Locutus Universal Intellivision Game Image (LUIGI) V1.0

6-Jul-2019, A


ONLY
ONE WAY

Background **5**
Design Criteria 5
Nomenclature 5
LUIGI Memory Organization Terminology 6
LUIGI File Format **7**
LUIGI File Header 7
Version 0 Header (16 bytes) 7
Version 1 Header (32 bytes) 7
Feature Flags 8
Compatibility Flags 9
Compatibility Field Version Number 10
JLP Flags 10
JLP Accelerator Enable 10
JLP Flash Save Game Size 11
Source Material UID Field (Version 1 and later) 11
Header Checksum: DOWCRC 12
Block Header 12
Block Types 12
Block Type 0x00: Start Encryption 13
Block Type 0x01: Memory Mapping, Permissions, and Page Flipping Tables 13
Memory Mapping Table 13
Permissions Table 14
Page Flipping Table 15
Mattel Page Flip Special Case Address Ranges 16
Interaction between Intellicart-style Bankswitching and Mattel-style Page Flipping 16
Block Type 0x02: Data Hunk 17
8-bit Data Block: 2 + N bytes 17
10-bit Data Block: 2 + ((N + 2) >> 2) + N bytes 17
16-bit Data Block: 1 + 2*N bytes 18
Reserved Encodings 18
Implementation Notes / Observations 18
Encoding Example Code 19
Encoded Example 20
Encoding Efficiency Analysis 20
Block Type 0x03: Program Metadata 21
Metadata Tags 22
Block type 0xFF: End of Data 23

Reference Material **24**
Reference Implementation: IEEE 802.3 CRC32 24
Reference Implementation: DOWCRC 25
Reference Implementation: Castagnoli, Brauer, Hermann CRC32/4 26
CFGVAR Reference 27
CFGVAR Value Type: String 29
CFGVAR Value Type: Number 29
CFGVAR Value Type: Date String 29
Date String Formats 29
Date String Field Definitions 30
CFGVAR Value Type: Compat 30
CFGVAR Value Type: ECS Enable 31
CFGVAR Value Type: Voice Enable 31
CFGVAR Value Type: INTV2 Compat 31
CFGVAR Default Values 31
Recommended License Strings 32
Intellicart Bankswitching Reference 33
Address Spaces 33
[mapping] Data Segments: Non-Bankswitched Address Ranges 33
[bankswitch] Segments: Bankswitched Address Ranges 33
[preload] Data Segments 34
[memattr] Segments 34
Bankswitch Registers 36
Locutus Intellicart Bankswitch Implementation 36
Intellicart RAM to Locutus RAM Mapping 36
Mapping ROM Features onto LUIGI 37
Bankswitch Attribute Lookup and Execution 37
Mattel Page Flipping Reference 38
CFG File Syntax 38
Flipping a Page Mattel Style 38
Empty Pages 38
Reset Behavior 38
Page Flip Attribute Lookup and Execution 38
Flipped Pages in Locutus RAM 39
Revision History **41**

Background

LUIGI is the on-flash format Locutus stores games in. The LUIGI format unifies the BIN+CFG and ROM formats. It simultaneously supports Mattel-style page flipping and Intellicart-style bankswitching. It also provides unique native features.

Design Criteria

LUIGI aspires to the following goals:

- **Compact.** Most games are a multiple of 4K words. Locutus allocates storage in 8K blocks. LUIGI avoids taking more blocks than needed by encoding data efficiently.
- **Simple.** The decoder runs in a constrained environment that is difficult to debug, and has few resources available—especially RAM.
- **Robust against transmission and storage errors.** The format should detect bit errors, and possibly be able to recover from single-bit errors.
- **Support all the things!**
    - Support Mattel page flipping.
    - Support Intellicart bankswitching.
    - Support the full 1MB of Locutus RAM, and perhaps larger RAMs in the future.

Nomenclature

- *Page flipping:* The Mattel technique for selecting among multiple 4K pages of ROM in a 4K window of memory.
- *Bankswitching:* The Intellicart technique for mapping various ranges of Intellicart RAM into one or more 2K-word half-pages.
- *Little Endian:* This stores quantities larger than one byte as a sequence of bytes starting with the least-significant bits. For example, it stores the 32-bit quantity `0xAABBCCDD` as `0xDD`, `0xCC`, `0xBB`, `0xAA`. LUIGI stores all multi-byte quantities in little endian.
- *Big Endian:* This stores bytes in the opposite order as Little Endian. The BIN format stores words in Big Endian order.

Aside: Many resources use the terms *page flipping* and *bankswitching* interchangeably. This document specifically assigns *page flipping* to the Mattel model and *bankswitching* to the Intellicart model. This is consistent with the history of both schemes: The Intellicart uses the `[bankswitch]` CFG section to indicate an Intellicart bankswitched segment. Mattel's documentation refers to its own strategy as paged ROMs.

LUIGI Memory Organization Terminology

Term	Meaning
byte	8 bits
word	16 bits
paragraph (para)	256 words
half-page	2K words (8 paragraphs)
page	4K words (16 paragraphs)
chapter (chap)	A group of pages that map to the same 4K address range.

LUIGI File Format

The LUIGI format consists of the following major sections:

- LUIGI File Header.
- Memory Mapping, Permissions, and Page Flipping Tables.
- Data Hunks.
- Optional Program Metadata.

The LUIGI format takes some inspiration from the PNG file format. It consists of a sequence of largely independent data blocks, each with a generic local header. This allows us to extend the format in the future as needed, while allowing older tools to process the parts they understand while ignoring (or passing through) the new block types.

The LUIGI file header has a fixed format that is not block oriented. This header identifies the file as a LUIGI file along with the LUIGI version. It also provides a set of feature flags, as described below.

LUIGI File Header

Version 0 Header (16 bytes)

Version 0 LUIGI files existed through the early part of Locutus development. This version was officially retired during beta testing. It is documented here for posterity.

Bytes		Field	Details
0	2	Magic Number	Fixed value 0x4C 0x54 0x4F, which is ASCII “LTO”.
3		Version	Fixed value 0x00.
4	11	Feature Flags	A 64-bit vector of feature flags. Corresponds 1:1 to first 64 feature flags in Version 1 LUIGI files.
12	14	*Reserved*	Fill with 0x00.
15		Header Checksum	CRC over entire header.

Version 1 Header (32 bytes)

Version 1 LUIGI files represent the production version of LUIGI. Version 1 differs from version 0 in the following ways:

- Extends feature flags from 64 bits to 128 bits.
- Adds source-material checksums.

Bytes		Field	Details
0	2	Magic Number	Fixed value 0x4C 0x54 0x4F, which is ASCII "LTO".
3		Version	Fixed value 0x01.
4	19	Feature Flags	A 128-bit vector of feature flags.
20	27	Unique ID (UID)	A unique identifier derived from the source material.
28	30	*Reserved*	Fill with 0x00.
31		Header Checksum	CRC over entire header.

Feature Flags

The LUIGI format provides room for up to 128 feature flag bits. LUIGI reserves undefined feature flags. Programs that directly generate new LUIGI files should set reserved fields to 0. Programs that manipulate existing LUIGI files should preserve the contents of reserved fields, to allow forward compatibility.

