

BeIA Miscellany

This document is confidential and may not be distributed without permission of Be Incorporated.

copyright © 2000 Be Incorporated

BeIA Miscellany

Table of Contents

Introduction	7
Boot Mode	9
1 The Modes	9
1.1 Tools Mode	9
1.2 Validate Mode	9
1.3 First Boot Mode	10
1.4 Normal Mode	10
2 Switching Between Modes	10
2.1 The bootmode Program	10
2.2 The BOOTMODE Variable	11
3 Implementation	11

The Update Mechanism 13

1	Update Terms	13
2	Update Players and Rules	13
2.1	The Update Server	14
2.2	The Update Daemon	14
2.3	Update Packages (Verification, Encryption, and Security)	14
2.4	The Update Scripts	15
3	Update Session Outline	15
3.1	Initiating an Update Session	15
3.2	Contacting the Server	15
3.3	Receiving a Response	16
3.4	Executing updatescript	16
3.5	Updating and After	16
4	Update Environment Variables	16
4.1	Defined in /etc/update/revision	17
4.2	Defined in /etc/update/update_url	17
4.3	Defined in /etc/update/updt_ident	17
4.4	Defined by the Update Daemon	17
5	Scripts	17
5.1	updt	18
5.1.1	Variables	18
5.1.2	How it Works	18
5.2	updatescript	18
5.2.1	Variables	19
5.2.2	How it Works	19
5.3	doupdate	19
5.3.1	Variables	19
5.3.2	How it Works	20
5.4	doupgrade	20
5.4.1	Variables	20
5.4.2	How it Works	21
5.5	doscript	21
5.5.1	Invocation	21
5.5.2	Variables	22
5.5.3	How it Works	22
6	Upgrade Alert Files	22
7	Check List	22
8	Configurability	23

Digital Signature Algorithm25

1	How the Digital Signature Algorithm Works	25
2	DSA on BeIA (dsasig)	25
2.1	Generating DSA Parameters	26
2.2	Generating Keys.	26
2.3	Signing a Document.	26
2.4	Verifying a Document.	26
2.5	Creating a Hash Value	26
3	Division of Labor	27
4	Default DSA Files	27

The Microshell29

1	Script Syntax and Grammar	29
2	Built-in Commands	30
2.1	Environment Variables	30
2.2	File System Navigation and Access	30
2.3	System Settings.	31
2.4	System Execution	31
2.5	Process Execution	31
2.6	Boolean Logic	32
2.7	File Existence	32
2.8	Miscellaneous.	32
3	External Commands	32
3.1	Script Interpreter	33
3.2	Communication with the Server and other Apps	33
3.3	RAM Disk Management	33
3.4	System Checks	33
3.5	File Decryption and Verification	33

BeIA Miscellany

Introduction

The sections in this chapter are:

- “Boot Mode” describes the different modes in which a BeIA device operates. You switch modes during the product creation cycle to move from development, to testing, to deployment.
- “The Update Mechanism” describes how to update a BeIA device by downloading new software from a remote update server.
- “Digital Signature Algorithm” tells you how to create digital signatures for validating update packages.
- “The Microshell” describes the commands you can use in the BeIA microshell (**bterm**).

BeIA Miscellany

Boot Mode

The developer or user environment that a BeIA device boots depends on the device's **boot mode**. This document describes the four boot modes, and explains how to switch between them.

1 The Modes

There are four boot modes:

- **Tools mode** is used during development and customization.
- **Validate mode** runs diagnostics on the device. This is the mode you want to be in when you duplicate your BeIA image.
- **First boot mode** is used when the user first boots the device.
- **Normal mode** is the normal operating mode—it's the mode the user sees on the second and subsequent boots.

1.1 Tools Mode

While you're developing your BeIA device, you want to be in **tools mode**. In this mode, the device boots into the browser, but you can go to the BeIA desktop environment by typing **alt+q**. From the desktop you can examine and modify files, launch a **bterm** window to download customization files (through **ftp**), and launch the browser (through **TellBrowser**) to test your BeIA configuration.

Tools mode is the default—when you receive a new BeIA image, it's automatically in tools mode.

When you're satisfied with your customization—in other words, when you're ready to copy your BeIA image onto other device—you run the **Make_Release_Image** program by clicking the application's icon on the desktop:

<<illo>>

Make_Release_Image removes the tools mode-only files (such as the **bterm** program), runs **checksum** on the rest of the files (for later validation), switches your device to **validate mode**, and then shuts down your device. At this point, your device's image is ready to be duplicated.

Note that you can put your device in validate mode (or any other mode) without removing the tools mode files by running the **bootmode** program from the command line, as described in “2 Switching Between Modes.”

1.2 Validate Mode

When you boot a device that's in **validate mode**, the BeIA validation test suite is run. The suite comprises some user-driven diagnostics—“Press a key,” “Click the mouse,” etc.—as well as a **checksum** test.

If the device passes the validation tests, it's automatically put into **first boot mode**. At this point, the device is ready to be shipped to an end user.

For more information on factory validation, see "Factory Validation."

1.3 First Boot Mode

When the user boots a device that's in **first boot mode** (and it's assumed that the user's first boot will be in this mode), the device launches the browser which immediately displays the HTML page defined in `$RESOURCES/firstboot.html`. This first boot page is expected to welcome the user, help him set things up, and so on. As with all local HTML pages, `firstboot.html` is fully customizable.

After launching the browser in first boot mode, the device is put into **normal mode**.

