SB boot image file format

Table of Contents:

- Glossary
- Introduction
- Basic types
- Boot image format
  - Image header
  - Section table
  - DEK dictionary
  - Section boot tags
  - Section data regions
  - Image authentication code
- Encryption details
  - Encryption process
  - Decryption process
- Boot commands
  - `ROM_NOP_CMD`
  - `ROM_TAG_CMD`
  - `ROM_LOAD_CMD`
  - `ROM_FILL_CMD`
  - `ROM_JUMP_CMD`
  - `ROM_CALL_CMD`
  - `ROM_MODE_CMD`
- File format versions
- References
- Revision History

Glossary

AES-128 - Rijndael cipher with block and key sizes of 128 bits.
Block cipher - Encryption algorithm that works on blocks of N={64, 128, ...} bits.
CBC - Cipher Block Chaining, a cipher mode that uses feedback between ciphertext blocks.
CBC-MAC - A message authentication code computed with a block cipher.
Cipher block - The minimum amount of data on which a block cipher operates.
Ciphertext - Encrypted data.
DEK - Data encryption key, a one time session key used to encrypt the bulk of the boot image.
ECB - Electronic Code Book, a cipher mode with no feedback between ciphertext blocks.
Hash - Digest computation algorithm.
KEK - Key encryption key, used to encrypt a session key or DEK.
MAC - Message authentication code. Provides integrity and authentication checks.
Message digest - Unique value computed from data using a hash algorithm. Provides only an integrity check unless encrypted.
Plaintext - Unencrypted data.
Rijndael - Block cipher chosen by the US Government to replace DES. Pronounced "rain-dahl".
Session key - Encryption key generated at the time of encryption. Only ever used once.
SHA-1 - Hash algorithm that produces 160-bit message digest.

Introduction

The entire boot image format is built around the requirements of AES-128, with its minimum block size of 128 bits or 16 bytes. AES-128 is the symmetric block cipher that is used for encrypted boot images. Using its block size as the base unit throughout the image makes it much easier to accommodate encryption.

In order to support multiple executables within one image, the format has the concept of sections. Each section can contain a standalone bootable image, or may be part of a larger sequence of sections. A boot command is provided that can be used to direct the bootloader to continue from another section at runtime.

There are a number of features of this format that are not useful for all applications or methods of reading. For instance, the section table is only useful if random access to the boot image is available. While the boot tags are most useful when booting from a streaming media. The goal here is to provide a great deal of capability to the image, regardless of how it is accessed.

Basic types

Throughout this document several basic C types are used to represent cipher blocks, keys, and other important elements. The definitions for these types are shown below.

```
//! An AES-128 cipher block is 16 bytes.
typedef uint8_t cipher_block_t[16];

//! An AES-128 key is 128 bits, or 16 bytes.
typedef uint8_t aes128_key_t[16];

//! A SHA-1 digest is 160 bits, or 20 bytes.
typedef uint8_t sha1_digest_t[20];

//! Unique identifier for a section.
typedef uint32_t section_id_t;
```

Boot image format

The boot image format consists of five distinct regions. First there is a plaintext header containing basic information about the image. A section table, also plaintext, comes afterwards. It describes each of the different sections within the image. For encrypted images, a key dictionary that is used to support multiple OTP keys then follows. Next, each section has its data, which is prefixed with a tag used by the bootloader. And finally, the image terminates with an authentication code for the entire image. Figure 1 shows the basic layout of a boot image.

The image format is designed to be read from streaming media without support for random access while requiring caching of as little data as possible. However, the format also includes features that are most useful when random access to the image is possible, for example through the SDK. For example, the image ends with an authentication code computed from the entire rest of the image. This isn't particularly useful for the ROM, but can be used by host-resident utilities to verify and authenticate boot images before using them.

