
character	kCharacter=1 (byte) Character code (byte) See “Special Case Types” on page 4-5
uniChar	kUnicodeCharacter=2 (byte) High byte of character code (byte) Low byte of character code (byte) See “Special Case Types” on page 4-5
binary*	kBinaryObject=3 (byte) Number of bytes of data (xlong) Class (object) Data See “Binary Object Data” on page 4-5
array*	kArray=4 (byte) Number of slots (xlong) Class (object) Slot values in ascending order (objects)
plainArray*	kPlainArray=5 (byte) Number of slots (xlong) Slot values in ascending order (objects) See “Special Case Types” on page 4-5
frame*	kFrame=6 (byte) Number of slots (xlong) Slot tags in ascending order (symbol objects) Slot values in ascending order (objects)
symbol*	kSymbol=7 (byte) Number of characters in name (xlong) Name (bytes)

Newton Streamed Object Format

Table 3-2 Encoding Table

Type	Encoded Version
string*	kString=8 (byte) Number of bytes in string (xlong) String (halfwords) See “Binary Object Data” and “Special Case Types” on page 4-5
precedent	kPrecedent=9 (byte) Precedent ID (xlong)
nil	kNIL=10 (byte) See “Special Case Types” on page 4-5
smallRect*	kSmallRect=11 (byte) Top value (byte) Left value (byte) Bottom value (byte) Right value (byte) See “Special Case Types” on page 4-5
largeBinary*	kLargeBinary=12 (byte) Class (object) Compressed? (non-zero means compressed) (byte) Number of bytes of data (long) Number of characters in compander name (long) Number of byte of compander parameters (long) Reserved (encode zero, ignore when decoding) (long) Compander name (bytes) Compander parameters (bytes) Data (bytes) See “Binary Object Data” on page 4-5

Immediate Objects

Immediate objects are represented by `kImmediate` followed by a `Ref` that gives the value of the immediate. Table 3-3 shows

- how an encoder can generate an encoded `Ref` from an immediate value
- how a decoder can test a coded `Ref` to determine the type

Newton Streamed Object Format

- how a decoder can convert the Ref into the original immediate value.
- The tests and conversions shown are C expressions.

Table 3-3 How to Construct the Ref of an Immediate

Type	How Encoders Generate	How Decoders Can Test	How Decoders Extract
integer	(v << 2) [arithmetic shift]	(r & 3) == 0	(r >> 2)
character	(v << 4) 6	(r & 0xF) == 6	(r >> 4) & 0xFFFF
TRUE	0x1A	r == 0x1A	--
NIL	2	r == 2	--
magicptr	(v << 2) 3	(r & 3) == 3	--

Binary Object Data

All binary object data is encoded in Newton format. In particular, real numbers and strings are encoded in big-endian byte order, and strings include a Unicode null terminator.

Special Case Types

The formats `character`, `uniChar`, `plainArray`, `string`, `nil`, and `smallRect` are special cases of more general types. Although each of these could be sent using the more general type, encoders can use specifiers for these types in order to reduce the size of the streamed data.

For example, suppose you have a rectangle specification like this:

```
aRect: {left: 10, top: 14, right: 40, bottom: 100}
```

This is a frame, and could be represented as a frame. The encoder would then need to send the name of each slot along with the slot values. On the other hand, if the encoder recognizes this is a rectangle specification, it can use the `smallRect` specifier. When using `smallRect`, the encoder does not need to send the slot names, since every `smallRect` has the same four slots.

Decoders of this format must properly decode these cases, but encoders do not have to generate them.

Example of Newton Streamed Object Format

Here is an example of a NewtonScript frame.

Newton Streamed Object Format

```
x := {name: "Walter Smith",
      cats: 2,
      bounds: {left: 10, top: 14, right: 40, bottom: 100},
      right: $\u2022,
      phones: ["408-996-1010", nil]};
x.phones[1] := SetClass("408-974-9094", 'faxPhone);
x.nameAgain := x.name;
```

The last two lines modify the frame in order to make the example more complex.

If you print this frame it prints as:

```
{ name: "Walter Smith",
  cats: 2,
  bounds: {left: 10, top: 14, right: 40, bottom: 100},
  uchar: $\u2022,
  phones: ["408-996-1010", <faxPhone, length 26>],
  nameAgain: "Walter Smith"}
```

The streamed representation of this frame is:

```
02060607046E616D650704636174730706626F756E6473070575636861720706
70686F6E657307096E616D65416761696E081A00570061006C00740065007200
200053006D0069007400680000000080B0E0A64280220220502081A0034003000
38002D003900390036002D003100300031003000000031A070866617850686F6E
65003400300038002D003900370034002D0039003000390034000000907
```

This streamed representation translates as:

```
02-version number
06-kFrame [ID 0]
06-length, 6 slots
Slot tags:
  07 (kSymbol) 04 (length of name) 6E616D65 ("name") [ID 1]
  07 04 63617473 ("cats") [ID 2]
  07 06 626F756E6473 ("bounds") [ID 3]
  07 05 7563686172 ("uchar") [ID 4]
  07 06 70686F6E6573 ("phones") [ID 5]
  07 09 6E616D65416761696E ("nameAgain") [ID 6]
Slot values:
  08-kString [ID 7]
  1A-length, 26 bytes
  00570061006C00740065007200200053006D0069007400680000 ("Walter Smith")
  00-kImmediate
  08-Ref of the integer 2
  0B-kSmallRect [ID 8]
```


Newton Streamed Object Format

```

0E (top=14) 0A (left=10) 64 (bottom=100) 28 (right=40)
02-kUnicodeCharacter
2022-The character code
05-kPlainArray [ID 9]
02-length, 2 slots
Slot values:
    08-kString [ID 10]
    1A-length, 26 bytes
    003400300038002D003900390036002D00310030003100300000 ("408-996-1010")
    03-kBinaryObject [ID 11]
    1A-length, 26 bytes
    Class:
        07(kSymbol) 08 (length of name) 66617850686F6E65 ("faxPhone")[ID 12]
        003400300038002D003900370034002D00390030003900340000 ("408-974-9094")
09-kPrecedent
07-ID 7 ("Walter Smith" object above)

```

Newton Streamed Object Format