P2 Assembly Programming

A Human-Centered Approach to Parallel Processing

June 2026

Version 3.0

Tutorial Philosophy

Learn by doing, celebrate progress, have fun!

Code Block Colors:

- **Green** – Spin2
- **Yellow** – PASM2
- **Purple** – CORDIC
- **Blue** – Multi-COG
- **Red** – Antipattern

Teaching Elements:

- **Sidetracks** – deeper dives
- **Medicine Cabinet** – simpler alternatives
- **Your Turn** – hands-on exercises
- **Interludes** – stories & context

Contents

Copyright and License

Trademarks

Disclaimer

Dedication

To deSilva — *Whose legendary P1 assembly tutorial taught a generation of programmers that assembly language could be approachable, enjoyable, and even fun. Your unique voice—combining technical precision with human warmth—showed us that great documentation teaches not just the mind, but speaks to the spirit of discovery.*

To the Propeller Community — *Who have spent countless hours exploring, documenting, and sharing their knowledge. From the early P1 pioneers to today's P2 innovators, your collective wisdom makes this manual possible.*

To Future Makers — *May you find in these pages the same joy of discovery that we experienced. The Propeller 2 is more than a microcontroller—it's an invitation to think differently about computing. Welcome to the journey.*

"*The best way to predict the future is to invent it.*" — Alan Kay

Acknowledgments

This manual stands on the shoulders of giants. We gratefully acknowledge:

Primary Contributors

deSilva - For creating the gold standard of microcontroller documentation with the P1 Assembly Tutorial. Your pedagogical approach, combining technical depth with human empathy, remains unmatched. This manual attempts to honor your legacy while adapting to the P2's capabilities.

Iron Sheep Productions LLC (Stephen M Moraco) - For extensive P2 documentation efforts, community tools, and the vision of creating an AI-optimized knowledge base. Your systematic approach to extracting and organizing P2 knowledge made this comprehensive manual possible.

Chip Gracey - Creator of the Propeller architecture. Thank you for giving us a microcontroller that thinks differently and challenges us to do the same.

Community Contributors

The Parallax Forums Community - Your questions, answers, code examples, and endless experimentation have created a living knowledge base that no single author could match.

Early P2 Adopters - Who dealt with evolving documentation, changing specifications, and still produced amazing projects that showed us what was possible.

Technical Reviewers

Special thanks to those who reviewed drafts, tested code examples, and provided invaluable feedback:

- The P2 Documentation Team at Parallax
- Community members who beta-tested examples
- Everyone who reported errors and suggested improvements

Inspiration

The MIT AI Lab - For showing us that technical documentation can have personality

Donald Knuth - For proving that programming texts can be literature

The Demoscene Community - For pushing hardware beyond its limits and inspiring us to do the same

Production Notes

This manual was created using:

- Knowledge extracted from official Parallax technical documentation and OBEX (Object Exchange) community contributions
- AI-assisted content generation trained on deSilva's writing style
- Community validation and real-world testing
- A commitment to making parallel processing accessible to everyone

"If I have seen further, it is by standing on the shoulders of giants." — Isaac Newton

Any errors, omissions, or dad jokes that fell flat are entirely the responsibility of the authors, not our distinguished contributors.

Preface: Welcome to the Journey

Well, here we are! You're about to embark on a journey into the heart of the Propeller 2, and I promise you, it's going to be quite different from what you might expect.

A Different Kind of Processor

The Propeller 2 isn't just another microcontroller. Oh no, it's something far more interesting. Imagine, if you will, eight independent processors (we **CALL** them COGs) all working together in perfect harmony, sharing a common memory space, yet each running their own programs at full speed. No interrupts fighting for attention, no complex priority schemes, just eight brains working in parallel.

And if you think this sounds terribly complicated, you're probably right… but here's the secret: it's actually simpler than traditional architectures once you understand the philosophy.

About This Manual

This manual follows in the footsteps of deSilva's legendary P1 tutorial. What does that mean? It means we're going to:

1. **Start with working code** - You'll be blinking LEDs before you know what hit you
2. **Learn by doing** - Theory is important, but nothing beats hands-on experience
3. **Have some fun** - Yes, assembly language can actually be enjoyable!
4. **Be honest about complexity** - When something is hard, we'll admit it and then show you how to handle it

Who Is This For?

Are you a complete beginner to assembly language? Welcome! We'll take good care of you.

Are you a grizzled veteran who's been writing assembly since the 6502? Welcome! The P2 will still surprise you.

Are you somewhere in between? Perfect! This is exactly where you want to be.

The only requirement is curiosity and a willingness to think a bit differently about how computers can work.

How to Use This Manual

The Fast Track — If you're impatient (and who isn't?), jump straight to Chapter 1. Get that LED blinking. Feel the satisfaction. Then come back here when you're ready for more.

The Scenic Route — Read the chapters in order. Each builds on the previous one, and I've hidden little gems of knowledge throughout that will make later chapters easier.

The Reference Approach — Already know what you're looking for? The table of contents and index are your friends. The appendices contain every instruction, every Smart Pin mode, every CORDIC operation.

What Makes the P2 Special?

Let me count the ways:

- **8 symmetric COGs** - No master/slave relationships, all COGs are equal
- **64 Smart Pins** - Each pin has its own processor for I/O operations
- **CORDIC engine** - Hardware trigonometry and coordinate transformations
- **Hardware multiply/divide** - Finally! Real math at hardware speed
- **512KB of RAM** - Shared by all COGs with deterministic access timing
- **No interrupts** - Well, actually there are interrupts, but we'll talk about why you probably don't want them

A Note on Our Approach

The best technical documentation remembers you're human. You'll get frustrated. You'll make mistakes. Your code won't work the first time (or the second, or sometimes even the third).

That's normal. That's learning. And that's why this manual provides plenty of "medicine" along the way - simpler alternatives, working examples, and the occasional moment of levity to keep your spirits up.

The deSilva Spirit

Throughout this manual, you'll encounter the teaching spirit of deSilva. When you see phrases like:

- "Well, …" - We're about to correct a common assumption
- "Uff!" - We just got through something complex
- "Have Fun!" - We mean it, this stuff is actually enjoyable

These aren't just quirks; they're signals that we remember you're human and we're on this journey together.

Ready?

Take a deep breath. Pour yourself your favorite beverage. Open your development environment.

Let's make some magic happen with the Propeller 2!

"*The Propeller architecture is based on the simple idea that the best way to avoid the complexity of interrupts is to have enough processors that you don't need them.*"
— Chip Gracey, creator of the Propeller

Turn the page, and let's blink that LED! →

Chapter 1: Your First Spin

Let's blink an LED and change your life

Why P2?

Before we dive into code, let me tell you why you're in for something different.

If you've fought with interrupt priority conflicts on an ARM, watched your timing jitter because of cache misses, or discovered that the UART you need is only available on pins you're already using… well, the P2 was designed by someone who got tired of those problems too.

Here's the P2 philosophy in a nutshell:

Instead of one processor fighting with interrupts, you get eight complete, identical processors (COGs) that run truly in parallel. Your serial handler never delays your motor control. Your sensor sampling never misses a deadline. Each task owns its own processor.

Instead of fixed peripherals, every one of the 64 pins contains its own programmable state machine. Any pin can become a UART, PWM output, quadrature encoder, ADC - whatever you need, wherever you need it.

Instead of timing that depends on cache luck, the hub memory has deterministic access. Your timing loops work the same way every time.

Instead of calling math libraries, there's a hardware CORDIC that gives you sine, cosine, and arctangent (via vector rotate and convert) with results ready in exactly 55 clocks. Every time.

Does this mean P2 is perfect for everything? Of course not. But if your projects involve multiple real-time tasks, precise timing, video or audio generation, or just running out of peripheral pins - you're in the right place.

For a full comparison to ARM, ESP32, Arduino, and PIC platforms, see Appendix A. But you probably want to blink that LED first, don't you?

The Hook: Making Light

I know you're absolutely crazy to have your first instruction executed, so let's not waste any time. Here's a complete PASM2 program that blinks an LED on pin 56 (that's the built-in LED on the P2 Eval board):

```
1 CON
2   _clkfreq = 200_000_000        ' 200 MHz system clock
3
```

continues on next page →

↪ *continued from previous page*

```
4  DAT
5  ' LED Blinker - Your first PASM2 program!
6          ORG     0               ' Start at COG address 0
7
8          DRVH    #56             ' Drive pin 56 high (LED on)
9          WAITX   ##50_000_000    ' Wait 0.25 seconds at 200MHz
10         DRVL    #56             ' Drive pin 56 low (LED off)
11         WAITX   ##50_000_000    ' Wait 0.25 seconds
12         JMP     #$-6            ' Jump back 6 longs (## adds hidden AUGD)
```

That's it! Five lines of code and you have a blinking LED. Load this into any COG and watch the magic happen.

