Inside OTA Packages

The system builds the updater binary from **`bootable/recovery/updater`** and uses it in an OTA package.

The package itself is a .zip file (**`ota_update.zip`**, **`incremental_ota_update.zip`**) that contains the executable binary **`META-INF/com/google/android/update-binary`** .

Updater contains several builtin functions and an interpreter for an extensible scripting language (**edify**) that supports commands for typical update-related tasks. Updater looks in the package .zip file for a script in the file **`META-INF/com/google/android/updater-script`**.

Note: Using the edify script and/or builtin functions is not a common activity, but can be helpful if you need to debug the update file.

Edify syntax

An edify script is a single expression in which all values are strings. Empty strings are *false* in a Boolean context and all other strings are *true*. Edify supports the following operators (with the usual meanings):

```
(expr )
 expr + expr  # string concatenation, not integer addition
 expr == expr
 expr != expr
 expr && expr
 expr || expr
 ! expr
 if expr then expr endif
 if expr then expr else expr endif
 function_name(expr, expr,...)
 expr; expr
```

Any string of the characters *a-z, A-Z, 0-9, _, :, /,* . that isn't a reserved word is considered a string literal. (Reserved words are **if else** then **endif.**) String literals may also appear in double-quotes; this is how to create values with whitespace and other characters not in the above set. \n, \t, \", and \\ serve as escapes within quoted strings, as does \x*##*.

The && and || operators are short-circuiting; the right side is not evaluated if the logical result is determined by the left side. The following are equivalent:

```
e1 && e2
if e1 then e2 endif
```

The ; operator is a sequence point; it means to evaluate first the left side and then the right side. Its value is the value of the right-side expression. A semicolon can also appear after an expression, so the effect simulates C-style statements:

```
prepare();
do_other_thing("argument");
finish_up();
```

Built-in functions

Most update functionality is contained in the functions available for execution by scripts. (Strictly speaking these are *macros* rather than *functions* in the Lisp sense, since they need not evaluate all of their arguments.) Unless otherwise noted, functions return **true** on success and **false** on error. If you want errors to abort execution of the script, use the **`abort()`** and/or **`assert()`** functions. The set of functions available in updater can also be extended to provide device-specific functionality (https://source.android.com/devices/tech/ota/device_code.html) .

`abort([msg])`

> Aborts execution of the script immediately, with the optional *msg*. If the user has turned on text display, *msg* appears in the recovery log and on-screen.

`assert(expr[, expr, ...])`

Evaluates each *expr* in turn. If any is false, immediately aborts execution with the message "assert failed" and the source text of the failed expression.

`apply_patch(`*`src_file`*`, `*`tgt_file`*`, `*`tgt_sha1`*`, `*`tgt_size`*`, `*`patch1_sha1`*`, `*`patch1_blob`*`, [...])`

Applies a binary patch to the *src_file* to produce the *tgt_file* . If the desired target is the same as the source, pass "-" for *tgt_file* . *tgt_sha1* and *tgt_size* are the expected final SHA1 hash and size of the target file. The remaining arguments must come in pairs: a SHA1 hash (a 40-character hex string) and a blob. The blob is the patch to be applied when the source file's current contents have the given SHA1.

The patching is done in a safe manner that guarantees the target file either has the desired SHA1 hash and size, or it is untouched—it will not be left in an unrecoverable intermediate state. If the process is interrupted during patching, the target file may be in an intermediate state; a copy exists in the cache partition so restarting the update can successfully update the file.

Special syntax is supported to treat the contents of Memory Technology Device (MTD) partitions as files, allowing patching of raw partitions such as boot. To read an MTD partition, you must know how much data you want to read since the partition does not have an end-of-file notion. You can use the string "MTD:*partition*:*size_1*:*sha1_1*:*size_2*: *sha1_2*" as a filename to read the given partition. You must specify at least one *(size, sha-1)* pair; you can specify more than one if there are multiple possibilities for what you expect to read.

`apply_patch_check(`*`filename`*`, `*`sha1`*`[, `*`sha1`*`, ...])`

Returns true if the contents of *filename* or the temporary copy in the cache partition (if present) have a SHA1 checksum equal to one of the given *sha1* values. *sha1* values are specified as 40 hex digits. This function differs from `sha1_check(read_file(`*`filename`*`), `*`sha1`*` [, ...])` in that it knows to check the cache partition copy, so `apply_patch_check()` will succeed even if the file was corrupted by an interrupted `apply_patch() update`.

`apply_patch_space(`*`bytes`*`)`

Returns true if at least *bytes* of scratch space is available for applying binary patches.

`concat(`*`expr`*`[, `*`expr`*`, ...])`

Evaluates each expression and concatenates them. The + operator is syntactic sugar for this function in the special case of two arguments (but the function form can take any number of expressions). The expressions must be strings; it can't concatenate blobs.

`file_getprop(`*`filename`*`, `*`key`*`)`

Reads the given *filename*, interprets it as a properties file (e.g. `/system/build.prop`), and returns the value of the given *key* , or the empty string if *key* is not present.

