Locutus File System and Wire Communication Protocol

4-Jul-2019, A


ONLY
ONE WAY

Background

The Locutus wire communication protocol specifies how Locutus and the host communicate with each other. End user interface software uses the wire protocol to configure Locutus, download ROM images, set up menus, and so forth. The protocol:

- Provides a mechanism to detect the presence of Locutus.
- Provides a series of commands that allow the host to manipulate Locutus.
- Provides a lightweight mechanism for Locutus to alert the host of issues.
- Is extensible, allowing for new features in the future.

Several of the commands in the wire protocol directly manipulate the Locutus File System (LFS). This document first describes the Locutus File System, along with its data structures. The remainder of the document describes the wire protocol itself, and the commands you can invoke via that protocol.

Locutus File System (LFS)

Concepts

The Locutus File System rests on a handful of unique concepts. The following table introduces several of these concepts. The remainder of the LFS section drills down into the most important details.

Term	Description
FTL	Flash translation layer. This manages the underlying flash storage, presenting the illusion of a standard read-write block device.
PBLK	Physical block. This refers to a physical 8K block of storage on the flash.
VBLK	Virtual block. This refers to an 8K block of storage presented by the FTL to the layers above it. FTL transparently maps VBLKs to PBLKs under the hood.
COW	Copy-On-Write. This is the mechanism that FTL uses to migrate a VBLK to a new PBLK should something try to write new data to the VBLK.
Directory	A container holding a list of files. Some OSes call this a folder instead.
File	A container holding metadata about a file, including its type, its name, its displayed color, and optionally pointers to data forks and/or an associated directory.
Fork	A container for data held in flash. It consists of a starting VBLK, a length in bytes, and a user-provided 24-bit UID.
UID	An arbitrary 24-bit unique identifier for a fork. External management software sets UIDs on forks to help identify those forks later. Current management software uses a 24-bit CRC to generate UIDs.

Filesystem Structure

Presentation Layer vs. Data Layer

LFS is *not* a hierarchical file system. In a hierarchical file system, files are designated by paths, and the path components represent nodes within a filesystem tree. Files form leaves on the tree[1]. LFS dispenses with this notion entirely.

LFS treats *directories* and *files* as a *presentation layer* over the data. Neither directories nor files play any role in managing the filesystem itself as far as LFS is concerned. Rather, LFS merely ferries the information from the outside file manager to the embedded menu system. The external file manager can construct a hierarchical view from the directories and files LFS provides; however, it's not required to do so.

In contrast, LFS directly manages *forks*. When you attach data to a fork, LFS manages the integrity of the data and its placement in flash for you. It controls where it is allocated and manages that as needed. The external file manager can upload and attach data to a fork, request to read data from a fork, and erase forks. LFS manages the rest of the details under the hood, such as what VBLKs hold the fork, and so on. Thus, the outside file manager only manages the presentation layer, while LFS manages the integrity and placement of the data.

This division of labor helps make LFS robust against interrupted updates. If an update gets interrupted, leaving the directory or file list in an inconsistent state, *no data is actually lost even if the file system looks corrupted.* LFS creates and destroys forks atomically.[2] Forks retain their data until deleted. When the external manager resumes an interrupted update, it merely needs to reconcile the set of forks stored on the device, and re-send the directory and file lists. It does not need to integrity-check the stored data or underlying flash storage media.

This division of labor also enables other interesting capabilities such as *deduping*. With deduping, multiple files can attach to the same game image fork, for example. Each of those files may have different details (different names, colors, save-game info, etc.) and may even live in different directories.

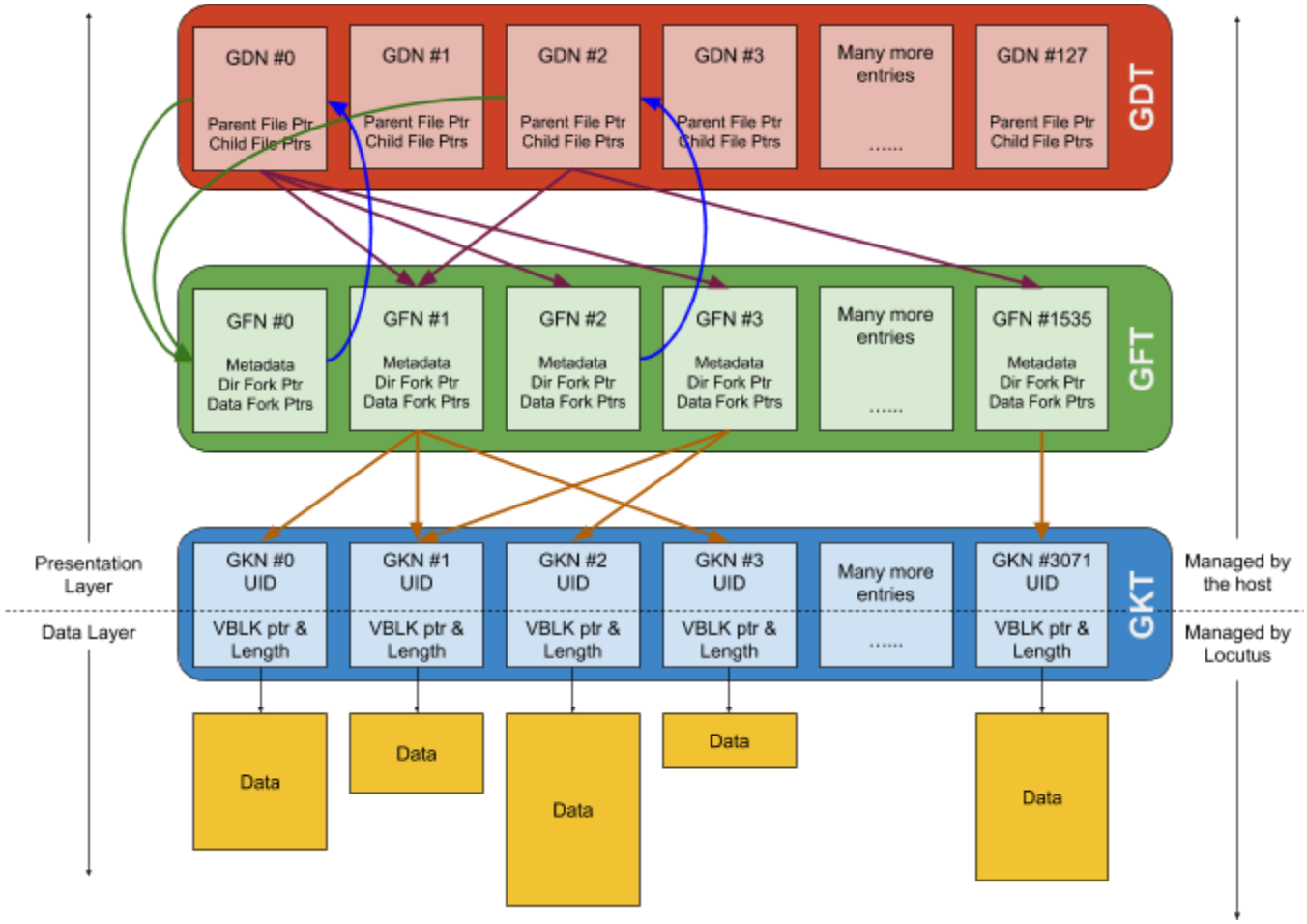
Global Tables

Every directory, file, and fork is assigned a *global* number. That is, every directory has a Global Directory Number (GDN), every file has a Global File Number (GFN), and every fork has a Global forK Number (GKN). These numbers are *not* context-sensitive. GFN #42 means the same thing even if it appears in every directory. GKN #42 refers to the same fork everywhere, regardless of how many files it's attached to.

These three numbers form indices into three corresponding global tables: The Global Directory Table (GDT), the Global File Table (GFT), and the Global forK Table (GKT). LFS's various APIs take GDN, GFN, and/or GKN arguments as input, and use those to index the GDT, GFT, and GKT respectively.

The following diagram shows the relationship between the Global Directory Table, the Global File Table, and the Global forK Table. Note that the connections shown are meant to be generally representative.

[1] In UNIX-style filesystems, there is an additional layer of indirection, inodes, which contain the permission information as well as the file contents; multiple files can point at the same inode. LFS's forks borrow from that.
[2] The underlying FTL is transaction-oriented with a full metadata journal. FTL ultimately provides the atomicity guarantees.



A quick legend for the connections:

- Curved **blue** lines from GFNs to GDNs represent directory fork links.
- Curved **green** lines from GDNs to GFNs represent parent-directory links.
- Straight **purple** lines from GDNs to GFNs show the membership of files in directories.
- Straight **orange** lines from GFNs to GKNs show the connection of data forks to files.

