

Chapter 35 - Disk and File I/O

Intuition Engine exposes one disk volume through a small MMIO block. BASIC uses the same block for LOAD, SAVE, BLOAD, COMPILE, TRANSPILE, ASSEMBLE, direct-mode DIR, and direct-mode TYPE. Machine code can use the registers directly, but the examples here use BASIC POKE8, POKE32, PEEK8, and PEEK32 so they can be typed on the machine.

35.1 Names and Volume Rules

Every name in this chapter is relative to the IE disk volume. "GAME.BAS" names an entry at the root of the volume. "MUSIC/TITLE.MOD" names an entry in a volume subdirectory.

Names are rejected if they:

- begin with /
- contain . .

A rejected name sets FILE_STATUS to 1 and FILE_ERROR_CODE to 3 (FILE_ERR_PATH_TRAVERSAL). Reads are case-insensitive when an exact-case match is not present. Writes create or replace the named entry; they do not append.

35.2 Register Block

The legacy block starts at \$F2200 and spans 32 bytes. Registers are 32-bit unless noted. IE64 also exposes the separate 64-bit FILE_DATA_PTR64 data-buffer register at \$F22B0 for read and write data buffers. This extends the architecture and does not replace the legacy block.

Address	Name	Access	Purpose
\$F2200	FILE_NAME_PTR	W	Pointer to a NUL-terminated name string
\$F2204	FILE_DATA_PTR	W	Pointer to the data buffer
\$F2208	FILE_DATA_LEN	W	Byte count for write; ignored by read
\$F220C	FILE_CTRL	W	Write 1 read, 2 write, 3 list
\$F2210	FILE_STATUS	R	0 OK, 1 error
\$F2214	FILE_RESULT_LEN	R	Bytes transferred by read or list
\$F2218	FILE_ERROR_CODE	R	Error code
\$F221C	FILE_READ_MAX	W	One-shot read cap; 0 is unbounded
\$F22B0	FILE_DATA_PTR64	R/W	IE64-only 64-bit data-buffer pointer

FILE_CTRL fires the operation immediately. There is no busy bit and no interrupt. When the write to FILE_CTRL returns, the status registers already describe the result.

Operation codes:

Code	Name
1	FILE_OP_READ

Code	Name
2	FILE_OP_WRITE
3	FILE_OP_LIST

Error codes:

Code	Name	Meaning
0	FILE_ERR_OK	Success
1	FILE_ERR_NOT_FOUND	Entry does not exist
2	FILE_ERR_PERMISSION	Operation was refused
3	FILE_ERR_PATH_TRAVERSAL	Name escaped the volume
4	FILE_ERR_RANGE	Staged data span is outside the File I/O address contract

35.3 Read

Set up a read like this:

1. Put a NUL-terminated name string in memory.
2. Write that address to FILE_NAME_PTR.
3. Write the destination buffer address to FILE_DATA_PTR, or on IE64 write a 64-bit destination address to FILE_DATA_PTR64.
4. Write 1 to FILE_CTRL.
5. Read FILE_STATUS.

If FILE_STATUS is 0, the file bytes are in the destination buffer and FILE_RESULT_LEN is the byte count. If status is 1, read FILE_ERROR_CODE.

The reader must provide enough destination memory. The disk block does not receive a destination capacity for reads. FILE_DATA_PTR is the legacy 32-bit path used by all CPUs. It must describe a span inside active RAM and below the sign-extended alias guard at \$FFFF0000. On IE64, FILE_DATA_PTR64 may instead describe a destination in high backed guest RAM. If the destination begins near the end of the low memory slice and continues into backed extended RAM, the bytes are still copied, provided every byte of the span is inside active RAM and the span does not enter the sign-extended alias guard. This is a byte-copy rule. It does not make a scalar word or long PEEK32/POKE32 valid when that one access straddles the same boundary.