1.4 Normal Mode

Normal mode is what the user normally sees: When the device is booted (or wakes up), the `$RESOURCES/login.html` file is evaluated by the browser. See <<customization spec>> for more information.

2 Switching Between Modes

Normally, you switch between modes by following the natural course of events, as described in the sections above. However, you can force a device into a particular mode through the **bootmode** command line program, or by setting the `BOOTMODE` variable in the file `/boot/home/config/settings/beia-bootmode`.

2.1 The bootmode Program

The **bootmode** program is typically used to set the device's boot mode state. The syntax is:

```
bootmode --set [ --altq ] tools | validate | firstboot | normal
```

The `altq` switch, if present, turns on the `alt+q` to the desktop feature. If `altq` is absent, `alt+q` will not exit to the desktop.

For example, to put a device in firstboot mode, but retain the `alt+q` feature, you would open a **bterm** window and type...

```
$ bootmode --set --altq firstboot
```

bootmode performs other system state maintenance as well: The `no_boot_keys` switch turns off access to the boot menu, and disables kernel debugging output:

```
$ bootmode --no_boot_keys
```

The `configure_recovery` switch reads a "factory settings" file:

```
$ bootmode --configure_recovery file
```

The factory settings file contains device configuration information that's described <<elsewhere>>.

If you invoke **bootmode** with no arguments, it prints, to standard output, the current boot mode.

Note that **Make_Release_Image** deletes the **bootmode** program (as well as **bterm** and other tools mode files). In other words, after you've run **Make_Release_Image** you'll no longer be able to switch between modes in this manner.

2.2 The BOOTMODE Variable

You can also set the mode by setting the `BOOTMODE` environment variable, stored in `/boot/home/config/settings/beia-bootmode`, to one of `"tools"`, `"validate"`, `"firstboot"`, or `"normal"`. To turn on or off the `alt+q` feature, set the `ALTQ` variable. For example:

```
BOOTMODE="tools"
ALTQ=1
```

This puts the device into tools mode, and turns on `alt+q`.

3 Implementation

When your BeIA machine boots, it runs the `/system/boot/Bootscript` script. Within this script is a command that tells the system to determine the current boot mode by reading the mode from the file:

```
/boot/home/config/settings/beia-bootmode
```

(The programs and HTML pages—**Make_Release_Image**, `firstboot.html`, etc.—that need to switch the mode do so by writing to this file.) The boot mode is stored as the `$BOOTMODE` environment variable.

Next, the bootscript runs the mode-specific script stored in...

```
/system/boot/Bootscript.$BOOTMODE
```

There are four mode-specific script files, `Bootscript.tools`, `Bootscript.validate`, `Bootscript.firstboot`, and `Bootscript.normal`.

You can edit these scripts, although you should understand that you may subvert the expected sequence of modes if you do so.

BeIA Miscellany

The Update Mechanism

Note: The update mechanism described here is made obsolete by the new BeIA MAP technology.

One of the features of a BeIA device is that its software can be easily upgraded. An upgrade can be as small as a single GIF or HTML file, or it can comprise major system components, such as servers or the browser. Since the device is always (or periodically) connected to the internet, the new software can be downloaded from a remote “update server” on the Web; the device itself then takes over and installs the new software automatically, often without the user being aware that an update has been installed.

This document describes the process by which the software and data on a user’s BeIA device is upgraded from a remote server, explains what a vendor must do to take part in the system, and describes how a vendor can modify the system.

1 Update Terms

- The term “client” means the user’s BeIA device that’s being updated.
- An “update package” is a compressed file that contains new data that’s applied to the client.
- The “update server” (or, simply, “server”) is the remote computer that provides update packages to the client.
- A “major update” refers to the installation (on the client) of an update package that’s greater than 1 megabyte or that touches critical system components that mustn’t be upgraded while the device is being used. During a major update, the client machine is effectively unusable—the browser is killed, and the machine reboots when the update has finished. The user is told when a major update is about to happen, and is allowed to defer the update. A major update is also called an “upgrade”.
- A “minor update” refers to the installation of an update package that’s less than 1 megabyte and that only replaces “safe” files (HTML, GIF, etc.). Minor updates happen without affecting the browser and don’t require that the client device be rebooted. The user should never be aware that a minor update is going on.

2 Update Players and Rules

The components of the update mechanism, and some of the rules that they play by, are described in the following sections.

2.1 The Update Server

The update server is a remote machine, provided by the device's ISP, that maintains and serves update packages. The URL of the update server is identified in the client's `/etc/update/update_url` file. Every client device must be apprised of its update server's URL before it's delivered to the user.

Note: Although it's possible for the vendor to change the update server's URL, "legacy" URLs will need to be supported until all client devices have been updated to use the new URL.

The update server is expected to provide or maintain:

- A database of update packages, indexed by language, local, and client hardware, as appropriate for the array of client devices it supports.
- An "version control script" that identifies the latest update package, and that provides the logic for bringing any client up-to-date. This may mean downloading a single update, a series of small updates, a single upgrade, an upgrade followed by a series of updates—it all depends on how old the client's files are. A functional version control script is provided as part of BeIA.
- Zip and digitally sign all packages that it makes available for downloading (see "2.3 Update Packages (Verification, Encryption, and Security)").

The server isn't expected to:

- Keep track of which version of the software a specific client device is running. The client knows its own version, and compares itself to the server when it runs the version control script.
- Push data onto a client. The update server never sends data to the client without being asked.
- Provide an HTML interface. The user never needs to—and probably shouldn't be allowed to—initiate an update, so the server doesn't need to present an interface to the database of update packages.

