JLP Programmer's View

JLP primarily emulates standard General Instrument ROM and RAM. Therefore most JLP programming consists of writing a standard Intellivision game and delivering it in standard "BIN+CFG" format. Therefore, for most purposes, programmers need not take JLP into account when programming.

JLP matters only when the game program needs one or more of the following:

- Additional RAM,
- Additional ROM beyond what fits in the Intellivision memory map,
- Access to JLP's hardware acceleration, or
- Access to JLP's flash storage.

The following sections describe how programs can access these additional features.

JLP Default Memory Map

In its default configuration, JLP provides the following memory map:

Address Range	Description
$0000 – $0FFF	Unavailable to JLP: used by Intellivision and peripherals
$1000 – $1FFF	Unavailable to JLP: used by EXEC
$2000 – $2FFF	Pages 0 and 2 – 15 available for ROM; $2000 page 1 used by ECS
$3000 – $3FFF	Unavailable to JLP: used by GROM and GRAM
$4000 – $47FF	Unavailable to JLP: used by ECS RAM
$4800 – $4FFF	Available for ROM (special 2K segment)
$5000 – $6FFF	Available for ROM
$7000 – $7FFF	Pages 1 – 15 available for ROM; $7000 page 0 used by ECS
$8000 – $803F	JLP Flash command registers
$8040 – $9F7F	JLP RAM

$9F80 – $9FFF	JLP accelerator features
$8000 – $9FFF	Available for ROM via special page-switch (see below)
$A000 – $DFFF	Available for ROM
$E000 – $EFFF	Pages 0 and 2 – 15 available for ROM; $E000 page 1 used by ECS
$F000 – $FFFF	Available for ROM

The remainder of the document assumes JLP in its default configuration. Because firmware defines most of JLP's features, one can customize JLP for a specific application. For example, JLP could easily provide 8-bit RAM at $D000 – $D3FF to support Chess and Triple Challenge, although the firmware currently does not provide this.

New code should target JLP's default feature set if at all possible.

JLP RAM

Feature Definition

JLP provides up to 8000 words of 16-bit RAM for game programs. JLP's 16-bit RAM resides at memory locations $8040 through $9F7F. The Intellivision and JLP both ascribe special meaning to locations $8000 – $803F. JLP maps its hardware acceleration features to $9F80 – $9FFF.

For MCUs that provide 8K words of RAM, the default JLP firmware fills $8040 to $9F7F with 16-bit RAM. For MCUs that provide only 4K words of RAM, the default JLP firmware fills $9000 – $9F7F with 16-bit RAM.

JLP 16-bit RAM behaves much like the 16-bit System RAM the Intellivision provides at $200 – $35F. JLP's RAM differs in the following ways:

- Fully supports the GI bus protocol. All CP-1600 code will execute correctly from JLP RAM, unlike the Intellivision System RAM.
- It is accessible at all times *except* during Flash operations. (See the Flash Access section below.) During Flash operations, the Intellivision cannot contact JLP.
- All Flash read and write operations copy data into or out of JLP RAM. To store or retrieve data from Intellivision System RAM, you must copy it to/from JLP RAM.

Otherwise, JLP RAM looks and behaves exactly like General Instrument compatible 16-bit RAM.

SDK-1600 Programming Notes

The file `cart.mac` provides a set of macros to simplify programming for JLP. The 42K memory map it offers automatically sets up a large address space for ROM *and* includes support for JLP's RAM.

To allocate memory from one of the several pools of memory in an Intellivision system, use one of the following macros:

Macro	Description	8-bit Scratch RAM	16-bit System RAM	8-bit ECS RAM	16-bit JLP RAM
`BYTEVAR`	Allocate a single byte variable.	X		X	
`BYTEARRAY`	Allocate an array of bytes	X		X	
`WORDVAR`	Allocate a single word variable		X		X
`WORDARRAY`	Allocate an array of words		X		X
`SCRATCH`	Allocate bytes in the Intellivision's internal 8-bit Scratch RAM	X			
`SYSTEM`	Allocate words in the Intellivision's internal 16-bit System RAM		X		
`ECSRAM`	Allocate bytes in the ECS's 2K of 8-bit RAM			X	
`CARTRAM`	Allocate words in JLP's 16-bit RAM				X

Note that `cart.mac` will only allocate ECS memory if you specify `REQ_ECS,` or use the `ECSRAM` macro.

Program Example

The following example declares a 96-word array in JLP RAM and a 5 word buffer in Intellivision System RAM:

```
FL_BUF  CARTRAM 96
FL_CMD  SYSTEM  5
```

Additional SDK-1600 / Emulation Note

The JLP emulation provided by jzIntv (enabled with --jlp or --jlp-savegame switches) provides its own RAM emulation at $8040 - $9F7F. Do **not** add code to your program to reserve RAM or ROM in this address range. For example, do **not** do the following:

```
ORG $8040, $8040, "=RW"
RMB $9F80 - $8040
```

This will result in a section such as the following in your program's .CFG file:

```
[memattr]
$8040 - $9F7F = RAM 16
```

This declares *additional* RAM that exists in parallel to the RAM provided by the JLP model in jzIntv. The two will conflict, eventually resulting in trouble, especially when trying to use JLP flash features.

JLP Paged ROM

Feature Definition

JLP can hold more ROM than the Intellivision can address at any one time. JLP makes this additional ROM available to the Intellivision through page-switching.

JLP uses "ECS-style" paged ROM as its default page-switching model. ECS refers to the *Entertainment Computer System*. This document refers to the format as ECS-style, as the ECS is the main released peripheral to use this page-switching model. The game *World Series Major League Baseball* is the only released game known to use this model. Some unreleased Mattel titles such as *Go For The Gold* also use this model.

The Mattel ECS page-switching model works by dividing the address space into 4K-word segments, and allowing you to map up to 16 different ROM images (or *pages*) into each of those 4K segments. That is, addresses `$0000` – `$0FFF` form one segment, `$1000` – `$1FFF` forms another segment, `$2000` – `$2FFF` forms yet another, and so on. Each 4K-word segment could have one of 16 different pages visible at one time if that segment includes paged ROM.

In this model, you can consider non-paged ROM segments as filling all 16 pages with the same data. Any attempt to page-switch these segments leaves the Intellivision with the same view of memory. For example, the Intellivision EXEC resides at `$1000` – `$1FFF`, and is not paged. So, attempting to select `$1000` page 0 leaves you with the same view of memory as selecting `$1000`

page 1, $1000 page 2 and so on.

The ECS EXEC, in contrast, consists of 3 paged ROM segments at $2000 page 1, $7000 page 0 and $E000 page 1. Therefore, other game ROMs (including JLP's) can map into these segments, either at different page numbers, or as non-paged ROM.