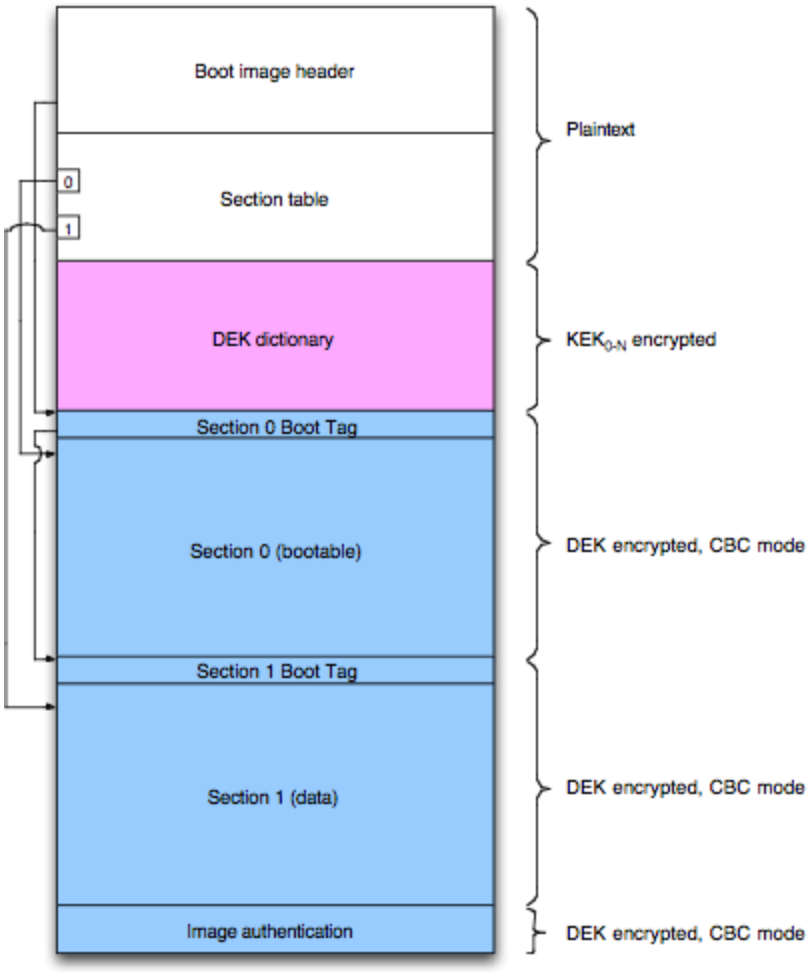

Figure 1. Boot image regions

The basic unit size of the format is that of an AES-128 cipher block, or 16 bytes. Every region in the file always starts on a cipher block boundary.

Every field within the image is formatted in little endian byte order.

Image header

The header of a boot image is always unencrypted. It provides required information about the image as a whole, as well as some useful pointers to the other regions within the image.

Image header size is always a round number of cipher blocks. Any padding bytes that are necessary to fill out the structure will always be set to random values. No padding is necessary if the header completely fills the last cipher block it occupies. The section table dictionary will follow immediately in that case.

The C structure definition for the image header follows:

```
struct boot_image_header_t
{
    union
    {
        sha1_digest_t m_digest;
        struct
        {
            cipher_block_t m_iv;
            uint8_t m_extra[4];
        };
    };
    uint8_t m_signature[4];
    uint8_t m_majorVersion;
    uint8_t m_minorVersion;
    uint16_t m_flags;
    uint32_t m_imageBlocks;
    uint32_t m_firstBootTagBlock;
    section_id_t m_firstBootableSectionID;
    uint16_t m_keyCount;
    uint16_t m_keyDictionaryBlock;
    uint16_t m_headerBlocks;
    uint16_t m_sectionCount;
    uint16_t m_sectionHeaderSize;
    uint8_t m_padding0[2];
    uint8_t m_signature2[4];
    uint64_t m_timestamp;
    version_t m_productVersion;
    version_t m_componentVersion;
    uint16_t m_driveTag;
    uint8_t m_padding1[6];
};
```

The fields of `boot_image_header_t` have the descriptions given in Table 1. The flags defined for the `m_flags` field are shown in Table 2.

Field	Description
`m_digest`	SHA-1 digest of all fields of the header prior to this one. The first 16 bytes (of 20 total) also act as the initialization vector for CBC-encrypted regions.
`m_signature`	Always has the value 'STMP'.
`m_majorVersion`	Major version of the boot image format, currently 1.
`m_minorVersion`	Minor version of the boot image format, currently 1 or 2.
`m_flags`	Flags associated with the entire image.
`m_imageBlocks`	Size of the entire image in blocks.
`m_firstBootTagBlock`	Offset from start of file to the cipher block containing the first boot tag.
`m_firstBootableSectionID`	Unique identifier of the section to start booting from.
`m_keyCount`	Number of entries in the DEK dictionary.
`m_keyDictionaryBlock`	Starting block number, from the beginning of the image, for the DEK dictionary.
`m_headerBlocks`	Size of the entire image header in blocks.
`m_sectionCount`	Number of sections.
`m_sectionHeaderSize`	Size in blocks of a section header.
`m_padding0`	Two bytes of padding to align `m_signature2` to a word boundary. Set to random values.
`m_signature2`	Always set to 'sgtl'. This second signature is only present in files with a minor version greater or equal to 1.
`m_timestamp`	Timestamp in microseconds size 1-1-2000 00:00 when the image was created.
`m_productVersion`	Product version.

