Boot Specification

v1.0

INNO EMBE PH

Contents

[1. Introduction to Boot Specification 3](#_Toc163478449)

[1.1. Background 3](#_Toc163478450)

[1.2. Target Architecture 3](#_Toc163478451)

[1.3. Boot Sources 3](#_Toc163478452)

[1.4. Other Considerations 3](#_Toc163478453)

[2. The Boot Specification 5](#_Toc163478454)

[2.1. ROM image format 5](#_Toc163478455)

[2.1.1. ROM Header 5](#_Toc163478456)

[2.1.2. Boot Info 7](#_Toc163478457)

[2.1.3. Version Info 8](#_Toc163478458)

[2.1.4. Product Info 9](#_Toc163478459)

[2.1.5. DFU Info 11](#_Toc163478460)

[2.1.6. Parameters Info 11](#_Toc163478461)

[2.2. Parameter Passing 12](#_Toc163478462)

[2.3. Device State 13](#_Toc163478463)

[3. Implementation Details 14](#_Toc163478464)

[3.1. Toolchain Support 14](#_Toc163478465)

[3.2. Implementation using the C Language 14](#_Toc163478466)

[3.2.1. Annotations 14](#_Toc163478467)

[3.2.2. Types and other Constants 14](#_Toc163478468)

[3.2.3. Structure Definitions 14](#_Toc163478469)

[4. Auto-versioning using Git Tagging 19](#_Toc163478470)

[5. ROM/Hex File Identification 20](#_Toc163478471)

# Introduction to Boot Specification

This chapter aims to describe some high level information on the boot specification. For consistency, this document does not discriminate between the terms Operating System (OS), Real-Time Operating System (RTOS), Embedded Software and Firmware (or Main Firmware). All of these bootloader targets will be referred to as just firmware from here on.

## Background

All firmware usually have their own bootloader implementation. Installing a new firmware on a device generally involves their own set of boot mechanisms each with their own boot-time interfaces and policies. This specification hopes to standardize all these customizations and interfaces so that a bootloader could *potentially load* any complying firmware. This specification *does not specify how bootloaders should work* – it only shows how the interfaces to the target firmware will operate. It also does not apply to systems already coming with their own bootloaders like the Linux kernel or other RTOS alternatives using u-boot.

## Target Architecture

The boot specification has it’s roots in Gigadevice’s GD32F[[1]](#footnote-1) MCU product line which generally uses the ARM Cortex-M4 architecture. Non ARM Cortex-M architectures may need a boot specification but don’t have one already, it is encouraged to use this specification as reference so that a variation of this specification, stripped of the ARM Cortex-M specific details could be adopted to allow for the target architecture.

## Boot Sources

As mentioned in the *Target Architecture* section, the boot specification was initially conceived and developed on Gigadevice’s GD32F MCU product line. This MCU family always boots from a reset vector address mapped to an internal flash. The internal flash technologies used almost always used 4KB flash pages and would either have erase sectors that are either 16KB aligned or 64KB aligned. This means that the original behavior of the bootloader is to *look for a target firmware to boot also in the same flash memory* where it is running from – in a different sector or partition.

## Other Considerations

Normally, any bootloader’s general responsibility is to bootstrap the main firmware by initializing the environment it expects to run inside of and loading it from some read-only medium into a location that is executable (like for example, the SDRAM).

* Firmware Download Support   
  This specification adds to the general responsibility to add some proprietary (but probably still common across the company’s product lines) feature like support for updating of firmware thru a MIDI-compliant bus (USB/UART).

Additionally, the USB specification has its own industry protocol called the DFU[[2]](#footnote-2) specification - this is not about supporting that industry protocol. Instead, this is about supporting the SysEx standard defined by MIDI[[3]](#footnote-3) with some vendor-specific handshaking placed inside.

* Binary Identification

Common information like version and target product can be easily and correctly be extracted from firmware binary/ROM image files because the information is embedded inside of the same file. It frees the users from using the filename as the source of information which can be an error prone and cumbersome policy to use.

* Maximum Bootloader Size

It is strongly suggested to keep bootloader as free from high level responsibilities as possible. Therefore, the maximum size of the bootloader is expected to not exceed 64 Kilobytes.

* Boot Areas

The bootloader (by default) will start to search for the rom image header(2.1.1) signature directly at the next 64 kilobyte address where the bootloader is located. Any system is free to define their own boot areas depending on the requirements and restrictions of the system they are developing. For example, in some MCUs which have their own hardcoded ROM code loader, this bootloader will instead act as an SPL (Secondary Program Loader) and will most likely load firmware from an external flash on a completely separate address space.