Bits		Description	CFGVAR[1]	Category
0	1	Intellivoice compatibility	voice_compat	Compatibility
2	3	ECS compatibility	ecs_compat	
4	5	Intellivision 2 compatibility	intv2_compat	
6	7	Keyboard Component compatibility	kc_compat	
8	9	Compatibility field version number	-	
10	11	TutorVision compatibility[2]	tv_compat	
12	15	*Reserved for compatibility flags*	-	
16	17	JLP Accelerator Enable	jlp_accel	JLP
18	21	*Reserved for JLP-related flags*	-	
22	31	JLP Flash Save Game Size (in sectors)	jlp_flash	
32		Enable Locutus' memory mapper at $1000 - $14FF[3]	lto_mapper	Locutus
33	62	*Reserved*		
63		Explicit vs. Implicit feature flags: 0 = Feature flags are the bin2luigi/rom2luigi defaults 1 = Feature flags are explicitly set by the user via CFGVAR or other means.		
64	127	*Reserved (Not present in Version 0 headers.)*		

[1] See the CFGVAR Reference for more details, including aliases for some CFGVARs.
[2] Only if compatibility field version number (bits [9:8]) are 01b or greater. Otherwise, this field is reserved.
[3] Documented in the yet-to-be-released Locutus Programmer's Guide.

Compatibility Flags

For the 2-bit compatibility fields above, LUIGI uses the following encoding:

Bit Pattern	Meaning	Details
00	**Incompatible**	The device *must not* be present when using this program.
01	**Tolerates**	Program operates correctly in the presence of this hardware.
10	**Enhanced**	Program provides extra functionality when this hardware is present.
11	**Requires**	Program requires this hardware to operate correctly.

The compatibility flag bits in the LUIGI header are descriptive, not prescriptive. They describe the contents of the LUIGI file rather than how the LUIGI file should be processed. Locutus' configuration establishes the policy that interprets and acts on these fields.

For example, consider ECS Compatibility. Locutus defines the following four ECS ROM Enable policies:

Bit Pattern	Name of Setting	Details
00	**Always**	ECS ROMs enabled always (Locutus does nothing).
01	**Compatible**	ECS ROMs disabled only for known ECS-incompatible titles.
10	**ECS Games**	ECS ROMs enabled only for known ECS-aware games.
11	**Never**	ECS ROMs disabled unless *absolutely* required.

This leads to the following truth table indicating whether Locutus enables the ECS ROMs based on the ECS compatibility field in the LUIGI file (rows) and the ECS compatibility policy (columns):

		Locutus ECS ROM Enable Policy			
		Always (00)	**Compatible (01)**	**ECS Games (10)**	**Never (11)**
ECS Compat. In LUIGI file	**Incompatible (00)**	ENABLED	disabled	disabled	disabled
	Tolerates (01)	ENABLED	ENABLED	disabled	disabled
	Enhanced (10)	ENABLED	ENABLED	ENABLED	disabled
	Requires (11)	ENABLED	ENABLED	ENABLED	ENABLED

Compatibility Field Version Number

This 2-bit field indicates how to interpret reserved bits `[15:10]`. When set to `00b`, bits `[15:10]` are considered *reserved* and should be ignored.

When the Compatibility Field Version Number is `01b` or higher, bits `[11:10]` become the TutorVision Compatibility field.

JLP Flags

JLP provides set of acceleration functions provided by the JLP cartridge board via memory mapped registers. This includes multiplication, division, CRC, and random number generation. JLP Acceleration also provides an 8000 × 16-bit RAM expansion at `$8040` - `$9F7F`, and the ability to save program data to flash.

Locutus emulates JLP's capabilities. The JLP Flags tell Locutus what JLP functionality to provide.

JLP Accelerator Enable

Bit Pattern	Name of Setting	Details
`00`	**Disabled**	JLP Acceleration support is completely disabled for this program.
`01`	**Accelerators On**	JLP Accelerators and RAM enabled at reset. No JLP Flash support, regardless of *JLP Flash Minimum Size*.[4]
`10`	**Accelerators Off; Flash enabled**	JLP Accelerators and RAM available, but disabled at reset. JLP Flash supported if *JLP Flash Minimum Size* is non-zero.
`11`	**Accelerators On; Flash enabled**	JLP Accelerators and RAM enabled at reset. JLP Flash supported if *JLP Flash Minimum Size* is non-zero..

The Disabled setting (`00b`) disables all support for JLP Accelerators. The program will not see the JLP Accelerators, and can not turn them on.

The remaining three settings enable support for JLP Accelerators. In settings `01b` and `11b`, Locutus makes the JLP Accelerators and associated RAM visible at reset. In setting `10b`, Locutus does not initially make JLP Accelerators visible; however, programs can make them visible as described below. In these modes, Locutus stores JLP Accelerator RAM in Locutus RAM addresses `$10040` - `$11F7F`.

The JLP Accelerators and associated RAM occupy a large fraction of the Intellivision address map. A program can turn JLP Accelerators *off* at run-time with a special JLP-specific page flip request: write `$6A7A` to `$8034`. Likewise, it can turn JLP Accelerators *on* by writing `$4A5A` to `$8033`.

[4] `bin2luigi` upgrades `jlp_accel` = 1; `jlp_flash` > 0 to `jlp_accel` = 3 when encoding a LUIGI.

When a program enables JLP Accelerator support via flags (`jlp_accel` >= `01b`) but accelerators are *off*, CPU accesses to addresses `$8000` - `$9FFF` behave as if JLP Accelerators do not exist.[5] Programs can map other RAM or ROM into this address range—including paged memory—and access it while JLP Accelerators are *off*.

JLP Flash Save Game Size

The JLP boards store flash data in 1.5KB sectors divided into 8 rows of 96 words (192 bytes) each. JLP exposes the amount of flash available through a pair of memory-mapped registers. A JLP board's flash capacity is determined by the total board capacity, minus the space occupied by firmware and the program itself. Programs consult these memory-mapped registers to determine actual flash capacity.

Locutus does not have the same constraints as a JLP board. Locutus supports flash save areas up to approximately 1MB, independent of the size of the program. The *JLP Flash Save Game Size* parameter tells Locutus how much JLP Flash space to allocate. Locutus advertises its configured JLP Flash capacity to programs in the same way JLP would. Locutus only makes JLP Flash storage available when *JLP Accelerator Enable* is `10b` or `11b`.

Each flash sector is 1.5KB. Locutus currently supports JLP Flash save areas up to approximately 1MB. The maximum legal value for *JLP Flash Save Game Size* is **682 sectors**. (682 × 1536 = 1,047,552 bytes)

Source Material UID Field (Version 1 and later)

The UID field assists in identifying the source material that corresponds to a particular LUIGI.

- `bin2luigi`:
  - Original BIN's CRC32 followed by original CFG's CRC32.
  - If no CFG provided, CFG CRC = `0x00000000`.
- `rom2luigi`:
  - Original ROM's CRC32 followed by "`.ROM`" (`0x2E, 0x52, 0x4F, 0x4D`).

Here, CRC32 refers to the standard IEEE 802.3 32-bit CRC used by tools such as ZIP. See Reference Implementation: IEEE 802.3 CRC32 for details.

Tools that directly generate LUIGI files should set this field to a unique ID derived from the source material. This can make it easier to distinguish two otherwise similar looking LUIGI files: Two files with different UIDs are considered different, while two files with the same UID *may* be the same and deserve closer examination.

Programs that set the UID should compute it in such a way that the *same* input produces the *same* UID, while two *different* inputs are highly likely to produce *different* UIDs, much like a good hash function.

Locutus itself does not use the UID field for any purpose.

[5] Other than writing `$4A5A` to `$8033`.

Header Checksum: DOWCRC

Both version 0 and version 1 use DOWCRC from Koopman & Chakravarty's paper for the header checksum. The checksum covers the entire header. See Reference Implementation: DOWCRC for details.

Block Header

Each data block after the header consists of a short, fixed header followed by an optional payload. The block header fields are designed to allow skipping blocks without examining their payloads.

Bytes		Field	Details
0		Block Type	Single byte field indicating the block type.
1	2	Payload Length	Length of the payload associated with this block: 0 .. 65535 bytes.
3		Header Checksum	DOWCRC checksum over { Block Type, Payload Length }.
4	7	Payload Checksum	CRC32/4 checksum for the payload. (Details below.)
8	?	Payload (optional)	Optional payload data associated with this block.