`format(`*`fs_type`*`, `*`partition_type`*`, `*`location`*`, `*`fs_size`*`, `*`mount_point`*`)`

Reformats a given partition. Supported partition types:

- fs_type="yaffs2" and partition_type="MTD". Location must be the name of the MTD partition; an empty yaffs2 filesystem is constructed there. Remaining arguments are unused.
- fs_type="ext4" and partition_type="EMMC". Location must be the device file for the partition. An empty ext4 filesystem is constructed there. If *fs_size* is zero, the filesystem takes up the entire partition. If *fs_size* is a positive number, the filesystem takes the first *fs_size* bytes of the partition. If *fs_size* is a negative number, the filesystem takes all except the last *|fs_size|* bytes of the partition.
- fs_type="f2fs" and partition_type="EMMC". Location must be the device file for the partition. *fs_size* must be a non-negative number. If *fs_size* is zero, the filesystem takes up the entire partition. If *fs_size* is a positive number, the filesystem takes the first *fs_size* bytes of the partition.
- mount_point should be the future mount point for the filesystem.

`getprop(`*`key`*`)`

Returns the value of system property *key* (or the empty string, if it's not defined). The system property values defined by the recovery partition are not necessarily the same as those of the main system. This function returns the value in recovery.

`greater_than_int(`*`a`*`, `*`b`*`)`

Returns true if and only if (iff) *a* (interpreted as an integer) is greater than *b* (interpreted as an integer).

`ifelse(`*`cond`*`, `*`e1`*`[, `*`e2`*`])`

Evaluates *cond*, and if it is true evaluates and returns the value of *e1*, otherwise it evaluates and returns *e2* (if present). The "if ... else ... then ... endif" construct is just syntactic sugar for this function.

`is_mounted(`*`mount_point`*`)`

Returns true iff there is a filesystem mounted at *mount_point*.

`is_substring(`*`needle`*`, `*`haystack`*`)`

Returns true iff *needle* is a substring of *haystack*.

`less_than_int(`*`a`*`, `*`b`*`)`

Returns true iff *a* (interpreted as an integer) is less than *b* (interpreted as an integer).

`mount(`*`fs_type`*`, `*`partition_type`*`, `*`name`*`, `*`mount_point`*`)`

Mounts a filesystem of *fs_type* at *mount_point*. *partition_type* must be one of:

- **MTD**. Name is the name of an MTD partition (e.g., system, userdata; see `/proc/mtd` on the device for a complete list).
- **EMMC.**

  Recovery does not mount any filesystems by default (except the SD card if the user is doing a manual install of a package from the SD card); your script must mount any partitions it needs to modify.

`package_extract_dir(`*`package_dir`*`, `*`dest_dir`*`)`

Extracts all files from the package underneath *package_dir* and writes them to the corresponding tree beneath *dest_dir*. Any existing files are overwritten.

`package_extract_file(`*`package_file`*`[, `*`dest_file`*`])`

Extracts a single *package_file* from the update package and writes it to *dest_file*, overwriting existing files if necessary. Without the *dest_file* argument, returns the contents of the package file as a binary blob.

`read_file(`*`filename`*`)`

Reads *filename* and returns its contents as a binary blob.

`run_program(`*`path`*`[, `*`arg`*`, ...])`

Executes the binary at *path*, passing *arg*s. Returns the program's exit status.

`set_progress(`*`frac`*`)`

Sets the position of the progress meter within the chunk defined by the most recent `show_progress()` call. *frac* must be in the range [0.0, 1.0]. The progress meter never moves backwards; attempts to make it do so are ignored.

`sha1_check(`*`blob`*`[, `*`sha1`*`])`

The *blob* argument is a blob of the type returned by `read_file()` or the one-argument form of `package_extract_file()` . With no *sha1* arguments, this function returns the SHA1 hash of the blob (as a 40-digit hex string). With one or more *sha1* arguments, this function returns the SHA1 hash if it equals one of the arguments, or the empty string if it does not equal any of them.

`show_progress(`*`frac`*`, `*`secs`*`)`

Advances the progress meter over the next *frac* of its length over the *secs* seconds (must be an integer). *secs* may be 0, in which case the meter is not advanced automatically but by use of the `set_progress()` function defined above.

`sleep(`*`secs`*`)`

Sleeps for *secs* seconds (must be an integer).

`stdout(`*`expr`*`[, `*`expr`*`, ...])`

Evaluates each expression and dumps its value to stdout. Useful for debugging.

`tune2fs(`*`device`*`[, `*`arg`*`, …])`

Adjusts tunable parameters *args* on *device*.

`ui_print([`*`text`*`, ...])`

Concatenates all *text* arguments and prints the result to the UI (where it will be visible if the user has turned on the text display).

`unmount(`*`mount_point`*`)`

Unmounts the filesystem mounted at *mount_point*.

`wipe_block_device(`*`block_dev`*`, `*`len`*`)`

Wipes the *len* bytes of the given block device *block_dev*.

`wipe_cache()`

Causes the cache partition to be wiped at the end of a successful installation.

`write_raw_image(`*`filename_or_blob`*`, `*`partition`*`)`

Writes the image in *filename_or_blob* to the MTD *partition*. *filename_or_blob* can be a string naming a local file or a blob-valued argument containing the data to write. To copy a file from the OTA package to a partition, use:
`write_raw_image(package_extract_file("zip_filename"), "partition_name");`

Note: Prior to Android 4.1, only filenames were accepted, so to accomplish this the data first had to be unzipped into a temporary local file.