* Memory Addresses and pointers

Addresses and pointers inside the headers data structures are always represented in 32-bit values. There is no planned support for 64-bit at the moment.

# The Boot Specification

We define at least three aspects of the boot specification:

1. The format of the firmware ROM image as seen by the bootloader. (**ROM image format**)
2. The format of information passed by the bootloader to the firmware, e.g. a simple parameter passing mechanism to a firmware. (**Parameter Passing**)
3. The state of the device when the bootloader transfers control to the firmware. (**Device State**)

## ROM image format

A firmware image is an ordinary 32-bit executable in the raw binary image format that contains on its first 4KB area, the ROM Header. The bootloader will search this region for the ROM header signature and if it cannot find it, the bootloader will always default to the firmware download state -e.g. there is no firmware to load.

### ROM Header

The ROM header is the data structure which contains all the information required by the bootloader to locate all other information about the firmware ROM image. It should be located inside the first 4KB area of the firmware ROM image. Additionally, it should be 256 byte aligned e.g. it must always start on an address that is divisible by 256.

If a field is not supported, then it should be zeroed out, e.g. NULL pointer address.

The layout of the ROM header is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Offset** | **Type** | **Length** | **Field Name** |
| 0 | u8 | 16 | ROM Header Signature |
| 16 | u32 | 4 | boot\_info pointer |
| 20 | u32 | 4 | version\_info pointer |
| 24 | u32 | 4 | product\_info pointer |
| 28 | u32 | 4 | dfu\_info pointer |
| 32 | u32 | 4 | parameters\_info pointer |
| 36 | u32 | 4 | Reserved |

**The fields of the ROM Header**

‘ROM Header Signature’

The ROM header signature is a 16 byte long field that identifies the ROM header , which must always be the string **“\x7f-ROM\_IMG-\xf7**”. The remaining unused space should be zeroed.

‘version\_info pointer’

Address where the bootloader can find the version\_info data structure.

‘boot\_info pointer’

Address where the bootloader can find the boot\_info data structure.

‘version\_info pointer’

Address where the bootloader can find the version\_info data structure.

‘product1\_info pointer’

Address where the bootloader can find the version\_info data structure.

‘dfu\_info pointer’

Address where the bootloader can find the dfu\_info data structure.

‘parameter\_info pointer’

Address where the bootloader can find the parameters\_info data structure.

### Boot Info

The boot info data structure contains information used by the bootloader to be able to load and pass control to firmware in a safe and correct manner. It can be located in any 16-byte aligned address inside the ROM image.

|  |  |  |  |
| --- | --- | --- | --- |
| **Offset** | **Type** | **Length** | **Field Name** |
| 0 | u8 | 16 | Boot Info Signature |
| 16 | u32 | 4 | Load Address |
| 20 | u32 | 4 | Initial SP address |
| 24 | u32 | 4 | Entry Point Address |
| 28 | u32 | 4 | ROM Size |

**The fields of the Boot Info Data Structure**

‘Boot Info Signature’

The Boot Info header signature is a 16 byte long field that identifies the Boot Info Data structure, which must always be the string **“\x7f-BOOTINFO-\xf7**”. The remaining unused space should be zeroed.

‘Load Address and ROM Size’

The Load Address and ROM Size contains the address and size where the bootloader is supposed to copy to ROM image onto before transferring control to it. In the case of GD32F MCUs or any similar implementation, the load address and size is generally not used since the firmware is already “loaded” onto its final executing location on the flash itself.

‘Initial SP and Entry Point Address’

ARM-7 based firmware requires to two things to be able to be started up properly, an initial stack pointer and an entry point. These two information are replicated here to have a common place where firmware entry point can be found. The bootloader will read the Initial SP Address set the SP register to this value, and set PC register to the entry point address to be able to transfer control to main firmware designed to run in ARM-7.

### Version Info

The Version info data structure contains information used by the bootloader to determine the version of the rom image. It can be located in any 16-byte aligned address inside the ROM image.

|  |  |  |  |
| --- | --- | --- | --- |
| **Offset** | **Type** | **Length** | **Field Name** |
| 0 | u8 | 16 | Version Info Signature |
| 16 | u8 | 1 | Major Version |
| 17 | u8 | 1 | Minor Version |
| 18 | u8 | 1 | Patch level |
| 19 | u8 | 32 | Pre Release string |
| 51 | u8 | 16 | Build Date |

**The fields of the Version Info Data Structure**

‘Version Info Signature’

The Version Info header signature is a 16 byte long field that identifies the Version Info data structure, which must always be the string **“\x7f-VERINFO-\xf7**”. The remaining unused space should be zeroed.

