

# Chapter 32 - Coprocessor and Cross-CPU Calls

---

The six processors of Part IV sit on the same Intuition Engine bus. They see shared RAM, they can reach the same cards, and they can pass work to each other without changing machines. A BASIC program running on IE64 can submit work to a 6502 service, and another CPU can poll for the answer later.

The mechanism is the **coprocessor system**. It consists of MMIO registers, one ring buffer per worker CPU, completion records, and five BASIC entry points: `COSTART`, `COSTOP`, `COCALL`, `COSTATUS`, and `COWAIT`.

The reader path in this chapter is BASIC first, then direct `POKE` and `PEEK` of the same registers. The runnable example does not require a service program to be present; it proves the status and error path that every real service call uses.

## 32.1 Model

---

A **coprocessor** is one of the six CPUs acting as a worker for another CPU. The calling CPU submits a request descriptor; the worker reads it, writes a response descriptor, and marks the ticket complete.

The normal sequence is:

1. Start a worker with `COSTART`.
2. Put request bytes in shared RAM.
3. Call `COCALL` to enqueue the request.
4. Use `COSTATUS` to poll, or `COWAIT` to wait.
5. Read the response bytes.
6. Stop the worker with `COSTOP` when it is no longer needed.

Worker state is reported through `COPROC_WORKER_STATE`:

State bit	Meaning
Clear	No worker is running for that CPU type.
Set	A worker for that CPU type is running.

Failures are always visible through `COPROC_CMD_STATUS` and `COPROC_CMD_ERROR`. `COCALL` also returns ticket 0 when enqueue fails, so a BASIC programme can reject the request immediately.

## 32.2 CPU Type Codes

---

These are the CPU type values used by BASIC and MMIO:

CPU type	Code	Service suffix	Worker RAM
IE32	1	.IE32	\$200000 to \$27FFFF
IE64	2	.IE64	\$3A0000 to \$41FFFF
6502	3	.IE65	\$300000 to \$30FFFF
M68K	4	.IE68	\$280000 to \$2FFFFFF
Z80	5	.IE80	\$310000 to \$31FFFF

CPU type	Code	Service suffix	Worker RAM
x86	6	.IE86	\$320000 to \$39FFFF

COSTART checks that the service suffix matches the requested CPU type. A mismatch reports an error rather than starting the wrong worker.

## 32.3 BASIC Entry Points

### 32.3.1 COSTART

```
COSTART cpuType, "serviceName"
```

Starts the named service on the selected CPU. The service must be available in IE storage and must match the CPU type. The command records success or failure in COPROC\_CMD\_STATUS and COPROC\_CMD\_ERROR.

### 32.3.2 COSTOP

```
COSTOP cpuType
```

Stops the worker for cpuType. Requests that have not completed are marked COPROC\_TICKET\_WORKER\_DOWN.

### 32.3.3 COCALL

```
ticket = COCALL(cpuType, op, reqPtr, reqLen, respPtr, respCap)
```

Enqueues a request. The request is reqLen bytes starting at reqPtr; the response can use up to respCap bytes at respPtr. The op value is a service-defined operation code.

For any non-zero reqLen or respCap, the corresponding pointer must be a public low32 span allocated by MEMALLOC. Invalid raw pointers raise ?FC ERROR. Zero-length request or response spans may use pointer 0.

CCALL returns a non-zero ticket on success. It returns 0 on enqueue failure; read COPROC\_CMD\_ERROR to learn why.

Think of the ticket as a bus receipt. The caller owns the request bytes, the worker owns the service code, and the coprocessor block is the clerk between them. That model is the same whether the worker plays audio, fills a table, prepares a sprite, or converts data for another chip.

### 32.3.4 COSTATUS

```
status = COSTATUS(ticket)
```

Polls a ticket without waiting:

Value	Constant	Meaning
0	COPROC_TICKET_PENDING	Queued, not yet started
1	COPROC_TICKET_RUNNING	Worker is processing
2	COPROC_TICKET_OK	Completed successfully
3	COPROC_TICKET_ERROR	Worker returned an error

Value	Constant	Meaning
4	COPROC_TICKET_TIMEOUT	Wait deadline expired
5	COPROC_TICKET_WORKER_DOWN	Worker is no longer running

### 32.3.5 COWAIT

```
COWAIT ticket [, timeoutMs]
```

Waits for a ticket to finish or for the timeout to expire. If the timeout is omitted, BASIC uses 1000 milliseconds. COWAIT is a statement, not a function; call CSTATUS(ticket) afterwards to read the final ticket status.