The payload CRC implements CRC32/4 from Castagnoli, Brauer, Hermann. This polynomial generates checksums with a minimum Hamming distance of 4 on all payloads up to 2^{31} - 1 bits, and a minimum Hamming distance of 6 on all payloads up to 5275 bits. See Reference Implementation: Castagnoli, Brauer, Hermann CRC32/4 for details.

Block Types

LUIGI defines the following block types. Other block types are reserved.

Block Type	Description
0x00	Start Encryption
0x01	Memory Mapping, Permissions, and Page Flipping Tables
0x02	Data Hunk
0x03	Program Metadata
0xFF	End of Data

Block Type 0x00: Start Encryption

This block introduces the start of encryption for encrypted LUIGI files.

The first 16 bytes of payload indicate the Device Random Unique IDentifier (DRUID) that the LUIGI file was encrypted for. If the DRUID is all zeros, then the LUIGI is encrypted to run on all Locutus devices.

The remainder of the payload as well as the remainder of the file (including block headers) is encrypted, and therefore cannot be parsed without decrypting it. Therefore, tools that encrypt LUIGI files should move any metadata blocks ahead of this block, so that metadata remains visible after encryption.

Currently, Left Turn Only handles all encryption in-house. To encrypt a program, contact Left Turn Only.

Block Type 0x01: Memory Mapping, Permissions, and Page Flipping Tables

This block provides the initial memory mapping and permissions for the entire memory map. The data payload for this block contains three fixed length tables:

1. Memory Mapping Table (512 bytes).
2. Permissions Table (256 bytes).
3. Page Flipping Table (512 bytes).

This block type should appear exactly once in a LUIGI file, and must be exactly 1280 bytes.

Memory Mapping Table

Each entry in this table corresponds to a 256 word paragraph (para) in the Intellivision memory map, mapping that para to a 256 word window in the Locutus external RAM. Each entry is 1 word (16 bits).

Locutus maps memory accesses to each para's address range to a range in Locutus' external RAM by concatenating the 16-bit value in this table to the lower 8 bits of the Intellivision address. This generates a 24-bit effective address. Expressed as pseudo-C code:

```
para      = (intv_addr >> 8) & 0xFF;        // Extract para from Intellivision address
mmap_addr = mmap_tbl[para];                 // Look up memory mapping (16 bits)
locu_addr = (mmap_addr << 8) | (intv_addr & 0xFF);  // Concatenate address bits
```

Locutus stores two copies of the memory mapping table internally: *initial* and *active*. It updates the active memory mapping table at runtime in response to bankswitch and page-flip requests. When Locutus sees a console reset, it copies the initial memory mapping table to the active table. This reinitializes the program's memory map and restores page-flips and bankswitches to their initial state.

Memory mapping caveats:

- **Locutus only has a 19-bit address space.** Locutus ignores the upper 5 bits of the generated address. Forward-compatible games must put zeros in the additional address bits so that future cartridges can provide greater memory capacity.
- **Forbidden Intellivision addresses.** Locutus does not allow programs to map *readable* memory into the following address spaces. It ignores reads to the following addresses:
    - `0x0000 - 0x04FF`
    - `0x1000 - 0x1FFF`
    - `0x3000 - 0x3FFF`
- **Intellicart Bankswitching.** The original Intellicart's bankswitch state is undefined when the game is loaded. The Intellicart does not change its bankswitch settings across reset. Locutus, however, reloads its memory map across reset, including Intellicart bankswitch settings.

Permissions Table

Each entry in this table corresponds to a 256 word paragraph in the Intellivision memory map, providing access permissions for that paragraph. Each entry is 1 byte (8 bits).

The permissions flags follow the same layout as the Intellicart ROM format:

Bit(s)		Name	Meaning
`0`		READ	Readability: `1` = readable, `0` = not-readable.
1		WRITE	Writability: `1` = writable, `0` = not-writable.
2		NARROW	Paragraph holds Narrow (8-bit) memory. `1` = writes only update lower 8 bits, `0` = writes update all 16 bits.
3		BANKSW	Paragraph is Intellicart-style bankswitched memory.
4	7	n/a	*Reserved.*

The read/write flags directly control whether Locutus responds to reads and writes in the corresponding paragraph. These two flags are fully independent: You can create write-only memory by setting `WRITE = 1` and `READ = 0`, for instance.

The NARROW flag controls whether writes to locations in that paragraph update all 16 bits or only the lower 8 bits. Locutus ignores this bit if `WRITE = 0`. To create 8-bit RAM with the upper 8 bits zeroed, pre-load zeros into the RAM space with data hunks, and then mark the range as `READ = 1`, `WRITE = 1`, `NARROW = 1`.

The BANKSW flag controls Intellicart bankswitching support. Intellicart bankswitching operates on 2K-word half-pages. Locutus only examines paragraphs that reside on 2K half-page boundaries to decide whether to remap memory in response to an Intellicart bankswitching write.

For example, a write to location $004C requests an Intellicart bankswitch on addresses $C000 - $C7FF. Locutus examines the BANKSW permission bit associated with address $C000's paragraph to determine whether to act on that write. It ignores the BANKSW permission bits associated with addresses $C100 - $C7FF. See Intellicart Bankswitching Reference for more details.

As with the memory mapping table, Locutus stores two copies of the permissions table internally: *initial* and *active*. Locutus updates the active permissions table at run time in response to page-flips. When Locutus sees a console reset, it copies the initial permissions table to the active permissions table. This reinitializes the program's memory permissions to their initial state.

Page Flipping Table

The page flipping table holds 256 entries, organized as 16 sets of 16 entries—aka. 16 *chapters* of 16 *pages*. Each set of 16 entries provides mappings for up to 16 4K-word pages associated with a 4K range of the Intellivision address map. Each entry in this table is 1 word (16 bits), divided into three fields:

Bits		Name	Description
0		READ	Readability: 1 = readable, 0 = not-readable.
1		WRITE	Writability: 1 = writable, 0 = not-writable.
2		NARROW	Page holds "narrow" (8-bit) memory. 1 = writes only update lower 8 bits, 0 = writes update all 16 bits.
3		PFE	Page-Flip Enable: Enable Mattel-style page flipping on this page.
4	15	Upper Address	Upper 12 address bits of the Locutus RAM address for the page.

The Page-Flip Enable (PFE) determines whether Locutus responds to page-flip requests in this address range. It should be set consistently across all pages within a given chapter. Locutus examines the PFE bit for the *target* page to decide whether to page-flip.

On a Mattel-style page-flip, Locutus uses the upper address bits to initialize the memory mapping bits for every paragraph in the corresponding 4K page. Each paragraph's memory mapping includes 16 upper address bits. Locutus copies the Upper Address field to bits [15:4] of the memory mapping, and initializes bits [3:0] to the paragraph number within the page.

Locutus copies the 3 permission bits to the paragraph permission flags for all paragraphs within that page. These 3 bits precisely correspond to bits [2:0] of the permissions table entry. Locutus sets bits [7:3] to 0. This clears BANKSW, which is stored in bit 3.

In the Mattel scheme, all page-flipped program ROMs flip to Page 0 at reset. Locutus implements this by reinitializing the memory map and permissions from the Initial Memory Mapping and Initial Permissions tables. To mimic Mattel's scheme, set these tables consistently so they switch to Page 0 on all Mattel page-flipped pages.

Locutus does not modify the page flipping table at run time. It only maintains one copy of the page flipping table.