‘Major, Minor, Patchlevel’

The Major, Minor, Patchlevel are concatenated together to what is know as the semantic[[4]](#footnote-4) versioning scheme.

‘Pre Release String’

A short null-terminated string that is used to describe a pre-release. Can be empty.

‘Build Date’

The build date records the date when the binary ROM image was built. The \_\_DATE\_\_ predefined macro defined by the C standard is the recommended way to fill up this field.

### Product Info

The Product info data structure contains information describing this ROM image e.g. what device this firmware is targeted to run, and other important device parameters. It can be located in any 16-byte aligned address inside the ROM image.

|  |  |  |  |
| --- | --- | --- | --- |
| **Offset** | **Type** | **Length** | **Field Name** |
| 0 | u8 | 16 | Product Info Signature |
| 16 | u8 | 63 | Manufacturer ID |
| 79 | u8 | 1 | Manufacturer ID Length |
| 80 | u8 | 63 | Model ID |
| 143 | u8 | 1 | Model ID Length |
| 144 | u8 | 64 | Model Code |
| 208 | u8 | 64 | Model Name |

**The fields of the Product Info Data Structure**

‘Product Info Signature’

The Product Info header signature is a 16 byte long field that identifies the Product Info data structure, which must always be the string **“\x7f-PRODINFO-\xf7**”. The remaining unused space should be zeroed.

‘Manufacturer ID’

The Manufacturer ID is 64 bytes wide field used to encode the Manufacturer Code. It is 63 bytes wide to account for future expansion and support for longer IDs. For example the Behringer Manufacturer Code is 00-20-32. The remaining unused space should be zeroed.

‘Manufacturer ID Length’

The Manufacturer ID Length describes the length of the Manufacturer ID field. In the prior example given the Behringer manufacturer ID is 00-20-32, therefore this field should be 3.

‘Model ID’

The Manufacturer ID is 64 bytes wide field used to encode the Model Code. It is 63 bytes wide to account for future expansion and support for longer IDs. For example the BM-11M Model ID is 00-01-57. The remaining unused space should be zeroed.

‘Model ID Length’

The Model ID Length describes the length of the Model ID field. In the prior example given the BM-11M Model ID is 00-01-57, therefore this field should be 3.

‘Model Code’

The Model Code is a Null-terminated string that describes the vendor specific model code of the product. For example, BM-11M model name is “0709-ALM”. The remaining unused space should be zeroed.

‘Model Name’

The Model Name is a Null-terminated string that describes the name of the product. For example, BM-11M model name is “BM-11M”. The remaining unused space should be zeroed.

### DFU Info

The DFU info data structure contains information used by the bootloader to be able determine where to find the DFU signature. It can be located in any 16-byte aligned address inside the ROM image. The DFU signature, when present signifies that it should not bootstrap the firmware but boot into DFU mode instead. The signature location is defined by the firmware, and should be located in non-volatile memory (flash). The DFU signature itself, is only just 4 bytes long.

|  |  |  |  |
| --- | --- | --- | --- |
| **Offset** | **Type** | **Length** | **Field Name** |
| 0 | u8 | 16 | DFU Info Signature |
| 16 | u32 | 4 | Address |
| 20 | u32 | 4 | Pattern |

**The fields of the DFU Info Data Structure**

‘DFU Info Signature’

The DFU Info signature is a 16 byte long field that identifies the DFU Info data structure, which must always be the string **“\x7f-DFUINFO-\xf7**”. The remaining unused space should be zeroed.

‘Address

The bootloader checks this address for the value defined in the pattern field.

‘Pattern’

If the value at the DFU Signature address matches this value, then bootloader will boot info DFU mode.

### Parameters Info

This is discussed in Parameter Passing (0).

## Parameter Passing

This boot specification provides an avenue where the bootloader may need to pass some additional information or flags to the firmware similar to running a program with an argument from the command line. This is the purpose of the Parameters Info data structure. It can be located in any 16-byte aligned address inside the ROM image.

|  |  |  |  |
| --- | --- | --- | --- |
| **Offset** | **Type** | **Length** | **Field Name** |
| 0 | u8 | 16 | Parameters Info Signature |
| 16 | u32 | 4 | Parameters Info Address |
| 20 | u32 | 4 | Parameters Info Length |

**The fields of the Parameters Info Data Structure**

‘Parameters Info Signature’

The Parameters Info signature is a 16 byte long field that identifies the Parameters Info data structure, which must always be the string **“\x7f-PARAMINFO-\xf7**”. The remaining unused space should be zeroed.