Basic Types

The remainder of this section uses the following type definitions for basic types. Integer quantities larger than 1 byte are stored in little endian format.

```
typedef uint16_t vblk_t; // Virtual block number; 0xFFFF if none.
typedef uint8_t  gdn_t;  // Global Directory Number (GDN), 0 - 127, or 0xFF
typedef uint16_t gfn_t;  // Global File Number (GFN), 0 - 1535, or 0xFFFF
typedef uint16_t gkn_t;  // Global forK Number (GKN), 0 - 3071, or 0xFFFF

typedef struct length24 {
    uint8_t len[3];      // 24-bit fork length in little-endian format.
} length24_t;
```

```
typedef struct uid24 {
    uint8_t uid[3];      // 24-bit unique-identifier.
} uid24_t;

enum file_type {
    FT_EMPTY = 0xFF,     // Free (empty) entry.
    FT_EXEC  = 0xFE,     // Executable file.
    FT_DIR   = 0xFD      // Directory file
};
typedef uint8_t file_type_t;

enum file_color {
    FC_BLACK      = 0,
    FC_BLUE       = 1,
    FC_RED        = 2,
    FC_TAN        = 3,
    FC_DARK_GREEN = 4,
    FC_GREEN      = 5,
    FC_YELLOW     = 6,
    FC_WHITE      = 7
};
typedef uint8_t file_color_t;
```

Directory Entries: Global Directory Table

Each GDT entry corresponds to a single directory. The GDT has room for up to 128 directories. Each directory holds pointers for up to 255 files. It also includes a special entry for *parent directory file.*

The following structure describes each GDT entry:

```
struct gdt_entry {
    gfn_t parent_directory;  // GFN of the parent directory
    gfn_t file_list[255];    // GFNs of the files in the directory
};
```

Each GDT entry occupies 512 bytes, laid out as follows:

Byte range		Description
0x000	0x001	Parent directory GFN
0x002	0x003	GFN of first file in directory
0x004	0x005	GFN of second file in directory
. . . many more entries . . .		
0x1FE	0x1FF	GFN of 255th file in directory

Each GDT entry stores its directory pointers as a GFNs, rather than GDNs. Directory entries in the GDT lack naming information and other metadata. LFS represents a directory by a GFT entry of type Directory (`0xFD`) that points to a corresponding GDT entry. GDT entries hold GFNs rather than GDNs, to connect to that metadata.

GDT entries have no directory-length field. GFN `0xFFFF` terminates a directory list that holds fewer than 255 valid entries. As a practical matter, Locutus treats the first "out of range" GFN as terminating the directory; however, for future-proofness sake, terminate the list with `0xFFFF` and fill all unused entries with `0xFFFF`.

File Entries: Global File Table

Each GFT entry represents a single file. Files contain multiple elements:

- **File Type.** Currently, Locutus recognizes two files types: Directory and Executable.
- **File Color.** This is the color the Intellivision-side UI uses to display the file.
- **Short Name.** This is a name that must fit within 18 characters.
- **Long Name.** This is a more descriptive name that must fit in 60 characters.
- **Directory GDN.** If this file corresponds to a directory, this provides the GDN.
    - For File Type = Directory, this points to the GDN for that directory.
    - For File Type ≠ Directory, Locutus assigns no meaning yet. In the future, it could be used to indicate a private file store for an executable or store other useful information.
- **Data Forks.** A file may have up to 7 data forks associated with it. Currently, Locutus only assigns meaning to the first 3 forks, and then only for file type == Executable.
    - **Fork 0:** Program image in LUIGI[3] format.
    - **Fork 1:** Program manual.
    - **Fork 2:** JLP flash emulation fork. ("JLP save game" information.)
    - **Forks 3 .. 6:** Reserved for future expansion.

The following structure describes each GFT entry:

```
struct gft_entry {
    file_type_t  type;           //  File type:  Free, Executable, Directory
    file_color_t color;          //  Display color (0 - 7)
    char         short_name[18]; //  Short name for the file; padded with NULs as needed.
    char         long_name[60];  //  Long name for the file; padded with NULs as needed.
    gdn_t        dir_gdn;        //  GDN of the directory fork. Set to 0xFF if none.
    uint8_t      rsvd_0;         //  For alignment purposes.  Set to 0xFF.
    gkn_t        fork[7];        //  GKNs for forks 0 through 6.
};
```

[3] LUIGI: Locutus Universal Intellivision Game Image.

Each GFT entry occupies 96 bytes, laid out as follows:

Byte range		Description
0x000		**File Type**. 0xFF = *Free*, 0xFE = *Executable*, 0xFD = *Directory*
0x001		**File Color.** Colors 0 - 7 correspond to Intellivision primaries.[4]
0x002	0x013	**Short Name.** Padded with NULs as needed to 18 characters.
0x014	0x04F	**Long Name.** Padded with NULs as needed to 60 characters.
0x050		**Directory GDN.** 0xFF if none.
0x051		**Reserved.** Set to 0xFF.
0x052	0x053	**Fork 0 Global forK Number (GKN).** Executable image,
0x054	0x055	**Fork 1 Global forK Number (GKN).** Manual text.
0x056	0x057	**Fork 2 Global forK Number (GKN).** JLP Flash storage.
0x058	0x059	**Fork 3 Global forK Number (GKN).** Reserved.
0x05A	0x05B	**Fork 4 Global forK Number (GKN).** Reserved.
0x05C	0x05D	**Fork 5 Global forK Number (GKN).** Reserved.
0x05E	0x05F	**Fork 6 Global forK Number (GKN).** Reserved.

From an implementation standpoint, the Intellivision user interface will display unknown file types; however, it won't let the user try to launch or interact with those files, since the UI doesn't know what to do with them.

For the directory GDN, 0xFF indicates that no directory is associated with this file. The current firmware expects the GDN to be 0xFF if the file type *is not* 0xFD (Directory); otherwise it expects the GDN to be a valid directory number if the file type *is* 0xFD.

Programs manipulating the short and/or long names must not assume the names have NUL terminators. An 18-character short name has no NULs. NULs are there for padding only, when the names are shorter than the field provided.

GFN #0: The Root Directory

While LFS is not strictly hierarchical, it does require a default entry point. GFN #0 serves as this entry point, pointing to the root directory. For proper operation, its file type must be set to 0xFD, and it must point at a valid GDN.

Because GFN #0 is not normally listed as a member of any directory, Locutus assigns alternate meanings to its short and long names. The short name becomes the *Device Name*, and is displayed at the top of screen when at the root menu. The long name becomes the *Device Owner*, and is displayed on the Device Information screen.

[4] Out of range color numbers are undefined / reserved.

When Locutus initializes the filesystem to factory-new state, it sets:

- GFN #0 Type == 0xFD (Directory).
- GFN #0 Color == 7 (White).
- GFN #0 Short Name (aka. Device Name) == "LTO Flash!".
- GFN #0 Long Name (aka. Device Owner) == "" (empty string).
- GFN #0 Directory Fork == GDN #0.
- GDN #0 Parent directory == GFN #0.
- GDN #0 Fork entries to 0xFFFF.

Data Forks: Global forK Table

Each entry in the GKT represents one data fork. A data fork is just a linear sequence of bytes on the flash. GKT entries also include a 24-bit UID to assist outside file managers when synchronizing files. Locutus assigns no actual meaning to the UID; however, in some cases Locutus needs to construct one.

The following structure describes each GKT entry:

```
struct gkt_entry {
    vblk_t     vblk;         // Virtual block number.
    length24_t length;       // 24-bit fork length.
    uid24_t    uid;          // 24-bit UID.
};
```

Each GKT entry occupies 8 bytes, laid out as follows:

Byte range		Description
0x000	0x001	Virtual block number that fork starts in, or 0xFFFF if fork is unused.
0x002	0x004	Length of the fork in bytes. Maximum fork length is 0x100000.
0x005	0x007	24-bit UID for the fork. Used by outside utilities to help identify forks.

Every data fork exists as a contiguous extent on the filesystem, specified by the starting VBLK and length. The exact span of VBLKs a given fork occupies is determined directly by the starting VBLK and fork length.

Storing each file's location as a starting VBLK plus length saves space by avoiding a separate "allocated block list" for each file. However, that can lead to filesystem fragmentation as forks get created and deleted. Therefore, LFS changes the starting VBLK for a fork as needed, if it needs to defragment the filesystem layout to make room for a new fork.

LFS maintains the actual mapping of forks to VBLKs. The external file manager cannot modify the starting VBLK for any fork, and can only specify the fork's length at fork creation time.

LFS currently does not offer a mechanism for changing the length of a fork once created. This restriction greatly simplifies the filesystem design and implementation.

On-Media Layout

The GDT, GFT, and GKT occupy the first several VBLKs of the flash as three contiguous arrays. This can be thought of as one larger structure:

```
struct global_tables {
    struct gdt_entry gdt[128];    // 128 Global Directory Table (GDT) entries.
    struct gft_entry gft[1536];   // 1536 Global File Table (GFT) entries.
    struct gkt_entry gkt[3072];   // 3072 Global forK Table (GKT) entries.
};
```

The Download Global Tables command sends a serial data payload matching `struct global_tables`.

The resulting global tables fill the first 29 VBLKs as follows:

Table	Entries	Entry Size (bytes)	Total Size (bytes)	VBLKs	VBLK Range	
GDT	128	512	65536	8	0	7
GFT	1536	96	147456	18	8	25
GKT	3072	8	24576	3	26	28

Most software does not care about the on-media layout for these tables. The Locutus Wire Protocol provides commands to manipulate the global tables that do not expose or rely on this layout.

Locutus Wire Protocol

The Locutus Wire Protocol is a simple command-response protocol over the USB serial link provided by Locutus. For robustness, checksums protect all commands and data payloads, ensuring both sides of the link agree on whatever actions need to be performed.

USB and Serial Parameters

Locutus uses an FT230X USB serial chip for its serial interface, programmed as follows:

USB Parameter	Value	Details
Product ID (PID)	`0x6015`	FT230X product ID.
Vendor ID (VID)	`0x0403`	Future Technology Devices International, LTD (FTDI).
Serial Number	`JZxxxxxx`	Uniquely set for each device to a random string.
Product	`“LTO Flash!”`	This string is programmed in at the factory.
Manufacturer	`“Left Turn Only”`	This string is programmed in at the factory.
Current Required (mA)	250mA	This is above the expected peak current draw of ~180mA.