## 32.4 Runnable Status Example

This listing intentionally submits a request when no 6502 worker is running. It is useful because it exercises the same command, ticket, and error registers that a real service call uses, without requiring a service program first.

Type this listing:

```
10 REM PUT REQUEST AND RESPONSE BUFFERS IN SHARED RAM
20 REQ=MEMALLOC(8,4):RESP=MEMALLOC(16,4)
30 POKE32 REQ,123
40 REM ASK CPU TYPE 3, THE 6502, TO RUN OPERATION 1
50 T=COCALL(3,1,REQ,4,RESP,4)
60 PRINT "TICKET ";T
70 PRINT "CMD ";PEEK32(&H000F2348)
80 PRINT "ERR ";PEEK32(&H000F234C)
90 PRINT "WORKERS ";PEEK32(&H000F2374)
```

Expected result:

```
TICKET 0
CMD 1
ERR 6
WORKERS 0
```

CMD 1 means command error. ERR 6 is COPROC\_ERR\_NO\_WORKER. Once a service is running, the same COCALL form returns a non-zero ticket instead.

Lines 20 and 30 create the same request buffer a real worker would read. Line 50 is the important control write hidden behind the BASIC function: it enqueues a request for CPU type 3. The three PEEK lines then show the command status, exact error code, and live worker mask in the MMIO block.

## 32.5 MMIO Register Block

The coprocessor block occupies \$F2340 to \$F238F. The extended status block occupies \$F23B0 to \$F23BF.

Address	Name	Access	Purpose
\$F2340	COPROC_CMD	W	Command to run
\$F2344	COPROC_CPU_TYPE	W	Target CPU type

Address	Name	Access	Purpose
\$F2348	COPROC_CMD_STATUS	R	0 ok, 1 error
\$F234C	COPROC_CMD_ERROR	R	Error code
\$F2350	COPROC_TICKET	R/W	Ticket ID
\$F2354	COPROC_TICKET_STATUS	R	Last-pollled ticket status
\$F2358	COPROC_OP	W	Service operation code
\$F235C	COPROC_REQ_PTR	W	Request buffer pointer
\$F2360	COPROC_REQ_LEN	W	Request buffer length
\$F2364	COPROC_RESP_PTR	W	Response buffer pointer
\$F2368	COPROC_RESP_CAP	W	Response buffer capacity
\$F236C	COPROC_TIMEOUT	W	Timeout in milliseconds
\$F2370	COPROC_NAME_PTR	W	Pointer to service-name string
\$F2374	COPROC_WORKER_STATE	R	Bitmask of running workers
\$F2378	COPROC_STATS_OPS	R	Total operations dispatched
\$F237C	COPROC_STATS_BYTES	R	Total bytes processed
\$F2380	COPROC_IRQ_CTRL	R/W	bit 0 enables IRQ on completion
\$F2384	COPROC_DISPATCH_OVERHEAD	R	Last dispatch overhead, nanoseconds
\$F2388	COPROC_COMPLETED_TICKET	R	Last completed ticket ID
\$F23B0	COPROC_RING_DEPTH	R	Selected CPU ring occupancy
\$F23B4	COPROC_WORKER_UPTIME	R	Selected worker uptime, seconds
\$F23B8	COPROC_STATS_RESET	W	Write 1 to clear statistics
\$F23BC	COPROC_BUSY_PCT	R	Rolling worker busy percentage

Read COPROC\_RING\_DEPTH and COPROC\_WORKER\_UPTIME only after writing COPROC\_CPU\_TYPE; the selected CPU type chooses which worker is queried.

COPROC\_BUSY\_PCT is calculated over the recent coprocessor activity window, about one second. Writing 1 to COPROC\_STATS\_RESET clears the operation and byte counters and starts a fresh busy-percentage window.

The 6502 and Z80 cannot address \$F2340 directly. They use a gateway at \$F200 to \$F24F. The gateway keeps the same register offsets, so \$F204 reaches COPROC\_CPU\_TYPE and \$F210 reaches COPROC\_TICKET. Byte writes compose 32-bit values in little-endian order.