‘Parameters Info Address’

This is the firmware defined space (located in RAM memory) where the firmware expects the bootloader to write its parameters to. Parameters are always NULL terminated strings.

‘Parameters Info Length’

This is the firmware defined space (located in RAM memory) where the firmware expects the bootloader to write its parameters to.

## Device State

Upon transferring control to the firmware, the following should all hold true.

* The ARM processor should be in thumb mode.
* All peripherals used in processing are left in their initial reset state like the ADC/DAC/USB, etc. Save for a few peripherals that plays a role in the early bootup phase like GPIOs used for the LEDs among other things.
* The stack and BSS regions used by the firmware is **not** zeroed out.
* All the general purpose registers should be zeroed out except for the SP and the PC.
* The DFU signature has not been tampered. If it is not cleared, firmware should clear it.
* All interrupts are disabled.
* MMU is turned off.
* Flash unit is in write protected mode.
* Memory ECC (if available) is initialized.
* Data L1 and Instruction cache (if available) are all turned off.

# Implementation Details

## Toolchain Support

* Compiler can configure a symbol to be placed in a section defined by the developer thru the use of linker directives. Using the GNU Compiler collection (GCC, GNU ld) should pose no problem.

## Implementation using the C Language

### Annotations

Using the GCC “\_\_attribute\_\_” extensions, we are able to create annotations (or labels) that can be used to fine tune our boot data structures.

**Define the building blocks of our annotations.**

We will build further annotations using these two.

#define \_\_section(x) \_\_attribute\_\_((section (x)))

#define \_\_aligned(x) \_\_attribute\_\_((aligned (x)))

**Define the intermediate annotations**

These will describe the actual boot header data structures.

#define ROM\_HDR\_ALIGNMENT 256

#define INFO\_ALIGNMENT 256

#define \_\_rom\_header \_\_section(“.romhdr”) \_\_aligned(ROM\_HDR\_ALIGNMENT)

#define \_\_data\_info \_\_section(“.info”) \_\_aligned(INFO\_ALIGNMENT)

### Types and other Constants

We also define a few types and constants we will be constantly using when build the data structures below.

typedef uint32\_t addr\_t;

#define SIGNATURE\_LENGTH 16

### Structure Definitions

From this point, we will define the boot header structures defined in this specification document. Additionally, we will also provide static helper methods that will abstract and provide the correct creation of objects of the boot header types.

**rom\_hdr**

struct rom\_hdr {

#define ROM\_HDR\_SIGNATURE “\x7f-ROM\_IMG-\xf7

uint8\_t signature[SIGNATURE\_LENGTH];

addr\_t boot\_info\_ptr;

addr\_t version\_info\_ptr;

addr\_t product\_info\_ptr;

addr\_t dfu\_info\_ptr;

addr\_t parameters\_info\_ptr;

uint32\_t reserved;

};

**helper code to create rom\_hdr structure:**

#define DECLARE\_ROM\_HDR(name, b, v, p, d, parm) \

static const struct rom\_hdr \_\_rom\_header name = { \

.signature = ROM\_HDR\_SIGNATURE, \

.boot\_info\_ptr = (const addr\_t)(b), \

.version\_info\_ptr = (const addr\_t)(v), \

.product\_info\_ptr = (const addr\_t)(p), \

.dfu\_info\_ptr = (const addr\_t)(d), \

.parameters\_info\_ptr = (const addr\_t)(parm), \

}

**boot\_info**

struct boot\_info {

#define BOOT\_INFO\_SIGNATURE “\x7f-BOOTINFO-\xf7

uint8\_t signature[SIGNATURE\_LENGTH];

addr\_t load\_address;

addr\_t initial\_sp;

addr\_t entry\_point;

uint32\_t rom\_size;

};

**helper code to create boot\_info structure:**

#define DECLARE\_BOOT\_INFO(name, l, I, e, r) \

static const struct boot\_info \_\_data\_info name = { \

.signature = BOOT\_INFO\_SIGNATURE, \

.load\_address = (const addr\_t)(l), \

.initial\_sp = (const addr\_t)(i), \

.entry\_point = (const addr\_t)(e), \

.rom\_size = (r), \

}

**version\_info**

struct version\_info {

#define VERSION\_INFO\_SIGNATURE “\x7f-VERINFO-\xf7

uint8\_t signature[SIGNATURE\_LENGTH];

uint8\_t major;

uint8\_t minor;

uint8\_t patch\_level;

uint8\_t build\_date[BUILD\_DATE\_LENGTH];