m_componentVersion	Component version.
m_driveTag	Identifier for the disk drive or partition containing this image.
m_padding1	Eight bytes of padding to fill out the cipher block. Set to random values.

Table 1. Image header fields.

Constant	Bit	Description
ROM_DISPLAY_PROGRESS	0	Turn on progress reports of executed commands.
ROM_VERBOSE_PROGRESS	1	Prints extra information in reports about executed commands. Applies only if ROM_DISPLAY_PROGRESS is also enabled.

Table 2. Boot image flags.

The m_majorVersion and m_minorVersion fields describe the version of the boot image format, not the version of the ROM (as in previous boot image formats). The major version field is currently 1. Any time this field is changed, the format is no longer backwards compatible with previous versions and a new bootloader is expected to be required. The minor version field should be incremented for any format changes that are backwards compatible with previous bootloader versions. For instance, adding a new field to the end of the image header is backwards compatible due to the presence of the m_headerBlocks field. So in this case only m_minorVersion should be incremented. But if the image header fields were reordered the current bootloader could no longer read the image and the m_majorVersion field must be incremented. See the file format versions table at the end of this document for more version details.

If the value of the m_keyCount is zero, then the boot image in fully unencrypted. The image is always encrypted if there is at least one key in the dictionary.

The SHA-1 digest of the header provides a basic integrity check for unencrypted images. It does not provide any extra security because it can simply be updated along with any changes made to the header.

Throughout the rest of the file, any time something is encrypted using CBC mode the first 16 bytes of the m_digest field are used as the initialization vector. The digest is random enough because the header differs for all boot images. The m_timestamp field, in addition to its nominal purpose, serves to guarantee that the plaintext header will be different for every boot image that is created. In addition to improving the randomness of the header digest, this is important because the header is authenticated with the OTP key.

In order to decrease bootloading time, the ROM can optionally use the m_firstBootTagBlock and m_firstBootableSectionID fields to jump to the first bootable section without having to parse the section header table. The m_firstBootTagBlock field contains the starting cipher block offset from the beginning of the image to the first boot tag. This field must not be 0 unless the image contains no sections (in which case it is invalid). From the first boot tag the ROM can search for the section whose unique ID matches m_firstBootableSectionID.

The m_keyDictionaryBlock field is also used to help the boot ROM speed its processing of the header. This value could be calculated from other header fields, but having it pre-calculated allows the ROM code to keep track of fewer header fields.

The m_productVersion and m_componentVersion fields contain version values that describe the firmware within the boot image. These fields use the following C structure defintion:

```
struct version_t
{
    uint16_t m_major;
    uint16_t m_pad0;
    uint16_t m_minor;
    uint16_t m_pad1;
    uint16_t m_revision;
    uint16_t m_pad2;
};
```

Within each of the major, minor, and revision fields of the version_t structure, the version number is in right-aligned BCD format. The default value for both versions is 999.999.999.

The m_padding0 and m_padding1 fields are used to align other fields and round out the structure size to an even cipher block. These bytes are set to random values when the image is created to add to the “whiteness” of the header for cryptographic purposes.

Section table

The section table is basically an index of the starting block and length for each section within a boot image. It also contains flags that apply solely to that section.

The table is always unencrypted and comes immediately after the plaintext image header and before the DEK dictionary, if the dictionary is present.

The C type definition for the section table and its entries is as follows:

```
struct section_header_t
{
    section_id_t m_identifier;
    uint32_t m_offset;
    uint32_t m_length;
    uint32_t m_flags;
};

struct section_table_t
{
    section_info_t m_sections[1];
};
```

The fields of `section_header_t` are described in Table 3. The flags defined for the `m_flags` field of `section_header_t` are as shown in Table 4.