The serial port it exposes should be configured with the following parameters. Take particular note of the parameters marked *mandatory*.

Serial Parameter	Value	Mandatory?	Details
Baud Rate	`2000000`	**YES**	This is fixed in the firmware; other baud rates will fail.
Data Bits	`8`	**YES**	This is fixed in the firmware; other settings will fail.
Parity Bits	`None`	**YES**	This is fixed in the firmware; other settings will fail.
Stop Bits	`1`	**YES**	This is fixed in the firmware; other settings will fail.
Flow Control	`Hardware Flow Control`	Highly recommended	The firmware expects to use hardware flow control and does not understand software flow control.
Latency Timer	`1ms`	Recommended	This is an optional setting; smaller values are better.

Protocol Structure

The protocol consists of the following elements:

- **Idle Beacon.** This announces the presence of Locutus, and the fact it is ready and able to accept commands.
- **Host to Locutus Command.** This is a command packet toward Locutus, asking it to perform some operation.
- **Locutus to Host Acknowledgement.** This ACKs or NAKs the command, and optionally provides a response payload. In case of a NAK, the exchange stops here.
- **Serial Data Payload.** For commands with a serial data payload (either Host to Locutus, or Locutus to Host), the serial data payload gets sent after the host receives an ACK.
- **Locutus to Host Final Response.** Locutus reports success or failure for the command, along with a checksum for all exchanged acknowledgement and payload bytes.

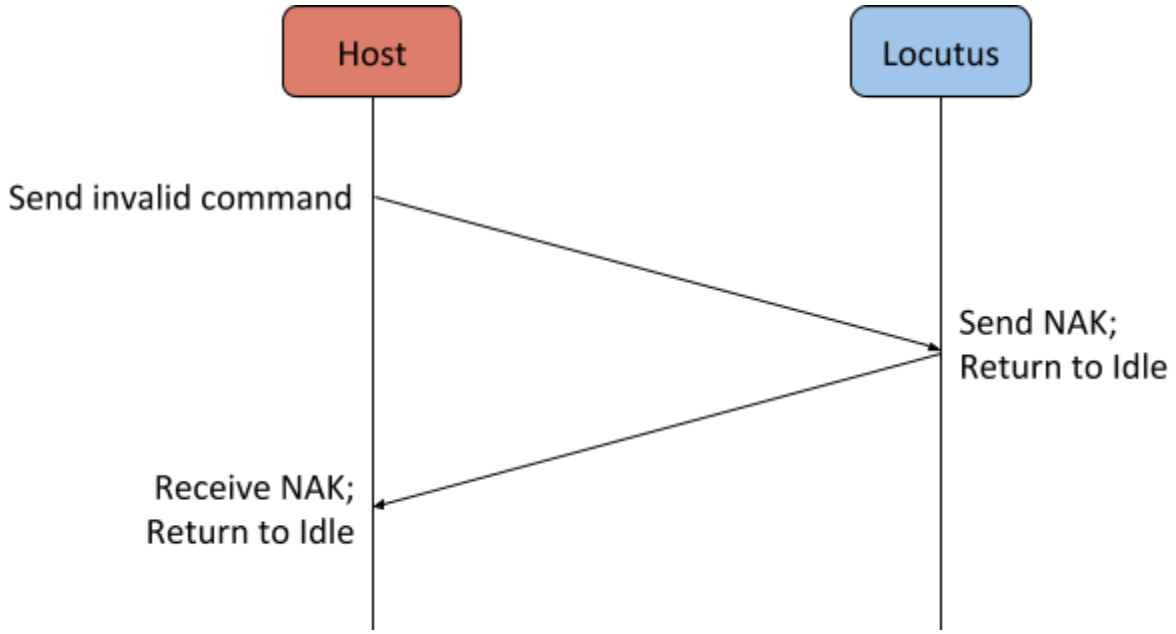
Protocol Timeline Diagrams

The following timeline diagrams illustrate the different command/response patterns. The sections that follow drill down into the actual protocol details.

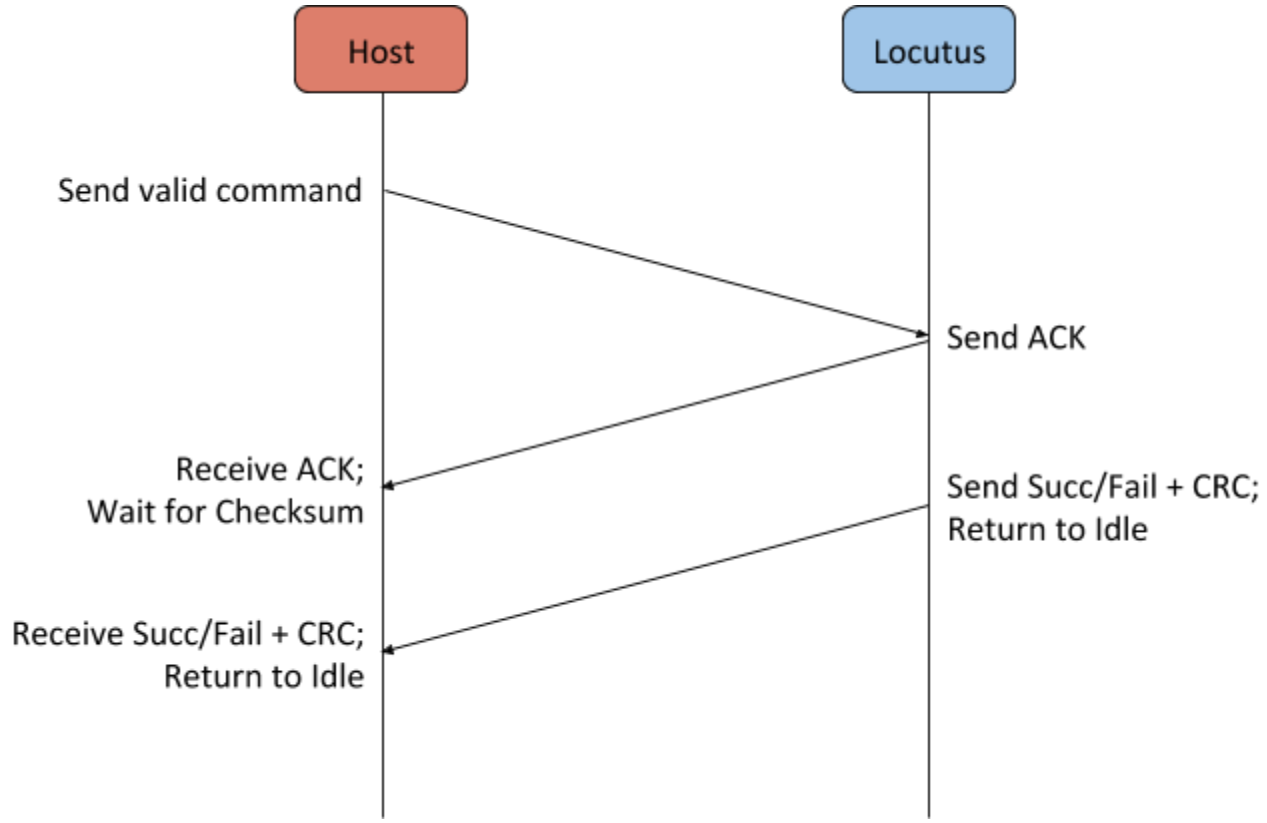
Each diagram assumes the host has synchronized to Locutus' idle beacon before sending the request.

Also, note that *any* command could time out if Locutus gets interrupted by a punctuating event, such as the Intellivision powering up or down, or the USB cable suddenly getting disconnected.