2.2 The Update Daemon

Running on every BeIA client device is an "update daemon". The daemon contacts the update server every four hours to check for, download, and install a new update package (or series of packages). If the previous check resulted in an error (because the server is down, or a package couldn't be verified, or some other problem), the daemon runs every 15 minutes until the error is resolved.

Note that the daemon won't connect to the update server unless the device already has a network connection in place. For example, the daemon doesn't wake up a sleeping device to check for a new update.

There's no user interface associated with the update check; it happens automatically in the background.

2.3 Update Packages (Verification, Encryption, and Security)

All update packages provided by the update server must be zipped and then digitally signed. On the client side, any package that's downloaded from the server is verified before it's unzipped; invalid packages are thrown away.

BeIA clients use the Digital Signal Algorithm (DSA) program **dsasig** to perform the digital signature verification. The public DSA key that the client uses to authenticate a package must be installed on the client device before it's delivered to the user. **dsasig** is described in the "Digital Signature Algorithm" document in the *BeIA Miscellany* chapter.

Note: Although **dsasig** will prevent Web pirates from altering your packages, it *isn't* a secure encryption scheme—it won't prevent them from stealing and reading your packages. Data encryption for security purposes is up to the vendor—BeIA doesn't supply a secure encryption scheme.

2.4 The Update Scripts

Most of the update mechanism is driven through a series of **Microshell** scripts that are executed on the client. For example, the update daemon contacts the update server by executing the “update script”; the server must provide a “version control script” that’s downloaded to and executed by the client; to download and install update packages, the client executes the “do update” and “do upgrade” scripts.

The scripts provided by BeIA live in the `/etc/update` directory; they’re examined in a later section “(5 Scripts)”

The BeIA scripts can be used as is, or they can be modified or replaced by the BeIA vendor.

3 Update Session Outline

An outline of the update mechanism looks like this:

1. **1. Initiating an Update Session.** The update daemon wakes up and (if an internet connection is already in place) executes the **updt** script.
2. **2. Contacting the Server.** **updt** sends a message to the update server that describes the client’s machine (language, locale, and product type—but *not* the client’s revision number).
3. **3. Receiving a Response.** The update server sends back an **updatescript** file that’s been fine-tuned for the client’s machine.
4. **4. Executing updatescript.** **updt** executes **updatescript**, which determines if the client machine is up-to-date: If it is, no action is taken (wait four hours and go to step 1); if it isn’t, **updatescript** downloads a new update package.
5. **5. Updating and After.** If this is a minor update, the update package is installed without alerting (or otherwise interrupting) the user. If it’s a major update, an alert lets the user proceed with or defer installation, after which the device rebooted.

The following sections go into greater detail about the process.

3.1 Initiating an Update Session

Whenever a BeIA client device connects to the network—because it was just turned on, or was just awakened, or because the network went down but is back up—the device’s update daemon (`/system/servers/updated`) initiates a new “update session” by executing the **updt** script (`/etc/update/updt`).

Note: Since it’s called by the update daemon, the `/etc/update/updt` script must exist. However, the contents of the script are completely configurable. The rest of the update mechanism is “driven” from **updt**—if you replace **updt**, you’ve effectively replaced the entire update mechanism.

3.2 Contacting the Server

updt sends a message to the update server (i.e. to the [UPDATE_URL](#) URL) by executing the **doupdate** script. (This script, described in detail later, is used to download small update packages.)

3.3 Receiving a Response

The update server looks at the arguments to the message sent by the client to determine the most recent update package that applies to the client's configuration. It then provides a version of the **updatescript** file that identifies the candidate package; the **updatescript** file is downloaded to the client as part of the **doupdate** script that the client is executing.

Note that the update server doesn't care (or even know) whether the update package that it chooses is already running on the client.

3.4 Executing updatescript

The first thing the client does after downloading **updatescript** is to verify its authenticity through the digital signature program **dsasig** (see "Digital Signature Algorithm" in the *BeIA Miscellany* chapter). This verification is part of the **doupdate** script.

The client then executes **updatescript** (again, through a line in **doupdate**) which is expected to contain the logic for determining the steps that are necessary to bring the client's machine up-to-date. If the client is up-to-date, then no further steps are taken and the update process will begin again in four hours when the update daemon wakes up and executes **updt**. If the client needs to be updated, **updatescript** executes **doupdate** (for minor updates) or **doupgrade** (for major updates).

Note that **updatescript** can also furnish its own download-and-install instructions, rather than depending on **doupdate/doupgrade**.

3.5 Updating and After

Minor updates (*a la doupdate*) are performed without notifying the user. After a minor update is successfully installed, the update daemon waits four hours and again executes **updt**. However, if a minor update installation fails, the daemon will try the installation again after fifteen minutes, and continue trying until the installation is successful.

In a major update (*a la doupgrade*), an alert is presented that gives the user a chance to defer the installation. If the user chooses to defer, the daemon tries again after fifteen minutes (i.e. the alert is presented every fifteen minutes until the user acquiesces). If the installation is successful, device will automatically reboot; when it comes back up and connects to the network, the daemon will execute **updt** and restart its four hour (or fifteen minute) cycle. If the installation of a major update fails, the recovery system takes over.

Warning: After a major system upgrade the client device may be in an inconsistent state. Because of this, you should call **reboot** as soon as possible after the upgrade. If you need to perform some clean up before rebooting, you should only call **Microshell**-internal commands; other (binary) commands may crash the system. See "The Microshell" in the *BeIA Miscellany* chapter for a list of the **Microshell**-internal commands.