Field	Description
`m_identifier`	Unique 32-bit identifier for this section.
`m_offset`	The starting cipher block for this section's data from the beginning of the image.
`m_length`	The length of the section data in cipher blocks.
`m_flags`	Flags that apply to the entire section.

Table 3. Section header fields.

Constant	Bit	Description
`ROM_SECTION_BOOTABLE`	0	The section is bootable and contains a sequence of bootloader commands.
`ROM_SECTION_CLEARTEXT`	1	The section is unencrypted. Applies only if the rest of the boot image is encrypted.

Table 4. Section flags.

The length of each entry in the section table comes from the `m_sectionHeaderSize` field of the image header. Entries are always a round number of cipher blocks long, being padded if necessary. And all entries in the table are the same length. In version 1 of the file format, section table entries are a single cipher block long and have no padding.

The total number of sections, and thus the number of entries in the section table, is given in the `m_sectionCount` field of the image header. This should always be at least 1 for a valid bootable image. If it is 0, then the image contains no boot commands and is considered invalid In addition, there must be at least one section with the `ROM_SECTION_BOOTABLE` flag set for an image to be valid.

The size of the section table is either `(header.m_sectionCount * header.m_sectionHeaderSize)` cipher blocks or `(header.m_sectionCount * header.m_sectionHeaderSize * 16)` bytes.

DEK dictionary

The key dictionary always follows the image header in the next cipher block in encrypted images. Unencrypted images do not have a DEK dictionary.

Its purpose is to allow a single boot image to work with any number of customer OTP keys. This is accomplished by generating a new key, the data encryption key (DEK), every time a boot image is generated. Except for this dictionary, the rest of the image is encrypted with this DEK. The dictionary is used to map from any given OTP key to the DEK in a secure manner, by encrypting the DEK with each OTP key to be supported. Thus the DEK is never available without a valid OTP key.

Each entry in the dictionary is comprised of two pieces of data: a message authentication code (MAC) and the encrypted DEK itself. The MAC acts as a check code, a known value that can be searched for. Otherwise there is no way to tell a valid decryption of the DEK from garbage.

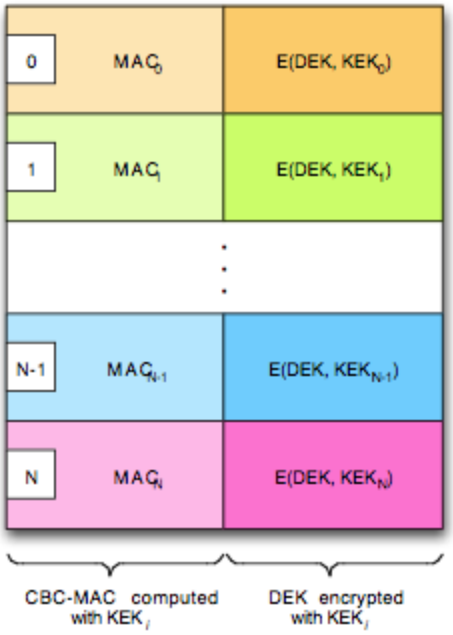

Figure 2. DEK dictionary

The message authentication code or MAC is generated using a technique called CBC-MAC. The header of the boot image and the section table, which are both always plaintext, are encrypted in CBC mode using the KEK for the given dictionary entry. The initialization vector for this encryption is always zero [2]. Only the last cipher block is retained throughout this process. The authentication code is the last cipher block.

The C type definition for the DEK dictionary is shown here:

```
struct dek_dictionary_entry_t
{
    cipher_block_t m_mac;
    aes128_key_t m_dek;
};

struct dek_dictionary_t
{
    dek_dictionary_entry_t m_entries[1];
};
```

The `m_dek` field in each entry is encrypted using the KEK in CBC mode using the IV from the image header. The CBC-MAC result, held in the `m_mac` field, is not encrypted. This is not necessary since it is generated from the secret OTP key.

The number of entries in the dictionary is determined from the `m_keyCount` field of the image header. The dictionary size will always be `header.m_keyCount * 2` cipher blocks, or `header.m_keyCount * 32` bytes. If `m_keyCount` is zero, then the DEK dictionary occupies no cipher blocks in the image and the entire image is unencrypted.

The only realistic limit on the size of the dictionary is boot time. Obviously, the more dictionary entries, the longer it will take to boot the device. But at least the algorithm to search for the DEK should be O(n).