FILE_DATA_LEN is write-side state. A read ignores it, even if it still contains the length from an earlier write. On a successful read, FILE_RESULT_LEN is the actual number of bytes copied. If the name is accepted but the read itself fails, FILE_RESULT_LEN is cleared to 0; use FILE_STATUS and FILE_ERROR_CODE as the final error test.

FILE_READ_MAX is an optional read cap. By default it is 0, which means unbounded: a read transfers the whole file. If you write a non-zero byte count to FILE_READ_MAX before triggering a read, a file larger than that count is refused with FILE_ERR_RANGE and **no bytes are copied** into the buffer, so a caller can bound a read to its own buffer size without first knowing the file length. The cap is one-shot: each read consumes it (it resets to 0), so it applies only to the very next read and never leaks into a later one. Writes and lists ignore it. The BASIC ASSEMBLE command uses this register to make sure an oversized assembly source is rejected before it can reach the assembler staging buffer. The BASIC TYPE command uses the same cap so an oversized text file is refused before any byte is printed.

The read is refused with FILE_ERR_RANGE if the destination span would reach \$FFFF0000 through the legacy pointer, wrap, start low and cross into the alias guard, or run past active RAM. In that case no partial copy is made and FILE_RESULT_LEN

is 0.

35.4 Write

Set up a write like this:

1. Put a NUL-terminated name string in memory.
2. Put the bytes to write in memory.
3. Write the name address to FILE_NAME_PTR.
4. Write the data address to FILE_DATA_PTR, or on IE64 write a 64-bit source address to FILE_DATA_PTR64.
5. Write the byte count to FILE_DATA_LEN.
6. Write 2 to FILE_CTRL.
7. Read FILE_STATUS.

Writing creates the entry if it does not exist and replaces it if it does. The register block has no append mode.

The write is refused with FILE_ERR_RANGE if the source span would reach \$FFFF0000 through the legacy pointer, wrap, start low and cross into the alias guard, or run past active RAM. The file is not partly written.

35.5 List

Set up a directory listing like this:

1. Put a NUL-terminated directory name in memory. An empty string lists the root.
2. Write the name address to FILE_NAME_PTR.
3. Write a destination buffer address to FILE_DATA_PTR.
4. Write 3 to FILE_CTRL.
5. Read FILE_STATUS.

On success, the buffer receives sorted text with CR LF after each entry and a final NUL byte. Directory entries have a trailing /. FILE_RESULT_LEN counts the text bytes but not the final NUL.

A listing is refused with FILE_ERR_RANGE if the text plus its final NUL byte would not fit in the staged destination span.

Directory listing uses the legacy staged destination path. Keep the listing buffer below \$FFFF0000 and inside active RAM.

35.6 BASIC Verbs

35.6.1 LOAD

```
LOAD "name"
```

LOAD reads a BASIC programme from disk, tokenises it, makes it the current programme, and clears variables. If the entry is not found, BASIC prints ?FILE NOT FOUND and keeps the previous programme.

35.6.2 SAVE

```
SAVE "name"
```

SAVE writes the current BASIC programme as detokenised numbered text. The saved text round-trips through LOAD.

35.6.3 BLOAD

```
BLOAD "name", addr
```

BLOAD reads raw bytes into memory at `addr`. It does not tokenise and it does not clear variables. Because the File I/O block uses the read-side 32-bit `FILE_DATA_PTR`, `addr` must be below 2^{32} ; otherwise BASIC reports `?FC ERROR`.

35.6.4 COMPILE

```
COMPILE "name"
```

COMPILE is a direct-mode command. It takes the stored BASIC programme, makes a native IE64 image from it inside the machine, and writes the result as a flat `.ie64` file. `COMPILE "DEMO"` writes `DEMO.ie64`; `COMPILE "DEMO.IE64"` keeps the suffix already supplied.

The output name is a simple filename, not a path. If the current programme was loaded from a subdirectory, the compiled image is written beside that loaded programme. If no programme has been loaded, it is written at the root of the disk volume.