What's Really Happening

Well, now that you've seen it work (you did try it, right?), let's talk about what's actually going on here.

The Instructions Decoded

`ORG 0` - This tells the assembler to start placing code at COG address 0. Every COG has its own private 512 longs (2KB) of memory, and execution always starts at address 0 when a COG is loaded.

`DRVH #56` - This drives pin 56 high (3.3V). The 'h' means high. The '#' means we're using an immediate value (the actual number 56) rather than the contents of register 56. One instruction, and your LED is on!

`WAITX ##50_000_000` - This waits for 50 million clock cycles. At 200 MHz, that's 0.25 seconds. Notice the double '##'? That means this is a 32-bit immediate value. Single '#' only gives us 9 bits.

`DRVL #56` - Drive low. LED off. You get the pattern.

`JMP #$-6` - Jump back 6 longs. The '$' means "current address", so '$-6' means "6 addresses back from here". Why 6? Each `##` immediate generates a hidden **AUGD** instruction (it augments the `WAITX` destination operand), so we have 6 longs total: **DRVH**, **AUGD**, **WAITX**, **DRVL**, **AUGD**, **WAITX**. Infinite loop achieved!

But Wait, There's More!

"Hold on," you might say, "how does this even get into the COG?"

Ah, excellent question! In the real world, you'd typically launch this from Spin2 (the high-level language) like this:

```
CON
  _clkfreq = 200_000_000  ' 200 MHz system clock

PUB main()
    COGINIT(COGEXEC_NEW, @blink_code, 0)  ' Start PASM2 in new COG
    repeat  ' Keep the main COG alive

DAT
        ORG     0
blink_code
        DRVH    #56
        WAITX   ##50_000_000
        DRVL    #56
        WAITX   ##50_000_000
        JMP     #$-6
```

The **COGINIT** instruction loads your PASM2 code from hub memory into a fresh COG and starts it running. Meanwhile, your Spin2 code keeps running in its own COG. You now have parallel processing!

Sidetrack

The Clock Preamble

Notice the `CON` section at the top of that example? Every P2 program needs to configure its system clock:

```
CON
  _clkfreq = 200_000_000  ' 200 MHz system clock
```

This tells the P2 to run at 200 MHz using your board's crystal oscillator. Without it, the chip runs at a sluggish ~20 MHz on its internal RC oscillator—and timing-dependent code (including DEBUG output) won't behave as expected.

At 200 MHz with most instructions taking 2 clocks, each COG executes approximately 100 million instructions per second (100 MIPS). With 8 COGs running in parallel, that's 800 MIPS of total processing power—and that's before Smart Pins start handling I/O autonomously.

From here on, we'll omit this preamble from examples to keep them focused on the concept being taught. When you create your own files, always include it at the top before your `PUB` or `DAT` sections.

Let's Make It Better

The blinker works, but it's a bit rigid, isn't it? What if we want to change the blink rate? Let's use a register:

```
        ORG     0

        MOV     delay, ##50_000_000     ' Set delay to 0.25 sec
                                        '  (at 200 MHz)

.blink  DRVH    #56                     ' LED on
        WAITX   delay                   ' Wait
        DRVL    #56                     ' LED off
        WAITX   delay                   ' Wait
        JMP     #.blink                 ' Repeat forever

delay   LONG    0                       ' Storage for delay value
```

Uff! Look at that - we're using a register now! The **MOV** instruction copies our delay value into a register (which we cleverly named 'delay'). Now we can change the blink rate by modifying just one value.

A note on terminology: P2 documentation often uses "register" to refer to any long in COG RAM. Unlike ARM or x86 where registers are a small, special set (R0-R15, EAX, etc.), every COG RAM location can be used as a general-purpose register. However, the last 16 locations (496-511) have special functions: addresses 496-503 are dual-purpose (usable as RAM if interrupts aren't used), and 504-511 are special-purpose registers (PTRA, PTRB, DIRA, DIRB, OUTA, OUTB, INA, INB). When you see "register" in P2 context, think "COG RAM location."

Understanding COGs

Here's something important: each COG is a complete processor with its own memory. When we loaded our blink program, it was copied from hub memory into COG memory. The COG then executes it independently, without any further connection to hub memory (unless we explicitly read or write to it).

Think of it like this:

- **Hub memory** is the meeting place (512KB shared by all)
- **COG memory** is private workspace (2KB per COG)
- Loading a COG is like making a photocopy - the COG gets its own copy to run

This is why our blinker keeps running even after the Spin2 code that launched it goes into an infinite repeat loop. The COG is independent!

Your Turn: Experiments

Now for the fun part. Try these modifications:

Experiment 1: Different Patterns

Make the LED blink in a pattern: short-short-long (like SOS):

```
1          ORG     0
2
3          MOV     short, ##20_000_000    ' 0.1 second (at 200 MHz)
4          MOV     long_d, ##60_000_000   ' 0.3 seconds (at 200 MHz)
5
6  .pattern DRVH    #56                     ' Short pulse 1
7          WAITX   short
8          DRVL    #56
9          WAITX   short
10
11         DRVH    #56                    ' Short pulse 2
12         WAITX   short
13         DRVL    #56
14         WAITX   short
15
16         DRVH    #56                    ' Long pulse
17         WAITX   long_d
18         DRVL    #56
```

continues on next page →

↪ *continued from previous page*

```
        WAITX   long_d

        JMP     #.pattern

short   LONG    0
long_d  LONG    0
```

Experiment 2: Multiple LEDs

Blink LEDs on pins 56 and 57 alternately:

```
        ORG     0

.loop   DRVH    #56                     ' LED 56 on
        DRVL    #57                     ' LED 57 off
        WAITX   ##50_000_000            ' 0.25 sec at 200 MHz

        DRVL    #56                     ' LED 56 off
        DRVH    #57                     ' LED 57 on
        WAITX   ##50_000_000            ' 0.25 sec at 200 MHz

        JMP     #.loop
```

Experiment 3: Fading (Advanced)

This one's a bit tricky - we'll use PWM to fade the LED:

```
        ORG     0

        DIRL    #56                     ' Reset the pin before configuring
        WRPIN   ##P_PWM_TRIANGLE, #56   ' Configure pin 56 for PWM
        WXPIN   ##$0100_0001, #56       ' Frame = 256 base periods
        DIRH    #56                     ' Enable the pin

.fade   WYPIN   level, #56              ' Set duty cycle
        WAITX   ##100_000               ' Small delay
        ADD     level, #1               ' Increment brightness
        AND     level, #$FF             ' Wrap at 256
        JMP     #.fade
```

continues on next page →

↪ *continued from previous page*

```
13
14 level   LONG    0
```

Don't worry if the PWM example seems complex - we'll cover Smart Pins in detail in Chapter 8!

The Medicine Cabinet

Feeling overwhelmed? Here's the simplified prescription:

Minimum viable blinker - Just 3 instructions:

```
1 .loop   DRVNOT  #56          ' Toggle pin 56
2         WAITX   ##50_000_000 ' 0.25s at 200MHz
3         JMP     #.loop       ' Repeat
```

The **DRVNOT** instruction toggles a pin - if it's high, make it low; if it's low, make it high. Sometimes simpler is better!

Sidetrack

Why Start at Address 0?

You might wonder why COG code always starts at address 0. It's actually quite elegant: When a COG is started with `COGINIT`, the hardware:

1. Stops the COG (if it was running)
2. Copies 504 longs from hub to COG memory (addresses 0-503); the top 8 (504-511) are special hardware registers, not loaded
3. Starts execution at COG address 0

This means every COG program starts fresh, with a clean slate. No residual state, no confusion. It's like each COG gets a fresh brain transplant every time it starts!

The last 16 longs (addresses 496-511) have special functions: 496-503 are dual-purpose (usable as RAM if interrupts not used), and 504-511 are special-purpose registers (PTRA, PTRB, DIRA, DIRB, OUTA, OUTB, INA, INB). We'll explore these later.