Section boot tags

Before each section data region there is a special tag cipher block that describes the following section. These tags are called boot tags because the boot ROM uses them to search for sections without having to maintain a copy of the entire section table in memory or re-read portions of the image from storage. Boot tags are always paired with a section data region—there is never one without the other. Another way to think of boot tags is as a section header local to the section contents.

The actual structure of a boot tag is that of the `ROM_TAG_CMD` bootloader command. Reusing the boot command structure for the boot tag simplifies the ROM code somewhat. The tag command contains duplicates of some of the fields from the section table entry for the section data region with which it is paired. The most important of these are the section identifier and the section length in blocks.

Because there is no padding allowed between sections, the section length effectively points to the next boot tag. This allows the boot ROM to easily search for section data regions by comparing identifiers and following the chain formed by boot tags. The last boot tag in an image will always have its `ROM_LAST_TAG` flag set to help the ROM know at what point to stop searching.

Section data regions

There are two types of section data regions. The first is a bootable region that contains a sequence of boot commands. Second is any non-bootable region that can contain arbitrary data that is not processed by the boot ROM. These regions may contain resources or other data to be used by customer applications.

The contents of a bootable region are simply a number of bootloader commands sequenced one after another. Bootable sections must always begin with a `ROM_TAG_CMD` bootloader command. See section 9 for more details about the structure of bootloader commands and the details of individual commands.

The ROM will simply skip over any non-bootable sections.

Section data regions must be ordered in the same sequence as they appear in the section table. That is, the data region for section number 1 must come after the data region for section number 0 within the boot image. Also, there must be no pad blocks inserted before or after section data regions, even though the format implicitly supports this by the use of cipher block pointers. These restrictions are intended to make the processing of the boot image by the ROM easier.

Image authentication code

Every boot image ends with an authentication code that is computed from the entire contents of the image (excluding the authentication code, of course). This code is a SHA-1 digest encrypted with the DEK using CBC mode. The authentication code consumes two cipher blocks in the image, with 3 words of padding added after the last word of the SHA-1 digest (because a SHA-1 digest is 160 bits and cipher blocks are 128 bits). The padding bytes are set to random values.

The digest is computed from the following components, in this order: plaintext header, plaintext section table, DEK dictionary, plaintext section contents.

Hash algorithms do not themselves provide authentication, only providing an integrity check. However, if the digest is encrypted with a secret key then it can be used to provide authentication.

In an unencrypted boot image, the image authentication code is of course also unencrypted. The code no longer provides authentication, but does still provide an integrity check over the entire image.

The authentication code will always be located starting at cipher block number `(header.m_imageBlocks - 2)`.

Encryption details

Encryption process

The process of encryption takes place solely within the elftosb utility as it converts ELF or S-record binaries into a boot image. The sequence below shows the steps that elftosb will take to encrypt an image.

1. Build plaintext image header
    a. Generate IV
    b. Compute SHA-1 over image header
2. Generate plaintext section table
3. Generate DEK
4. For every KEK:
    a. Read KEK key file
    b. Compute CBC-MAC over plaintext image header with IV=0
    c. Encrypt DEK with KEK in CBC mode with IV from header
    d. Combine unencrypted CBC-MAC and encrypted DEK into dictionary entry
5. For every section:
    a. Generate a `ROM_TAG_CMD` as the boot tag for this section
    b. Encrypt the boot tag using CBC mode with IV from header
    c. Generate plaintext section contents
    d. Encrypt the section contents using CBC mode with IV from header
6. Compute SHA-1 digest of image
7. Encrypt image digest using CBC mode with IV from header

Decryption process

The decryption process takes place within the ROM. In addition, there will be a host utility program that can decrypt a boot image for testing purposes.