Non-paged ROMs may coexist with a paged ROM in the same 4K segment, so long as the program switches off all paged ROMs in that segment. To accomplish this, the program merely selects a page number for that segment that does not match any of the paged ROMs present in that segment. For example, *World Series Major League Baseball* (WSMLB) has non-paged ROM at $E000 that conflicts with the paged ECS ROM at $E000 page 1. WSMLB switches segment $E000 to select page 0, which effectively switches off the ECS ROM in this range. Most Intellivision games above 28K will likely use this model. The initialization code in SDK-1600's `cart.mac` automatically switches off the ECS EXEC ROMs as part of its startup sequence.

Banked ROMs behave as follows:

- Reset selects page 0 in all segments.
- A write to location $xFFF with a value of the form $***x***A5***y*** selects page '***y***' in segment $***x***000 – $***x***FFF.
- A given paged ROM within a segment responds only when its page number is selected for that segment. This allows multiple paged ROMs to coexist in the same segment at different page numbers.

To specify a paged ROM to jzIntv and the JLP tools, you must use the BIN+CFG game format. In the CFG, indicate the paged ROM segments by adding the PAGE keyword to one or more 4K segments in the [mapping] section. Example:

```
; No paged ROMs:
[mapping]
$0000 - $0FFF = $5000
$1000 - $1FFF = $6000
$2000 - $2FFF = $A000
$3000 - $3FFF = $B000


; Alternate version with paged ROM at $A000 pages 3 and 4:
[mapping]
$0000 - $0FFF = $5000
$1000 - $1FFF = $6000
$2000 - $2FFF = $A000 PAGE 3
$3000 - $3FFF = $A000 PAGE 4
```

For a more complex example, consider the configuration used by *Go For The Gold*. *Go For The Gold* is an animated title screen that wraps four largely unmodified Mattel sports titles. It uses page-switching to switch in one of the four titles based on the user's selection. When used with a suitably prepared binary image, the following configuration allows *Go For The Gold* to function correctly(*) on JLP:

```
[mapping]
$0000 - $0FFF = $5000 PAGE 0
$1000 - $1FFF = $6000
$2000 - $2FFF = $5000 PAGE 6  ; Skiing
$3000 - $3FFF = $5000 PAGE 5  ; Hockey
$4000 - $4FFF = $5000 PAGE D  ; Boxing
$5000 - $5FFF = $5000 PAGE 8  ; Basketball
```

(*)GFTG is not bug free, at least when assembled from the released versions of Skiing, Hockey, Boxing and Basketball; it introduces a timing glitch in Skiing's initialization that causes it to occasionally launch incorrectly.

SDK-1600 Programming Note

AS1600 currently does not provide direct support for this page-switching model. It is possible to use page-switching without AS1600 support; however, you must construct additional tooling to do so. If you need more than 42K for your game, or need to use page-switching for other purposes, please contact LTO for assistance. *[This is no longer true; AS1600 now has native support for paged ROMs. Details to be added here. Short version: "ORG addr:page" begins assembling in a paged ROM section. Paged ROM only supported with BIN+CFG format. Paged ROM segments are 4K words on 4K boundaries. —*JZ 06-Nov-2014*]*

Program Example

The following code snippet disables the ROMs in the ECS by switching ROM segments `$2000`, `$7000` and `$E000` all to page 15, which contains no ROM.

```
MVII    #$2A5F, R0
MVO     R0,     $2FFF
MVII    #$7A5F, R0
MVO     R0,     $7FFF
MVII    #$EA5F, R0
MVO     R0,     $EFFF
```

Special Page-Switch: JLP RAM vs. Additional ROM

Feature Definition

JLP allows you to switch JLP RAM and accelerator features on and off. While JLP RAM and accelerator features are off, you can access ROM in this address space if your program maps any ROM in this address range.

This feature exists primarily to accommodate existing software that requires ROM in the range `$8000` – `$8FFF` or `$9000` – `$9FFF`, as well as to accommodate software that needs to work with an Intellivision Keyboard Component, as the Keyboard Component maps its own 16K x 10-bit

RAM at $8000 – $BFFF. Most programs developed for JLP do not need this feature.

To switch JLP RAM and accelerator features off, write $6A7A to location $8034. To switch JLP RAM and accelerator features back on, write $4A5A to location $8033.

Program Example

The following snippet turns JLP RAM off, then back on.

```
; Turn off JLP RAM
MVII  #$6A7A,     R0
MVO   R0,         $8034

; Turn on JLP RAM and accelerators
MVII  #$4A5A,     R0
MVO   R0,         $8033
```

JLP Multiply and Divide Accelerators

Feature Definition

The PIC24H MCU includes fast hardware multiply and divide support. JLP exposes its hardware multiplier and divider to the Intellivision through a series of memory mapped registers, mimicking a dedicated peripheral one might have found in an enhanced 80s machine.

Each of these mathematical accelerator functions works the same way: Write the arguments to one pair of input locations, and read the results from another pair of output locations. Each command takes its input from a different pair of locations, but writes its output to the same pair of output locations: $9F8E, $9F8F.

JLP computes a new result when *either* input operand changes. Furthermore, it holds the input operands and results indefinitely. This allows you to reuse an argument multiple times. See how this helps in the example at the end of this section.

The following table lists all of the provided functions and the memory locations they reside in:

Name	Description	1st oper	2nd oper	Result in $9F8F	Result in $9F8E
MPYSS	Signed 16-bit by signed 16-bit multiply into 32-bit result.	$9F80	$9F81	Upper 16 bits of product	Lower 16 bits of product
MPYSU	Signed 16-bit by unsigned 16-bit multiply into 32-bit result.	$9F82	$9F83	Upper 16 bits of product	Lower 16 bits of product
MPYUS	Unsigned 16-bit by signed 16-bit multiply into 32-bit result.	$9F84	$9F85	Upper 16 bits of product	Lower 16 bits of product
MPYUU	Unsigned 16-bit by unsigned 16-bit multiply into 32-bit result.	$9F86	$9F87	Upper 16 bits of product	Lower 16 bits of product
DIVSS	Signed 16-bit by signed 16-bit divide with remainder.	$9F88	$9F89	Remainder	Quotient
DIVUU	Unsigned 16-bit by unsigned 16-bit divide with remainder	$9F8A	$9F8B	Remainder	Quotient

Interrupts and Atomicity

These accelerator features require multiple memory accesses to perform a single operation. Furthermore, they all write their results to the same output locations. This presents a problem if your program uses these features from both an interrupt and non-interrupt context. Unless you take steps to ensure otherwise, the CP-1610 may take an interrupt right in the middle of code that tries to use these accelerators.

You have two options: Only use the accelerators in a single context—always outside an interrupt handler or always inside an interrupt handler—or make all uses of the accelerators *atomic* with respect to interrupts. In other words, to use the accelerators from both interrupt and non-interrupt contexts, you must make the uses in non-interrupt contexts *uninterruptible.*

The CP-1610 offers two ways to do this:

1. Use DIS and EIS to explicitly disable and re-enable interrupts. This works best for longer sequences of instructions.
2. Use sequences of uninterruptible instructions (MVO, shifts) to block interrupts.