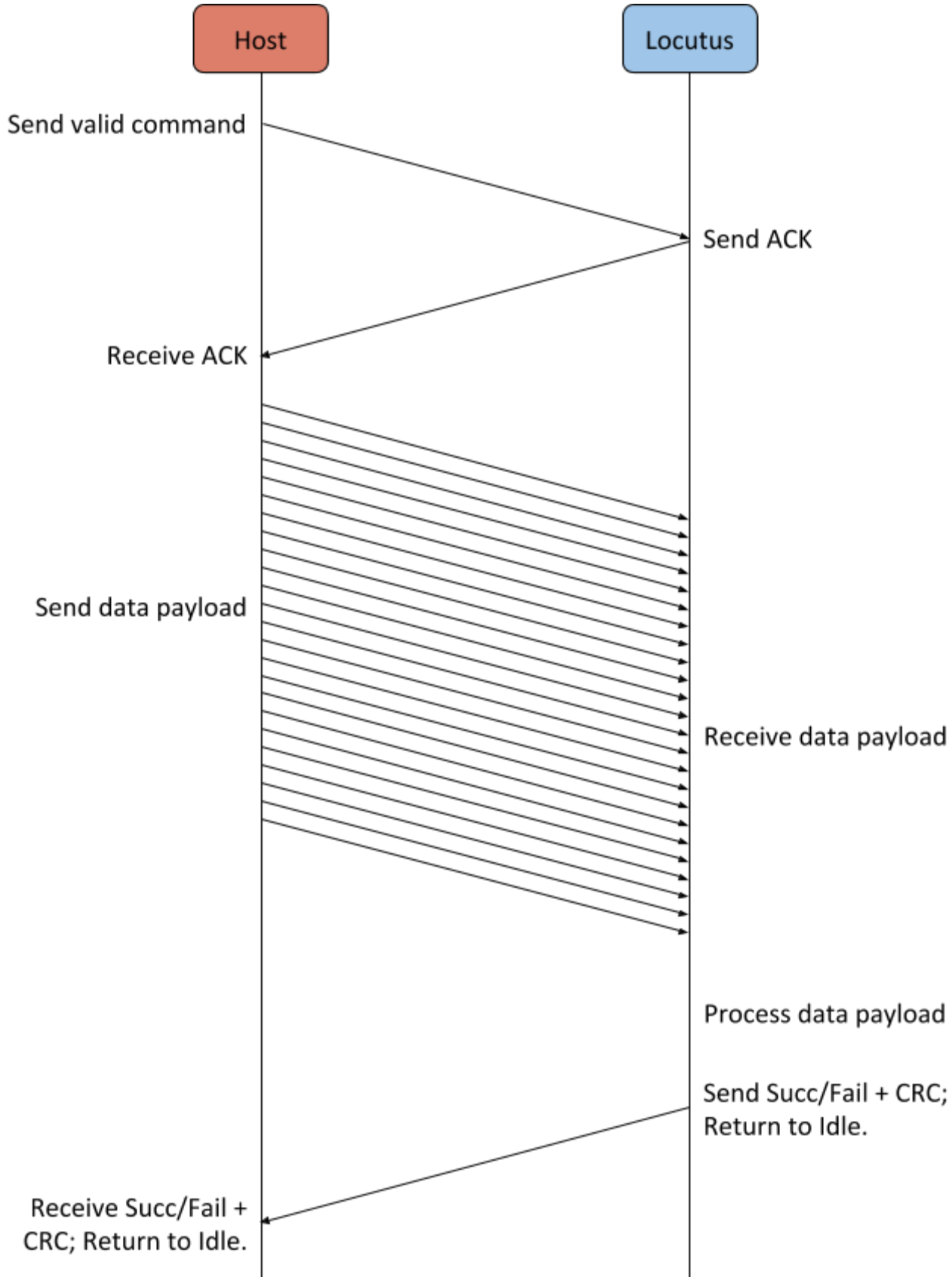
NAK’d Command



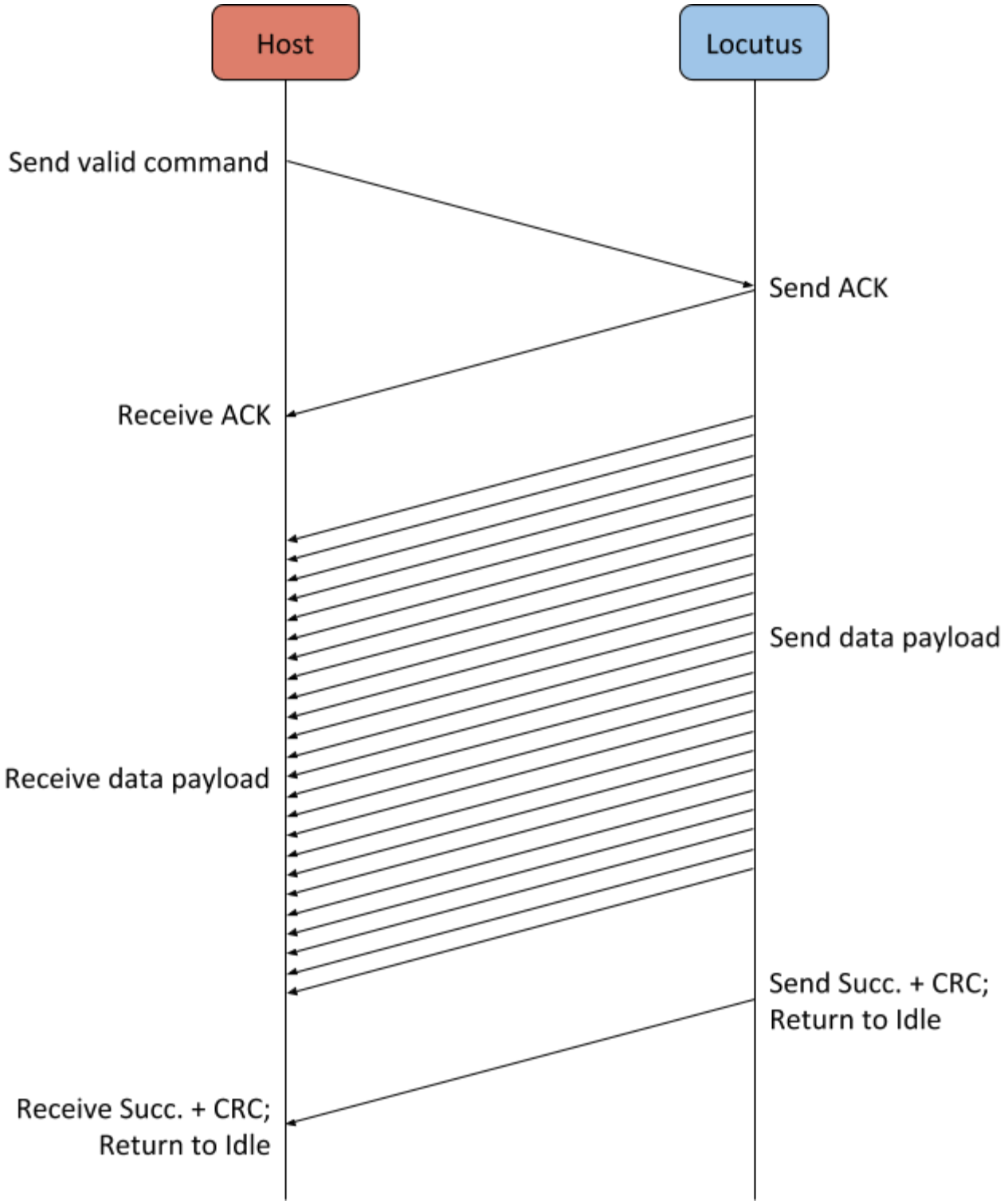
ACK’d Command Without Serial Data Payload



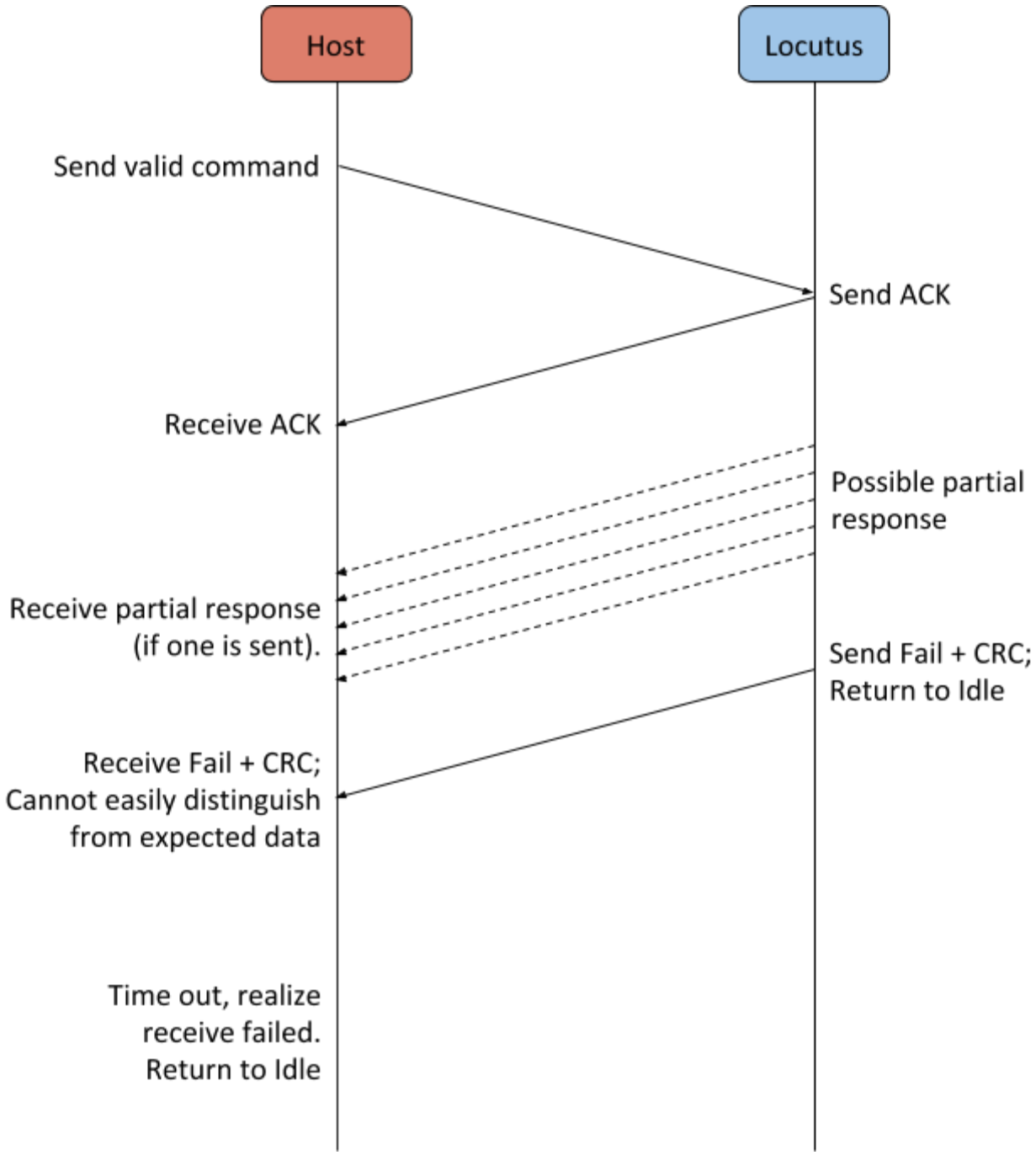
ACK’d Command With Host-to-Locutus Serial Data Payload



ACK’d Command With Locutus-to-Host Serial Data Payload (Successful)



ACK'd Command With Locutus-to-Host Serial Data Payload (Unsuccessful)



This particular pattern should be rare, as Locutus makes every attempt to validate arguments up-front so it isn't surprised with an error mid-stream. If such a timeout does occur, the last 5 bytes received before the timeout are likely the error-return code and CRC.