Mattel Page Flip Special Case Address Ranges

Page	Behavior
$0xxx	Page flip requests ignored.
$1xxx	Page flip requests ignored unless: • lto_mapper = 0 • lto_mapper = 1, and WRITE = 1 for $1F00 - $1FFF.
$4xxx	Page flip requests only affect $4800 - $4FFF, to accommodate ECS RAM at $4000 - $47FF. The target page in Locutus RAM occupies the upper 2K of a 4K-aligned segment.
$8xxx	Page flip requests ignored if JLP Accelerators are currently active: Writes to $8FFF modify JLP RAM at $8FFF, while write to $9FFF are ignored.
$9xxx	

Interaction between Intellicart-style Bankswitching and Mattel-style Page Flipping

The Mattel and Intellicart schemes work very differently from each other. In the Mattel scheme, each 4K page in the address map may have multiple 4K ROM pages mapped to it. The Intellicart scheme exposes a 64K-word backing RAM through one or more flexible 2K-word half-pages in the Intellivision address map. Locutus attempts to make the two schemes coexist.

The active Permissions Table determines whether a given 2K half-page responds to Intellicart bankswitch requests. To establish Intellicart bankswitched segments, set the BANKSW bit in the corresponding paragraphs.

The active Page Flipping Table determines whether a given 4K page responds to Mattel-style page-flip requests. To establish a Mattel-style page-flipped segment, set the PFE bit in the corresponding Page Flipping Table entry.

As long as nothing sets BANKSW and PFE within the same 4K Intellivision address range, the two schemes do not interact. If a program *does* set both bits within an overlapping address range, Locutus honors BANKSW until the first page flip. Locutus clears BANKSW on a page-flip. This disables bankswitching on that range until something sets BANKSW again.

Locutus' behavior across reset differs somewhat from the Intellicart or CC3. The Intellicart and CC3 leave their bankswitched segments *undefined* at startup, although in practice they don't change across reset. In order to support Mattel-style page flipping, Locutus reinitializes its memory mappings to the initial mapping and permissions tables across Intellivision reset. This also resets the Intellicart bankswitched segments.

Block Type `0x02`: Data Hunk

Data hunks provide 16-bit data intended to populate a range of Locutus external RAM. LUIGI packs the data hunk with a scheme that limits the cost of large stretches of 8-bit and 10-bit data, such as is typically found in Intellivision games.

A data hunk payload consists of:

- 3 byte address (24-bit word address)
- Packed data

The data hunk header specifies the *encoded* payload length. The *decoded* word length is implicit in the encoded data.

The packed data consists of a series of blocks of the following types:

- 8-bit data block
- 10-bit data block
- 16-bit data block

A start byte introduces each block. It indicates the block type and length. Each block type specifies how to translate the start byte to the block length.

8-bit Data Block: 2 + N bytes

- Start byte: `0x01` to `0x3F`
  - `N = start_byte`.
- `N-1` 8-bit data bytes.
- One 16-bit word, LS-byte first.

10-bit Data Block: 2 + ((N + 2) >> 2) + N bytes

- Start bytes: `0x40` to `0xBF`
  - `N = start_byte - 0x3F`
- `ceil((N - 1) / 4)` data packets of the following form:
  - MSBs byte:
    - Bits `[7:6]` are MSBs for 1st decle
    - Bits `[5:4]` are MSBs for 2nd decle
    - Bits `[3:2]` are MSBs for 3rd decle
    - Bits `[1:0]` are MSBs for 4th decle
  - Up to 4 more data bytes providing LSBs of decles
  - Last packet is *short*—no 2nd, 3rd, or 4th byte—if `N-1` is not a multiple of 4.
- One 16-bit word, LS-byte first.

16-bit Data Block: 1 + 2*N bytes

- Start byte: `0xC0` to `0xFD`
    - `N = start_byte - 0xBF`
- N 16-bit words, LS-byte first

Reserved Encodings

- Start bytes `0x00`, `0xFE`, and `0xFF` are reserved for future expansion.

Implementation Notes / Observations

Each block type terminates with a 16-bit word. This reduces the overhead due to occasional 16-bit words that appear in otherwise mostly 10-bit program code, such as 16-bit immediate constants in the middle of a stream of 10-bit opcodes.

8-bit blocks exist to compress text and graphic data that is only 8 bits wide. 8-bit blocks provide a net benefit even with fairly short strings. A single 8-bit value followed by a 16-bit word takes 4 bytes to encode (start byte, data byte, trailing word). That uses fewer bytes than opening an equivalent 10-bit or 16-bit block.

Encoding Example Code

```
0240 0100                       MVO     R0,     $100
0040                            SWAP    R0
0240 0101                       MVO     R0,     $100 + 1

02B8 7A5F                       MVII    #$7A5F, R0
0240 7FFF                       MVO     R0,     $7FFF

0004 0154 0042                  CALL    CLRSCR

02BC 8040                       MVII    #$8040, R4
02B9 1F4F                       MVII    #$9F8F - $8040, R1
0004 0154 0046                  CALL    FILLZERO

0002                            EIS

01C0                            CLRR    R0
0240 012E                       MVO     R0,     TSKACT
0240 0102                       MVO     R0,     PRG_RAM_OK

0004 0154 0030                  CALL    WAIT
0001                            DECLE   1

0004 0154 0051      @@loop:     CALL    DO_MENU
56A8                            DECLE   MAIN_MENU

0004 0150 0366                  CALL    RUNQ
0000                            HLT
0220 0009                       B       @@loop
```

Encoded Example

```
46                  10-bit block, N = 7
92 40 00 40 40          Decles 240 100 040 240
60 01 B8                Decles 101 2B8
5F 7A                   Word 7A5F

41                  10-bit block, N = 2
80 40                   Decle 240
FF 7F                   Word 7FFF

44                  10-bit block, N = 5
12 04 54 42 BC          Decles 004 154 042 2BC
40 80                   Word 8040

41                  10-bit block, N = 2
80 B9                   Decle 2B9
4F 1F                   Word 1F4F

50                  10-bit block, N = 17
10 04 54 46 02          Decles 004 154 046 002
66 C0 40 2E 40          Decles 1C0 240 12E 240
44 02 04 54 30          Decles 102 004 154 030
04 01 04 54 51          Decles 001 004 154 051
A8 56                   Word 56A8

45                  10-bit block, N = 6
1C 04 50 66 00          Decles 004 150 336 000
80 20                   Decle 220
09 00                   Word 0009
```

Encoding Efficiency Analysis

LUIGI encoded this example in 62 bytes. The original input was 39 words (78 bytes)—34 holding decles, and 5 holding 16-bit words. The 62 byte encoded example consists of:

- 6 bytes block headers
- 56 bytes payload

If we could somehow encode the data with no framing at all—that is, pack 10-bit data so that 4 decles takes 5 bytes, and 16-bit data takes 2 bytes—then the example above would require 53 bytes total. LUIGI's decle/word payload encoding is therefore nearly ideal: 56 bytes vs. 53 bytes. The block framing on top of that incurs an additional ~11% overhead.

For a pure 10-bit ROM, the block framing adds less than 1.25% overhead: 2 bytes per 128 decle block. A 128 decle block encodes to 162 bytes: 159 bytes of optimally packed payload, 1 header byte, and 2 bytes of 16-bit word.

For long 8-bit segments, the block framing overhead approaches 3.2%, since the maximum 8-bit block size is 63 bytes. A 63-byte block of 8-bit data requires 65 total bytes: 1 header byte, 62 data bytes, and 2 bytes of 16-bit word.

For long 16-bit segments, the block framing overhead approaches 0.8%, since the maximum 16-bit block is 62 words long. A 62-word block requires 125 bytes: 1 byte of framing, and 124 bytes holding the 62 16-bit words.

Block Type `0x03`: Program Metadata

This block acts like an ID3 tag, providing information about the program contained in the LUIGI file. This block is optional. Locutus currently ignores it; however, the GUI may use it to populate the menu. If/when Locutus does parse this metadata, it will be for display purposes only. No tag affects how Locutus interprets the game. All tags are optional.