Common Gotchas

Before we move on, let me save you some debugging time:

1. **Forgetting the ##** - Using `WAITX #25_000_000` will NOT wait for 0.25 seconds! Single # only allows values up to 511.
2. **Wrong pin number** - The P2 Eval board's LEDs are on pins 56-63. The P2 Edge module might have different assignments.
3. **Clock setup required** - P2 boots at ~20MHz (internal RC oscillator). Most programs configure 200MHz with a crystal. Our examples assume 200MHz - adjust **WAITX** values if your clock differs.
4. **COG already running** - If you `COGINIT` to a specific COG that's already running something else, it will be stopped and replaced. Use `COGEXEC_NEW` to automatically find a free COG.

What We've Learned

Let's celebrate what you've accomplished:

- Written your first PASM2 program
- Controlled hardware (LED) directly
- Used immediate values (# and ##)
- Created loops with **JMP**
- Understood COG independence
- Modified code for different patterns

That's quite a lot for Chapter 1!

Coming Up Next

In Chapter 2, we'll take our "Architecture Safari" and explore:

- How 8 COGs really work together
- The hub memory system and the "egg beater"
- Why the P2 doesn't need interrupts
- How to make COGs talk to each other

But for now, enjoy your blinking LED. You've just taken your first step into parallel processing!

Have Fun! And remember, every expert was once a beginner who kept their LED blinking when everyone else gave up.

Chapter 2: Architecture Safari

Eight brains are better than one

The Propeller Philosophy

Before we dive into the technical details, let's talk philosophy. Why would anyone design a microcontroller with eight processors?

The answer is beautifully simple: to avoid complexity.

"Wait," you might say, "eight processors sounds MORE complex, not less!"

Well, consider the traditional approach:

- One processor trying to do everything
- Interrupts constantly breaking your flow
- Priority levels to juggle
- Context switching overhead
- Race conditions and timing nightmares

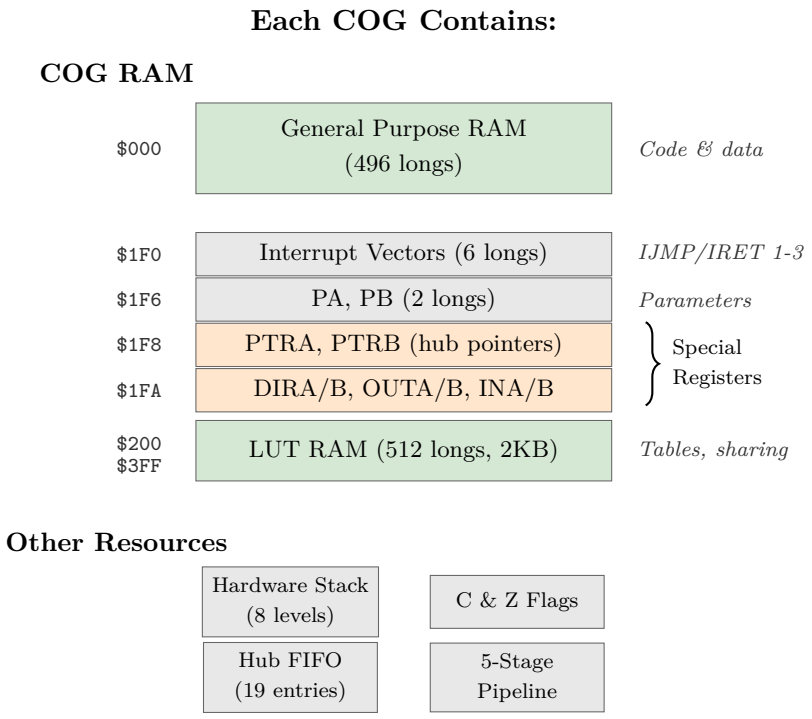
Now consider the Propeller way:

- Eight processors, each doing one thing well
- No interrupts needed (why interrupt when you have a dedicated processor?)
- No priorities (all COGs are equal)
- Deterministic timing (you know EXACTLY when things happen)
- True parallel processing (not time-slicing)

It's like the difference between one stressed-out juggler trying to keep eight balls in the air versus eight relaxed people each tossing one ball. Which seems simpler?

COG Anatomy 101

Let's dissect a COG and see what makes it tick:



But here's the beautiful part: COGs are identical. There's no "master" COG or "special" COG. Any COG can do anything any other COG can do. Democracy in silicon!

The 512-Long Limit

Each COG has exactly 512 longs (2048 bytes) of memory. The first 496 longs are yours to use for code and data. The last 16 are special registers (but not like P1 - we'll get to that).

"Only 496 instructions?" you might cry. "That's tiny!"

Well, yes and no. Remember:

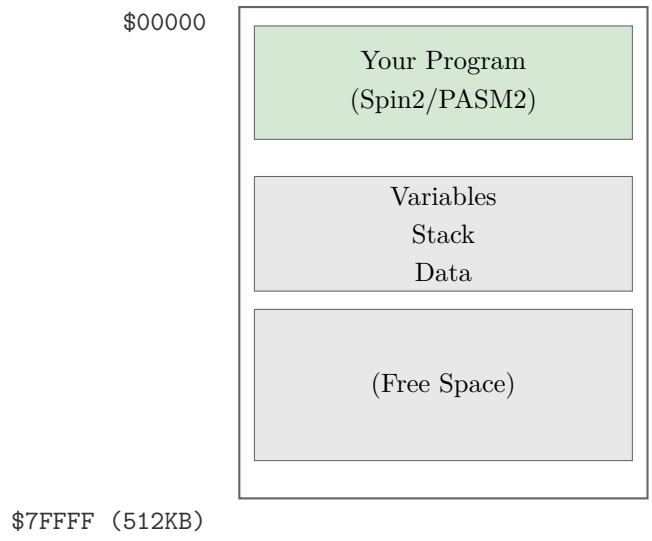
1. PASM2 instructions are powerful - one instruction often does what takes several in other processors
2. You have EIGHT of these COGs
3. There's hub execution mode for larger programs (Chapter 10)
4. Most real-time tasks **FIT** easily in 496 instructions

Think of it like haiku - the constraint forces elegance.

Meet the Hub: The Meeting Place

The hub is where COGs come together. It's 512KB of RAM shared by all COGs, and it's where the magic of cooperation happens.

Hub Memory Layout:

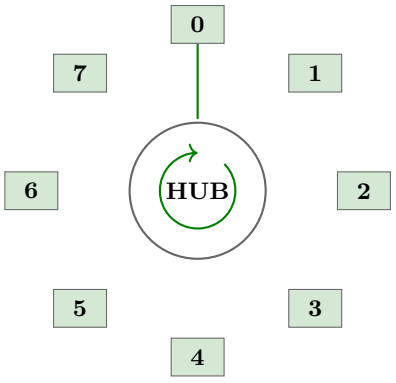


The Egg Beater Revolution

Now here's where P2 gets clever. In P1, COGs took turns accessing the hub in a round-robin fashion. If you missed your slot, you waited for the wheel to come around again.

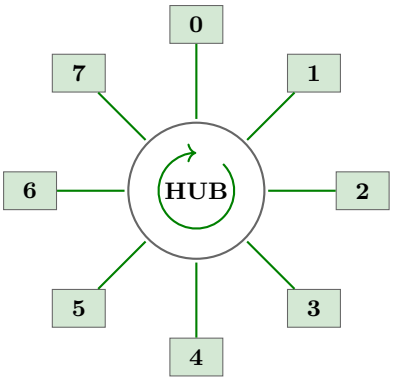
P2 uses what we call the "egg beater" model. Imagine eight beaters (COGs) all whipping through the same bowl (hub) simultaneously, but their paths are cleverly arranged so they never collide:

P1 Traditional:



COGs take turns
Each waits up to
16 clocks

P2 Egg Beater:



All COGs access
simultaneously!
Max 8 clocks wait

The practical result? Hub access is MUCH faster and more predictable. Instead of waiting up to 16 clocks (P1), you wait at most 8 clocks (P2), and often less if you align your accesses properly.

Let's See COGs in Action

Here's a simple demonstration of multiple COGs working together:

```
' Multi-COG LED Pattern Demo
PUB main() | i
    repeat i from 0 to 3
        COGINIT(COGEXEC_NEW, @cog_code, 56 + i)  ' Start 4 COGs
    repeat  ' Main COG just watches

DAT
        ORG     0
cog_code
        MOV     pin_num, ptra          ' Pin number was passed in via PTRA

.loop   DRVNOT  pin_num                ' Toggle our LED
        SHL     pin_num, #24           ' Pin number to bits 24-31
        OR      pin_num, ##10_000_000  ' Combine with delay
        WAITX   pin_num                ' Wait (varies per COG!)
        SHR     pin_num, #24           ' Restore pin number
        JMP     #.loop

pin_num LONG    0
```

What's happening here:

1. The main Spin2 code starts 4 COGs
2. Each COG gets a different pin number (56, 57, 58, 59)
3. Each COG blinks its LED at a slightly different rate
4. All four LEDs blink independently and simultaneously!