4 Update Environment Variables

The client device defines a set of environment variables that identify the update server and describe some client parameters. The variables are maintained as text in various files on the client, or are defined by the update daemon.

Variables are defined one-per-line in this format:

variable=value

If the value is a string, it should be quoted. Numeric values needn't be; for example:


```
UPDATE_URL="http://www.myISP.com/updates/"
REVISION=23
```

To read a file's variables (from the **Microshell** or from a script) you [source](#) the file:

```
source /boot/home/config/beia-redirect
```

Environment variables are referred to with the [\\$](#) operator. For example:

```
echo $REVISION
```

The variables are listed by in the sections below.

Note: Some files use the [setenv](#) expression ("[setenv variable value](#)") to set their variables' values. The [setenv](#) expression is equivalent to the "[variable=value](#)" formula shown above.

4.1 Defined in /etc/update/revision

/etc/update/revision contains a single variable:

- **REVISION.** A value (typically numeric) that describes the latest major update that was downloaded to the client device. The value of this variable is reset whenever a new major update is installed.

4.2 Defined in /etc/update/update_url

/etc/update/update_url contains a single variable:

- **UPDATE_URL.** The URL of the update server for this client.

4.3 Defined in /etc/update/updt_ident

/etc/update/updt_ident contains a single variable:

- **updt_ident.** A value (typically numeric) that describes the latest minor update that was downloaded to the client device. The value of this variable is reset whenever a new minor update is installed.

4.4 Defined by the Update Daemon

The update daemon sets these variables:

- **RELOAD.** This is a boolean variable that's used to determine whether a new update script needs to be downloaded from the server, as explained in "[5.1 updt.](#)"
- **SCRIPT_URL.** The address of a script you want to download and execute via the **doscript** script. The value of **SCRIPT_URL** is set through a message to the update daemon. See "[5.5 doscript](#)" for more information.

5 Scripts

The update scripts are located in /etc/update/. They are:

- **updt:** Responsible for downloading and executing the latest **updatescript**.
- **updatescript:** Downloadable script responsible for version control logic.
- **doupdate:** Non-interactive script used for minor updates.

- **douppgrade**: Interactive script used for major updates.
- **doscript**: Convenient, non-interactive script that can download and execute arbitrary, unzipped scripts.

The scripts are described in the following sections. Since the scripts are text files, you can easily modify or replace them. As explained earlier, only `/etc/update/updt` is hard-coded as a pathname (because it's called from the update daemon). All other script are invoked, directly or indirectly, from **updt**.

All scripts return 0 on success and non-0 on failure.

5.1 updt

Declared in: `/etc/update/updt`

updt sets up the update environment, (possibly) downloads a new **updatescript** file from the update server, and then executes the script. **updt** is invoked by the update daemon—you shouldn't invoke it directly from your own scripts.

5.1.1 Variables

updt expects the following environment variable to be set:

- **RELOAD**: This is set by the update daemon to be **true** if this is a normal four hour update check, and **false** if this is a fifteen minute check (see “3.5 Updating and After” for more information on the four hour vs. fifteen minute check). **RELOAD** is used to by **updt** to determine whether a new **updatescript** file should be downloaded (**true**) or if the existing **updatescript** should be used (**false**).

5.1.2 How it Works

1. **source** the environment variables in `/boot/home/config/settings/beia-redirect`.
2. Check to see if the system is awake (the **nut** command), continuing only if it is, and returning 0 otherwise.
3. If **RELOAD** is **true**, set the update variables (**UPDATE_...**) that are used by **doupdate**, and then invoke that script. This downloads the **updatescript** file from the update server and places it in `/etc/update/`. If **RELOAD** is **false**, this step is skipped.
4. Check for the existence of `/etc/update/updatescript`, and then execute it (if it exists). The value returned by **updatescript** is returned by **updt**. If **updatescript** doesn't exist, **updt** returns 0.

5.2 updatescript

Declared in: `/etc/update/updatescript`

updatescript contains the logic that determines whether the client device needs to be upgraded, and, if it does, what upgrade path it should take (major update, minor update, major update followed by one or more minor updates, etc.). The client downloads **updatescript** from the update server during an update session, as described in “5.1 updt.” The default version—which is provided for example purposes only—is described here.

5.2.1 Variables

updatescript expects the following variables to be set:

- **REVISION**: Defined in `/etc/update/revision`, this is a numeric value of the latest major update that was installed on the client device.
- **UPDT_IDENT**: Defined in `/etc/update/updt_ident`, this is a numeric value of the latest minor update that was installed on the client device.

updatescript is expected to set these variables to values it brings from the server:

- **CUR_REVISION**: The value of the latest major update available (for this device) on the server.
- **CUR_IDENT**: The value of the latest minor update available (for this device) on the server.

As it's running, **updatescript** defines these variable:

- **NEXT_REVISION**: The value of the major update that's being installed right now. This is only significant if the upgrade path passes through multiple major upgrades.
- **UPDT_STATUS**: Update installation error code.