The block payload consists of a series of sub-records. Each sub-record begins with a 1 byte tag and 1 byte length. That is followed by up to 255 bytes of data, as indicated by the length byte.

> *Note:* Each tag has a maximum payload length of 255 bytes. `bin2luigi` and `rom2luigi` truncate longer tags to fit within 255 bytes. For *Miscellaneous* tags, the 255 byte limit applies to the combined payload comprising variable name, `'='` delimiter, and value string.

Most tags hold strings. LUIGI supports UTF-8 string data. However, Locutus' menu software is only able to display the ASCII subset. Therefore, the Locutus GUI may discard or modify UTF-8 characters outside the ASCII range `0x20` - `0x7E` when populating the menu from LUIGI metadata.

Each tag may appear more than once. This allows a LUIGI to specify multiple authors, multiple release dates, and so on.

For most tags, jzIntv and other software tracks all of the values associated with the tag. Two tags, however—*Program Name* (`0x00`) and *Program Short Name* (`0x01`)—only permit one value. If these tags appear more than once, LUIGI does not define which instance takes precedence. The *Single / Multi* column in the summary table below indicates which tags support single values vs. multiple values.

Metadata Tags

Tag	Description	Encoding	Equivalent CFGVAR	Single / Multi
0x00	Program Name	String	name	Single
0x01	Program Short Name	String	short_name	Single
0x02	Author	String	author	Multi
0x03	Publisher	String	publisher	Multi
0x04	Date Released	Numeric	year / release_date	Multi
0x05	License	String	license	Multi
0x06	Description	String	description	Multi
0x07	Miscellaneous	String	(Any unmatched variable)	n/a
0x08	Game Artist	String	game_art_by	Multi
0x09	Music Composer	String	music_by	Multi
0x0A	Sound Effects Creator	String	sfx_by	Multi
0x0B	Voice Actor	String	voices_by	Multi
0x0C	Documentation Writer	String	docs_by	Multi
0x0D	Concept Creator	String	concept_by	Multi
0x0E	Box/Manual/Overlay Artist	String	box_art_by	Multi
0x0F	More Information Pointer	String	more_info_at	Multi

LUIGI encodes unmatched CFGVARs under the *Miscellaneous* tag as a string of the form "`var=value`". It concatenates the variable name, a single `'='`, and the string form of the value. This encoding is unambiguous, as variable names cannot contain `'='`.

LUIGI encodes the release date with a variable-length encoding. The date format defines 8 1-byte fields, as shown below. The release date tag itself may hold fewer than 8 bytes. The missing bytes are assumed to be 0. This allows the user to specify dates with different degrees of precision. For example, a release date tag with 1 byte only specifies the release year.

Byte	Meaning	Valid range
0	Year - 1900	0 - 255
1	Month	1 - 12
2	Day	1 - 31
3	Hour	0 - 23

4	Minute	0 - 59
5	Second	0 - 60
6	Timezone Offset Hours	-12 - +12
7	Timezone Offset Minutes	0 - 59

The timezone offset is given by *Hours**60 + *Minutes*. *Minutes* is always *positive*. LUIGI encodes the timezone -0130 as -2 *Hours*, +30 *Minutes*.

The seconds field allows values as large as 60 to account for leap seconds.

Block type 0xFF: End of Data

This isn't actually a block type. If the LUIGI decoder sees 0xFF where it expects a block type, it stops reading the file immediately and does not read any further.

Block type 0xFF allows streaming a raw LUIGI file in environments where it's not possible to inform the decoder ahead of time what the actual file length is. In serial streaming applications, the sender should not send any more bytes associated with the LUIGI after sending block type 0xFF.

Reference Material

Reference Implementation: IEEE 802.3 CRC32

- Polynomial: `0xEDB88320`.
- Right shifting.
- Initial value: `0xFFFFFFFF`.
- Inverted result.

```
uint32_t ref_ieee802_crc32_update(uint32_t crc, const uint8_t byte) {
    int i;

    crc ^= byte;

    for (i = 0; i < 8; i++) {
        crc = (crc >> 1) ^ (crc & 1 ? 0xEDB88320ul : 0);
    }

    return crc;
}

uint32_t ref_ieee802_crc32_block(const uint8_t *const block,
                                 const int length) {
    uint32_t crc = 0xFFFFFFFFul;
    int i;

    for (i = 0; i < length; i++) {
        crc = ref_ieee802_crc32_update(crc, block[i]);
    }

    return crc ^ 0xFFFFFFFFul;
}
```

Test vectors:

Input	Output
0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F	0xCECEE288
0x4A 0x5A 0x6A 0x7A	0x9B04D72C
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00	0x6522DF69
0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF	0x2144DF1C

Reference Implementation: DOWCRC

This CRC is described in [Koopman & Chakravarty's paper](#).

- Polynomial: `0x98`.
- Right shifting.
- Initial value: `0x00`.
- Non-inverted result.

```
uint8_t ref_dowcrc_update(uint8_t crc, const uint8_t byte) {
    int i;

    crc ^= byte;

    for (i = 0; i < 8; i++) {
        crc = (crc >> 1) ^ (crc & 1 ? 0x98 : 0);
    }

    return crc;
}

uint8_t ref_dowcrc_block(const uint8_t *const block, const int length) {
    uint8_t crc = 0;
    int i;

    for (i = 0; i < length; i++) {
        crc = ref_dowcrc_update(crc, block[i]);
    }

    return crc;
}
```

Test vectors:

Input	Output
0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F	0x00
0x4A 0x5A 0x6A 0x7A	0xB8
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00	0x00
0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF	0x84

Reference Implementation: Castagnoli, Brauer, Hermann CRC32/4

This CRC is described in Castagnoli, Brauer, Hermann.

- Polynomial: `0x82F63B78`.
- Right shifting.
- Initial value: `0x00000000`.
- Non-inverted result.

```
uint32_t ref_crc32_4_update(uint32_t crc, const uint8_t byte) {
    int i;

    crc ^= byte;

    for (i = 0; i < 8; i++) {
        crc = (crc >> 1) ^ (crc & 1 ? 0x82F63B78ul : 0);
    }

    return crc;
}

uint32_t ref_crc32_4_block(const uint8_t *block, const int length) {
    uint32_t crc = 0;
    int i;

    for (i = 0; i < length; i++) {
        crc = ref_crc32_4_update(crc, block[i]);
    }

    return crc;
}
```

Test vectors:

Input	Output
0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F	0x9BB99201
0x4A 0x5A 0x6A 0x7A	0x02CB247E
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00	0x00000000
0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF	0xC44FF94D

CFGVAR Reference

jzIntv and its related tools such as `bin2rom`, `rom2bin`, `bin2luigi`, `luigi2bin`, `rom2luigi`, and `as1600` make use of configuration variables (CFGVARs) to carry metadata that describes the program. In the BIN+CFG format, these variables appear in the `[vars]` section of the CFG file. Likewise, `as1600` provides the CFGVAR directive to specify a configuration variable. This specification refers to these variables as CFGVARs.

No formal specification exists for CFGVARs. jzIntv *et al*, however, define a set of CFGVARs they understand and assign meaning to.

Some CFGVARs are *multi-valued*, meaning that the CFGVAR can appear more than once to specify multiple values for the variable. The tools track all of the provided values. This makes it possible to specify multiple publishers, authors, release dates, and so on, when it makes sense to do so.

If a non-multi-valued CFGVAR appears more than once (either in CFGVAR directives in an assembly file, or as variables in a `[vars]` section of a CFG file), one of the provided values is kept. However, *which* value is kept is unspecified.