Idle Beacon

When Locutus is idle and able to accept new commands, it periodically outputs the string **“**LOCUTUS\n**”**, approximately once per second, although sometimes faster or slower. This is referred to as the *idle beacon*.

The exact sequence, in hexadecimal, is:

```
4C 4F 43 55 54 55 53 0A                                 # LOCUTUS\n
```

Outside software can detect the presence of Locutus by looking for this beacon. Any software which interfaces to Locutus should take care to actively receive and drain these beacons from the serial port.

Suppose the function `recv_byte()` returns the next byte from the serial port, or a negative value if an error occurs such as a receive-timeout or other receive error. The following C function synchronizes with Locutus' beacon, if Locutus is sending one. It returns `0` on success, or a negative value indicating the serial port error.

```
int wait_beacon(void) {
    int r = 0;

    while (true) {
        /* If an error occurs, return the error */
        if (r < 0) return r;

        /* Synchronize with beacon string; discard all other input. */
        if ((r = recv_byte()) != 'L') continue;
        if ((r = recv_byte()) != 'O') continue;
        if ((r = recv_byte()) != 'C') continue;
        if ((r = recv_byte()) != 'U') continue;
        if ((r = recv_byte()) != 'T') continue;
        if ((r = recv_byte()) != 'U') continue;
        if ((r = recv_byte()) != 'S') continue;
        if ((r = recv_byte()) == 0xA) break;
    }

    /* Once we receive a complete beacon, return 0, indicating we're sync'd. */
    return 0;
}
```

Host to Locutus Command

All host to Locutus commands fit the following structure:

Bytes		Name	Description
0x00		Start of Header	Fixed constant 0xAA
0x01		Command	Single byte containing the command number
0x02		Inv. Command	Single byte containing the command number, XOR'd with 0xFF
0x03		End of header	Fixed constant 0x55
0x04	0x07	Arg0	First argument
0x08	0x0B	Arg1	Second argument
0x0C	0x0F	Arg2	Third argument
0x10	0x13	Arg3	Fourth argument
0x14	0x17	Checksum	Standard CRC32 of the preceding 20 bytes.

Every command starts with this fixed 24 byte record. When invoking a command that takes fewer than 4 arguments, the host should fill the unused arguments with 0xFFFFFFFF.

If Locutus accepts the command, it replies with 0xAA to indicate it received the command successfully. If Locutus detects an error in the command, it will reply with 0xEE. Otherwise, the command may time out. Locutus will start sending "LOCUTUS\n" idle beacons when it's ready for a new command.

Some commands include a serial data payload. For payloads sent by the host, the host must not send the payload until Locutus acknowledges the command with 0xAA. For payloads sent by Locutus, Locutus sends the payload after the acknowledgement.

Commands

Commands fall into three major categories: General, LFS-oriented, and Firmware Upgrade.

Category	Number	Description	PC	MM
General	0x00	Ping	✔	✔
	0x01	Garbage Collect	✔	✔
	0x02	Download and Play	✘	✔
	0x03	Set Configuration	✔	✘
	0x04	Download Error Log	✔	✔
	0x05	Download Crash Log	✔	✔
	0x06	Erase Crash Log	✔	✔
	0x07	*Reserved for Grab / Release Control*	-	-
	0x08-0x0F	*Reserved*	-	-
LFS	0x10	Get Filesystem Statistics	✔	✔
	0x11	Get Dirty Flags	✔	✔
	0x12	Set Dirty Flags	✔	✘
	0x13	Download Global Tables	✔	✔
	0x14	Upload Data Block to RAM	✔	✘
	0x15	Download Data Block from RAM	✔	✘
	0x16	Checksum Data Block in RAM	✔	✘
	0x17	Update Global Directory Table from Data Block in RAM	✔	✘
	0x18	Update Global File Table from Data Block in RAM	✔	✘
	0x19	Create Fork from Data Block in RAM	✔	✘
	0x1A	Read Fork to Data Block in RAM	✔	✘
	0x1B	Update Fork UID	✔	✘
	0x1C	Delete Fork	✔	✘
	0x1D	Delete File	✔	✘
	0x1E	Delete Directory	✔	✘
	0x1F	Reinitialize (reformat) LFS	✔	✘
FW Upgrade	0x20	Query Firmware Revisions	✔	✔
	0x21	Validate Firmware Image in RAM	✔	✘
	0x22	Erase Secondary Firmware	✔	✘
	0x23	Program Secondary Firmware (w/ implied validate)	✔	✘

Locutus operates in three main modes:

- **PC Target Mode:** Locutus is connected to a host, but not a powered Intellivision.
- **Menu Mode:** Locutus is connected to a powered Intellivision, but not playing a game.
- **Game Mode:** Locutus is connected to a powered Intellivision and is playing a game.

Locutus only operates its wire protocol in PC Target Mode and Menu Mode. The **PC** and **MM** columns in the table above indicate which commands Locutus supports in PC Target Mode and Menu Mode.

General Commands

Command 0x00: Ping

Command 0x01: Garbage Collect

The Ping and Garbage Collect commands provide a mechanism to periodically check on Locutus' status.

As far as Locutus is concerned, the Ping command is a pure no-op. Locutus performs no actions in response to a Ping. The Garbage Collect command is similar to Ping; however, it additionally suggests to Locutus to pump its FTL garbage collection process; however, this suggestion is merely a hint.

Both commands take no arguments and always succeed. Both commands return a 32-byte serial data payload:

Bytes		Description
`0x00`	`0x0F`	Device Random Unique IDentifier (DRUID)
`0x10`	`0x1F`	Status and Configuration Flags

The flag bits form a 128-bit vector. Bits 0 - 31 are status flags. Bits 32 - 127 are configuration flags that control the behavior of Locutus. For fields with multiple values, boldface indicates the intended factory default.

Bit Range		Description
`0`		Intellivision power status (0 = off, 1 = on)
1		Errors pending in error log (0 = none)
2		Crash log present (0 = none)
3		*Reserved to indicate grabbed/not-grabbed*
4	31	*Reserved*
32	33	Intellivision II Compatibility Mode ● 00 = Disabled ● 01 = Conservative ● **10 = Aggressive** ● 11 = Reserved (behaves as aggressive)
34	39	*Reserved for extending Intellivision II compatibility. Set to all 1s for now.*

40	41	ECS Compatibility Mode ● 00 = ECS ROMs enabled always (Locutus does nothing) ● 01 = ECS ROMs disabled only for known ECS-incompatible titles ● **10 = ECS ROMs enabled only for known ECS-aware games** ● 11 = ECS ROMs disabled unless absolutely necessary
42	47	*Reserved for extending ECS compatibility mode bits. Set to all 1s for now.*
48	49	Title Screen Mode ● 00 = Never ● 01 = Once per session ● 10 = Reserved (behaves as *once per session*) ● **11 = Always**
50	51	Save Menu Position ● 00 = Never ● **01 = During session only** ● 10 = Reserved (behaves as *during session only*) ● 11 = Always
52		Background GC ● 0 = Disabled ● **1 = Enabled**
53		Keyclicks[5] ● 0 = Disabled ● 1 = Enabled
54		Configuration Menu Enable (FW 4416 and later) ● 0 = Disable configuration menu on console ● **1 = Enable configuration menu on console**
55		Memory Initialization on Game Load[6] ● 0 = Randomized ● **1 = Zeroed**
56	125	*Reserved*
126		Internal to Locutus. Reads as 1, and cannot be modified.
127		Reset to Factory Default: ● 0 = On next boot, reset to factory default configuration ● **1 = Retain current configuration**

[5] The intended factory default was “enabled,” but I believe most went out with “disabled.”
[6] Feature not yet released, but work in progress.

Command 0x02: Download and Play

The Download and Play command expects two things: A file length and a LUIGI file as-is.

```
int download_and_play( uint32_t length );
```

The host provides the file length as an argument, and sends an unmodified LUIGI file as a serial data payload. Locutus returns a success or failure indication plus checksum, as with any other command.

Locutus decodes the LUIGI file directly to external RAM, as if loading it from flash. If Locutus detects an error in the LUIGI file, it politely receives the remaining bytes indicated, and then returns an error. Otherwise, it returns success before switching into Game Mode.

After Locutus switches to Game Mode, it passes control of the serial link to the loaded game. The Locutus Wire Protocol remains inactive until the cartridge resets to PC Target or Menu Mode.

Command 0x03: Set Configuration

The Set Configuration command reconfigures Locutus' behavior. It takes three 32-bit arguments that correspond to configuration/status bits 32 through 127, defined under Ping/Garbage Collect above.

```
int set_configuration( uint32_t bits0, uint32_t bits1, uint32_t bits2 );
```

This call always succeeds. Argument definitions:

- `bits0` corresponds to bits 32..63 of the configuration/status bits
- `bits1` corresponds to bits 64..95 of the configuration/status bits
- `bits2` corresponds to bits 96..127 of the configuration/status bits

Locutus blindly stores the data provided and performs no validation on the received configuration. Software that modifies the configuration bits should preserve the values it finds in reserved fields, in order to ensure compatibility with future firmware. The current firmware sets reserved fields to 1 by default. Locutus is not guaranteed to store reserved fields, so external controllers should not store additional data here.