5.2.2 How it Works

1. Set the **CUR_REVISION** and **CUR_IDENT** numbers.
2. Retrieve the value of **REVISION** by `source`'ing `/etc/update/revision`. If the file doesn't exist, set **REVISION** to 0.
3. Retrieve the value of **UPDT_IDENT** by `source`'ing `/etc/update/updt_ident`. If the file doesn't exist, set **UPDT_IDENT** to 0.
4. Set **UPDT_STATUS** to 0.
5. Major update: Compare **REVISION** to **CUR_REVISION**; if they match, we don't need to do a major update—skip to step 6. If they don't match, set the proper variables and call **doupgrade**. If **doupgrade** is successful, update the `/etc/update/revision` file, spawn a process that will reboot the device, and then exit with 0. If **doupgrade** fails, exit with non-0.
6. Minor update: Compare **UPDT_IDENT** to **CUR_IDENT**; if they match, we don't need to do a minor update—exit with 0. If they don't match, set the proper variables and call **doupdate**. If **doupdate** is successful, update the `/etc/update/updt_ident` file, spawn a process that will reboot the device, and then exit with 0. If **doupdate** fails, exit with non-0.
7. Note that **updatescript** never performs a major update *and* a minor update.

5.3 doupdate

Declared in: `/etc/update/doupdate`

doupdate is used to download, verify, and unzip small (less than 1 MB) update packages. It operates without interrupting the user—no alert panel is presented by the script.

5.3.1 Variables

doupdate expects the following variables to be set:

- **UPDATE_PACKAGE**: A string identifier that names the resource that you want to download from the update server. The name must be meaningful to the server.
- **UPDATE_TEMP**: The pathname of the file on the client device that you want the update package to be copied to. **UPDATE_TEMP** should be an absolute pathname; the file is automatically deleted after the package is installed.
- **UPDATE_TARGET**: The directory on the client device into which the update package will be installed.
- **UPDATE_KEY**: The path to the public key used to verify the package. **UPDATE_KEY** can be a relative or absolute pathname; relative names are appended to `/etc/update/keys/`.

doupdate also uses the variables defined in `/boot/home/config/settings/beia-redirect` (see “Update Environment Variables”).

5.3.2 How it Works

1. Send a **wget** command to the update server (**UPDATE_URL**) requesting that **UPDATE_PACKAGE** be downloaded to the client (as **UPDATE_TEMP**).
2. Verify the package by running **dsasig** on the downloaded file using **UPDATE_KEY** as the public key.
3. If the verification fails, **UPDATE_TEMP** is deleted and **doupdate** returns non-0. If it succeeds, **UPDATE_TEMP** is **unzip**'d into **UPDATE_TARGET**.
4. If **unzip** fails, **UPDATE_TEMP** is deleted and **doupdate** returns non-0. If it succeeds, the **UPDATE_TEMP** file is deleted and **doupdate** returns 0.

5.4 doupgrade

Declared in: `/etc/update/doupgrade`

doupgrade performs a major update. Because of the disruption to the user that it causes, upgrades should only be performed when large or delicate parts of the system are being replaced. When executed, **doupgrade** displays an “Upgrade?” alert that lets the user **Upgrade Now** or **Defer**. If the user defers, the “Upgrade?” alert is presented again every 15 minutes; when the user finally consents, **doupgrade** shuts down the browser and presents an alert that shows the running status of the upgrade (and that warns the user not to interrupt the process).

If the upgrade completes successfully, the device reboots itself. If it fails, the browser is restarted.

5.4.1 Variables

doupgrade expects these variables to be set:

- **UPDATE_PACKAGE**, **UPDATE_TARGET**, **UPDATE_KEY**: See the **doupdate** variables (“5.3.1 Variables”).
- **UPDATE_SIZE**: The size of the update package file, in megabytes.

Note that **doupgrade** doesn't use the **UPDATE_TEMP** variable.

doupgrade also uses the variables defined in `/boot/home/config/settings/beia-redirect` (see “Update Environment Variables”).

5.4.2 How it Works

1. Send a message to **TellBrowser** that causes the upgrade alert to be displayed, and capture the user's response (**Install now** or **Defer**). (The upgrade alert is describe in the next major section.)
2. If the user wants to defer, exit with non-0. This will put the update daemon into fifteen minute mode, and the upgrade alert will reappear after fifteen minutes. If the user wants to proceed with the update...
3. ...shut down the browser, and send a message to **TellBrowser** telling it to display the upgrade status alert.
4. Download the update package ([UPDATE_PACKAGE](#)) into a specially-built RAM disk, verify it with **dsasig**, and **unzip** it into the target directory ([UDPATE_TARGET](#)).
5. If the package is successfully installed, tell the user by putting up a new alert, then [sync](#) the file system and [reboot](#) the device. If the installation is unsuccessful, tell the user, tear down the RAM disk, and return non-0.

5.5 doscript

Declared in: `/etc/update/doscript`

doscript is used to download, verify, and execute an abitrary, non-zipped script. **doscript** isn't an automated part of the update system so you don't *have* to use it for anything. And if you do want to use it, you have to explicitly tell the update daemon to run it (by sending a message, as explained below). Of course, you could just execute **doscript** yourself; the advantage of having the update daemon run it is that the execution will be synchronous with the rest of the update mechanism. In other words, by telling the update daemon to execute **doscript**, you are guaranteed that the script won't be executed while an update is going on.

Typically, **doscript** is tied to a UI object that the user can press—push the button and **doscript** is executed.

doscript always copies the downloaded file to `/etc/update/script`.

Note: Keep in mind that the file that **doscript** downloads *must not* be zipped.