};

**helper code to create version\_info structure:**

#define \_\_DECLARE\_VERSION\_INFO(name, maj, min, pl, b) \

static const struct version\_info \_\_data\_info name = { \

.signature = VERSION\_INFO\_SIGNATURE, \

.major = (const addr\_t)(l), \

.minor = (const addr\_t)(i), \

.patch\_level = (const addr\_t)(e), \

.build\_date = (b), \

}

#define DECLARE\_VERSION\_INFO(name, maj, min, pl) \

\_\_DECLARE\_VERSION\_INFO(name, maj, min, pl, \_\_DATE\_\_)

The helper code to a create version\_info structure provides two levels of creation so that we can automate the input of the build date and not have to pass any date explicitly in the code.

**product\_info**

struct product\_info {

#define PRODUCT\_INFO\_SIGNATURE “\x7f-PRODINFO-\xf7

#define MANUFACTURER\_ID\_LENGTH 63

#define MODEL\_ID\_LENGTH 63

#define MODEL\_CODE\_LENGTH 63

#define MODEL\_NAME\_LENGTH 64

uint8\_t signature[SIGNATURE\_LENGTH];

uint8\_t mfg\_id[MANUFACTURER\_ID\_LENGTH];

uint8\_t mfg\_id\_len;

uint8\_t model\_id[MODEL\_ID\_LENGTH];

uint8\_t model\_id\_len;

uint8\_t model\_code[MODEL\_CODE\_LENGTH];

uint8\_t model\_name[MODEL\_NAME\_LENGTH];

};

**helper code to create product\_info structure:**

#define DECLARE\_PRODUCT\_INFO(name, mfg, mfg\_len, mdl\_id, \

mdl\_len, mdl\_code, mdl\_name) \

static const struct product\_info \_\_data\_info name = { \

.signature = PRODUCT\_INFO\_SIGNATURE, \

.mfg\_id = (mfg), \

.mfg\_id\_len = (mfg\_len), \

.model\_id = (mdl\_id), \

.model\_id\_len = (mdl\_len) \

.model\_code = (mdl\_code), \

.model\_name = (mdl\_name), \

}

**dfu\_info**

struct dfu\_info {

#define DFU\_INFO\_SIGNATURE “\x7f-DFUINFO-\xf7

uint8\_t signature[SIGNATURE\_LENGTH];

addr\_t address;

uint32\_t pattern;

};

**helper code to create dfu\_info structure:**

#define DECLARE\_DFU\_INFO(name, a, p) \

static const struct dfu\_info \_\_data\_info name = { \

.signature = DFU\_INFO\_SIGNATURE, \

.address = (a), \

.pattern = (p), \

}

**parameters\_info**

struct parameters\_info {

#define PARAMETERS\_INFO\_SIGNATURE “\x7f-PARAMINFO-\xf7

uint8\_t signature[SIGNATURE\_LENGTH];

addr\_t address;

uint32\_t length;

};

**helper code to create parameter\_info structure:**

#define DECLARE\_PARAMETER\_INFO(name, a, l) \

static const struct dfu\_info \_\_data\_info name = { \

.signature = PARAMETERS\_INFO\_SIGNATURE, \

.address = (a), \

.length = (l), \

}

**Sample usage:**

Using the helper APIs above, firmware code can easily incorporate the boot header specification in a few (more readable) lines.

#define COMMANDLINE\_LENGTH 64

#define char command\_line[COMMANDLINE\_LENGTH];

#define BEHRINGER\_MANUFACTURER\_ID “\x00\x20\x32”

#define BEHRINGER\_MANUFACTURER\_ID\_LEN 3

#define BM11M\_MODEL\_ID “\x00\x01\x57”

#define BM11M\_MODEL\_ID\_LEN 3

#define BM11M\_MODEL\_CODE “0709-ALM”

#define BM11M\_MODEL\_NAME “BM-11M”

#define BM11M\_MAJOR\_VERSION 1

#define BM11M\_MINOR\_VERSION 2

#define BM11M\_PATCH\_LEVEL 3

#define BM11M\_LOAD\_ADDR 0x08000000

#define BM11M\_INITIAL\_SP 0x0C000000

#define BM11M\_ENTRY\_POINT (BM11M\_LOAD\_ADDR)