In most cases, you will want to use the first approach. That strategy works in all circumstances, but it also consumes the most cycles. Here is an example that multiplies `R0` x `R1` into `R3:R2`:

```
        DIS                     ; Disable interrupts
        MVO     R0,     $9F80   ; First operand to MPYSS
```

```
        MVO     R1,     $9F81   ; Second operand to MPYSS
        MVI     $9F8E,  R2      ; Lower 16 bits of 32-bit result
        EIS                     ; Enable interrupts after next MVI
        MVI     $9F8F,  R3      ; Upper 16 bits of 32-bit result
```

This sequence is fully atomic with respect to interrupts. The EIS appears before the last MVI because EIS *itself* is an uninterruptible instruction, meaning that no interrupt can occur between EIS and the instruction that follows it. By pulling EIS above the final MVI, we keep interrupts disabled for the minimum amount of time.

You can use the second approach when you only need 16 bits of the result and want to absolutely minimize the time you disable interrupts. For example, if you only need the lower 16 bits of R0 x R1, the following sequence is atomic because MVO is uninterruptible:

```
        MVO     R0,     $9F80   ; First operand to MPYSS
        MVO     R1,     $9F81   ; Second operand to MPYSS
        MVI     $9F8E,  R2      ; Lower 16 bits of 32-bit result
```

This can be useful if you know the result of your operation fits in 16 bits.

Program Example

The following example demonstrates converting a number to decimal using the unsigned divide accelerator. This can be useful for displaying scores on-screen, for example.

```
DIVUU.num   EQU     $9F8A   ; Numerator for the divide
DIVUU.den   EQU     $9F8B   ; Denominator for the divide
DIVUU.quo   EQU     $9F8F   ; Quotient
DIVUU.rem   EQU     $9F8E   ; Remainder

; Decimalize the contents of R0, writing the five digits to the buffer @R4
            MVII    #10,    R1
            DIS                         ; Disable interrupts
            MVO     R1, DIVUU.den       ; Set up the denominator for divides
            MVO     R0, DIVUU.num       ; \
            MVI     DIVUU.rem, R0       ;  |- First digit (rightmost)
            MVO@    R0,        R4       ; /
            MVI     DIVUU.quo, R0
            MVO     R0, DIVUU.num       ; \
            MVI     DIVUU.rem, R0       ;  |- Second digit
            MVO@    R0,        R4       ; /
            MVI     DIVUU.quo, R0
            MVO     R0, DIVUU.num       ; \
            MVI     DIVUU.rem, R0       ;  |- Third digit
            MVO@    R0,        R4       ; /
            MVI     DIVUU.quo, R0
            MVO     R0, DIVUU.num       ; \
```

```
MVI     DIVUU.rem, R0       ;  |- Fourth digit
MVO@    R0,        R4       ; /
MVI     DIVUU.quo, R0
MVO     R0, DIVUU.num       ; \
EIS                         ;  | (Enable interrupts after MVI)
MVI     DIVUU.rem, R0       ;  |- Fifth digit (leftmost)
MVO@    R0,        R4       ; /
```

This particular example reuses the argument ‘10’ in the denominator of the divide for all 5 digits, resulting in a considerable savings.

JLP CRC-16 Accelerator

Feature Definition

CRC stands for Cyclic Redundancy Check. CRCs compute a hash function of a string of inputs using specialized arithmetic that can then be used as a checksum. A small change in the input leads to a large change in the output. Many common file formats and transmission protocols use CRCs for this reason, as a small corruption will generate a large difference in CRC. Random number generators often make use of CRCs and related codes for similar reasons. You can read more about CRCs on Wikipedia here:

http://en.wikipedia.org/wiki/Cyclic_redundancy_check

JLP provides a CRC-16 accelerator with a fixed polynomial. CRC-16 means that it generates a 16-bit checksum / hash. Fixed polynomial means that JLP only computes one of the many possible CRC-16 hash functions. JLP’s CRC is a right-shifting CRC with the polynomial 0xAD52. See the Reference Code section below to see exactly what that means via reference Perl and assembly code.

Programs access the CRC-16 accelerator through 2 memory mapped registers:

- `$9FFC`: Write only. Write data to checksum to this location.
- `$9FFD`: Read-write. Write this location to initialize the checksum. Read this location to get the current checksum after writing the data to checksum to `$9FFC`.

Games and other programs might use the accelerator a few different ways:

1. Production test to verify the game ROM is programmed into JLP correctly.
2. Verify that data in JLP Flash was stored and retrieved correctly.
3. Generate a deterministic and repeatable set of “random” numbers.

The sections below explore each of those potential uses.

Note: The CRC-16 accelerator has similar atomicity concerns to the multiply and divide accelerators, since it is a single resource that programs could conceivably use from both an interrupt and non-interrupt context. See the “Interrupts and Atomicity” section above for a detailed exploration of the issue.

Program Example: Data Verification

CRCs excel at detecting corruption in small blocks of data. In this mode, programs typically compute the CRC over a block of data and compare it to a CRC value computed previously over the same block of data. If the CRCs match, the program assumes the data is correct. If the CRCs mismatch, the program knows the data is incorrect.

Production test code can use the accelerator to verify that each of the ROM segments contains the expected ROM image. Currently shipping titles such as Christmas Carol incorporate production test code that checksums each 4K ROM segment to verify its CRC.

Code that saves and loads data from JLP flash can verify that the data survived the round trip by checking its CRC. When saving data to flash, the code computes the CRC and saves it with the data. When retrieving data from flash, the code computes the CRC again and compares it against the saved CRC.

Both of these uses benefit from a “block CRC”, that is code that reads an entire block of RAM or ROM, and returns a checksum for that block. The following function implements a block CRC suitable for both purposes.

```
;; ======================================================================== ;;
;;  CRC16                                                                   ;;
;;                                                                          ;;
;;  Compute a CRC-16 on a block of memory.  Implements a right-shifting     ;;
;;  CRC with the polynomial 0xAD52.  This version uses JLP acceleration.    ;;
;;                                                                          ;;
;;  INPUTS: CRC16                                                           ;;
;;      R1  Number of words to checksum                                     ;;
;;      R4  Base address to start checksum                                  ;;
;;                                                                          ;;
;;  INPUTS: CRC16.1                                                         ;;
;;      R0  Initial checksum                                                ;;
;;      R1  Number of words to checksum                                     ;;
;;      R4  Base address to start checksum                                  ;;
;;                                                                          ;;
```

```
;;  OUTPUTS                                                             ;;
;;      R0  Final checksum                                              ;;
;;      R4  Points past end of 4K segment                               ;;
;;      R1..R3 unmodified                                               ;;
;; ======================================================================= ;;
CRC16       PROC
            CLRR    R0
@@1:        MVO     R0,     $9FFD
            PSHR    R1
            PSHR    R2
            PSHR    R3

            MVII    #$9FFC, R2
            SARC    R1,     2
            ANDI    #$3FFF, R1
            BNC     @@even
            MVI@    R4,     R0
            MVO@    R0,     R2
@@even:     BNOV    @@no2

            MVI@    R4,     R0
            MVO@    R0,     R2
            MVI@    R4,     R0
            MVO@    R0,     R2
@@no2:      BEQ     @@done

@@loop:
            MVI@    R4,     R0
            MVI@    R4,     R3
            MVO@    R0,     R2
            MVO@    R3,     R2
            MVI@    R4,     R0
            MVI@    R4,     R3
            MVO@    R0,     R2
            MVO@    R3,     R2
            DECR    R1
            BNEQ    @@loop

@@done:     MVI     $9FFD,  R0
            PULR    R3
            PULR    R2
            PULR    R1
            JR      R5
            ENDP
```

Program Example: Deterministic Random Number Generator

In this model, the program needs to generate a sequence of "random" numbers that always occur in the same order. This can be useful in a number of situations.