Configuration bits are stored on flash in dedicated storage.

Command 0x04: Download Error Log

Locutus maintains a circular buffer of error and warning messages generated by various parts of the firmware. This command downloads that circular buffer.

```
int download_error_log( void );
```

This call always succeeds. Locutus sends the error log as a serial data payload.

The serial data payload contains:

- A single byte indicating the current error log index.
- 128 bytes with the error log data.

Locutus treats the error log index as a circular pointer within the 128 byte error log. This pointer initializes to 0 when Locutus resets. The Ping and Garbage Collect commands set the Errors Pending flag whenever the current error log pointer differs from the last-sent error log. After Download Error Log, Locutus updates its notion of the last-sent error position.

Error log entries vary in length from 1 to 4 16-bit words. The first word always has the following format:

Bits		Description
0	11	Error ID within this module.
12	13	Module ID: ● 00 = FTL ● 01 = LFS ● 10 = External Flash ● 11 = LUIGI
14	15	Record length in words: ● 00 = 1 word ● 01 = 2 words ● 10 = 3 words ● 11 = 4 words

For records larger than 1 word, the additional words provide 16-bit values whose meaning depends on the specific error. The tuple `{ error_id, module_id, fw_revision }` identify a specific error message. That, along with the error payload, generally require access to the firmware source to interpret fully.

Most entries in the error log arise from violating a constraint associated with a filesystem command. These errors are benign, and result from the host making an unreasonable request.

If Locutus detects an internal inconsistency, however, it may write one or more errors in the error log.

Command 0x05: Download Crash Log

In the unfortunate circumstance of a firmware crash, Locutus writes a crash log to a dedicated portion of flash. The Download Crash Log command provides a mechanism to retrieve it.

```
int download_crash_log( void );
```

This command always succeeds. Locutus sends the crash log as a serial data payload, even if its empty. The crash log is always *exactly* 131072 bytes.

Command 0x06: Erase Crash Log

This command erases the crash log if present.

```
int erase_crash_log( void );
```

This command always succeeds.

Locutus File System Commands

Most filesystem interactions are broken into two distinct stages: Moving data between the host and Locutus RAM, and moving data between Locutus RAM and filesystem structures. This frees the commands that interact with the filesystem from dealing with data transfer issues to and from the host.

Command 0x10: Get Filesystem Statistics

The Get Filesystem Statistics command collects certain health and usage statistics about the Locutus File System, and returns them as a serial data payload. These same parameters inform Locutus' remaining flash lifetime computation. The command takes no arguments.

```
int lfs_get_filesystem_statistics( void );
```

The serial data payload is 256 bytes, and corresponds to the following C structure:

```
struct lfs_stats
{
    uint16_t   avail_vblocks;       // Available virtual blocks
    uint16_t   total_vblocks;       // Total virtual blocks
    uint16_t   clean_pblocks;       // Number of 'clean' physical blocks
    uint16_t   total_pblocks;       // Total physical blocks
    uint32_t   psect_erasures;      // Number of physical sector erasures
    uint32_t   msect_erasures;      // Number of metadata sector erasures
    uint32_t   log_wrap_count;      // Number of times journal log wrapped
    uint8_t    reserved[236];       // Reserved; filled with zeros
};
```

Command 0x11: Get Dirty Flags

Command 0x12: Set Dirty Flags

Locutus provides a set of 32 dirty flags that an outside program can use to record its progress. The Get Dirty Flags and Set Dirty Flags commands allow the outside program to manipulate these flags.

```
int lfs_get_dirty_flags( void );
int lfs_set_dirty_flags( uint32_t flags );
```

Locutus only assigns minimal meaning to the flags: Any non-zero flag indicates that the filesystem is in an inconsistent state, and that the user should re-synchronize with the management software. Locutus does not assign a meaning to any of the individual bits within the flags.

The Get Dirty Flags command sends the dirty-flag word as a 32-bit serial data payload, rather than as a return value. It always returns success unless it was unable to send the serial data payload.

The Set Dirty Flags command records the new dirty-flag word in the filesystem metadata. It returns success unless it was unable to record the flags in the filesystem metadata.

Command 0x13: Download Global Tables

The Download Global Tables command downloads the global structures that describe the filesystem.

```
int lfs_download_global_tables( void );
```

Locutus sends the Global Directory Table (GDT), Global File Table (GFT), and Global forK Table (GKT) as a serial data payload. This call returns success unless it was unable to send the data payload.

The serial data payload itself is described in On-Media Layout above.

Command 0x14: Upload Data Block to RAM

Uploads data from the host to Locutus RAM.

```
int lfs_upload_data_block_to_ram( uint32_t addr, uint32_t length );
```

Constraints:

- The span [*addr*, *addr + length*) must fit within RAM.
- The *addr* must be even.

Returns an error if the arguments are invalid, or a serial error occurs while receiving the serial data payload. Otherwise, returns success.

The host must upload *length* bytes of data via serial after sending this command. Locutus copies the specified data to the cartridge external RAM.

Command 0x15: Download Data Block from RAM

Downloads data from Locutus RAM to the host.

```
int lfs_download_data_block_from_ram( uint32_t addr, uint32_t length );
```

Constraints:

- The span [*addr, addr + length*) must fit within RAM.
- The *addr* must be even.

Returns an error if the arguments are invalid, or a serial error occurs while sending the serial data payload. Otherwise, returns success.

Locutus sends the requested data as a serial data payload.

Command 0x16: Checksum Data Block in RAM

Compute a CRC32 for a span of Locutus RAM.

```
int lfs_checksum_data_block_in_ram( uint32_t addr, uint32_t length );
```

Constraints:

- The span [*addr, addr + length*) must fit within RAM.
- The *addr* must be even.

Returns an error if the arguments are invalid, or a serial error occurs when sending the checksum as a serial data payload. Otherwise, returns success.

LFS computes the CRC32 checksum of RAM for the indicated range of addresses. It then sends the resulting checksum as a 32-bit serial data payload.

This command is useful for validating a download from the host to RAM, or for comparing the contents of a fork stored on Locutus with a fork on the host without actually downloading it.

Command 0x17: Update Global Directory Table from Data Block in RAM

Updates a portion of the Global Directory Table (GDT) from an image in RAM.

```
int lfs_update_gdt_from_ram( uint32_t first_gdn, uint32_t count,
                             uint32_t addr );
```

Constraints:

- The span [*first_gdn, first_gdn + count*) must be within [0, 127].
- The span [*addr, addr + count*512*) must fit within RAM.
- The *addr* must be even.

Returns an error if the arguments are invalid, or a filesystem error occurs.

Updates the indicated span of the Global Directory Table with records stored in Locutus RAM. This is the mechanism by which external management software populates the GDT.

LFS does not actually validate or interpret the contents of the GDT entries provided by the host. The menu software uses it to construct a menu display.

Command 0x18: Update Global File Table from Data Block in RAM

Updates a portion of the Global File Table (GFT) from an image in RAM.

```
int lfs_update_gft_from_ram( uint32_t first_gfn, uint32_t count,
                             uint32_t addr );
```

Constraints:

- The span [*first_gfn, first_gfn + count*) must be within [0, 1535].
- The span [*addr, addr + count*96*) must fit within RAM.
- The *addr* must be even.

Returns an error if the arguments are invalid, or a filesystem error occurs.

Updates the indicated span of the Global File Table with records stored in Locutus RAM. This is the mechanism by which external management software populates the GFT.

LFS does not actually validate or interpret the contents of the GFT entry provided by the host. The menu software uses it to construct a menu display and to launch programs.

Command 0x19: Create Fork from Data Block in RAM

Creates a new fork in the filesystem, and updates the Global forK Table (GKT).

```
int lfs_create_fork_from_ram( uint32_t gkn, uint32_t addr, uint32_t length,
                              uint32_t uid );
```

Constraints:

- The *gkn* must be within [0, 3071].
- The *gkn* must currently be unallocated.
- The span [*addr, addr + length*) must fit within RAM.
- The *addr* must be even.
- The *length* must be non-zero.
- The filesystem must have sufficient free space.

Returns an error if the arguments are invalid, or a filesystem error occurs.

Creates a new fork with the specified GKN, and assigns it the given 24-bit UID. LFS ignores bits 24 to 31 of the provided UID. LFS does not interpret UIDs; rather, these are primarily for external software's convenience.

Fork creation is atomic. On success, LFS creates the file and updates the GKT as a single atomic action. On failure, the GKT and filesystem contents remain unmodified.

Command 0x1A: Read Fork to Data Block in RAM

Copies data from a fork to Locutus RAM.

```
int lfs_read_fork_to_ram( uint32_t gkn, uint32_t addr,
                          uint32_t fork_ofs, uint32_t length );
```

Constraints:

- The *gkn* must be within [0, 3071].
- The *gkn* must currently be allocated.
- The span [*addr, addr + length*) must fit within RAM.
- The span [*fork_ofs, fork_ofs + length*) must fit within [0, *fork_length*).
- The *addr* must be even.
- The *fork_ofs* must be even.
- The *length* must be non-zero.

Returns an error if the arguments are invalid, or a filesystem error occurs.

Copies the indicated span of data from the fork to Locutus RAM. An external program may then operate on that fork data however it desires. Common operations include downloading the fork or performing a checksum on it.