1. Read first cipher block of image header. The `m_keyCount` field in the first cipher block tells if the image is encrypted or not. If the image is encrypted, `m_keyCount` wil be non-zero.
2. As the image header is read, compute the CBC-MAC over it using the OTP key.
3. For each entry in the DEK dictionary
    a. Does `m_mac` field match the computed CBC-MAC? If not, skip to next entry.
    b. If `m_mac` matches, decrypt DEK using OTP key and exit loop.
4. For each of the section table and any section data regions that is are to be read:
    a. Decrypt the region using the DEK in CBC mode with IV from header

Boot commands

A bootable section in an image contains a sequence of boot commands and any data required by those commands. The commands are processed in a linear sequence starting with the first. Each boot command occupies a single cipher block, plus any cipher blocks required for data associated with that command. The C structure definition for a boot command is as follows:

```
struct boot_command_t
{
    uint8_t m_checksum;
    uint8_t m_tag;
    uint16_t m_flags;
    uint32_t m_address;
    uint32_t m_count;
    uint32_t m_data;
};
```

The commands described in this section are chosen to allow the greatest flexibility in construction of boot images using the fewest number of command types. For the most part, the individual fields of `boot_command_t` vary in exact meaning between each command and are described below.

Because the `m_checksum` field is always calculated in the same way for every command, it deserves special mention here. This field provides a cheap and easy way to verify that the cipher block contains a valid bootloader command. While 8 bits is certainly not enough to act as a solid defense against either corruption or intended changes, it is far better than nothing.

The checksum is computed in the following manner:

```
boot_command_t bootCommand;
uint8_t * bytes = reinterpret_cast<uint8_t *>(&bootCommand);
uint8_t checksum = 0x5a;
int i;

// Unroll this loop for better optimization.
for (i = 1; i < sizeof(bootCommand); ++i)
{
  checksum += bytes[i];
}
```

Note that the checksum is computed only over bytes 1 through 15 of the `boot_command_t` structure for each boot command. Put another way, any additional cipher blocks of data following a command are not included in the checksum. Also note that the initial checksum value is 0x5a instead of zero. This is to prevent an all-zero command from also having a zero checksum.

The `m_tag` fields of each boot command contains a unique byte value that identifies which command it structure describes. The list of boot command tag values is shown in Table 5.

Command Tag Value	Command Tag Mnemonic
0x00	ROM_NOP_CMD
0x01	ROM_TAG_CMD
0x02	ROM_LOAD_CMD
0x03	ROM_FILL_CMD
0x04	ROM_JUMP_CMD

0x05	ROM_CALL_CMD
0x06	ROM_MODE_CMD

Table 5. Boot command tag values.

Any values of m_tag that do not match those listed in Table 5 are invalid. If encountered, the bootloader will stop and report an error.

ROM_NOP_CMD

This command is a no-operation. The bootload simply skips over it. All fields except the m_tag fields are ignored by the bootloader and can contain any value. However, until other uses are documented for these fields they should contain the values presented in Table 6.

Field	Description
m_checksum	Simple checksum, which comes to 0x5a when all other fields are zeroes.
m_tag	0x00 or ROM_NOP_CMD
m_flags	0
m_address	0
m_count	0
m_data	0

Table 6. No-op command fields.

ROM_TAG_CMD

Used as a kind of “key frame” that describes a section. Another way to think of it is as a local section header. It contains most of the fields from the section’s entry in the section table.

This command in not expected to appear within the command stream in a bootable section, and the bootloader will just ignore it if it is present. The purpose of this command definition is to describe the structure of the boot tag cipher block. Boot tags use the exact same structure as boot commands to make the bootloader’s job that much easier.

Field	Description
m_checksum	Simple checksum of the other fields of boot_command_t.
m_tag	0x01 or ROM_TAG_CMD
m_flags	Bit 0: ROM_LAST_TAG
m_address	The m_tag field from the section header.
m_count	The number of cipher blocks that the data for this section occupies. This also happens to be the number of cipher blocks until the next boot tag (except for the last one).
m_data	The m_flags field from the section header.

Table 7. Hint Tag command fields.

The flag ROM_LAST_TAG must be set on the boot tag for the last section. It is used to indicate that there are no more boot tags in the image.

Note that unlike the section table entries, the ROM_TAG_CMD cannot grow in size to add more fields.

ROM_LOAD_CMD

This command is followed by an arbitrary number of cipher blocks that contain data to be loaded into memory starting at the location specified by the m_address field of boot_command_t. The m_count field contains the number of bytes to be loaded to this location in memory.

Field	Description
m_checksum	Simple checksum of the other fields of boot_command_t.

m_tag	0x02 or ROM_LOAD_CMD
m_flags	Bit 0: DCD load (MX28 only)
m_address	Memory address to which the data will be stored.
m_count	Number of bytes to load. This is also the number of valid bytes in the data cipher blocks following this command.
m_data	CRC-32 over the data to be loaded.