COMPILE also writes a same-name text listing of the generated IE64 instructions. That assembly text is self-contained: when the programme needs runtime support, the support bytes and any bundled tokenised programme data are written as labelled `dc.b` data. The listing is for inspection or later assembly. RUN uses the `.ie64` image.

Not every stored line can become a standalone image. Direct-mode commands such as `RUN A0T`, `COMPILE`, `TRANSPILE`, `ASSEMBLE`, and `DIR` or `TYPE` are rejected. A standalone image cannot use `LOAD`. For `POKE`, `POKE8`, `POKE16`, `POKE32`, and `POKE64` inside a standalone image, integer-literal operands can be lowered directly; other expressions use the resident runtime path.

If the stored programme is empty, COMPILE prints `?NO CODE TO COMPILE` and writes no image.

35.6.5 TRANSPILE

```
TRANSPILE "name"
```

TRANSPILE is a direct-mode command. It runs the first half of COMPILE: BASIC converts the stored programme to IE64 assembly text and writes the matching assembly source file. It does not assemble that source and it does not write `name.ie64`.

The output name follows the same rule as COMPILE. If the current programme was loaded from a subdirectory, the assembly source is written beside that loaded programme. If no programme has been loaded, it is written at the root of the disk volume.

Use TRANSPILE when you want to inspect the native IE64 source that BASIC would compile, or when you want to assemble it later with ASSEMBLE.

If the stored programme is empty, TRANSPILE prints `?NO CODE TO COMPILE` and writes no source file.

35.6.6 ASSEMBLE

```
ASSEMBLE "name"
```

ASSEMBLE is a direct-mode command. It reads the matching assembly source file, assembles it inside the machine at `PROG_START`, and writes `name.ie64`. The stored BASIC programme is not used and is not changed.

The source may contain IE64 instructions, including MOV_T, labels, branch and JSR targets, dc.b, dc.w, dc.l, dc.q, align, and the standard symbolic constants known to the in-machine assembler. A conventional constants include line is accepted as a no-op so the same source can still name those constants. The zero-test branch source forms BEQZ, BNEZ, BLTZ, BGEZ, BGTZ, and BLEZ are accepted as compare-and-branch instructions against R0. Other include files, org, equ, macros, conditionals, unknown mnemonics, and unresolved symbols are rejected.

A missing source file, an unreadable source file, or a source file of about 1 MB or larger reports ?FILE ERROR IN 0. An assembly error reports ?COMPILE ERROR IN 0, and no .ie64 file is written.

This short prompt session shows the three related commands:

```
10 PRINT "MADE INSIDE IE"  
COMPILE "MADE"  
TRANSPILE "MADE"  
ASSEMBLE "MADE"  
RUN "MADE.IE64"
```

COMPILE writes both the flat image and the generated assembly source. TRANSPILE writes only the assembly source. ASSEMBLE reads that source and writes MADE.ie64 again. The final RUN starts the flat IE64 image.

35.6.7 DIR

```
DIR  
DIR "subdir"
```

DIR is a direct-mode command. It lists the root or the named directory and prints entries separated by CR LF. Its output depends on the current disk volume, so it is shown here as a syntax template rather than a transcript with fixed expected output.

35.6.8 TYPE

```
TYPE "name"
```

TYPE is a direct-mode command. It reads a text file from the disk volume and prints it to the screen. The name is required and must be quoted. It may name a file in a subdirectory, subject to the same volume rules as LOAD, SAVE, and DIR.

Before printing anything, BASIC reads the file into the resident File I/O buffer and checks that it is text. Tab, line feed, carriage return, ordinary printable characters, and bytes \$80 through \$FF are accepted. NUL, DEL, and other low control bytes are treated as binary data. If such a byte is found, BASIC prints ?NOT A TEXT FILE and does not print the file contents.