Command 0x1B: Update Fork UID

Updates the UID field of an existing fork.

```
int lfs_update_fork_uid( uint32_t gkn, uint32_t uid );
```

Constraints:

- The GKN must be within [0, 3071].
- The GKN must currently be allocated.

Returns an error if the arguments are invalid, or a filesystem error occurs.

Copies the new UID to the fork's entry in the Global forK Table (GKT).

Command 0x1C: Delete Fork

Deletes a fork from the filesystem.

```
int lfs_delete_fork( uint32_t gkn );
```

Constraints:

- The *gkn* must be within [0, 3071].
- The *gkn* must currently be allocated.

Returns an error if the arguments are invalid, or a filesystem error occurs.

Deletes the fork data from the filesystem and marks the fork deallocated in the Global forK Table (GKT). Deletion is atomic: On success, the fork is deleted; on failure, the filesystem remains unmodified.

Command 0x1D: Delete File

Deletes a file from the filesystem.

```
int lfs_delete_file( uint32_t gfn );
```

Constraints:

- The *gfn* must be within [0, 1035].

Returns an error if the arguments are invalid, or a filesystem error occurs.

Overwrites the specified entry in the Global File Table (GFT) with `0xFF`, thus marking the file as deleted. Files in LFS are just collections of pointers to forks, along with name metadata. Thus, "deleting" a file does not actually release any storage. To release storage, you must delete the forks as a separate step.

If you delete GFN #0, LFS forces the file type to 0xFD (Directory). This prevents GFN #0 from ever truly being deleted. The file representing the root directory can never be truly deleted.

Command 0x1E: Delete Directory

Deletes a directory from the filesystem.

```
int lfs_delete_directory( uint32_t gdn );
```

Constraints:

- The *gdn* must be within [0, 127].

Returns an error if the arguments are invalid, or a filesystem error occurs.

Overwrites the specified entry in the Global Directory Table (GDT) with 0xFF, thus marking the directory as deleted. Directories in LFS are just collections of file numbers to display, along with a pointer to a parent directory. Thus, "deleting" a directory does not actually release any storage. To release storage, you must delete files and forks as separate steps.

If you delete GDN #0, LFS forces the file parent directory pointer to 0x0000 (itself). This prevents GDN #0 from ever truly being deleted. The directory entry representing the root directory can never be truly deleted.

Command 0x1F: Reinitialize (Reformat) LFS

Reformats the filesystem.

```
int lfs_reinitialize( uint32_t magic );
```

Constraints:

- The *magic* must be the magic number 0x4A5A6A7A.

Returns an error if the arguments are invalid, or a filesystem error occurs.

Erases the entire filesystem, and reinitializes it with a default root directory record.

In FW 2357 and later, Locutus also forcibly wraps the metadata journal log. This pushes a new metadata snapshot to flash and empties the metadata log. That establishes a new baseline for the flash translation layer below the filesystem.

Ordinarily, this is invisible to the user. Metadata log wraps do, however, contribute to Locutus' flash lifetime estimate. If you make many calls to this command, you can sharply reduce Locutus' estimate of remaining flash life, regardless of whether you've *actually* reduced its flash lifetime.

Firmware Upgrade Commands

To upgrade firmware on Locutus:

- Upload the firmware image into Locutus RAM starting at address 0.
- Validate the firmware image in RAM. *(Optional, but recommended.)*
- Erase the secondary firmware image if present.
    - **NOTE:** This step will cause Locutus to reboot into the primary factory firmware image, even if it was already running the primary image!
- Wait for a "L0CUTUS\n" beacon.
- Query firmware revisions. Secondary FW revision should report as all 1s.
- Tell Locutus to program the firmware staged in RAM. (Locutus will revalidate the image before programming.)
    - On success, Locutus will reboot into the new secondary image when done.
    - On failure, Locutus will reboot into the primary image when done, even though it's already running the primary image.
- Wait for a "L0CUTUS\n" beacon.
- Query firmware revisions to validate the upgrade succeeded.

Command 0x20: Query Firmware Revisions

This requests the currently stored firmware revisions in Locutus—primary (aka. "factory") and secondary (aka. "upgrade").

```
int get_firmware_revisions( void );
```

On success, sends a 12-byte serial data payload containing three firmware revisions:

```
int32_t  primary_revision;          // Factory installed immutable firmware
int32_t  secondary_revision;        // Upgrade firmware, or -1 if not present
int32_t  active_revision;           // Currently running firmware
```

Under normal circumstances, the active revision will equal the secondary revision if present, or primary revision otherwise.

The FW version number is intended to represent a unique build of the software, based on the source control repository. Locutus appends two additional bits of information to this number. The FW version number is in bits 31:2, while bits 1 and 0 contain additional information:

- Bit 0 is set for the secondary firmware, and cleared for the primary firmware.
- Bit 1 is set if the firmware was built from a repository with modified files. This indicates that the firmware revision number might not uniquely identify a particular build of the software.

Command 0x21: Validate Firmware Image

This validates a firmware image that was previously staged in RAM starting at address 0.

```
int validate_firmware_image( void );
```

The currently running firmware runs a validation algorithm on Locutus RAM starting at address 0. Returns success (`0x00`) if the RAM contents appear to be a valid firmware upgrade image, and failure (`0xFF`) otherwise.

This is a completely optional step in the firmware upgrade process, as the actual firmware upgrade command repeats this validation.

Command 0x22: Erase Secondary Firmware Image

This prepares Locutus to receive a new secondary firmware image.

```
int erase_secondary_firmware( void );
```

Erases the secondary firmware image if present. This command may be issued when the secondary firmware is present and active. Locutus always reboots to the primary firmware image upon receiving this command, whether or not the secondary firmware is active or present.

After issuing this command, the calling software should wait for a "LOCUTUS\n" beacon before continuing.

Command 0x23: Program Firmware Image

This tells Locutus validate and program a new secondary firmware image.

```
int program_firmware_image( void );
```

This command can only be invoked from the primary firmware image. Returns an error if invoked from the secondary firmware image, or if the secondary firmware has not been erased.

Locutus validates the firmware image stored in RAM starting at address 0. If the image validation succeeds, Locutus programs it into the secondary firmware slot. Upon success, Locutus boots into the new secondary image. Upon failure, Locutus reboots into the primary.

After issuing this command, the calling software should wait for a "LOCUTUS\n" beacon before continuing.

Reference and Miscellaneous Material

Locutus Wire Protocol CRC32

The Locutus wire protocol computes CRC32 in the standard way, similar to how ZIP computes it.

CRC32 Parameters

Locutus CRC32 specified in [Rocksoft format:]

Rocksoft Parameter	Description	Value
`Name`	Algorithm name.	"CRC-32"
`Width`	Field width in bits.	32
`Poly`	CRC polynomial, with leading bit removed.	`0x04C11DB7`
`Init`	Initialization value for checksum.	`0xFFFFFFFF`
`RefIn`	Reflect Input (ie. reverse bit order).	`True`
`RefOut`	Reflect Output (ie. reverse bit order).	`True`
`XorOut`	Value to XOR checksum with at end.	`0xFFFFFFFF`
`Check`	Expected checksum for "123456789".	`0xCBF43926`

Alternate, simplified view:

- Polynomial: `0xEDB88320`. (Reflected polynomial.)
- Initial value: `0xFFFFFFFF`.
- Right-shifting. (This is equivalent to "Reflect Input, Reflect Output" in Rocksoft's nomenclature)
- Final checksum XOR'd with `0xFFFFFFFF`.

CRC32 Reference Implementation

```
#include <stddef.h>
#include <stdint.h>

uint32_t crc32_update_byte(uint32_t crc, const uint8_t byte) {
    crc ^= byte;

    for (int i = 0; i != 8; ++i)
        crc = (crc >> 1) ^ (crc & 1 ? 0xEDB88320u : 0);

    return crc;
}
```

```
uint32_t crc32_block(const uint8_t *const data, const size_t length) {
    uint32_t crc = 0xFFFFFFFFu;

    for (size_t i = 0; i != length; ++i)
        crc = crc32_update_byte(crc, data[i]);

    return crc ^ 0xFFFFFFFFu;
}
```

Locutus User Interface CRC24

The default user interface for LTO Flash! (aka. LUI, the *Locutus User Interface* or *LTO Flash User Interface*) generates UIDs for forks by computing a 24-bit CRC on the contents of the data fork.

Note that Locutus does not ascribe any meaning to the UID field on a fork. LUI, however, uses this CRC24 to generate UIDs from data forks. So, if you wish to reconcile with LUI, this CRC may be useful.

LUI CRC24 Parameters

LUI CRC24 specified in Rocksoft format:

Rocksoft Parameter	Description	Value
`Name`	Algorithm name.	`"LUI CRC-24"`
`Width`	Field width in bits.	`24`
`Poly`	CRC polynomial, with leading bit removed.	`0x864CFB`
`Init`	Initialization value for checksum.	`0xB704CE`
`RefIn`	Reflect Input (ie. reverse bit order).	`False`
`RefOut`	Reflect Output (ie. reverse bit order).	`False`
`XorOut`	Value to XOR checksum with at end.	`0x000000`
`Check`	Expected checksum for "123456789".	`0x21CF02`

Alternate, simplified view:

- Polynomial: `0x864CFB`.
- Initial value: `0xB704CE`.
- Left-shifting.
- Final checksum is not XORed with anything.