COG Communication: How They Talk

COGs are independent, but they're not isolated. They can communicate through hub memory:

Method 1: Simple Variables

```
' COG 1: Writer
        MOV     value, #42
        WRLONG  value, ##$1000  ' Write to hub address $1000

' COG 2: Reader
        RDLONG  result, ##$1000 ' Read from hub address $1000
```

Method 2: Locks (When It Matters)

When multiple COGs might write to the same location, we need locks:

```
' Get a lock
.try_lock
        LOCKTRY lock_id wc     ' Try to get lock (C=1 if we got it)
  if_nc JMP     #.try_lock     ' Keep trying until we get it

        ' Critical section - we have the lock!
        RDLONG  value, ##shared_addr
        ADD     value, #1
        WRLONG  value, ##shared_addr

        LOCKREL lock_id        ' Release the lock

lock_id LONG    0              ' Lock 0-15
```

Method 3: Mailboxes (Elegant)

A mailbox is just a hub location where COGs leave messages:

```
' COG A: Leave a message
        WRLONG  message, ##mailbox

' COG B: Check for messages
.check  RDLONG  data, ##mailbox wz
   if_z JMP     #.check        ' Keep checking if empty
        WRLONG  #0, ##mailbox  ' Clear mailbox
        ' Process the message in 'data'
```

The Timer: Everyone Gets One

Each COG has its own 64-bit timer, always counting system clocks. This is incredibly useful:

```
' Method 1: Simple delay
        GETCT   start_time
        ADDCT1  start_time, ##1_000_000
        WAITCT1                 ' Wait exactly 1,000,000 clocks

' Method 2: Periodic events
        GETCT   time
.loop   ADDCT1  time, ##10_000_000
        WAITCT1                 ' Wait for next 10M clock interval
        DRVNOT  #56             ' Toggle LED
        JMP     #.loop          ' Perfectly periodic!
```

The beauty? Each COG's timer is independent. No shared resource conflicts!

Why No Interrupts? (Usually)

Here's a controversial P2 feature: it HAS interrupts, but you probably shouldn't use them. Why?

Because with 8 COGs, you don't need interrupts! Instead of interrupting important work, just dedicate a COG to monitoring whatever would have triggered the interrupt:

```
' Traditional (with interrupts):
' Main code runs, gets interrupted, handles event, returns

' Propeller way:
' COG 1: Main code runs uninterrupted
' COG 2: Watches for event continuously
pin_watcher
        TESTP   #BUTTON_PIN wc
    if_c JMP    #button_pressed
        JMP     #pin_watcher

button_pressed
        WRLONG  ##1, ##button_flag  ' Signal other COGs
        JMP     #pin_watcher
```

No interrupt latency, no context switching, no priority inversion. Just dedicated, deterministic monitoring.

Real-World Example: Parallel Sensors

Let's read four different sensors simultaneously:

```
' Each COG runs this with different parameters
sensor_reader
        RDLONG  sensor_pin, ptra[0]    ' Get pin assignment
        RDLONG  hub_addr, ptra[1]       ' Where to store results

read_loop
        ' Read sensor (simplified - real sensors need protocols)
        TESTP   sensor_pin wc
        RCL     value, #1               ' Accumulate bits

        ' Every 32 reads, store to hub
        INCMOD  counter, #31 wc
   if_c WRLONG  value, hub_addr
   if_c ADD     hub_addr, #4

        JMP     #read_loop

sensor_pin LONG 0
hub_addr   LONG 0
value      LONG 0
counter    LONG 0
```

Four COGs running this code = four sensors being read truly simultaneously. Try doing that with a single processor and interrupts!

The Medicine Cabinet

Feeling overwhelmed by all this parallel processing? Here's your prescription:
Start simple: Use just one or two COGs at first

```
' Just two COGs - main program and one helper
PUB main()
    COGINIT(COGEXEC_NEW, @helper, 0)
    ' Your main code here
```

Debug one COG at a time: Get each COG working alone before combining

```
' Test COG in isolation first
debug_cog
        DRVH    #MY_DEBUG_LED  ' Visual confirmation it's running
        ' Your actual code here
```

Use Spin2 for coordination: Let the high-level language handle the complex stuff

```
' Spin2 manages COGs, PASM2 does the real-time work
PUB orchestrator()
    startSensorCog(0)
    startMotorCog(1)
    startCommsCog(2)
    ' Spin2 coordinates, PASM2 executes
```

Sidetrack

The Philosophy of Parallel

The Propeller's design philosophy comes from a simple observation: in the real world, things happen in parallel, not in sequence.
Consider your car:

- The engine runs continuously
- The radio plays independently
- The climate control maintains temperature
- The dashboard updates displays
- The ABS monitors wheel speed

continues on next page →

↪ *continued from previous page*

These aren't taking turns - they're all happening simultaneously. The Propeller models this reality directly. Instead of one processor frantically time-slicing between tasks, you have eight processors each focused on their job.
It's not just different - it's more natural.

Common Gotchas

Save yourself some debugging time:

1. **COG RAM is copied, not shared** - Changes in COG RAM don't affect hub RAM unless you explicitly write them back
2. **COGs start at 0** - Always! Your code better be there.
3. **Hub addresses are byte addresses** - COG addresses are **LONG** addresses. Don't mix them up!

```
    RDLONG  value, ##$1000  ' Reads from hub byte address $1000
    MOV     value, $100     ' Moves from COG long address $100 (256)
    ' Note: COG RAM is only 512 longs ($000-$1FF)!
```

4. **PTRA/PTRB are your friends** - These special registers make hub access much easier
5. **COGs are truly independent** - Stopping one COG doesn't affect others (unless they're waiting for it)

What We've Learned

Look at what you now understand:

- Why eight processors is simpler than one with interrupts
- How COGs are structured and limited
- The hub memory system and egg beater access
- Multiple ways for COGs to communicate
- Why interrupts are usually unnecessary
- How to think in parallel

Your Turn: Experiments

Experiment 1: COG Counter

Start COGs to increment different hub locations. With **COGEXEC**_NEW, the loop will start up to 7 new COGs (since COG 0 runs Spin2):

```
PUB main() | i
    repeat i from 0 to 7
        COGINIT(COGEXEC_NEW, @counter, $1000 + (i * 4))
    repeat
        ' Monitor the counters in hub RAM

DAT
        ORG     0
counter MOV     hub_ptr, ptra          ' Our hub address arrived in PTRA
.loop   RDLONG  value, hub_ptr
        ADD     value, #1
        WRLONG  value, hub_ptr
        WAITX   ##1_000_000
        JMP     #.loop

hub_ptr LONG    0
value   LONG    0
```

Experiment 2: Parallel Pattern

Make 8 LEDs display a moving pattern, with each COG controlling one LED:

```
' Each COG gets LED pin in ptra
        ORG     0
        RDLONG  pin, ptra
        RDLONG  delay, ptrb      ' Different delay per COG

.flash  DRVH    pin
        WAITX   delay
        DRVL    pin
        WAITX   delay
        SHL     delay, #1        ' Double the delay
        CMP     delay, ##100_000_000 wcz
```

continues on next page →

↪ *continued from previous page*

```
12      if_a MOV     delay, ##1_000_000  ' Reset if too long
13           JMP     #.flash
```

Coming Up Next

In Chapter 3, "Speaking PASM2", we'll dive deep into the instruction set:

- The anatomy of an instruction
- Conditional execution that will blow your mind
- Math operations that actually make sense
- Why PASM2 is unlike any assembly you've used

But for now, appreciate what you've learned: you understand the Propeller's parallel philosophy. That's not just technical knowledge - it's a new way of thinking about computing.

Have Fun! Remember, parallel processing isn't harder - it's different. And different can be wonderful.