CFGVAR	Value Type	Multi-valued	Description
`name`	String	No	Name of the program. Locutus recognizes the first 60 characters.
`short_name`	String	No	Short-hand name of the program. Locutus recognizes the first 18 characters.
`publisher`	String	Yes	Name of the program's publisher.
`author`	String	Yes	Author / programmer of the program.
`game_art_by`	String	Yes	Artist for the in-game artwork.
`music_by`	String	Yes	Composer and/or arranger for in-game music.
`sfx_by`	String	Yes	Sound effects creator.
`voices_by`	String	Yes	Voice actors for in-game voices.
`docs_by`	String	Yes	Author of the game documentation.
`concept_by`	String	Yes	Originator of the game concept. (e.g. Carol Shaw for River Raid)
`box_art_by`	String	Yes	Artist for the game box, overlays, manuals, etc.
`more_info_at`	String	Yes	Location (e.g. URL) where more information can be found

			about this program.
year	Number	Yes	The year this program was released.
release_date	Date String	Yes	The date this program was released.
build_date	Date String	Yes	The date this program was built / compiled.
license	String	Yes	The license under which this program is offered. See Recommended License Strings.
description desc	String	Yes	A description of this program.
version	String	Yes	Version of the program. This is a freeform string that's meant to be used to distinguish different variations of a program.
ecs_compat	Compat	No	The program's compatibility with the ECS.
ecs	ECS Enable	No	Specifies whether to enable ECS when loading this program. jzIntv and LUIGI map this onto an ecs_compat value.
voice_compat	Compat	No	The program's compatibility with the Intellivoice.
voice	Voice Enable	No	Specifies whether to enable Intellivoice when loading this program. jzIntv and LUIGI map this onto voice_compat.
intv2_compat	Compat	No	The program's compatibility with Intellivision II.
intv2	INTV2 Compat	No	The program's compatibility with Intellivision II.
kc_compat	Compat	No	The program's compatibility with the Keyboard Component.
tv_compat	Compat	No	The program's compatibility with the TutorVision.
lto_mapper	0 or 1	No	Specifies whether to enable the LTO memory mapper.
jlp_accel jlp	0 - 3	No	Specifies whether to enable JLP's accelerators. See JLP Accelerator Enable for encoding.
jlp_flash	Number	No	Specifies minimum number of 1.5K JLP flash sectors the program expects for JLP flash support. See JLP Flash Save Game Size for interpretation.

CFGVAR Value Type: String

Strings are the most generic CFGVAR value type. Historically, the tools support ASCII strings here.

In CFG files, strings may appear in quotes. However, quotes are optional if the string does not contain any characters from the following set, or characters outside the range `0x21` - `0x7E`.

`;` **`[`** **`]`** **`$`** **`=`** **`-`** **`,`** **`\`** *<space>* *<tab>*

Prior to SVN revision 1862, jzIntv *et al* had poor support for non-printing characters in strings. As of SVN revision 1862, jzIntv *et al* now *escape* non-printing characters when generating CFG files. When consuming CFG files, jzIntv *et al* interpret escape sequences of the form **`\`***nnn*, where *nnn* is an octal value, and **`\x`***nn*, where *nn* is a hexadecimal value.

As of SVN revision 1862, jzIntv *et al* also support UTF-8 for string-valued variables. Locutus' built-in menu software, however, does not understand UTF-8 characters. Programs should constrain themselves to 7-bit ASCII for `name` and `short_name`.

LUIGI limits strings to 255 bytes. `bin2luigi` and `rom2luigi` truncate strings to fit within 255 bytes.

CFGVAR Value Type: Number

This value type represent pure numeric values.

The CFG parser used by jzIntv *et al* performs a fuzzy match on a parsed value to determine its radix. It interprets values that begin with a `$` as hexadecimal values. It interprets values that consist solely of the digits `0` - `9` as decimal. It interprets values that consist of the digits `0` - `9` *and* `A` - `F` as hexadecimal.

CFGVAR Value Type: Date String

CFGVAR date strings provide variable resolution dates. At their least precise, they specify a year. At their most precise, they specify the time down to the second, in a particular timezone specified to the minute.

Any date string that includes more than just the year must appear in quotes. jzIntv *et al* also allow **`'/'`** in place of **`'-'`** in dates; however, you cannot mix **`'-'`** and **`'/'`** in the date portion of the string.

Date String Formats

YYYY	Year only
`"`*YYYY*`-`*MO*`"`	Year and month
`"`*YYYY*`-`*MO*`-`*DD*`"`	Year, month, and day

"YYYY-MO-DD HH"	Year, month, day, and hour
"YYYY-MO-DD HH:MI"	Year, month, day, hour, and minutes
"YYYY-MO-DD HH:MI:SS"	Year, month, day, hours, minutes, and seconds
"YYYY-MO-DD HH:MI:SS +hh"	Year, month, day, hours, minutes, seconds, and timezone hours
"YYYY-MO-DD HH:MI:SS +hhmm"	Year, month, day, hours, minutes, seconds, timezone hours and minutes.
"YYYY-MO-DD HH:MI:SS +hh:mm"	

Date String Field Definitions

Field	Definition
YYYY	Year. Recommend 4 digits to avoid ambiguity. Years in the range 0 to 99 will have 1900 added. Years in the range 100 to 1900 are invalid.
MO	One or two digit month.
DD	One or two digit day.
HH	One or two digit hour in 24-hour time.
MI	One or two digit minutes.
SS	One or two digit seconds.
+hh	Two digit timezone hour offset. **-hh** for timezones west of UTC, and **+hh** for timezones east of UTC.
mm	Two digit timezone minute offset. (May be one digit if colon is present.)

CFGVAR Value Type: Compat

Compatibility CFGVARs are numeric values in the range 0 to 3. Values outside this range are undefined.

Value	Meaning	Details
0	**Incompatible**	The device must not be present when using this program.
1	**Tolerates**	Program operates correctly in the presence of this hardware.
2	**Enhanced**	Program provides extra functionality when this hardware is present.
3	**Requires**	Program requires this hardware to operate correctly.

CFGVAR Value Type: ECS Enable

The `ecs` CFGVAR is a 0 or 1 value that specifies whether the program requires the ECS. This CFGVAR is inherited from INTVPC. When set to 1, INTVPC enables the ECS for the program. jzIntv *et al* map it onto `ecs_compat` as follows:

ecs	ecs_compat
0	1 (Tolerates)
1	3 (Requires)

CFGVAR Value Type: Voice Enable

The `voice` CFGVAR is a 0 or 1 value that specifies whether the program requires the Intellivoice. This CFGVAR is inherited from INTVPC. When set to 1, INTVPC enables the Intellivoice for the program. jzIntv *et al* map it onto `voice_compat` as follows:

voice	voice_compat
0	1 (Tolerates)
1	2 (Enhanced)

CFGVAR Value Type: INTV2 Compat

The `intv2` CFGVAR is identical to `intv2_compat`, except that it is restricted to the values 0 or 1. It specifies whether the program is compatible with the Intellivision II.

intv2	intv2_compat
0	0 (Incompatible)
1	1 (Tolerates)

CFGVAR Default Values

jzIntv and LUIGI assign default values to a subset of the CFGVARs. That is, if the CFG leaves the variable unspecified, these tools pick a default value. In LUIGI, the *Explicit vs. Implicit feature flag bit* indicates whether these parameters were populated from the defaults.

CFGVAR	Default	Notes
ecs_compat	1 (Tolerates)	Equivalent to `ecs = 0`
voice_compat	1 (Tolerates)	Equivalent to `voice = 0`
intv2_compat	1 (Tolerates)	Equivalent to `intv2 = 1`
kc_compat	1 (Tolerates)	
tv_compat	1 (Tolerates)	
jlp_accel	0 if `jlp_flash = 0` 2 if `jlp_flash > 0`	If the config supplies *exactly one* of `jlp_accel` or `jlp_flash`, the *supplied* flag influences the default value for the *omitted* flag.
jlp_flash	0 if `jlp_accel = 0` 4 if `jlp_accel >= 2`	

Recommended License Strings

While the license CFGVAR is a free-format string, programmers should consider using the following strings for well-known licenses. For the Creative Commons licenses, consider appending the version number as well—e.g. `CC BY-ND-SA 4.0`.