## 32.6 Commands

Write input registers first, then write one of these values to COPROC\_CMD:

Code	Constant	Effect
1	COPROC_CMD_START	Start a worker
2	COPROC_CMD_STOP	Stop a worker

Code	Constant	Effect
3	COPROC_CMD_ENQUEUE	Submit a request and return a ticket
4	COPROC_CMD_POLL	Poll a ticket's status
5	COPROC_CMD_WAIT	Wait for ticket completion or timeout
6	COPROC_CMD_START_MEM	Start a worker from a guest-RAM image

After the command write, read COPROC\_CMD\_STATUS. If it is 1, read COPROC\_CMD\_ERROR.

COPROC\_CMD\_START\_MEM uses the selected COPROC\_CPU\_TYPE and reads the service image from guest RAM at COPROC\_REQ\_PTR for COPROC\_REQ\_LEN bytes. It is the raw MMIO form used by self-contained programmes that already carry their worker image in memory. Ordinary BASIC programmes normally use CSTART for named worker files.

## 32.7 Error Codes

Code	Constant	Meaning
0	COPROC_ERR_NONE	Success
1	COPROC_ERR_INVALID_CPU	COPROC_CPU_TYPE is not in 1 to 6
2	COPROC_ERR_NOT_FOUND	Service was not found
3	COPROC_ERR_PATH_INVALID	Service name is malformed
4	COPROC_ERR_LOAD_FAILED	Service could not be loaded
5	COPROC_ERR_QUEUE_FULL	Worker's ring buffer has no free slots
6	COPROC_ERR_NO_WORKER	No worker is running for this CPU type
7	COPROC_ERR_STALE_TICKET	Ticket has already been reaped

## 32.8 Ring Buffer Layout

Each worker has a 16-slot ring in shared RAM:

Item	Address or size
Mailbox base	\$790000
Ring base	$\$790000 + \text{cpuIndex} * \$300$
Ring header	head, tail, capacity bytes
Request area	16 request descriptors, 32 bytes each
Response area	16 response descriptors, 16 bytes each

Request descriptor words:

Offset	Meaning
\$00	Ticket
\$04	CPU type
\$08	Operation code

Offset	Meaning
\$0C	Timeout
\$10	Request pointer
\$14	Request length
\$18	Response pointer
\$1C	Response capacity

Response descriptor words:

Offset	Meaning
\$00	Ticket
\$04	Ticket status
\$08	Service result code
\$0C	Response length

Most BASIC programmes never touch these descriptors directly. They are included so machine-code services can be written and checked by hand in IE Mon.

## 32.9 Positive Service Pattern

When a service is present, the BASIC side looks like this:

```

10 REQ=MEMALLOC(8,4)
20 RESP=MEMALLOC(16,4)
30 POKE32 REQ,42
40 COSTART 3,"MUL.IE65"
50 T=COCALL(3,1,REQ,4,RESP,4)
60 IF T=0 THEN PRINT "SUBMIT FAILED":END
70 COWAIT T,1000
80 S=COSTATUS(T)
90 IF S<>2 THEN PRINT "STATUS ";S:END
100 PRINT "ANSWER ";PEEK32(RESP)
110 COSTOP 3

```

The service defines operation 1, reads the request word at REQ, writes a result word at RESP, then marks the ticket COPROC\_TICKET\_OK. The calling side does not need to know which instructions the worker used; it only needs the request and response format.

### 32.10 IRQ on Completion

Bit 0 of COPROC\_IRQ\_CTRL enables an interrupt when any ticket completes. The interrupt vector depends on the listening CPU; see Chapter 31, section 31.3. The handler reads COPROC\_COMPLETED\_TICKET to learn which ticket finished.

This is useful when the listening CPU has other work to do and does not want to block in COWAIT. The usual pattern is to submit several requests, return to other work, and let the completion interrupt wake up the response handler.

## 32.11 What Comes Next

---

Chapter 33 covers IE Mon, the interactive monitor that lets you single-step any CPU, examine registers, set breakpoints and watchpoints, and inspect the bus from a command prompt. Chapter 34 covers IE Script, the scripting surface that drives the monitor from BASIC-like programs.