Table 8. Load command fields.

The number of cipher blocks following the command is `(m_count + 15) / 16`. This means that there may be up to 15 bytes of padding in the last data cipher block. Pad bytes will always be filled with random data. See Figure 3 for an example of how the cipher blocks are arranged for a load command with a data size of 18 bytes.

Figure 3. Load command cipher blocks.

There are no restrictions on alignment for the `m_address` or `m_count` fields. It is up to the ROM implementation to decide how to best optimize loading of data. Thus there is no guarantee on the order in which the data will be written to memory.

The `m_data` field contains a CRC-32 value computed over the data following the command header block. Any pad bytes in the last data cipher block are included in the CRC-32 calculation.

If bit 0 of the `m_flags` fields is set, the MX28 ROM will interpret the command as a DCD (Device Configuration Descriptor) load. In this case, the data being loaded is a HAB4 DCD, described in the HAB4 documentation. After the data is loaded into memory at the target address, the HAB library in ROM executes the DCD.

ROM_FILL_CMD

This bootloader command is used to fill regions of memory with a bit pattern. The fill pattern is always a full 32 bits wide, but a byte aligned fill length and target address are fully supported.

Field	Description
m_checksum	Simple checksum of the other fields of boot_command_t.
m_tag	0x03 or ROM_FILL_CMD
m_flags	Always 0.
m_address	The starting memory address to which the fill pattern will be written.
m_count	Number of bytes to fill.
m_data	The fill pattern. Always replicated across the word regardless of the pattern size.

Table 9. Fill command fields.

The fill pattern, regardless of its actual size, must be spread across the entire `m_data` field. So a pattern that is a byte wide must be replicated four times across `m_data`, and twice for half-word patterns.

When filling, the pattern is adjusted so that the most significant byte is aligned with the first byte to be filled. Figure 4 demonstrates what this looks like.

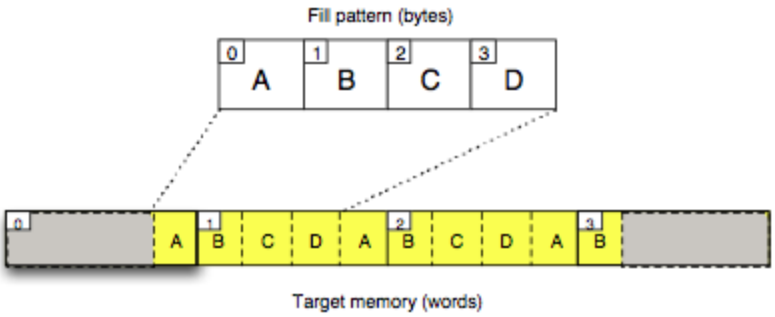

Figure 4. Fill pattern alignment.

Note that this command is guaranteed to use word writes between any unaligned ragged edges. This enables the use of the fill command as a word poke operation to write to registers.

ROM_JUMP_CMD

When the bootloader encounters this command, bootloading stops and CPU control is transferred to the function residing at `m_address`. The contents of `m_data` are passed as a single argument to the function. The ROM does not expect to regain control of the CPU after this command is executed.

Field	Description
`m_checksum`	Simple checksum of the other fields of `boot_command_t`.
`m_tag`	0x04 or `ROM_JUMP_CMD`
`m_flags`	Bit 0: HAB jump (MX28 only)
`m_address`	Address that the PC will be set to.
`m_count`	0
`m_data`	Argument to pass to the entry point in R0.

Table 10. Jump command fields.

The prototype of the function executed by `ROM_JUMP_CMD` is as follows:

```
void jump_function( uint32_t arg );
```

Note the void result. If the function does return, the bootloader fails with error ERROR_ROM_LDR_JUMP_RETURNED.

For the MX28 only, if bit 0 of `m_flags` is set then `m_address` contains the address of an IVT table as defined by HAB4 instead of an actual entry point address. The IVT then contains the entry point address, as well as other information such as code signing details. See the HAB4 documentation for details about the IVT structure.

ROM_CALL_CMD

Like the `ROM_JUMP_CMD`, the `ROM_CALL_CMD` also invokes a function residing at `m_address` and passes the value `m_data` as the its argument. The first and most important difference between the two commands is a semantic one, in that the function invoked by `ROM_CALL_CMD` is expected to relinquish control and return to the ROM to allow bootloading to continue. In addition, this command adds a second optional argument to the function prototype. This second argument in combination with the function's return value can be used to tell the bootloader to jump to another section in the current boot image or prepare for an entirely new boot image.