Chapter 3: Speaking PASM2

Learning the native tongue

The Hook: One Instruction, Many Powers

Look at this single PASM2 instruction:

```
1           ADD     value, #1 wc
```

This one line:

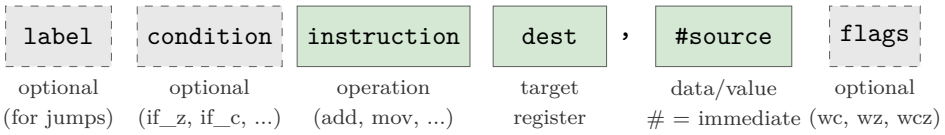
- **ADDS** 1 to 'value'
- Optionally sets the carry flag
- Executes in exactly 2 clock cycles
- Can be conditional
- Can even modify itself!

In most processors, that would take multiple instructions. In PASM2, it's just one. Let's learn to speak this powerful language.

Instruction Anatomy 101

Every PASM2 instruction follows the same basic pattern:

PASM2 Instruction Format:



Let's dissect a real instruction:

Example: `if_z add total, #10 wc`

The Basic Vocabulary

Moving Data Around

The **MOV** family - your bread and butter:

```
' Basic move
        MOV     dest, source    ' dest = source
        MOV     x, #42          ' x = 42 (immediate)
        MOV     x, ##70000      ' x = 70000 (32-bit immediate)

' But wait, there's more!
        mvn     dest, source    ' dest = NOT source (inverted)
        ABS     dest, source    ' dest = |source|
        NEG     dest, source    ' dest = -source

' And the mind-blowing ones
        ALTD    dest, source    ' Modify NEXT inst's dest field!
        ALTS    dest, source    ' Modify NEXT inst's source field!
```

Well, that escalated quickly! Don't worry about **ALTD**/ALTS yet - just know they exist and they're amazing.

Math Without Tears

P2 has hardware multiply and divide. Let that sink in. Hardware. Multiply. And. Divide.

```
' Addition and subtraction
        ADD     x, y            ' x = x + y
        SUB     x, y            ' x = x - y
        ADDS    x, y            ' Signed add
        SUBS    x, y            ' Signed subtract

' The revolution: hardware multiply!
        MUL     x, y            ' x = x * y (low 32 bits)
        MULS    x, y            ' Signed multiply

' And hardware divide!
        QDIV    x, y            ' Start division x/y
        GETQX   result          ' Get quotient
        GETQY   remainder       ' Get remainder
```

Here's a complete multiply example:

```
' Simple 16x16->32 multiply (2 clocks)
        MOV     x, #123
        MOV     y, #456
        MUL     x, y            ' Result: 123 * 456 = 56088 in x
        ' Uses lower 16 bits of each operand!

' For full 32x32->64 multiply, use CORDIC:
        QMUL    x, y            ' Start multiply (uses full 32 bits)
        ' ... 55 clocks of other work ...
        GETQX   low             ' Lower 32 bits of result
        GETQY   high            ' Upper 32 bits of result
```

Uff! In the old days, we'd write loops for this. Now hardware does it!

Logic Operations

Your Boolean friends:

```
        AND     x, mask         ' x = x AND mask
        OR      x, bits         ' x = x OR bits
        XOR     x, toggle       ' x = x XOR toggle
        NOT     x               ' x = NOT x (XOR with $FFFFFFFF)

' Bit manipulation
        BITL    x, #5           ' Clear bit 5 of x
        BITH    x, #5           ' Set bit 5 of x
        BITNOT  x, #5           ' Toggle bit 5 of x
        TESTB   x, #5 wc        ' Test bit 5, result in C flag
```

Shifting and Rotating

Moving bits around:

```
        SHL     x, #3           ' Shift left 3 bits
        SHR     x, #3           ' Shift right 3 bits
        SAR     x, #3           ' Arithmetic shift right (signed)
        ROL     x, #3           ' Rotate left 3 bits
        ROR     x, #3           ' Rotate right 3 bits
```

continues on next page →

↪ *continued from previous page*

```
6
7  ' Variable shifts (amount in register)
8          SHL     x, y            ' Shift x left by y bits
9
10 ' Fancy ones
11         REV     x               ' Reverse bit order (!!)
12         MERGEB  x               ' Merge bits, not bytes (inverse of SPLITB)
```

Flow Control: Jump!

Unconditional Jumps

```
1          JMP     #target         ' Jump to target
2          JMP     target          ' Jump to address in target register
3
4  ' Relative jumps
5          JMP     #$-4           ' Jump back 4 longs (addresses)
6          JMP     #$+8           ' Jump forward 8 instructions
```

Conditional Execution (The Magic)

Here's where PASM2 gets beautiful. ANY instruction can be conditional:

```
1  if_z    ADD     x, #1           ' Only add if Z flag set
2  if_nz   ADD     x, #1           ' Only add if Z flag clear
3  if_c    ADD     x, #1           ' Only add if C flag set
4  if_nc   ADD     x, #1           ' Only add if C flag clear
```

The basic conditions:

Condition	Meaning
`if_z`	If Z flag set (result was zero)
`if_nz`	If Z flag clear (result not zero)
`if_c`	If C flag set (carry/borrow occurred)
`if_nc`	If C flag clear
`if_c_and_z`	If both C and Z set
`if_c_or_z`	If either C or Z set
`if_c_eq_z`	If C equals Z
`if_c_ne_z`	If C not equal to Z

And the comparison conditions (use after **CMP**):

Condition	Meaning
`if_a`	If above (unsigned greater)
`if_b`	If below (unsigned less)
`if_ae`	If above or equal
`if_be`	If below or equal

The Call/Return Dance

```
        CALL    #subroutine     ' Call subroutine
        RET                     ' Return from subroutine

' But here's the P2 twist - CALL uses internal stack
subroutine
        ' Do something useful
        RET                     ' Returns to instruction after CALL

' You get 8 levels of hardware stack!
```

The _RET_ Prefix: Return With Benefits

Here's a clever trick the P2 offers. What if you could execute an instruction *and* return from a subroutine in one go? That's exactly what the `_RET_` prefix does.

```
' Normal way: Two instructions
add_and_return
        ADD     x, y            ' Do the add
        RET                     ' Then return (RET is a ~4-cycle branch)

' _RET_ way: One instruction!
add_and_return
        _ret_   ADD     x, y    ' Add AND return (saves 2 cycles)
```

The `_RET_` prefix says: "Execute this instruction, then return." It's like getting a free return ticket with your instruction. The add happens, flags get set normally, and then—pop!—you're back at the caller.

When does _RET_ NOT return?

Here's the catch: if the instruction itself branches, no return happens. The branch wins:

```
        _ret_   JMP     #somewhere      ' JMP wins - no return
        _ret_   CALL    #helper         ' CALL wins - no return
        _ret_   DJNZ    count, #loop    ' Branch? No return. Zero? Return!
```

That last one is interesting! If `count` isn't zero, **DJNZ** branches and no return. But when `count` hits zero, no branch occurs, so you get your return. Clever, right?

One-Instruction Subroutines

This is where `_RET_` really shines:

```
' Toggle pin 0 - entire subroutine is ONE instruction!
toggle_led
        _ret_   DRVNOT  #0              ' Toggle and return

' Read all inputs - also just one instruction
read_inputs
        _ret_   MOV     result, ina     ' Copy INA and return

' Usage:
        CALL    #toggle_led             ' Blink!
        CALL    #read_inputs            ' result now has INA
```

A normal subroutine needs at least two instructions (the work + **RET**). With `_RET_`, you can have genuinely single-instruction subroutines. Your code gets smaller and faster.

The Medicine: _RET_ Quick Reference

Pattern	What Happens
`_ret_ add x, y`	ADD executes, then return
`_ret_ jmp #label`	JMP executes, NO return (branch wins)
`_ret_ djnz n, #loop`	If n>0: branch, no return. If n=0: no branch, return

Important: Unlike `RET wcz`, the `_RET_` prefix does NOT restore C and Z flags from the stack. If you need flag restoration, use the regular `RET wcz` instruction.

Labels: Naming Your Places

You've been using labels throughout this chapter without us properly introducing them. How rude of me! Let's fix that.

Global Labels: The Big Signposts

A global label is just a name at the start of a line:

```
1 DAT             ORG
2
3 send_byte       RDBYTE  x, ptr          ' Global label
4                 WYPIN   x, tx_pin
5                 RET
6
7 receive_byte    TESTP   rx_pin    wc    ' Another global label
8                 RDPIN   x, rx_pin
9                 RET
```

Global labels are visible everywhere in your DAT block. You can jump to them, **CALL** them, reference them from Spin2 - they're your main signposts.