For example, consider a head-to-head puzzle game where each player gets a sequence of puzzle pieces he or she must place. To keep the game fair, both players should receive the same pieces in the same order. But, to keep the game interesting, the game should give a different sequence of pieces in different games.

Or, perhaps you have a game engine with "random" elements, but you want to be able to record demos by only recording player inputs. Doom's engine worked this way. You can see its deterministic random function here: http://spatula-city.org/~im14u2c/dl/m_random.c

To generate deterministic random functions with JLP's CRC accelerator:

- Set up a "seed" in memory to denote the game's notion of "current random number".
- To generate the next random number, perform the following steps:
    a. Copy the seed from RAM to location `$9FFD`
    b. Write a fixed value to `$9FFC`
    c. Copy the updated seed from `$9FFD` back to RAM.
- Use the updated seed as the next random number. If you need to scale it to a particular range of values, you could use it as an input to an unsigned multiply (with the other input being the range), and take the upper 16 bits of the product as the range-limited random value.

The following code demonstrates. It assumes two players, with both seeds stored next to each other in memory. `NEXTRAND` takes an argument in `R1` specifying which player's seed to fetch. It returns the new random number in `R0`.

```
SEEDS       SYSTEM  2       ; Allocate two random number seeds in System RAM

NEXTRAND    PROC
            ADDI    #SEEDS,     R1
            MVI@    R1,         R0      ; Get player's seed
            MVO     R0,         $9FFD   ; Use seed as CRC checksum value
            CLRR    R0                  ; \_ Write fixed value 0 to update checksum
            MVO     R0,         $9FFC   ; /
            MVI     $9FFD,      R0      ; Get updated CRC checksum
            MVO@    R0,         R1      ; Store as player's seed
            JR      R5                  ; Return
            ENDP
```

That's fairly straightforward. Two caveats:

1. The random sequence this generates never changes. Starting with a different seed just time-shifts the sequence.
2. The sequence repeats after 65535 steps.

Neither of these is a complete deal breaker. For one thing, not all games are sensitive to either of these parameters. Doom, for example, has a single sequence with only 256 values in it, and this sequence repeats after 256 steps.

For games that are sensitive to either or both of these properties, you have a few options.

1. Add a second word in RAM that stores the "fixed value" to update the seed with. Pick this "fixed value" randomly at the same time you pick your starting seed. This will give you a different path through the 65535 seed values even if you start with the same seed. (Note: Do not use $5AA4 as the fixed value—it generates a very short cycle.)
2. Combine the output of multiple deterministic random number generators running at different rates. For example, add a second seed, and update that seed every seventh iteration. XOR the values of both seeds to get your final random number. (7 is relatively prime to 65535, so the resulting sequence will be 7 times longer than the base generator's sequence. You can pick larger number that's relatively prime to 65535 to get a longer sequence.)
3. Combine both 1 and 2 together.

The following code demonstrates the first option, as it is simpler and is straightforward to implement. The second and third options are expensive enough that you might consider a different approach than the CRC-16 accelerator.

```
SEEDS       SYSTEM  2       ; Allocate two random number seeds in System RAM
SALT        SYSTEM  1       ; Random value that perturbs the random number sequence

NEXTRAND    PROC
            ADDI    #SEEDS,     R1
            MVI@    R1,         R0      ; Get player's seed
            MVO     R0,         $9FFD   ; Use seed as CRC checksum value
            MVI     SALT,       R0      ; \_ Write fixed value from SALT
            MVO     R0,         $9FFC   ; /  to update checksum
            MVI     $9FFD,      R0      ; Get updated CRC checksum
            MVO@    R0,         R1      ; Store as player's seed
            JR      R5                  ; Return
            ENDP
```

In any case, it's worth thinking about how much randomness you *really* need. For some purposes, such as shuffling a deck of cards, you need a surprising amount of randomness to ensure you can generate all 52! orderings with equal probability. For most others (look at Doom, for example), you need surprisingly little.