#define BM11M\_ROM\_SIZE 0x80000

DECLARE\_PARAMTERS\_INFO(param\_info, command\_line, COMMANDLINE\_LENGTH);

DECLARE\_DFU\_INFO(dfu\_info, DFU\_PATTERN\_ADDRESS, DFU\_PATTERN);

DECLARE\_PRODUCT\_INFO(product\_info,

BEHRINGER\_MANUFACTURER\_ID,

BEHRINGER\_MANUFACTURER\_ID\_LEN,

BM11M\_MODEL\_ID,

BM11M\_MODEL\_ID\_LEN,

BM11M\_MODEL\_CODE,

BM11M\_MODEL\_NAME);

DECLARE\_VERSION\_INFO(version\_info,

BM11M\_MAJOR\_VERSION,

BM11M\_MINOR\_VERSION,

BM11M\_PATCH\_LEVEL);

DECLARE\_BOOT\_INFO(boot\_info,

BM11M\_LOAD\_ADDR,

BM11M\_INITIAL\_SP,

BM11M\_ENTRY\_POINT,

BM11M\_ROM\_SIZE);

DECLARE\_ROM\_HDR(rom\_hdr,

&boot\_info,

&version\_info,

&product\_info,

&dfu\_info,

&param\_info);

# Auto-versioning using Git Tagging

Firmware versioning is usually kept inside a header file that defines the major, minor, patch level macros. What this boot specification proposes is that versioning should be done via git tagging instead. Generation of the version header file is done early \_before\_ any compilation takes place.

The git command describe with the --match modifier can be used to give the latest version tag that can be traced from the latest commit.

Tagging should follow the convention vX.Y.Z\*\*\*\*\*\*\*.

With this pattern, we can query the git repository via the following command:

**GIT DESCRIBE**

git describe --tag --abbrev=0 --match v\*.\*.\*

The output of which can be processed by python script to generate a valid version.h header file which firmware code can include. So it can use the versioning coming from the git repository tag.

# ROM/Hex File Identification

One benefit that is also considered a highlight of the boot specification is that it can be used as a binary identification (see 1.4) for firmware files. These firmware files are in the form of either an Intel hex or a raw binary.

A classic and straightforward way to add information to a firmware release is to use the filename. But this can quickly become cumbersome and is also error prone in that filenames can easily be renamed by anyone.

For example:

The filename for the firmware release for a hypothetical device “BM-11M” can be the following: ***bm-11m.hex***. We can go all sorts of crazy for the file naming, for example, to add versioning information we can append the versioning to the filename like so: **bm-11m\_v1.2.3.hex**. As you can see, adding additional information to the file by just appending new labels to it will quickly become cumbersome and error prone.

With the information standardized by the boot specification, we are able to create scripts and/or software that can read the information embedded inside the binary or hex file itself and not have to rely on the filename. At the same time, we can add more information about the binary while maintaining a clean and simple file naming convention.

A ROM file (testfw) that is compliant with the boot specification will look like this when inspected inside a hex editor:

A screenshot of a computer

Description automatically generated

Figure 1 - hexdump of binary with boot specification metadata

Notice the familiar signatures embedded in the file itself. We created a proof of concept script that can read the info off the file (whether binary or hex) and report them properly to the user. See examples that follow:

A screenshot of a computer

Description automatically generated

Figure 2 - reporting information of firmware in binary ROM form

A screenshot of a computer

Description automatically generated

Figure 3 - reporting information of the same firmware above in Intel hex form

From the screenshots above, we can see the easy extraction of information from the firmware file itself. Information like the version (**v1.2.3**), manufacturer ID **(00 20 32)**, model code **(0709-ALM)**, model name **(BM-11M),** even the build date **(Mar 17 2024)** – irrespective of the filename of the firmware, which is in this case just “testfw” for raw binary and “testfw.hex” for Intel hex are successfully extracted and reported to the user.

1. GD32 ARM Cortex-M4 Microcontrollers, https://www.gd32mcu.com/en/product/m4 [↑](#footnote-ref-1)
2. USB Device Firmware Upgrade Version 1.1, https://www.usb.org/sites/default/files/DFU\_1.1.pdf [↑](#footnote-ref-2)
3. MIDI, https://midi.org/specs [↑](#footnote-ref-3)
4. Semantic Versioning, https://semver.org/ [↑](#footnote-ref-4)