Line endings are made suitable for the terminal as the file is printed. A file with line feed, carriage return, or carriage return plus line feed line endings displays with each new line at the left edge. If the file does not end with a line break, BASIC adds one before returning to the Ready prompt.

The file must fit in the File I/O buffer. If it is too large, BASIC prints ?FILE TOO LARGE before any byte is staged. If the name is not found, BASIC prints ?FILE NOT FOUND; other file errors print ?FILE ERROR.

This is a prompt session, not a stored programme:

```
SAVE "NOTE.BAS"  
TYPE "NOTE.BAS"
```

The first command writes the current programme as text. The second command prints that saved text file back to the screen.

35.7 Typed MMIO Example

This BASIC listing writes two bytes to NOTE.TXT, clears the buffer, reads the file back, and prints the status and byte values.

```
10 REM NAME BUFFER AND DATA BUFFER
20 N=&H00720000:D=&H00720100
30 REM "NOTE.TXT",0
40 POKE8 N,78:POKE8 N+1,79:POKE8 N+2,84:POKE8 N+3,69
50 POKE8 N+4,46:POKE8 N+5,84:POKE8 N+6,88:POKE8 N+7,84
60 POKE8 N+8,0
70 REM FILE DATA "IE"
80 POKE8 D,73:POKE8 D+1,69
90 POKE32 &H000F2200,N
100 POKE32 &H000F2204,D
110 POKE32 &H000F2208,2
120 POKE32 &H000F220C,2
130 PRINT "WRITE ";PEEK32(&H000F2210)
140 REM CLEAR THE BUFFER AND READ THE FILE BACK
150 POKE8 D,0:POKE8 D+1,0
160 POKE32 &H000F220C,1
170 PRINT "READ ";PEEK32(&H000F2210)
180 PRINT "LEN ";PEEK32(&H000F2214)
190 PRINT PEEK8(D);PEEK8(D+1)
```

Expected result:

```
WRITE 0
READ 0
LEN 2
73 69
```

Lines 40 to 60 build the byte string "NOTE.TXT",0. Lines 80 to 120 provide the two data bytes and fire the write operation. Line 150 clears the RAM buffer so the readback cannot be mistaken for leftover data. Lines 160 to 190 fire the read, print the byte count, and then print the two returned bytes.

35.8 Small-CPU Access

Full-address CPUs write the register block directly. The 6502 and Z80 use their documented MMIO translation apertures to reach the same block. The block also accepts byte-width writes: four writes to consecutive byte offsets compose a 32-bit register value in little-endian order.

Writing byte 0 of FILE_CTRL with 1, 2, or 3 triggers the matching operation. Writes to the upper bytes of FILE_CTRL do not trigger an operation.

35.9 Limits and Side Effects

Names longer than 255 bytes are truncated before lookup. Reads ignore FILE_DATA_LEN; writes use it as the number of bytes to copy from FILE_DATA_PTR. Successful reads and lists set FILE_RESULT_LEN to the number of bytes returned. Lists

set `FILE_RESULT_LEN` to 0 on failure. Reads whose path is accepted but whose file cannot be read also set `FILE_RESULT_LEN` to 0. After a failed write, do not rely on `FILE_RESULT_LEN`.

`FILE_ERR_RANGE (4)` means the staged transfer span could not be represented safely: it would reach the sign-extended alias guard at `$FFFF0000` through the legacy pointer, wrap, start low and cross into that guard, or run past active RAM. The block refuses the whole transfer.

Read destinations are ordinary bus addresses. They may be in low RAM or backed extended RAM, and the transfer may cross that boundary because the file block writes one byte at a time. Use the IE64 `FILE_DATA_PTR64` extension for high read or write data buffers that cannot be represented by the legacy pointer. The program must still choose an address range that is large enough for the file.

The block is synchronous and single-operation. Program code should not change `FILE_NAME_PTR`, `FILE_DATA_PTR`, or `FILE_DATA_LEN` while an operation is in progress, although in normal use there is no observable busy interval.