Reference Code

The following functions define the exact CRC function JLP implements. The first is an assembly function that replicates the block-CRC functionality demonstrated above, using a lookup table for speed:

```
;; ===================================================================== ;;
;;  CRC16                                                                ;;
;;                                                                       ;;
;;  Compute a CRC-16 on a block of memory.  Implements a right-shifting  ;;
;;  CRC with the polynomial 0xAD52.                                      ;;
;;                                                                       ;;
;;  The CRC16T table has the original datum XORed in with the precomputed ;;
;;  poly value so that we don't need to mask it out of our CRC.  We can  ;;
;;  just SWAP and XOR and it'll cancel out.                              ;;
;;                                                                       ;;
;;  INPUTS: CRC16                                                        ;;
;;      R1  Number of words to checksum                                  ;;
;;      R4  Base address to start checksum                               ;;
;;                                                                       ;;
;;  INPUTS: CRC16.1                                                      ;;
;;      R0  Initial checksum                                             ;;
;;      R1  Number of words to checksum                                  ;;
;;      R4  Base address to start checksum                               ;;
;;                                                                       ;;
;;  OUTPUTS                                                              ;;
;;      R0  Final checksum                                               ;;
;;      R4  Points past end of 4K segment                                ;;
;;      R1..R3 unmodified                                                ;;
;; ===================================================================== ;;

CRC16   PROC
        CLRR    R0
@@1:
        PSHR    R5
        PSHR    R3
        PSHR    R2
        PSHR    R1

        MVII    #$FF,   R3
        MVII    #CRC16T,R5
@@loop:
        XOR@    R4,     R0      ;    8 Merge in next data word

        MOVR    R0,     R2      ;    6 \
        ANDR    R3,     R2      ;    6  |- Index into crc16t[].
        ADDR    R5,     R2      ;    6 /
        SWAP    R0              ;    6 Shift left by 8
```

```
        XOR@    R2,     R0      ;    8 XOR in CRC update for lower 8

        MOVR    R0,     R2      ;    6 \
        ANDR    R3,     R2      ;    6  |- Index into crc16t[].
        ADDR    R5,     R2      ;    6 /
        SWAP    R0              ;    6 Shift left by 8
        XOR@    R2,     R0      ;    8 XOR in CRC update for lower 8

        DECR    R1
        BNEQ    @@loop

        PULR    R1
        PULR    R2
        PULR    R3
        PULR    PC
        ENDP

CRC16T  PROC
        DECLE   $0000, $C035, $DACF, $1AFA, $EF3B, $2F0E, $35F4, $F5C1
        DECLE   $84D3, $44E6, $5E1C, $9E29, $6BE8, $ABDD, $B127, $7112
        DECLE   $5303, $9336, $89CC, $49F9, $BC38, $7C0D, $66F7, $A6C2
        DECLE   $D7D0, $17E5, $0D1F, $CD2A, $38EB, $F8DE, $E224, $2211
        DECLE   $A606, $6633, $7CC9, $BCFC, $493D, $8908, $93F2, $53C7
        DECLE   $22D5, $E2E0, $F81A, $382F, $CDEE, $0DDB, $1721, $D714
        DECLE   $F505, $3530, $2FCA, $EFFF, $1A3E, $DA0B, $C0F1, $00C4
        DECLE   $71D6, $B1E3, $AB19, $6B2C, $9EED, $5ED8, $4422, $8417
        DECLE   $16A9, $D69C, $CC66, $0C53, $F992, $39A7, $235D, $E368
        DECLE   $927A, $524F, $48B5, $8880, $7D41, $BD74, $A78E, $67BB
        DECLE   $45AA, $859F, $9F65, $5F50, $AA91, $6AA4, $705E, $B06B
        DECLE   $C179, $014C, $1BB6, $DB83, $2E42, $EE77, $F48D, $34B8
        DECLE   $B0AF, $709A, $6A60, $AA55, $5F94, $9FA1, $855B, $456E
        DECLE   $347C, $F449, $EEB3, $2E86, $DB47, $1B72, $0188, $C1BD
        DECLE   $E3AC, $2399, $3963, $F956, $0C97, $CCA2, $D658, $166D
        DECLE   $677F, $A74A, $BDB0, $7D85, $8844, $4871, $528B, $92BE
        DECLE   $2D52, $ED67, $F79D, $37A8, $C269, $025C, $18A6, $D893
        DECLE   $A981, $69B4, $734E, $B37B, $46BA, $868F, $9C75, $5C40
        DECLE   $7E51, $BE64, $A49E, $64AB, $916A, $515F, $4BA5, $8B90
        DECLE   $FA82, $3AB7, $204D, $E078, $15B9, $D58C, $CF76, $0F43
        DECLE   $8B54, $4B61, $519B, $91AE, $646F, $A45A, $BEA0, $7E95
        DECLE   $0F87, $CFB2, $D548, $157D, $E0BC, $2089, $3A73, $FA46
        DECLE   $D857, $1862, $0298, $C2AD, $376C, $F759, $EDA3, $2D96
        DECLE   $5C84, $9CB1, $864B, $467E, $B3BF, $738A, $6970, $A945
        DECLE   $3BFB, $FBCE, $E134, $2101, $D4C0, $14F5, $0E0F, $CE3A
        DECLE   $BF28, $7F1D, $65E7, $A5D2, $5013, $9026, $8ADC, $4AE9
        DECLE   $68F8, $A8CD, $B237, $7202, $87C3, $47F6, $5D0C, $9D39
        DECLE   $EC2B, $2C1E, $36E4, $F6D1, $0310, $C325, $D9DF, $19EA
        DECLE   $9DFD, $5DC8, $4732, $8707, $72C6, $B2F3, $A809, $683C
        DECLE   $192E, $D91B, $C3E1, $03D4, $F615, $3620, $2CDA, $ECEF
        DECLE   $CEFE, $0ECB, $1431, $D404, $21C5, $E1F0, $FB0A, $3B3F
        DECLE   $4A2D, $8A18, $90E2, $50D7, $A516, $6523, $7FD9, $BFEC
```

```
        ENDP
```

This next implementation in Perl demonstrates updating the seed for a single 16-bit update, without a lookup table:

```
## ========================================================================= ##
##  CRC16   Compute JLP CRC-16:  Given crc + data, compute new crc.         ##
## ========================================================================= ##
sub crc16($$)
{
    my ($crc, $data) = @_;

    $crc ^= $data;

    foreach (0 .. 15)
    {
        $crc = (($crc >> 1) ^ ($crc & 1 ? 0xAD52 : 0)) & 0xFFFF;
    }

    return $crc;
}
```

JLP Non-deterministic Hardware Random Number Generator

Feature Definition

For programs that want a non-deterministic (or, at least, a much less deterministic) random number sequence, JLP provides a hardware random number generator. For this generator, JLP monitors the Intellivision bus and mixes what it sees into its random number seed.

Because JLP's MCU runs with a different clock than the Intellivision CPU, JLP's alignment to the Intellivision clock varies. JLP's clock is at least 40 times as fast as the Intellivision's. Therefore, JLP can observe system-specific "garbage" on the Intellivision bus. Furthermore, JLP will observe every user input to the system indirectly, since JLP sees every bus cycle that reads a controller or other input, even though it doesn't respond to that input.

Programs access JLP's hardware random number generator by reading location $9FFE. JLP will return the current random number state as a 16-bit number. JLP will also scramble the random number with a CRC after the read, so that two consecutive reads return different numbers.

JLP's hardware random number generator collects entropy (aka. randomness) through variation in the Intellivision bus cycles. This means that two random numbers collected close in time will be more "related" than two random numbers collected further apart in time.

For most purposes, this doesn't matter. JLP generates sufficiently random numbers for the vast majority of games, regardless of how closely the game reads them in time. In the unlikely event that your program needs to generate "cryptographically strong" random numbers, assume that JLP collects very little entropy between consecutive reads of the random number register.

Program Example

The following example reads two random numbers from JLP into R0 and R1:

```
MVI     $9FFE, R0
MVI     $9FFE, R1
```

JLP Flash

Feature Definition

JLP stores your program in the MCU's on-chip flash. In most cases, the program image does not fill the entire available flash. Rather than waste that storage space, JLP exposes it to programs as non-volatile storage space.

The MCU's flash storage consists of two related structures:

- **Rows**: Each row holds 96 16-bit words. You can only write row-sized blocks.
- **Sectors**: Each sector consists of 8 rows. You can only erase by the sector.

JLP's flash support exposes this structure to the programmer through a set of memory mapped command registers that allow you to copy JLP RAM to a row in flash, copy a row of flash to JLP RAM, and erase a flash sector. The following table lists all the memory mapped registers provided by JLP's flash support:

Address	Name	Read/write	Description
$8023	JF.first	Read-only	Returns first valid flash row number
$8024	JF.last	Read-only	Returns last valid flash row number
$8025	JF.addr	Write-only	Address in JLP RAM to operate on
$8026	JF.row	Write-only	Flash row to operate on
$802D	JF.wrcmd	Write-only	Copy JLP RAM to flash. Must write the value $C0DE.
$802E	JF.rdcmd	Write-only	Copy flash to JLP RAM. Must write the value $DEC0.
$802F	JF.ercmd	Write-only	Erase flash sector. Must write the value $BEEF.