License String	License
GPLv2	GNU General Public License, Version 2
GPLv2+	GNU General Public License, Version 2 or later
GPLv3	GNU General Public License, Version 3
GPLv3+	GNU General Public License, Version 3 or later
BSD2	BSD 2 Clause
BSD3	BSD 3 Clause
CC CC0	Creative Commons, no restrictions
CC BY	Creative Commons, attribution alone
CC BY-SA	Creative Commons, attribution, share alike
CC BY-NC	Creative Commons, attribution, non-commercial use
CC BY-ND	Creative Commons, attribution, no derivatives
CC BY-NC-SA	Creative Commons, attribution, non-commercial use, share alike
CC BY-ND-SA	Creative Commons, attribution, no derivatives, share alike

Intellicart Bankswitching Reference

The Intellicart[6] provides 64K words of RAM for programs. It also provides a memory mapping scheme—that jzIntv and LUIGI refer to as *bankswitching*—to make all of that memory available to the Intellivision. Refer to the official Intellicart documentation and/or the IntelliWiki Intellicart page for additional documentation. This section contains a summary.

Address Spaces

The Intellicart operates in *two* address spaces: The Intellivision address space as seen by the CP1610 CPU, and the Intellicart address space. The Intellicart examines every memory access the CPU makes, and determines:

- Whether to respond to the access.
- What address in Intellicart address space the Intellivision address maps to.

The Intellicart treats Intellivision addresses `$0040` - `$005F` specially: it places its bankswitch registers in this address range. Writes to this address range control Intellicart bankswitching. The Intellicart does not respond to reads in this address range.

`[mapping]` Data Segments: Non-Bankswitched Address Ranges

`[mapping]` data segments in a CFG file specify blocks of data to load and map directly into the Intellivision address map. Each segment consists of a range of offsets in the BIN file, and a target address to map to in the Intellivision address map. These segments are *not* bankswitched.

Example:

```
[mapping]
$0000 - $0FFF = $5000    ; Map first 4K words of BIN to Intellivision address $5000.
```

For non-bankswitched addresses, the Intellicart sets the Intellicart address exactly equal to the Intellivision address. Suppose a program maps the half-page at `$5000` - `$57FF` as non-bankswitched. This half-page appears at `$5000` - `$57FF` in the Intellivision address map, and also resides at `$5000` - `$57FF` in the Intellicart RAM.

`[bankswitch]` Segments: Bankswitched Address Ranges

`[bankswitch]` segments specify address ranges in the Intellivision address map that should be bankswitched. The Intellicart rounds these address ranges up to half-page boundaries.

`[bankswitch]` segments do not specify any data to load. Use `[preload]` data segments (below) to load data for use with `[bankswitch]` segments.

[6] And CC3. This section applies equally to the Intellicart and CC3.

Example:

```
[bankswitch]
$6000 - $67FF              ; Mark Intellivision address $6000 - $67FF as bankswitched.
```

The Intellicart remaps bankswitched Intellivision addresses to the Intellicart RAM as follows:

- Look up the bankswitch offset for this 2K half page. That is, index based on the upper 5 address bits of the Intellivision address (bits [15:11]).
- Zero out the upper 5 Intellivision address bits.
- Add the bankswitch offset to the modified Intellivision address (modulo 64K) to compute the Intellicart address.

Pseudo-code:

```
half_page  = (intv_addr >> 11) & 0x1F;       // Extract half-page number (0 - 31).
bsw_offset = bsw_table[half_page] << 8;      // Look up bankswitch offset (8 bits).
icart_addr = ((intv_addr & 0x07FF) + bsw_offset) & 0xFFFF;  // Apply offset.
```

[preload] Data Segments

[preload] data segments in a CFG file specify blocks of data to load from the BIN file directly into Intellicart RAM. Each segment consists of a range of offsets in the BIN file, and a target Intellicart RAM address.

Example:

```
[preload]
$1000 - $1FFF = $3000    ; Preload words $1000 - $1FFF from BIN into Intellicart at $3000.
```

Unlike [mapping] data segments, [preload] data segments do not modify the Intellivision address map. Rather, they exist solely to populate Intellicart RAM. They are useful for initializing data that might be accessed through a bankswitched half-page later. They are also useful for initializing RAM that is marked *narrow,* by pre-zeroing the upper byte.

bin2luigi extends the interpretation of [preload] segments. It allows preload target addresses anywhere within Locutus' 512K word address map ($00000 - $7FFFF), rather than restricting to the Intellicart's 64K address range.

[memattr] Segments

[memattr] segments in a CFG file change the memory attributes of an Intellivision address range without loading data into them. Each segment consists of an address range, a memory type, and memory width. Like [bankswitch] segments, [memattr] segments only modify memory attributes and do not load data.

Example:

```
[memattr]
$8000 - $9FFF = RAM 16   ; 16-bit RAM at addresses $8000 - $9FFF.
```

The Intellicart and CC3 support the memory types below. Only CC3 and Locutus support Narrow 8-bit RAM and WOM.

Type Designation	Meaning
RAM 8	Readable, Writable, Narrow (8-bit) memory
WOM 8	Writable, Narrow (8-bit) memory. (WOM = Write Only Memory)
ROM 8	Readable, Narrow[7] memory. (ROM = Read Only Memory)
RAM 16	Readable, Writable memory.
WOM 16	Writable memory.
ROM 16	Readable memory.

Usage note: You can configure WOM underneath Intellivision components such as the GRAM, to capture any writes made to that address range. The WOM will *not* interfere with other devices as it only responds to writes.

[7] The Narrow attribute is meaningless here as it only affects writes.

Bankswitch Registers

The Intellicart provides 32 byte-wide, write-only bankswitch registers. Each register controls the bankswitch offset for a different 2K half-page. The Intellicart only examines bits [7:3] of each write.

Register Address	Address Range		Register Address	Address Range
$40	$0000 - $07FF		$50	$0800 - $0FFF
$41	$1000 - $17FF		$51	$1800 - $1FFF
$42	$2000 - $27FF		$52	$2800 - $2FFF
$43	$3000 - $37FF		$53	$3800 - $3FFF
$44	$4000 - $47FF		$54	$4800 - $4FFF
$45	$5000 - $57FF		$55	$5800 - $5FFF
$46	$6000 - $67FF		$56	$6800 - $6FFF
$47	$7000 - $77FF		$57	$7800 - $7FFF
$48	$8000 - $87FF		$58	$8800 - $8FFF
$49	$9000 - $97FF		$59	$9800 - $9FFF
$4A	$A000 - $A7FF		$5A	$A800 - $AFFF
$4B	$B000 - $B7FF		$5B	$B800 - $BFFF
$4C	$C000 - $C7FF		$5C	$C800 - $CFFF
$4D	$D000 - $D7FF		$5D	$D800 - $DFFF
$4E	$E000 - $E7FF		$5E	$E800 - $EFFF
$4F	$F000 - $F7FF		$5F	$F800 - $FFFF

Locutus Intellicart Bankswitch Implementation

Intellicart RAM to Locutus RAM Mapping

Locutus maps Intellicart RAM to Locutus RAM addresses $00000 - $0FFFF.

The Intellicart and CC3 compute all bankswitch address translations modulo 64K. While the scheme *could* be extended to larger address spaces, no prior implementation exists with more than 64K. Therefore, Locutus implements the same 64K restriction, and masks all translated addresses to 16 bits.