Local Labels: The Little Helpers

But here's a problem. What if every routine needs a loop? You can't have two labels called `loop` - the assembler would be terribly confused.

Enter local labels. Prefix a name with a dot (.) and it becomes local:

```
1 DAT             ORG
2
```

continues on next page →

↪ *continued from previous page*

```
send_byte       RDBYTE  x, ptr
.loop           TESTP   tx_pin    wc    ' Local to send_byte
        if_nc   JMP     #.loop
                WYPIN   x, tx_pin
                RET

receive_byte    TESTP   rx_pin    wc    ' New scope begins here
        if_nc   JMP     #.wait
.wait           TESTP   rx_pin    wc    ' Local to receive_byte
        if_nc   JMP     #.wait
                RDPIN   x, rx_pin
.loop           SHR     x, #24          ' Different .loop, OK!
                RET
```

Each global label starts a new "scope". The `.loop` under `send_byte` is completely separate from the `.loop` under `receive_byte`. You can reuse `.loop`, `.done`, `.retry`, `.exit` to your heart's content.

The Colon Alternative

You might also see local labels with a colon prefix:

```
:loop           DJNZ    count, #:loop   ' Same as .loop
```

Both : and . work identically. I prefer the dot - it's what modern convention has settled on - but you'll see both in the wild.

Reference Operators: Finding Your Labels

When you reference a label, you need to tell the assembler what you want:

```
' In COG code (after ORG):
        JMP     #my_routine     ' # = immediate COG address
        CALL    #.helper        ' # works for local labels too
        MOV     x, #data_table  ' Get COG address of data

' For hub addresses (used with Spin2):
        MOV     ptr, @hub_data  ' @ = hub address of label
```

The `#` means "immediate value" - use this for jumps and calls within COG code. The `@` means "hub address" - use this when passing addresses to Spin2 or for hub memory operations.

Scope Boundaries: When Local Labels Reset

Here's the rule: **every global label or data definition starts a new local scope**.

```
1 func_a          MOV     x, #1           ' Scope #1 begins
2 .loop           DJNZ    x, #.loop       ' .loop in scope #1
3
4 data_block      LONG    0, 0, 0, 0      ' Scope #2 begins (data!)
5
6 func_b          MOV     y, #2           ' Scope #3 begins
7 .loop           DJNZ    y, #.loop       ' .loop in scope #3, OK!
8 .done           RET
```

This is wonderfully useful - your utility routines can all use `.loop` and `.done` without stepping on each other's toes.

The Medicine: Quick Reference

What	Syntax	Example
Global label	`name`	`my_routine`
Local label	`.name` or `:name`	`.loop`, `:done`
Jump to label	`#label`	`jmp #.loop`
Hub address	`@label`	`mov ptr, @data`

Common Gotchas

1. **Forgetting the dot**: `loop` is global, `.loop` is local. If you accidentally create a global `loop`, you'll get conflicts.
2. **Scope surprise**: Data definitions (`LONG`, `WORD`, `BYTE`) also start new scopes. If you put data between two parts of a routine, your local labels won't work!
3. **The 30-character limit**: For compatibility with all tools, keep label names under 30 characters. `this_is_a_really_long_label_name` might cause trouble.

Data in DAT Blocks: Your Program's Pantry

Speaking of data definitions starting new scopes… we should probably talk about how to actually declare data! You've seen snippets like `counter LONG 0` scattered through our examples, but there's a whole world of data declaration waiting for you.

The Three Sizes

Just like Goldilocks, you have three choices:

```
DAT
my_byte         BYTE    $FF             ' 1 byte (8 bits)
my_word         WORD    $1234           ' 2 bytes (16 bits)
my_long         LONG    $DEADBEEF       ' 4 bytes (32 bits)
```

When do you use each? Well:

- **BYTE** for characters, small counters, flags, or when every byte of memory counts
- **WORD** for medium values, 16-bit peripherals, or when **BYTE** is too small but **LONG** is wasteful
- **LONG** for everything else - addresses, large numbers, and "I don't want to think about it"

If you're unsure, use **LONG**. Memory is cheap, debugging overflow errors is not.

Multiple Values on One Line

Here's a convenience - you can list multiple values after a single type:

```
DAT
primes          LONG    2, 3, 5, 7, 11, 13, 17, 19
gpio_pins       BYTE    16, 17, 18, 19, 20, 21
message         BYTE    "Hello, P2!", 0     ' String with null term
```

The assembler just lays them out consecutively in memory. That string? It's just bytes - each character followed by a zero at the end.

Arrays: The Repetition Trick

Need 100 bytes of zeros? Don't type them all out:

```
DAT
buffer          BYTE    0[100]          ' 100 zero bytes
lookup_table    WORD    $FFFF[64]       ' 64 words, all $FFFF
scratch_pad     LONG    0[32]           ' 32 longs of zeros
```

The `[count]` syntax repeats the value. This is your friend for buffers, tables, and anywhere you need initialized storage.

You can even combine values and repetition:

```
DAT
mixed_init      BYTE    $AA, $BB, 0[10], $CC, $DD
                '       ^    ^   ^^^^^   ^    ^
                '       Two vals, then 10 zeros, then two more
```

BYTEFIT and WORDFIT: The Safety Net

Here's a subtle trap. What if you accidentally write:

```
DAT
oops            BYTE    1000            ' 1000 won't fit in a byte!
```

The assembler will silently truncate 1000 to 232 (the low 8 bits). Your program will run, but with wrong values. Debugging that is no fun at all.

Enter the safety net:

```
DAT
safe_byte       bytefit 100, 200, 255   ' OK - all fit in a byte
danger_byte     bytefit 100, 200, 300   ' ERROR! 300 > 255, error!

safe_word       wordfit 1000, 50000     ' OK - all fit in a word
danger_word     wordfit 1000, 70000     ' ERROR! 70000 > 65535
```

Use `BYTEFIT` and `WORDFIT` when you want the assembler to check that your constants actually **FIT**. It's like having a compiler catch your mistakes before they become 3 AM debugging sessions.

Alignment: Keeping Things Tidy

The P2 is quite forgiving about alignment - it can read a LONG from any address. But aligned access is faster and cleaner. Sometimes you need to force alignment:

```
DAT
some_bytes      BYTE    1, 2, 3         ' 3 bytes
                ALIGNW                   ' Align to word boundary
next_word       WORD    $1234           ' Now properly aligned

more_bytes      BYTE    "ABC"           ' 3 bytes
                ALIGNL                   ' Align to long boundary
next_long       LONG    $DEADBEEF       ' Now on a 4-byte boundary
```

When does alignment matter? Mostly when you're:

- Mixing data sizes in the same DAT block
- Creating structures that Spin2 code will access
- Optimizing for maximum hub access speed

If you're just starting out, don't worry about it. Add alignment when you hit problems.

A Complete Example

Let's put it all together:

```
DAT             ORG

' Your code goes here
entry           MOV     ptra, ##buffer_addr
                MOV     count, #BUFFER_SIZE
.fill           WRBYTE  fill_value, ptra++
                DJNZ    count, #.fill
                JMP     #$

' Constants (read-only, effectively)
fill_value      BYTE    $55
BUFFER_SIZE     LONG    256

' Lookup tables
                ALIGNL
sin_table       WORD    0, 1608, 3212, 4808, 6393   ' Sine table
                WORD    7962, 9512, 11039, 12540    ' ... continues

' Working storage
                ALIGNL
buffer_addr     LONG    0               ' Set by Spin2 at startup
temp            LONG    0
result          LONG    0

' Reserve uninitialized space
                ALIGNL
scratch         RES     16              ' Reserve 16 longs
```

Notice the pattern: code first, then constants, then working storage, then reserved space. This keeps things organized and makes your DAT block readable.

The Medicine: Quick Reference

Declaration	Meaning	Example
`byte val`	8-bit value	`byte $FF`
`word val`	16-bit value	`word $1234`
`long val`	32-bit value	`long $DEADBEEF`
`val[n]`	Repeat n times	`byte 0[100]`
`bytefit`	Byte with range check	`bytefit 100, 200`
`wordfit`	Word with range check	`wordfit 1000, 50000`
`alignw`	Align to word	(no value)
`alignl`	Align to long	(no value)
`res n`	Reserve n longs	`res 16`

Common Gotchas

1. **Forgetting the label**: Every piece of data needs a label if you want to reference it. Anonymous data just wastes space.
2. **String termination**: `BYTE "Hello"` doesn't include a null terminator. Add `, 0` if you need one!
3. **RES is in longs**: `RES 10` reserves 10 *longs* (40 bytes), not 10 bytes. This trips up everyone at least once.
4. **Alignment after RES**: `RES` doesn't affect alignment. If you need alignment after reserved space, add an explicit `ALIGNL` or `ALIGNW`.