Determining Flash Capacity From Software

The registers JF.first and JF.last give the range of valid flash row numbers on a given device. These numbers vary based on the size of your program and the capacity of the MCU (small, medium or large). The available storage is always a multiple of 8 rows, so there are no "partial sectors."

Sectors always start on a row number that is a multiple of 8, and consist of that row plus the 7 that follow it. The first valid row number is always a multiple of 8, and always corresponds to the first row of the first erase sector. The last row number is always *one less* than a multiple of 8, and always corresponds to the last row of the last erase sector.

Executing Flash commands

At a high level, each of the JLP flash commands has the same overall structure:

- Set up the address in JLP RAM to operate on. (Not needed for JF.ercmd.)
- Set up the row in flash to operate on.
- Write a special "key value" to the command register for the desired command.
- Wait for the operation to complete.

The last two steps are perhaps the most complicated. JLP flash operations can take up to 28ms to complete. During the JLP flash operation, JLP's MCU stalls execution, and JLP ceases responding to the Intellivision.

Therefore, your program must take special steps to correctly handle flash commands:

- Execute the code that triggers the command and waits for its completion from the Intellivision's System RAM.

- Place the stack and interrupt service routines in System RAM if you enable interrupts.
- Do not access JLP RAM or ROM while the JLP flash command executes.

The following reference code demonstrates one way to achieve this. The function `JF.INIT` copies a small (5 words) routine to Intellivision System RAM that triggers the command, waits for its completion, and services interrupts. The function `JF.CMD` sets up and executes a flash command. Call `JF.INIT` to set everything up before calling `JF.CMD`.

```
;; ======================================================================== ;;
;;  JLP "Save Game" support                                                 ;;
;; ======================================================================== ;;
JF.first    EQU     $8023
JF.last     EQU     $8024
JF.addr     EQU     $8025
JF.row      EQU     $8026

JF.wrcmd    EQU     $802D
JF.rdcmd    EQU     $802E
JF.ercmd    EQU     $802F
JF.wrkey    EQU     $C0DE
JF.rdkey    EQU     $DEC0
JF.erkey    EQU     $BEEF

JF.write:   DECLE   JF.wrcmd,   JF.wrkey    ; Copy JLP RAM to flash row
JF.read:    DECLE   JF.rdcmd,   JF.rdkey    ; Copy flash row to JLP RAM
JF.erase:   DECLE   JF.ercmd,   JF.erkey    ; Erase flash sector

JF.SYSRAM   SYSTEM  5           ; 5 words in Intv for support routine

JF.SV       CARTRAM 6           ; 6 words in JLP for register / ISR save
JF.SV.ISR   EQU     JF.SV + 0
JF.SV.R0    EQU     JF.SV + 1
JF.SV.R1    EQU     JF.SV + 2
JF.SV.R2    EQU     JF.SV + 3
JF.SV.R4    EQU     JF.SV + 4
JF.SV.R5    EQU     JF.SV + 5

;; ======================================================================== ;;
;;  JF.INIT         Copy JLP save-game support routine to System RAM        ;;
;; ======================================================================== ;;
JF.INIT     PROC
            PSHR    R5
            MVII    #@@__code,  R5
            MVII    #JF.SYSRAM, R4
            REPEAT  5
            MVI@    R5,         R0      ; \_ Copy code fragment to System RAM
            MVO@    R0,         R4      ; /
            ENDR
            PULR    PC

            ;; === start of code that will run from RAM
@@__code:   MVO@    R0,         R1      ; JF.SYSRAM + 0: initiate command
            ADD@    R1,         PC      ; JF.SYSRAM + 1: Wait for JLP to return
            JR      R5                  ; JF.SYSRAM + 2:
            MVO@    R2,         R2      ; JF.SYSRAM + 3: \__ simple ISR
            JR      R5                  ; JF.SYSRAM + 4: /
            ;; === end of code that will run from RAM
            ENDP
```

```
;; ======================================================================== ;;
;;  JF.CMD          Issue a JLP Flash command                               ;;
;;                                                                          ;;
;;  INPUT                                                                   ;;
;;      R0  Slot number to operate on                                       ;;
;;      R1  Address to copy to/from in JLP RAM                              ;;
;;      @R5 Command to invoke:                                              ;;
;;                                                                          ;;
;;              JF.write -- Copy JLP RAM to Flash                           ;;
;;              JF.read  -- Copy Flash to JLP RAM                           ;;
;;              JF.erase -- Erase flash sector                              ;;
;;                                                                          ;;
;;  OUTPUT                                                                  ;;
;;      R0 - R4 not modified.  (Saved and restored across call)             ;;
;;      JLP command executed                                                ;;
;;                                                                          ;;
;;  NOTES                                                                   ;;
;;      This code requires two short routines in the console's System RAM.  ;;
;;      It also requires that the system stack reside in System RAM.        ;;
;;      Because an interrupt may occur during the code's execution, there   ;;
;;      must be sufficient stack space to service the interrupt (8 words).  ;;
;;                                                                          ;;
;;      The code also relies on the fact that the EXEC ISR dispatch does    ;;
;;      not modify R2.  This allows us to initialize R2 for the ISR ahead   ;;
;;      of time, rather than in the ISR.                                    ;;
;; ======================================================================== ;;
JF.CMD      PROC

            MVO     R4,         JF.SV.R4    ; \
            MVII    #JF.SV.R0,  R4          ;  |
            MVO@    R0,         R4          ;  |- Save registers, but not on
            MVO@    R1,         R4          ;  |  the stack.  (limit stack use)
            MVO@    R2,         R4          ; /

            MVI@    R5,         R4          ; Get command to invoke

            MVO     R5,         JF.SV.R5    ; save return address

            DIS
            MVO     R1,         JF.addr     ; \_ Save SG arguments in JLP
            MVO     R0,         JF.row      ; /

            MVI@    R4,         R1          ; Get command address
            MVI@    R4,         R0          ; Get unlock word

            MVII    #$100,      R4          ; \
            SDBD                            ;  |_ Save old ISR in save area
            MVI@    R4,         R2          ;  |
            MVO     R2,         JF.SV.ISR   ; /
```

```
MVII    #JF.SYSRAM + 3, R2      ; \
MVO     R2,         $100        ;  |_ Set up new ISR in RAM
SWAP    R2                      ;  |
MVO     R2,         $101        ; /

MVII    #$20,       R2          ; Address of STIC handshake
JSRE    R5,  JF.SYSRAM          ; Invoke the command

MVI     JF.SV.ISR,  R2          ; \
MVO     R2,         $100        ;  |_ Restore old ISR
SWAP    R2                      ;  |
MVO     R2,         $101        ; /

MVII    #JF.SV.R0,  R5          ; \
MVI@    R5,         R0          ;  |
MVI@    R5,         R1          ;  |- Restore registers
MVI@    R5,         R2          ;  |
MVI@    R5,         R4          ; /
MVI@    R5,         PC          ; Return

ENDP
```

The following subsections describe the flash commands themselves.