5.5.1 Invocation

To invoke **doscript**, you use the global [beos](#) JavaScript object to send a “command request” message to the update server:

```
beos.sendMessageSync("application/x-vnd.be-UPDT" , "ScRq" ,
    "command", "string", "doscript",
    "SCRIPT_URL", "string", serverScriptURL);
```

The parts of the message are:

- "application/x-vnd.be-UPDT" is the update daemon's signature.
- "ScRq" is understood (by the update daemon) to indicate that this is a “command request” message.
- The "command"/"string"/"doscript" triplet identifies (by string name) the command you want to execute (i.e. **doscript**).
- The triplet on the final line sets the [SCRIPT_URL](#) environment variable to the `serverScriptURL` that you supply.

You can retrieve the script's return code by looking at `beos.reply` when the function returns.

For more information on the `beos` JavaScript object, see “JavaScript” in the *BeIA Development* chapter.

5.5.2 Variables

doscript expects this environment variable to be set:

- `SCRIPT_URL`: Address of the script you want to download.

5.5.3 How it Works

1. **source** the environment variables in `/boot/home/config/settings/beia-redirect` (just in case).
2. Remove the existing `/etc/update/script` file.
3. Download `SCRIPT_URL` from the server and copy it to `/etc/update/script`.
4. Verify `/etc/update/script`, set its execution bits (`chmod 755`), and execute the script.
5. If the script can't be verified, remove it and return non-0; otherwise, return 0.

6 Upgrade Alert Files

The template files used for the upgrade alerts are:

- `custom/resources/$LANGUAGE/Alerts/alert2.html` is the html template for the upgrade alert.
- `custom/resources/$LANGUAGE/Alerts/upgradeinfo.html` is the html template for the upgrade status alert.
- `custom/resources/$LANGUAGE/Alerts/doupgrade.txt` contains the strings that the alerts display.

7 Check List

Before a BeIA client device is shipped the following needs to be done:

- A machine should be configured to act as a server. The sever must be able to provide the appropriate **updatescript** file based on the arguments described in “**3.2** Contacting the Server.”
- Update packages (and **updatescript** files) on the server need to be **zip**'d and digitally signed.
- The client device's `/etc/update/update_url` file needs to identify the URL of the server.
- The client device needs to be configured with the correct server parameters.
- A new set of DSA keys must be created.
- The alert panels and strings should be modified or localized.
- The sample **updatescript** should be modified to suit your update needs.

8 Configurability

The most important ingredients in the BeIA update mechanism are the update scripts. These scripts control the communication with the update server, and provide the logic and rules for determining how the scripts are written in the **Microshell** scripting language (see “The Microshell” in the *BeIA Miscellany* chapter) and are completely configurable by the vendor.

The vendor can also configure the alert panel and text that’s used to warn the user of an impending upgrade.

The process that drives the update mechanism—the BeIA update daemon (`/system/servers/updated`)—is not configurable. The update daemon’s primary responsibility is to execute, every four hours, the script that contacts the update server. As mentioned above, you can’t (currently) change the four hour timing. Also, the fact that the daemon doesn’t automatically create a connection to the network if a connection isn’t already in place can’t be changed.

BeIA Miscellany

Digital Signature Algorithm

A digital signature is a verification code that's used to ensure that a document comes, untampered, from a known source. The BeIA updater mechanism expects digital signatures to be added to all packages that are sent from the update server to a client device. It uses the Digital Signature Algorithm (DSA) as its signature method.

1 How the Digital Signature Algorithm Works

The Digital Signature Algorithm uses a pair of cryptographic keys (a “private” key and a “public” key), a hash function:

- The keys, which are generated by the document's sender, are related in that a document encrypted with the private key can only be decrypted with the public key. The document sender “hides” the private key but makes the public key known to the document's recipient (and to anyone else—the whole point of DSA is that publicizing the public key doesn't violate the integrity of the system).
- It doesn't matter what hash function is used as long as the sender and recipient use the same function.

Before sending a document, the sender creates a “message digest” by running the document through the hash function. The sender then creates a digital signature by encrypting the message digest with the private key. Both the document and the digital signature (i.e. the private key-encrypted message digest) are sent to the recipient.

When it receives a digitally-signed document, the recipient runs the document through the hash function to create his own message digest, and decrypts the digital signature using the public key to recreate the sender's message digest. Then it's simply a matter of comparing the two digests—if they're identical, the document is authentic and hasn't been tampered with.

2 DSA on BeIA (dsasig)

On BeIA, digital signatures are used to check the authenticity and integrity of *all* documents (scripts and update packages) that are sent from the update server to the client device. Update requests sent from the client to the server don't need to be signed since these requests don't—or shouldn't—contain executable data or private information.

The default DSA implementation on BeIA centers around the **dsasig** command line program. The server uses **dsasig** to generate new DSA parameters and keys, and to encrypt outgoing documents. The client uses dsasig to verify incoming documents. These operations are invoked by **dsasig**'s first argument:

- **mkparams**: generate a new set of DSA parameters (used by the other operations).
- **mkkey**: generate a pair of keys.
- **sign**: generate a signature and append it to an out-going document.

- **verify**: strip the signature from an in-coming document and test it for authenticity.

One other **dsasig** operation—the 160-bit hash routine invoked by **sha1**—can be used by the client or the server wherever a hash value is of use. **sha1** is invoked during **sign** so it isn't needed (directly) as part of the digital signature mechanism.

2.1 Generating DSA Parameters

```
dsasig mkparams keyLength outputFile
```

Generates a new set of DSA parameters of the specified key length (1024 is recommended) and writes the parameters to *outputFile*. The client and server must have identical copies of the DSA parameters file.