Including External Files: FILE

Sometimes you have binary data sitting in a file - a font bitmap, a sound sample, a pre-computed lookup table. Rather than manually converting it to hex values (ugh!), you can pull it straight in:

```
1 DAT
2 font_data       file    "myfont.bin"        ' Import entire file
3 sound_sample    file    "beep.raw"          ' Raw audio data
4 lut_table       file    "precalc.dat"       ' Pre-computed values
```

The assembler reads the file at compile time and drops its raw bytes right into your DAT block. The label gives you a way to reference where the data starts.

Where does it search?

1. Same folder as your source file
2. Library paths (if configured)

Practical example - embedded bitmap:

```
DAT             ORG
entry           MOV     ptr, ##@splash_screen
                ' ... display routine using ptr

' The image data lives right here in your code
splash_screen   file    "logo_128x64.bin"   ' 1024 bytes of pixel data
```

No conversion scripts, no copy-paste errors. Just reference the file and it's part of your program. The icing on the cake? If you update the file, recompile, and your program has the new data.

String and Data Generation Methods

Beyond manually typing values, you have some helpers for creating data:

@"text" - Inline String Address

Need a string address without declaring a separate label?

```
        MOV     ptra, @"Hello!"     ' ptra points to "Hello!" in hub
        CALL    #print_string

        MOV     ptra, @"Error: "    ' Another string, inline
        CALL    #print_string
```

The `@"text"` syntax creates the string in hub memory and gives you its address. It's like an anonymous label for a string constant. Each unique string gets stored once, even if you reference it multiple times.

STRING("text") and LSTRING("text")

These work similarly but in different contexts:

```
' In Spin2 code (not PASM):
DEBUG(STRING("Temperature: "))    ' Zero-terminated string address
DEBUG(LSTRING("Status"))          ' Length byte first, then string
```

`STRING()` returns the hub address of a zero-terminated string - same as what C programmers expect. `LSTRING()` puts a length **BYTE** at the front, which is handy when you need to know the string length without scanning for null.

BYTE[], WORD[], LONG[] - Data Arrays

In Spin2, you can create inline data arrays:

```
' Spin2 examples:
lookup := BYTE[10, 20, 30, 40, 50]    ' Returns address of byte array
config := LONG[$DEAD_BEEF, $CAFE_BABE]
```

These are primarily Spin2 features, but they generate hub data that your PASM code can access if you know the addresses.

The Pattern:

Method	Result	Use Case
`file "name"`	Raw binary data	Images, audio, lookup tables
`@"text"`	String address	Quick inline strings in PASM
`STRING("text")`	Zero-terminated string address	Spin2 string constants
`LSTRING("text")`	Length-prefixed string	When you need length upfront

The Flags: C and Z (and Q!)

Flags are your friends. They remember things:

```
' Z flag - was the result zero?
        SUB     x, y wz         ' Set Z if x-y equals zero
if_z    JMP     #equal          ' Jump if they were equal

' C flag - did we carry/borrow?
        ADD     x, y wc         ' Set C if addition overflowed
if_c    JMP     #overflow       ' Handle overflow

' Both at once!
        CMP     x, y wcz        ' Compare and set both flags
if_a    JMP     #x_greater      ' Jump if x > y (unsigned)
```

The Q flag is special - it's used by CORDIC operations (Chapter 7).

Special Instructions That Will Blow Your Mind

SKIP - The Instruction Skipper

```
        SKIP    ##%00001010    ' LSB first: 1=skip, 0=execute
        ADD     x, #1          ' Executed (bit 0 = 0)
        ADD     y, #1          ' Skipped! (bit 1 = 1)
        ADD     z, #1          ' Executed (bit 2 = 0)
        SUB     a, #1          ' Skipped! (bit 3 = 1)
        SUB     b, #1          ' Executed (bit 4 = 0)
        ' ... bit N controls the Nth instruction after SKIP
```

This is like having conditional execution on steroids!

REP - Hardware Loops

```
        REP     #4, #5         ' Repeat next 4 instructions 5 times
        ADD     x, #1
        SUB     y, #1
        ROL     z, #1
        ROR     w, #1
        ' These 4 instructions execute 5 times total
        ' No loop overhead!
```

ALTD/ALTS - Instruction Modification

```
' Modify the next instruction's destination
        MOV     index, #10
        ALTD    index, #array  ' Next instruction's dest = array+10
        MOV     0-0, value     ' Actually moves to array[10]!
```

This replaces self-modifying code from P1. Much cleaner!

Sidetrack

Hub-Exec Note: All ALTx instructions — **ALTI**, **ALTS**, **ALTD**, **ALTR**, **ALTB**, **ALTSN**, **ALTSB**, **ALTSW**, **ALTGN**, **ALTGB**, **ALTGW** — work identically in cog-exec and hub-exec modes. They act on the next pipelined instruction regardless of whether

continues on next page →

↪ *continued from previous page*

it came from cog/LUT memory or the hub-prefetch FIFO. So when you graduate to hub execution in Chapter 10, every ALTx trick you learn here still works.

Real-World Example: Fast Memory Copy

Let's combine what we've learned:

```
' Fast block copy using REP
fast_copy
        MOV     ptra, ##source_addr     ' Source pointer
        MOV     ptrb, ##dest_addr       ' Destination pointer

        REP     #2, ##256               ' Repeat 256 times
        RDLONG  temp, ptra++            ' Read and increment
        WRLONG  temp, ptrb++            ' Write and increment
        ' 1024 bytes (256 longs) copied with no loop overhead!

temp    LONG    0
```

The Medicine Cabinet

Feeling overwhelmed? Here's your simplified prescription:

Minimum Instructions to Know

```
' Moving data
        MOV     dest, source    ' Copy data

' Math
        ADD     dest, source    ' Addition
        SUB     dest, source    ' Subtraction

' Logic
        AND     dest, source    ' AND operation
        OR      dest, source    ' OR operation

' Flow
        JMP     #label          ' Jump
        CALL    #label          ' Call subroutine
        RET                     ' Return

' Flags
        CMP     x, y wcz        ' Compare and set flags
  if_z  JMP     #label          ' Conditional jump
```

Master these 10 instructions and you can write real programs!

Common Gotchas

1. **Immediate values**:
   - `#value` for 9-bit immediates (0-511)
   - `##value` for 32-bit immediates
   - Forgetting # uses the register at that address!
2. **Flag confusion**:
   - `wz` sets Z flag, `wc` sets C flag, `wcz` sets both
   - No flag update means flags unchanged
3. **PTRA/PTRB are special**:

```
    RDLONG  x, ptra++       ' Read and auto-increment
    RDLONG  x, ++ptra       ' Pre-increment then read
    RDLONG  x, ptra--       ' Read and auto-decrement
    RDLONG  x, ptra[5]      ' Read from ptra + 5*4
```

4. **Address confusion**:
   - COG addresses are in longs (0-511)
   - Hub addresses are in bytes (0-524287)

Your Turn: Experiments

Experiment 1: Conditional Counter

Count up if button pressed, down if not:

```
        ORG     0

.loop   TESTP   #BUTTON_PIN wc ' Test button
if_c    ADD     counter, #1    ' Increment if pressed
if_nc   SUB     counter, #1    ' Decrement if not

        WRLONG  counter, ##HUB_ADDR ' Display count
        WAITX   ##1_000_000
        JMP     #.loop

counter LONG    0
```

Experiment 2: Pattern Matcher

Find a pattern in data:

```
        ORG     0

        MOV     pattern, ##$DEADBEEF
        MOV     ptra, ##data_start

.search RDLONG  value, ptra++
        CMP     value, pattern wz
if_z    JMP     #.found
```

continues on next page →

↪ continued from previous page

```
9           CMP     ptra, ##data_end wcz
10  if_b    JMP     #.search
11          JMP     #.not_found
12
13  .found  ' Pattern found!
14          DRVH    #SUCCESS_LED
15          JMP     #$
16
17  .not_found
18          DRVH    #FAIL_LED
19          JMP     #$
```

Experiment 3: Speed Test

Compare multiply methods:

```
1  ' Method 1: Hardware multiply
2          GETCT   start_time
3          MUL     x, y
4          GETCT   end_time
5          SUB     end_time, start_time
6          ' Result: 2 clocks!
7
8  ' Method 2: Shift and add (old school)
9          GETCT   start_time
10         ' ... shift/add loop here
11         GETCT   end_time
12         ' Result: Many more clocks!
```

Sidetrack

Why PASM2 Is Different

Most assembly languages are thin wrappers over hardware. PASM2 is different - it's designed for humans:

1. **Symmetry**: Every instruction can use every addressing mode
2. **Orthogonality**: Features combine predictably
3. **Conditional everything**: Not just jumps, ANY instruction
4. **No special cases**: General-purpose registers, no accumulator

continues on next page →

↪ *continued from previous page*

This isn't accident - it's philosophy. The P2 was designed to make assembly programming pleasant.