Field	Description
`m_checksum`	Simple checksum of the other fields of `boot_command_t`.
`m_tag`	0x05 or `ROM_CALL_CMD`
`m_flags`	Bit 0: HAB call (MX28 only)
`m_address`	Address of the function to call.

m_count	0
m_data	Argument to pass to the function in R0.

Table 11. Call command fields.

The full prototype of the function executed by ROM_CALL_CMD is as follows:

```
int call_function( uint32_t arg, uint32_t * resultId );
```

The value of the m_data field is passed in the first argument to the function. The second argument is a pointer to a word that the function can modify to return a section or image ID.

The return value determines what happens when call_function() returns and whether *resultId is examined. Possible return values are shown in Table 12.

Return value	Action
< 0	Negative values are errors.
0=SUCCESS	Success. Continue executing commands in the current section.
1=ROM_BOOT_SECTION_ID	Switch to the section with the ID of *resultId.
2=ROM_BOOT_IMAGE_ID	Restart bootloader in expectance of a new boot image. The *resultId value is passed to the driver when it's initialization function is called again.
> 2	Ignored, same as SUCCESS.

Table 12. Call command return values.

The two positive return codes have special meanings. If the function returns ROM_BOOT_SECTION_ID then the bootloader begins searching for a section of the current image that has an ID equal to the value returned through resultId. This section must follow the current section in the image or it will not be found as the bootloader only searches forward through the image. If no section with a matching unique identifier is found the boot fails with an error.

If the function return ROM_BOOT_IMAGE_ID then the bootloader prepares itself to start reading an entirely new boot image file and signals this to the current boot driver by calling its initialization function again. The value returned through resultId is the ID of a boot image; the meaning of the image ID is specific to each boot driver, and not all boot drivers support switching to new image files. The behaviour is undefined when switching boot images with a driver that does not support this functionality.

Only if the return value is ROM_BOOT_SECTION_ID or ROM_BOOT_IMAGE_ID is the value pointed to by resultId examined when the bootloader resumes execution. Because of this and how the ARM ABI works, functions that do not expect to return ROM_BOOT_SECTION_ID or ROM_BOOT_IMAGE_ID can shorten their prototype to the following:

```
int call_function_short( uint32_t arg );
```

Just as with ROM_JUMP_CMD, if bit 0 of m_flags is set on the MX28 then the command expects a HAB4 IVT address in m_address instead of an entry point address.

ROM_MODE_CMD

A boot image will use this command when there is a need to switch boot modes. The new boot mode is passed in the m_data field of the boot_command_t structure. This possible values for this field are the same values read from the boot mode pins or boot mode eFuse bits.

Field	Description
m_checksum	Simple checksum of the other fields of boot_command_t.
m_tag	0x06 or ROM_MODE_CMD
m_flags	0
m_address	0
m_count	0

`m_data`	New boot mode value. This value depends on the boot modes supported by the chip.

Table 13. Mode command fields.

When the bootloader encounters this command, it will effectively reset its state machine and reinstantiate itself to load from the new boot device. For instance, a boot image loaded from I2C EEPROM can force the bootloader to switch to NAND. The new boot device must be formatted as if the original boot mode were set to that device to begin with, and it must contain a complete boot image.

File format versions

Versions are listed as Major.minor.

Version	Description
1.0	Original version.
1.1	Added `m_signature2` and `m_driveTag` to file header.
1.2	File makes use of HAB4 command variants (for i.MX28).

Table 14. File format versions.

References

[1] B. Schneier, *Applied Cryptography Second Edition: protocols, algorithms, and source code in C*, New York, NY: John Wiley & Sons, ISBN: 0-471-11709-9.
[2] M. Bellare, J. Kilian, and P. Rogaway, "The security of the cipher block chaining message authentication code", *Journal of Computer and System Sciences*, Vol. 61, No. 3, Dec 2000, pp. 362-399.

Revision History

Revision	Date	Author	Description
0.0	03/24/06	creed	Created document
0.1	04/11/06	creed	First complete draft
0.9	08/10/06	creed	Update to match shipping STMP3700 ROM
0.10	01/20/11	creed	Cleaning up and importing to wiki