Creating a new set of parameters takes a significant amount of time.

2.2 Generating Keys

```
dsasig mkkey paramsFile keyFile
```

Generates a new private/public key pair. *paramsFile* defines the length of the keys (use **dsasig mkparams** to generate this file).

The private and public keys are written to *keyFile.priv* and *keyFile.pub*, respectively (*keyFile* can be a full path name). Access to the private key file should be protected; the public key file can be made public without risk of compromising the private key. The two key files must be kept in the same directory.

When **dsasig mkkey** is run, the user is prompted for a password that's used to encrypt the private key. This same password must be used during **dsasig sign** when the key is used to create a digital signature.

2.3 Signing a Document

```
dsasig sign paramsFile keyFile documentFile
```

Creates a digital signature based on *paramsFile* and *keyFile* (which must point to the same files as in **dsasig mkkey**) and appends it to *documentFile*. During this process, the user is prompted for the private key password that was defined in **dsasig mkkey**.

If the generated signature is invalid (because the wrong private key password was used, for example) the program exits with an error (non-zero). If the program is successful, *documentFile* is ready to be sent.

2.4 Verifying a Document

```
dsasig verify paramsFile keyFile documentFile
```

Verifies the authenticity of the in-coming *documentFile* using the parameters in *paramsFile* and the public key in *keyFile.pub*. If the document is judged to be inauthentic, the program exits with an error (non-zero). If the document is authentic, *documentFile* will be left in a state that's ready to be read.

2.5 Creating a Hash Value

```
dsasig sha1 documentFile
```

Calculates and prints (to standard output) a 160-bit **SHA1** value for *documentFile*. You never need to invoke this operation directly when you're creating a digital signature; it's invoked for you as part of the signing operation (**dsasig sign**). However, you can use this operation wherever a 160-bit hash value is of use (such as in generating a value for the **CUR_IDENT** variable).

3 Division of Labor

When you're setting up your update server and creating new clients, the process should work like this:

- `dsasig mkparams` is run on the server. The parameter file that's generated is then copied to all client machines.
- `dsasig mkkey` is run on the server. The `key.priv` file is kept on the server (only!) and `key.pub` is copied to all client machines.
- When the server wants to send a document to the client, it runs `dsasig sign` on the document. Note that the same parameters and keys can be used to sign any number of documents.
- When the client receives a document from the server, it runs `dsasig verify`—and throws the document away if the function doesn't return 0.

If the private key is compromised—if the server suspects that someone has broken the private key and is sending bogus update packages—the server will have to regenerate a new key pair, and send the new public key to its clients. Of course, the clients will have to use the old key to in order to get the new one; coordinating all of this is the server's responsibility. It's anticipated that if a server wants to update the public key, it will have to be moved to a new URL, and the old URL maintained until all clients have been updated—which may be difficult to determine.

4 Default DSA Files

BeIA includes a set of default DSA parameters and keys. A server can use these defaults when it's being tested and debugged. Under no circumstances should a server use the defaults as part of the final, shipping product.

The following files are needed by the updater for signatures to work:

- DSA Parameters (`/etc/update/keys/params`): DSA parameters used by all keys
- Public Key (`/etc/update/keys/master.pub`): This key is used to verify all update scripts and packages.
- Private Key (`/etc/update/keys/master.priv` on the server only): This key is used to sign documents.

The password for the sample private key is "B80ACE4C3F8F0D".

BeIA Miscellany

The Microshell

The **Microshell** is a lightweight shell that's used to interpret commands and execute scripts. It's similar to, although less powerful than, other UNIX-type command shells such as **sh**, **cs**h, and **bash**. **Microshell** scripts are used extensively by the BeIA update mechanism.

This document provides the basic rules of **Microshell** syntax and grammar, and lists the shell's built-in commands.

1 Script Syntax and Grammar

- Microshell scripts must start with the line `#!/bin/sh`.
- Each line is an entire, separate expression (or part of an `if` expression, as explained later).
- A line can contain up to 4096 characters.
- Lines starting with `#` are comments.
- Leading whitespace is ignored (indentation isn't significant).
- Environment variables are referenced as `$VAR` (`$PATH`, `$SCRIPT_URL`, etc.)
- Arguments are referenced as `$N` (`$1`, `$2`, `$3`, and so on).
- `$?` holds the return value from the previously invoked command.
- `\n`, `\t`, `\r` are newline, tab, and carriage return characters, respectively.
- Use backslash to “uninterpret” a character (e.g. `\$` or `\\`). protect an entire word by putting it in double quotes.
- You can create branching logical expressions through `if/else/fi` and `while/wend` blocks. Each statement must be on its own line:

```
if expression
    expression
else
    expression
fi
while expression
    expression
wend
```

- `if` and `while` blocks can be nested to a depth of 128.
- When evaluating the commands in an expression, **Microshell** first looks through its list of built-in command (below), then it looks in the search path (`$PATH`). If you want to refer to a specific command file, use the pathname to the command.

2 Built-in Commands

The following commands are built into the **Microshell**. These commands are safe to use after a major update.

2.1 Environment Variables

`env`

Print all the environment variables to standard output.

`read var`

Set the environment variable *var* to the value on the next line read from standard input.

`setenv var value`

Set the environment variable *var* to *value*

2.2 File System Navigation and Access

All pathname arguments can be relative or absolute paths. Relative paths are appended to the current working directory.

`cat path`

Print the contents of the file *path* to standard out.