LUI CRC24 Reference Implementation

```
#include <stddef.h>
#include <stdint.h>

uint32_t lui_crc24_update_byte(uint32_t crc, uint8_t byte) {
    crc = (crc ^ ((uint32_t)byte << 16)) & 0xFFFFFFu;

    for (int i = 0; i != 8; ++i)
        crc = ((crc << 1) ^ (crc & 0x800000u ? 0x864CFBu : 0)) & 0xFFFFFFu;

    return crc;
}

uint32_t lui_crc24_block(const uint8_t *const data, const size_t length) {
    uint32_t crc = 0xB704CEu;

    for (size_t i = 0; i != length; ++i)
        crc = lui_crc24_update_byte(crc, data[i]);

    return crc;
}
```

Locutus Flash Translation Layer and Flash Lifetime

Locutus incorporates a statically wear-leveled flash translation layer (FTL), along with a dedicated metadata journal log. The metadata journal occupies a different portion of the media than the data itself, and each has different wear characteristics.

Locutus computes the remaining flash life based on a conservative estimate of the number of Erase/Write cycles each flash cell may be subjected to, along with the number of actual erasures applied to both data and metadata portions of the media.

Flash Media Characteristics

Locutus employs a 32MB SPI flash module which provides a mix of 4KB and 64KB erase sectors—32 × 4KB, plus 510 × 64KB. Locutus uses the 4KB sectors for metadata and crash logs, and the remainder for filesystem data. Each 64K erase sector corresponds to 8 PBLKs as exposed by the FTL.

The underlying flash media supports a nominal 100,000 erase/write cycles on each sector.

Main Data Store: 64K Sectors

The main data store consists of 510 64K erase sectors organized as 4080 PBLKs. The FTL advertises a capacity of 3760 VBLKs to the filesystem above it, and reserves 320 PBLKs of capacity for wear leveling. This additional capacity gives the FTL room to reallocate VBLKs to PBLKs, garbage collect, and wear-level.

The Get Filesystem Statistics command refers to these sectors as physical sectors. You can track the number of times these sectors have been erased by examining `psect_erasures` returned by that command.

Copy On Write (COW)

The FTL allows software to update the contents of a VBLK through a mechanism known as *copy on write* (COW). The FTL allocates a new PBLK for the VBLK, and merges the write data with the previous contents of the VBLK. Upon successful completion, the FTL directs future accesses to that VBLK to the newly allocated PBLK and marks the old PBLK as free and dirty.

The FTL records allocation and migration steps in the metadata journal log. COW, combined with the journal, enables atomic updates to the media: if the update gets interrupted before it completes—say, by losing power—the VBLK retains its previous value. The FTL marks newly allocated—but incompletely updated—PBLK as free and dirty.

Wear Leveling

The FTL spreads erasure wear among all PBLKs not currently allocated to VBLKs. Since each physical erase sector holds 8 PBLKs, an incremental garbage collector migrates allocated VBLKs to different PBLKs as needed to pack them into a minimal number of physical erase sectors over time.

FTL only migrates data to pack allocated PBLKs into physical erase sectors as needed. It does not migrate stable data for wear leveling purposes. This strategy is known as *static wear leveling*. With this strategy, FTL spreads erasures over the *free* physical erase sectors rather than all physical erase sectors.

On completely full media, the FTL keeps 40 physical erase sectors (320 PBLKs) in reserve for wear leveling.

Metadata and Crash Store: 32 × 4K Sectors

The metadata and crash store consists of 32 4K erase sectors, divided into three main regions:

- Block Map Snapshots: 4 sectors.
- Metadata Journal: 24 sectors.
- Crash Log: 4 sectors.

The Get Filesystem Statistics command refers to these sectors as *metadata sectors*. You can track the number of times *any* of these sectors has been erased by examining `msect_erasures` returned by that command.

That said, the number of metadata journal log wraps (`log_wrap_count`) is a better estimate of flash health for this region of flash, as explained below.

Block Map Snapshots

The FTL maintains multiple data structures to track which PBLKs are allocated (Physical Block Allocation Map, aka. PBAM), which PBLKs are dirty (Physical Block Dirty Map, aka. PBDM), and the mapping of VBLKs to PBLKs (Virtual to Physical Map, aka. V2PM). The FTL stores a snapshot of the V2PM and PBDM, along with other information, such as configuration bits, in the first few sectors of the metadata area. FTL does not store a copy of the PBAM in flash, as FTL can recreate it at run time from the V2PM.

The FTL actually stores *two* snapshots, each with a version number. The copy with the highest version number is the most recent version. FTL reconstructs the V2PM, PBDM, and PBAM at run time by loading the most recent snapshot, and then replaying the metadata journal to update it. Whenever FTL wraps the metadata journal, it erases the older snapshot and replaces it with a current snapshot.

As a result, FTL erases each snapshot at *half* the rate of the metadata journal—once for every two log wraps.

Metadata Journal

As the FTL updates the media, it records its actions in the metadata journal. The journal records VBLK allocation, deallocation, and migration; changes to configuration words; the start and end of atomic transactions; physical sector erasures; and so on.

The journal contains sufficient information for the FTL to reconstruct the state of the flash from the most recent block map snapshot plus a replay of the journal. The metadata journal log is sized so that it wraps infrequently.

Each entry in the journal cannot be changed once written without erasing an entire 4K sector. FTL appends to the journal until the journal fills up. Once the journal fills up, FTL writes a new block map snapshot and erases the metadata journal. This is referred to as a *log wrap.*

The number of times the metadata journal log wraps also exactly determines the number of times each of the metadata sectors associated with the journal get erased. The journal's metadata sectors see more erasures than any other metadata sector. Therefore, the flash lifetime calculation only considers the number of times the metadata journal log has wrapped.

Crash Log

Locutus dedicates 16K of the metadata portion of flash to store crash information, should the firmware unexpectedly crash. This is an exceedingly rare event, and has only been seen once in production.

Therefore, for purposes of flash lifetime computation, the crash log area does not contribute to Locutus' flash lifetime computation at all.

Lifetime Calculation

As described above, Locutus divides its flash storage into two main zones: data and metadata. Each has distinct wear characteristics, but both are designed to wear evenly.

To compute the remaining flash lifetime for Locutus, compute the lifetime for each zone separately and take the smaller number. Locutus' menu software and the default Locutus User Interface (LUI) use the following formulas based on numbers acquired from the Get Filesystem Statistics command:

- Data Remaining Life = 100% - 100% × `psect_erasures` / 2,000,000[7]
- Metadata Remaining Life = 100% - 100% × `log_wrap_count` / 50,000

[7] The built in menu software actually approximates by replacing 100% × `psect_erasures` / 2000000 with 3% × psect_erasures / 65536. This is roughly equivalent to 100% × `psect_erasures` / 2,164,533.

Each of these computations derates the flash lifetime by a factor of 2 relative to what the flash manufacturer advertises for their flash device.

Data Remaining Life Calculation Detail

The data remaining life calculation assumes that all erasures are spread evenly among only 40 physical erase sectors—that is, it assumes completely full media.

Under these assumptions, the expected flash lifetime for the data store is 40 × 100,000 = 4,000,000 erasures. Derating the flash lifetime to 50,000 E/W cycles reduces this to 2,000,000 erasures.

In practice, the media is rarely full. The FTL wear levels over all free PBLKs. For media that is 20% full, the FTL has around 430 erase sectors it can use for wear leveling. That allows for 43,000,000 erasures before hitting the underlying media limit, well above the advertised maximum of 2,000,000.

Metadata Remaining Life Calculation Detail

As described above, the metadata journal log represents the most-erased portion of the metadata partition. The number of log wraps directly indicates the number of times FTL has erased the metadata journal log sectors.

Therefore, the remaining flash life is a direct function of the number of log wraps. This computation divides by 50,000—rather than the 100,000 the flash claims to support—to derate the flash lifetime by a factor of 2 and provide some engineering margin.

Data Forks Created By Locutus

Most data forks stored in Locutus are created and managed by external software. That said, Locutus does create some data forks of its own to implement all of Locutus' functionality. The following sections describe the data forks Locutus creates as well as their properties.

Menu Position Fork

Locutus' Intellivision menu interface stores menu navigation information in a data fork linked from GFN #0, fork #1—normally reserved for a manual. The menu software creates this fork with size 8192 bytes, and UID = `0xC0FFEE`. If the menu software sees some other fork attached here, it will delete the attached fork and create a new one with these attributes.

The menu software saves navigation information in this fork. Currently, it stores no other data.

JLP Flash Emulation Forks

Locutus emulates JLP's flash save capability through an additional fork—fork #2—that stores the emulated flash data. Locutus connects the flash save state for the currently executing program to fork #2 of the GFN that the menu software launched.

Locutus computes an *ad hoc* checksum of the currently executing LUIGI to assign a UID, and computes a size based on the `jlp_flash` attribute stored in the loaded LUIGI file. Locutus deletes any existing JLP flash fork if the checksum does not match or its size is incorrect, and creates a new one.

Download & Play JLP Flash Emulation Forks

Locutus extends its JLP flash save capability to its Download & Play function. For programs run via Download & Play, Locutus attaches the JLP flash image to GFN #0, fork #2. Locutus reuses this fork if it determines it's safe to do so; otherwise, it deletes the existing fork and creates a new one.

Locutus applies the same criterion—the *ad hoc* checksum and requested flash size—to determine whether to reuse a Download & Play flash save fork. If a game requests JLP flash support, Locutus will create the fork if it does not exist, and otherwise will delete and recreate the fork if the computed UID and size do not match the current fork attached to GFN #0, fork #2.

If a program loaded via Download & Play does not request JLP flash support, Locutus does not examine GFN #0, fork #2.

Revision History

Date	Notes
15-Apr-2019, A	Initial public release.
4-Jul-2019, A	Minor wording and formatting tweaks.