`JF.rdcmd`: Copy Flash to JLP RAM

This operation copies a single row of flash to the specified address in JLP RAM. JLP will fill 96 locations of JLP RAM starting at the address specified in `JF.addr`. Reads typically execute very quickly (a few Intellivision CPU instruction cycles).

JLP drops the command if:

- The row number is out of range—ie. less than `JF.first` or greater than `JF.last`.
- One or more RAM addresses fall outside the JLP RAM range.

Empty rows of flash read as $FFFF.

`JF.wrcmd`: Copy JLP RAM to Flash

This operation copies 96 words of JLP RAM from the address specified in `JF.addr` into the row of flash specified in `JF.row`. Once the write completes, the value will remain in flash until erased. Writes can require up to 1.8ms to complete.

JLP drops the command if:

- The requested row of flash is not empty. You can not overwrite a row of flash without

erasing the sector that contains that row.
- The row number is out of range—ie. less than `JF.first` or greater than `JF.last`.
- One or more RAM addresses fall outside the JLP RAM range.

`JF.ercmd`: Erase a Flash Sector

This operation erases 8 rows (768 words) in flash. It erases the flash sector containing the row specified in `JF.row`, as well as the other 7 rows that share the same flash sector. Sector erasure takes up to 28ms to complete.

JLP drops the command if the row number is out of range—ie. less than `JF.first` or greater than `JF.last`.

Flash Endurance: Wear Leveling

Unlike RAM, flash memory wears a little every time you erase or write it. Flash works by depositing a charge on the gate of a transistor. There is no direct connection, however: Insulator completely surrounds the gate, isolating it. Programming and erasing a bit requires forcing a charge *through* the insulator. Over time, the insulator breaks down, reducing its ability to hold the charge on the gate. For greater detail on how that process works, see here:

https://secure.wikimedia.org/wikipedia/en/wiki/Flash_memory#NOR_flash

The flash provided by JLP's MCU withstands a minimum of 10,000 erase/write cycles. That means each bit cell can be erased and rewritten at least 10,000 times before it becomes unreliable. Note that wear is physically localized to the sector involved: Erasing and writing sector X does not wear on sector Y if they are different sectors.

Therefore programs should spread their writes around multiple flash sectors, so that they do not wear out the flash prematurely. Because your program also cannot overwrite a row without first erasing it, your program must do a little extra work to use the flash effectively.

Round Robin Writing with Data Versioning

The round robin approach works as follows: Always write to the least-recently-written row, or lowest numbered row in case of ties. This automatically spreads writes evenly across all rows, because it erases and rewrites rows in a fixed, rotating sequence. All rows see exactly the same number of writes. This provides the best possible wear leveling.

The question then remains: How do we use this for saving game or other state? One effective model treats the *total* set of saved game/program state as a single, monolithic structure. When the program writes an updated copy of this structure, it writes to the next row (or rows) in the rotation. When it reads the saved copy of the structure, it scans all the rows and finds and loads the most recently written copy.

That's the basic idea. The following structure provides a straightforward implementation:

- Add a data version field to your data:
    - This is separate from the program data itself. Typically, two words of version information provide a large enough range of version numbers.
    - Ensure version `$FFFFFFFF` is not a valid version number. This allows distinguishing an empty flash row (due to erasure) from a row with game data.
- Provide two main functions, LOAD and SAVE, to manage the flash.
    - LOAD:
        - Scans the flash and return the most recently saved version of the data.
        - If none exists, it returns a default version of the data. For example, if you intend to store a high score table, it would return a default high score table with default high scores.
        - Record the row number of the *next* row in flash as the "next save" location. The row after the most-recently-saved version is always the "oldest" row. The "SAVE" code will use this information.
    - SAVE:
        - Compare the data to save with the most recently saved version. (Optional, but highly recommended. Alternately, your program could ensure it only calls SAVE with updated data.)
            - If they are identical, do nothing. Stop here.
            - If they differ, continue.
        - Increment the version number of the data.
        - Erase flash sectors if necessary to make room for the data.
        - Write the data to flash.
        - Update the "next save" row to point after the saved data.
- Call LOAD at least once before any call to SAVE to initialize the flash and synchronize your program with flash.

The LOAD and SAVE functions abstract the flash commands away from your program, turning flash into a simple persistent data store that persists some or all of your program state. LOAD ensures the program sees the persistent state from flash, while SAVE ensures flash stores the most recent version of the data.

The following reference code implements the scheme above, with the following assumptions:

- Save data fits in 94 words (one 96 word row, minus 2 words for version information)
- Builds upon `JF.CMD` and `JF.INIT` reference code above.
- Stack in System RAM.
- `FL_LOAD` always calls `JF.INIT`.
- The program calls `FL_LOAD` at least (and usually only) once before any call to `FL_SAVE`. It need not call `FL_LOAD` more than once.

- The macros `SYSTEM` and `CARTRAM` allocate memory in System RAM and JLP RAM respectively. These macros are part of `cart.mac`.
- All data stored across 32 rows in flash at the "bottom" of flash; remainder of flash open for other uses. This allows a minimum of 320,000 rewrites, which should be more than enough for most programs.
- The program only calls `FL_SAVE` if the data to save actually changes—for example, the player achieves a new high score. `FL_SAVE` does not filter out redundant stores itself, your code that calls it needs to.
- No CRC or other protection on saved data.
- The program copies data into / out of `FL_BUF`; the reference code otherwise uses it as scratch space.

The routine `FL_LOAD` scans the 32 rows in its pool, looking for the most recent version of the data. It ignores rows that have `$FFFF` in the upper word of the version number, thereby making version numbers `$FFFF0000` – `$FFFFFFFF` "invalid". It starts version numbers at `$00000000`. Given the limited number of erase/rewrite cycles the flash supports, it's impossible to reach the range of invalid version numbers under normal circumstances.

Once it finds the highest version-number row, it copies that to `FL_BUF`, and stores some details about its version number and row in flash. If the flash is empty, `FL_LOAD` zeroes out the `FL_BUF`. **Note**: For your program, you may want to replace that last step with code that initializes the buffer to some default value appropriate for your program.