Mapping ROM Features onto LUIGI

`bin2luigi` and `rom2luigi` adhere to the Intellicart's rules for `[mapping]` and `[preload]` data segments, with the following exceptions:

- `bin2luigi` allows a `[preload]` data segment to specify a target address anywhere in Locutus' 512K-word address map, rather than restricting to the first 64K.
- `bin2luigi` preloads paged data segments at the top of Locutus RAM rather than within the Intellicart RAM space. Paged data segments are outside the Intellicart memory mapping model.

Bankswitch Attribute Lookup and Execution

When Locutus sees a write to an Intellicart bankswitch register (Intellivision addresses `$40` - $5F), it examines the Permissions Table to determine whether to respond to the write.

The Permissions Table has 256 entries—one for each 256 word paragraph. When looking up bankswitch permissions, Locutus only looks at the first paragraph in a half-page. Locutus computes the lookup index in a manner similar to the following pseudo-code:

```
// Get the half-page number (0-31). Decoding is squirrely due to how the Intellicart rotates the bits.
half_page = ((intv_addr & 0xF) << 1) | ((intv_addr & 0x10) >> 4);
para      = half_page << 3;        // Convert to paragraph number (0-255).
perms     = perm_tbl[para];        // Look up permission bits.
is_banksw = (perms >> 3) & 1;      // Extract BANKSW bit.
```

If the corresponding BANKSW bit is set in the Permissions Table, Locutus rewrites the appropriate entries in the Memory Mapping Table accordingly. Pseudo-code:

```
if (is_banksw) {
  const uint16_t bsw_offset = intv_data;
  for (int i = 0; i < 8; i++) {
    // Compute memory mapping for bankswitch. Recall that mmap_tbl[] provides bits 23:8 of
    // the Locutus RAM address, so there's no address shift here. The byte mask limits us to 64K.
    mmap_tbl[para + i] = (bsw_offset + i) & 0xFF;
  }
}
```

Mattel Page Flipping Reference

The Mattel page flipping scheme maps up to 16 4K pages into a single 4K page of the Intellivision address map. LUIGI refers to these 16 pages as a *chapter*.[8]

CFG File Syntax

jzIntv and `bin2luigi` extend the CFG format to comprehend Mattel paged ROMs. In the `[mapping]` section, programs indicate a paged ROM segment with the PAGE keyword, followed by a page number, as in the following example:

```
[mapping]
$0000 - $0FFF = $5000             ; Non-paged ROM segment
$1000 - $1FFF = $E000 PAGE 0      ; Paged ROM segment
$2000 - $2FFF = $E000 PAGE 1      ; Paged ROM segment
```

Flipping a Page Mattel Style

In the Mattel scheme, programs request a page flip on a 4K page by writing $*x*A5*y* to location $*x*FFF, where:

- *x* is the upper 4 bits of the 4K address range being flipped.
- *y* is the page (0 - 15) to flip to within the chapter.

For example, writing `$EA51` to location `$EFFF` flips to `$E000 PAGE 1`, as x = `$E`, and y = `$1`.

Empty Pages

Mattel's scheme accounts for distributed address decode, and the possibility that a given page might not be present in a chapter. It's perfectly legal to flip to a not-present page. Doing so does not affect future page-flips; rather, it just deselects all present ROMs in the corresponding 4K page.

Reset Behavior

All paged ROM segments flip to page 0 at reset, regardless of whether page 0 in a given chapter is empty. Locutus implements this behavior by copying the initial Memory Mapping Table to the active Memory Mapping Table.

Page Flip Attribute Lookup and Execution

Locutus relies on the Page Flipping Table to determine whether to respond to a page flip request. It looks up the Page Flip Enable (PFE) in a manner similar to the following pseudo-code in response to a write:

[8] Chapter is LUIGI's term, not Mattel's.

```
is_pageflip = 0;                                          //  Default: Not a page-flip.
if ((intv_addr & 0x0FFF) == 0x0FFF &&                     //  Last word on page.
    (intv_addr & 0xF000) == (intv_data & 0xF000) &&       //  Written data == $x...
    (intv_data & 0x0FF0) == 0x0A50) {                     //  Written data == $.A5.
  chap  = (intv_addr >> 12) & 0xF;    //  Get the chapter number ('x')
  page  = intv_data & 0xF;            //  Get the page number ('y')
  index = (chap << 4) | page;         //  Compute index into page-flip table
  flip  = flip_tbl[index];            //  Look up the page-flip permissions
  is_pageflip = (flip >> 3) & 1;      //  Extract the PFE bit.
}
```

When Locutus does detect a page-flip (`is_pageflip == 1` in the above pseudo-code), it remaps memory in a manner similar to the following pseudo-code:

```
if (is_pageflip) {
  const uint32_t flip_addr = (flip & 0xFFF0);  //  Address bits to concatenate.
  const uint8_t  perms     = flip & 0x7;       //  Permissions, minus PFE.
  const uint16_t para      = chap << 4;        //  Paragraph number, for indexing.

  //  For most address ranges, flip all 16 paragraphs in the page.
  if (chap != 0x4) {
    for (int i = 0; i < 16; i++) {
      mmap_tbl[para + i] = flip_addr + i;
      perm_tbl[para + i] = perms;
    }
  }
  //  For $4xxx, only flip the last 8 paragraphs in the page ($4800 - $4FFF), to avoid ECS RAM.
  else {
    for (int i = 8; i < 16; i++) {
      mmap_tbl[para + i] = flip_addr + i;
      perm_tbl[para + i] = perms;
    }
  }
}
```

Flipped Pages in Locutus RAM

Paged ROM segments can theoretically live anywhere in Locutus' RAM space, so long as they start on a 4K address boundary. In the case of $4800's paged segment, each of its pages also occupies 4K words of Locutus RAM, but only the last 2K words of each page is used by Locutus' page flipping logic.

By default, `bin2luigi` collects all paged ROM segments and packs them at the top of Locutus' RAM. The current version of `bin2luigi` packs paged ROM segments along the lines of the following pseudo-code:

```
// pg_data[chap][page] holds all of the collected paged data segments, if any.
// pg_perm[chap][page] holds the corresponding permissions.

locu_addr = 0x80000;    // One past the last address in Locutus.

for (chap = 0xF; chap >= 0x0; --chap) {     // Step backward by Intellivision address.
  bool has_flip = false;

  for (page = 0xF; page >= 0x0; --page) {  // Step backward by page number.
    if (pg_data[chap][page]) {              // Is there data at this chap/page combo?
      // Yes: Step backward in Locutus' address map and load the data there.
      locu_addr -= 0x1000;
      preload_data(pg_data[chap][page], chap, page, locu_addr);

      // Configure the flip-table to remember this locu_addr and perms.
      index = (chap << 4) | page;           // Compute index into page-flip table.
      flip_tbl[index] =
          ((locu_addr >> 8) & 0xFFF0)       // Upper address bits for flip.
        | (pg_perm[chap][page] & 7);        // Remember permissions for page.

      // Remember that this is a page-flipped chapter.
      has_flip = true;
    }
  }

  // If this was a page-flipped chapter, set Page Flip Enable for all pages in the chapter.
  if (has_flip) {
    for (page = 0x0; page <= 0xF; page++) {
      index = (chap << 4) | page;           // Compute index into page-flip table.
      flip_tbl[index] |= (1 << 3);          // Bit 3 is the Page Flip Enable.
    }
  }
}
```

Once this algorithm completes, the paged ROM segments will be in increasing-chapter order and increasing-page order within each chapter at the top of Locutus' RAM.

Revision History

Date	Notes
3-Apr-2019, A	Initial semi-public release.
15-Apr-2019, A	Added Intellicart reference and Mattel page-flip reference sections, along with much pseudo-code. Added CFGVAR reference section. Added encryption block definition. Describe where JLP RAM and page-flipped segments live in Locutus RAM. Many wording tweaks and clarifications.
15-Apr-2019, B	Fix minor error in description of Intellicart bankswitch writes.
6-Jul-2019, A	Minor grammar, formatting, and wording fixes.