`cd path`

Change the current working directory to *path*.

`chmod mode path`

Change the access mode for *path*.

`filecat path string`

Writes *string* to the end of the file given by *path*. If *string* contains whitespace, it must be quoted; `\n` adds a newline and `\t` adds a tab. For example:

```
filecat myFile "Column A\tColumn B\nValue one\tValue two\n"
```

`ln -s pathTo [pathFrom]`

Creates a symbolic link (*pathFrom*) to *pathTo*. If *pathFrom* isn't supplied, the link is placed in the current working directory and given the same leaf name as *pathTo*.

`ls [-l] [path]`

List the names of the files in *path*. If no *path*, the current working directory is used. `-l` creates a detailed listing.

`mkdir path`

Create a new directory at *path*.

`mountcfs directoryPath device`

Mount *device* as a compressed file system, and point *directoryPath* at it. The directory is created if it doesn't already exist.

`mv sourcePath destPath`

Move the *sourcePath* file or directory to *destPath*.

`pwd`

Print the current working directory.

`rm path1 [path2 path3 ...]`

Remove the file(s) identified by *path*[*N*].

`rmdir path`

Remove the directory identified by *path*.

2.3 System Settings

`getdialupsettings path`

Read the user name, user password, and dial-up phone number from *path*, and stuff them into the environment variables `R_USER`, `R_PASSWD`, and `R_PHONE`, respectively.

`put_bios_date var`

Set *var* to hold the BIOS release date string.

`put_bios_vendor var`

Set *var* to hold the BIOS vendor string.

`put_bios_version var`

Set *var* to hold the BIOS version string.

`putserialnumber var`

Set *var* to hold the system serial number.

2.4 System Execution

`addfirstpartition`

Mount the primary partition if it isn't already mounted. Returns 0 on success and non-0 on failure.

`reboot`

Kill all processes and restart the device. (You may want to call `sync` before rebooting—it isn't called for you.)

`rescan [component]`

Republish the list of drivers for the named component. If component is excluded, all drivers are republished.

`sync`

Synchronize the file system by writing all pending file changes to the storage device. When this command returns, the file system has been synced.

2.5 Process Execution

`exit [value]`

Exit this shell (or script) and return *value*.

`kill signal pid | name`

Send a signal to the thread designated by *process id* or by *name*.

`run path`

Execute the file identified by *path*. The file can be an executable binary file, or a **Microshell** script. The file is executed in its own context; in this sense, `run` is similar to `sh -c` (see “**3.1** Script Interpreter” for details).

`sleep seconds`

Pause this process for a number of seconds.

`source path`

Execute the script identified by *path*.

`spawn path`

Execute the file identified by *path* as a background process.

`waitfor name`

Pause until the named thread appears.

2.6 Boolean Logic

`empty var`

Return `true` if `strlen(var)` is 0.

`eq var value`

Return `true` if `var` is equivalent to `value` (both values taken as strings).

`false`

Return non-0.

`isneg var`

Returns `true` if `var` is less than 0.

`neq var value`

Return `true` if `var` is not equivalent to `value`.

`not expr`

Invert the boolean value of `expr`.

`true`

Return 0.

2.7 File Existence

`exists path`

`true` if `path` exists.

`isfile path`

`true` if `path` is a plain file.

`isdir path`

`true` if `path` is a directory.

2.8 Miscellaneous

`echo args`

Echo `args` to standard output.

`help`

print this list

3 External Commands

Listed below are the programs that you can run from a shell or script, but that aren't built into Microshell itself. You mustn't use these programs after a major update; you have to reboot the device first.

Unless otherwise noted, all of the external commands are located in `/bin`.

3.1 Script Interpreter

`sh [-c] scriptFile`

Execute *scriptFile* as a Microshell script. Note that the file, when executed with `sh`, needn't start with the `'#!/bin/sh'` line

`-c` means to setup a separate context for the execution. This means that:

- In a separate context, environment variables that are set within *scriptFile* will not affect the same variables in the caller.
- If *scriptFile* uses `exit` to terminate, the `-c` switch prevents the exit from terminating the caller.

When you're executing a script from within another script, you almost certainly want to use the `-c` switch.

If you want the environment variables in *scriptFile* to affect the caller, you should `source` the file rather than execute it with `sh`.

If you need to execute a script after a major update, use the built-in `run` command.

3.2 Communication with the Server and other Apps

- `wget`: Used to retrieve files via HTTP or FTP over the Internet.
- `sndcmd`: Sends messages to others apps and waits for reply.

3.3 RAM Disk Management

- `setvolumesize`: Allocates and frees memory needed by the RAM disk.
- `mkcfs`: Initializes the RAM disk to contain the BeIA compressed file system.
- `/dev/disk/virtual/ram/0`: The RAM disk device driver.
- **`mount`** and **`unmount`**: Mount and unmount the RAM disk.

3.4 System Checks

`nut mode [args]`

Check some aspect of the system, as determined by *mode*:

- `check`. Return `true` if the system is awake, and `false` otherwise.
- `toggle`. Wake the system if it's asleep (return `false`) ; put it to sleep if it's awake (return `true`).
- `sleep device`. Put *device* to sleep.
- `wakeup device`. Wake *device* up.
- `ioctl code value device`. Invoke the *ioctl code*, with the integer argument *value*, on *device*.
- `port command portname`. Invoke *command* on the port given by *portname*.

3.5 File Decryption and Verification

- `unzip`: File integrity check and decompression.
- `dsasig`: DSA digital signature app.