The `FL_SAVE` code relies on the data structure initialized by `FL_LOAD`. The program packs the information to store into `FL_BUF` and then calls `FL_SAVE`. `FL_SAVE` then copies the data into flash and updates the data structures. If the row it's saving to is the the first row of a sector, it erases the sector first.

```
;; ========================================================================= ;;
;;  Data Structures used by FL_LOAD, FL_SAVE                                 ;;
;; ========================================================================= ;;
FL_BUF          CARTRAM     96          ; Working buffer for rd/wr to flash
FL_DATA         EQU         FL_BUF + 2  ; Portion of buffer with user data

FL_INFO         CARTRAM     4           ;
FL_INFO.brow    EQU         FL_INFO + 0 ; 'Best' row found in flash
FL_INFO.bseq    EQU         FL_INFO + 1 ; 'Best' sequence number (two words)
FL_INFO.base    EQU         FL_INFO + 3 ; Base row to use in flash

;; ========================================================================= ;;
;;  FL_LOAD     Load data from persistent storage.                           ;;
;; ========================================================================= ;;
FL_LOAD         PROC
                PSHR    R5
                CALL    JF.INIT             ; Initialize flash routines

                ; Set up the base row for all flash activities.
                ; The reference code copies this out of JF.first, to make
                ; it easier to extend later by changing the value of
                ; FL_INFO.base.  FL_INFO.base must be a multiple of 8.
                MVI     JF.first,       R0  ;
                MVO     R0,   FL_INFO.base  ; Save base row

                ; Initialize the best row and sequence number
                MVII    #$FFFF,         R1
                MVII    #FL_INFO,       R2
                MOVR    R2,             R4
                MVO@    R1,             R4  ; FL_INFO.brow = -1 (no best)
                COMR    R1
                MVO@    R1,             R4  ; \_ FL_INFO.bseq = 00000000
                MVO@    R1,             R4  ; /

                ; Register allocation for @@__scan_loop:
                ;   R4:  FL_BUF, scratch
                ;   R3:  Rows remaining (32 downto 1)
                ;   R2:  FL_INFO
                ;   R1:  scratch
                ;   R0:  Flash sector number
                MVII    #32,            R3
                MVI     FL_INFO.base,   R0  ; get base row of our pool
@@__scan_loop:  MVII    #FL_BUF,        R1  ; \
                CALL    JF.CMD              ;  |- Read row R0 into FL_BUF
                DECLE   JF.read             ; /

                MVII    #FL_BUF,        R4
                MVI@    R4,             R1  ; \
                CMPI    #$FFFF,         R1  ;  |- skip row if MSW of row is FFFF
                BEQ     @@__next_row        ; /
```

```
                MOVR    R2,             R5  ; \_ R5 = &FL_INFO.bseq
                INCR    R5                  ; /

                CMP@    R5,             R1  ; Better than current best?
                BNC     @@__next_row        ; Smaller:  Nope.
                MVI@    R4,             R1  ; Load the LSW into R1
                BNEQ    @@__new_best        ; Bigger:   Yep

                CMP@    R5,             R1  ; MSW the same, so check LSW
                BNC     @@__next_row        ; Smaller:  Nope
                ; assume "equal to" can't happen.

@@__new_best:   MOVR    R2,             R5  ; R5 = &FL_INFO.brow
                MVI     FL_BUF + 0,     R4
                MVO@    R0,             R5  ; FL_INFO.brow = this row
                MVO@    R4,             R5  ; FL_INFO.bseq[0]  (MSW)
                MVO@    R1,             R5  ; FL_INFO.bseq[1]  (LSW)

@@__next_row:   INCR    R0                  ; \
                DECR    R3                  ;  |- Check up to 32 flash rows
                BNEQ    @@__scan_loop       ; /

                MVI@    R2,             R0  ; \   (FL_INFO.brow)
                TSTR    R0                  ;  |- Did we find any saved
                BMI     @@__no_data         ; /   data in flash?

                MVII    #FL_BUF,        R1  ; \
                CALL    JF.CMD              ;  |- Read 'best' row into FL_BUF
                DECLE   JF.read             ; /

                ; Program's data now available at FL_DATA (aka FL_BUF + 2).
@@__leave:      PULR    PC                  ; Return to the caller.

                ; No saved data found.  Initialize buffer to default value.
@@__no_data:    MVI     FL_INFO.base,   R0  ; \   Set "best row" to end of this
                ADDI    #31,            R0  ;  |- flash pool so that the next
                MVO@    R0,             R2  ; /   write goes to first row.

                ; NOTE:  The following loop merely zeros the buffer.
                ; Put different code here if your program requires it.
                MVII    #FL_BUF,        R4
                MVII    #96,            R1
                CLRR    R0
@@__zero:       MVO@    R0,             R4
                DECR    R1
                BNEQ    @@__zero

                PULR    PC
                ENDP
```

```
;; ========================================================================= ;;
;;  FL_SAVE     Copy data into JLP flash.                                    ;;
;;                                                                           ;;
;;  This code copies the 94 words of data at FL_DATA into JLP's flash       ;;
;;  storage.  It expects FL_INFO to contain valid information about the     ;;
;;  state of flash, either from a previous call to FL_LOAD or FL_SAVE.      ;;
;;                                                                           ;;
;;  This code also expects the JF.CMD state to already be initialized.      ;;
;; ========================================================================= ;;
FL_SAVE         PROC
                PSHR    R5

                ; Increment to next row in pool, modulo 32
                MVI     FL_INFO.brow,   R0  ; Get absolute row number
                SUB     FL_INFO.base,   R0  ; Subtract off the base row
                INCR    R0                  ; \_ Next row, modulo 32
                ANDI    #31,            R0  ; /
                ADD     FL_INFO.base,   R0  ; Add back the base row
                MVO     R0,   FL_INFO.brow  ; Store updated row

                ; If we moved into the first row of a sector, erase the sector
                MOVR    R0,             R1
                ANDI    #7,             R1
                BNEQ    @@__no_erase

                CALL    JF.CMD
                DECLE   JF.erase
@@__no_erase:
                ; Increment sequence number for this record
                MVI     FL_INFO.bseq+0, R2  ; MSW if sequence number
                MVI     FL_INFO.bseq+1, R1  ; LSW if sequence number

                ADDI    #1,             R1  ; Increment the sequence number
                ADCR    R2

                MVO     R2, FL_INFO.bseq+0
                MVO     R1, FL_INFO.bseq+1

                ; Insert sequence number in FL_BUF
                MVO     R2, FL_BUF + 0
                MVO     R1, FL_BUF + 1

                ; Now copy this into the row # that's in R0
                MVII    #FL_BUF,        R1
                CALL    JF.CMD
                DECLE   JF.write

@@__leave       PULR    PC
                ENDP
```

Simple Flags and Other Rarely Changing Data

If you need to store simple flags or values that do not change very often, you may be able to avoid the complication of wear leveling. Values that will change less than 10,000 times during the life of the cartridge do not need wear leveling.

Examples include:

- Permanently unlocking phases of the game–eg. Christmas Carol's "boss beaten" flag
- Cartridge customization (serial number, purchaser's name, etc.)
- Production-test-passed flag
- Game preferences–eg. controller settings, etc.

For these sorts of values, you can simply use the read, write and erase commands with each update, and pick a flash row relative to `JF.first` to store the data.

Note: Left Turn Only recommends that you include a production test feature in your program. This simplifies product testing, as the cartridges now can "test themselves." LTO provides a standard production test that you can integrate with your program if you desire. It uses flash in the manner described above to store the "production test passed" flag.

Tooling and Development

Left Turn Only recommends using jzIntv and SDK-1600 to develop and test your game on the computer of your choice. LTO also provides hardware adaptors to allow testing your JLP-based game in-system using a CC3 alongside a JLP board.

[... work in progress …]