What We've Learned

- Instruction anatomy and structure
- Basic data movement and math
- Hardware multiply and divide (!)
- Conditional execution on any instruction
- Special instructions (SKIP, **REP**, ALT*)
- Flag operations and testing
- Why PASM2 is human-friendly

Coming Up Next

Chapter 4, "The Hub Connection", explores:

- Reading and writing hub memory
- The FIFO and fast block transfers
- Hub execution mode
- Sharing data between COGs

You now speak basic PASM2. Time to learn how COGs communicate!

Have Fun! Remember, PASM2 isn't like other assembly languages - it's actually enjoyable!

Chapter 4: The Hub Connection

How COGs share and care

The Hook: Instant Communication

```
' COG 1: Leave a message
        WRLONG  ##$DEADBEEF, ##$1000

' COG 2: Get the message
        RDLONG  message, ##$1000
        ' message now contains $DEADBEEF!
```

That's it - COGs talking through hub memory. But there's so much more…

Reading from Hub

The basics are simple:

```
        RDBYTE  value, hubaddr    ' Read 1 byte
        RDWORD  value, hubaddr    ' Read 2 bytes (word)
        RDLONG  value, hubaddr    ' Read 4 bytes (long)

' With PTRA/PTRB magic
        RDLONG  value, ptra++     ' Read and increment pointer
        RDLONG  value, ++ptra     ' Increment then read
        RDLONG  value, ptra[5]    ' Read from ptra + 5*4
```

Writing to Hub

Just as easy:

```
        WRBYTE  value, hubaddr    ' Write 1 byte
        WRWORD  value, hubaddr    ' Write 2 bytes
        WRLONG  value, hubaddr    ' Write 4 bytes

' The mighty block transfer
```

continues on next page →

↪ *continued from previous page*

```
        SETQ    #16-1             ' Transfer 16 longs
        RDLONG  buffer, hubaddr   ' Reads 16 longs in one go!
```

The FIFO Pipeline

Here's where P2 gets serious about speed:

```
' Start the FIFO
        RDFAST  #0, ##data_start  ' Start fast read

' Now read at maximum speed
.loop   RFLONG  value             ' Read from FIFO
        ' Process value
        DJNZ    count, #.loop     ' Decrement and jump if not zero

' No hub timing worries - FIFO handles it all!
```

Real-World Example: Video Buffer

```
' Fast screen clear using block transfer
' Note: SETQ/SETQ2 maximum is 511 (for 512 longs)
' For larger fills, we loop in 512-long chunks
clear_screen
        MOV     hub_ptr, ##screen_buffer
        MOV     chunks, ##640*480/512   ' Number of 512-long chunks
        MOV     color, ##$00_00_00_00   ' Black

.loop   SETQ    #512-1                  ' Transfer 512 longs (max)
        WRLONG  color, hub_ptr          ' Fill this chunk
        ADD     hub_ptr, ##512*4        ' Advance 512 longs (2KB)
        DJNZ    chunks, #.loop
        ' Full screen cleared with minimal loop overhead!
```

The Medicine Cabinet

Simple hub access pattern:

```
' Just use PTRA for everything
        MOV     ptra, ##hub_address
        RDLONG  value, ptra++
        ' That's all you really need!
```

Your Turn: Experiments

Now you know the moves. Try these to make them stick:

1. **Round-trip:** Write a long to hub at address `$1000`, then read it back into a different register. Verify the values match using a comparison and a flag.
2. **Block fill:** Use **SETQ** + **WRLONG** to fill 64 consecutive longs in hub memory with the value `$DEAD_BEEF`. Then read them back one by one and confirm.
3. **FIFO race:** Use **RDFAST** to stream a sequence of longs from hub. Compare the throughput to a loop of individual **RDLONG** instructions. The difference should astonish you.
4. **Pointer arithmetic:** Set **PTRA** to a hub address, then use **RDLONG** `val, ptra[3]` to read the third **LONG** ahead — without moving the pointer. Experiment with `++`, `--`, and indexed forms.

Common Gotchas

Before you pull your hair out debugging hub access:

1. **Forgetting the `##`** — Hub addresses are 20-bit. A bare `#address` only encodes 9 bits, so you'll hit the wrong memory. Always use `##` for hub addresses outside the 0–511 range.
2. **Long-aligned access** — **RDLONG** and **WRLONG** work on **LONG** boundaries. If your address isn't a multiple of 4, the P2 silently masks the low bits. For byte-precise access, use **RDBYTE** / **WRBYTE**.
3. **SETQ block size** — **SETQ** `#N-1` transfers `N` longs (not `N-1`). The `-1` is because the encoded field is "count minus one." Off-by-one bugs love this one.
4. **PTRA vs. PTRB** — Both are pointer registers, but they're independent. Don't assume **PTRA** holds anything if you've been working with **PTRB**.
5. **Hub-exec vs. cog-exec timing** — In cog-exec, **RDLONG** is 9–16 cycles. In hub-exec, it's 9–26 cycles. Inner loops that hammer hub should run from COG memory when possible.

What We've Learned

Look at all the ground we've covered:

- How to read and write hub memory from COG code
- Why pointer registers (PTRA, PTRB) exist and when to use them
- How the FIFO pipeline accelerates sequential hub reads
- Block transfers with **SETQ** for moving big chunks at once
- The pattern for clearing or filling buffer-sized regions

You now have the entire COG Hub vocabulary at your disposal.

Coming Up Next

In Chapter 5, we'll unleash the P2's mathematical muscle:

- Hardware multiply and divide that don't require library calls
- 64-bit arithmetic without sweating
- A preview of the CORDIC coprocessor (it gets a full chapter later)
- Fixed-point math for fast trigonometry

But first, take a break. Hub access is one of those topics where letting it sit a day helps it lock in. Try one of the experiments above tomorrow.

Have Fun! And remember — the FIFO is doing real work even when your code looks idle. Trust it.

Chapter 5: Mathematics Unleashed

Hardware multiply and divide - finally!

The Hook: Hardware Multiply

```
        MUL     x, y              ' 16x16->32 bit unsigned multiply
        ' Result in x (lower 16 bits of each operand used)

        ' For full 32x32->64 bit multiply, use CORDIC:
        QMUL    x, y              ' Start 32x32->64 multiply
        ' ... other work (55 clocks) ...
        GETQX   low               ' Get lower 32 bits
        GETQY   high              ' Get upper 32 bits
```

Remember doing this with shifts and **ADDS**? Those days are over!

The Multiplication Revolution

```
' Unsigned multiply
        MUL     result, value     ' result = low 32 bits

' Signed multiply
        MULS    result, value     ' Signed version

' Scale by a power of two
        SHR     result, #1        ' Scale by 0.5 (divide by 2)
```

Division Without Tears

```
' Start division
        QDIV    dividend, divisor ' Start the operation

' Get results (takes 55 clocks)
        GETQX   quotient          ' Get quotient
        GETQY   remainder         ' Get remainder

```

continues on next page →

↪ *continued from previous page*

```
' Fractional division
        QFRAC   numerator, denominator
        GETQX   fraction         ' 32-bit fraction
```

64-Bit Operations

```
' 64-bit add
        ADD     low1, low2 wc
        ADDX    high1, high2

' 64-bit multiply (uses CORDIC)
        QMUL    x, y            ' Start 32x32->64 multiply
        ' ... 55 clocks ...
        GETQX   low             ' Lower 32 bits
        GETQY   high            ' Upper 32 bits
```

Real-World Example: Fixed-Point Math

```
' 16.16 fixed point multiply (uses CORDIC for full precision)
fixed_mul
        QMUL    a, b            ' Start 32x32->64 unsigned multiply
        ' ... 55 clocks (do other work) ...
        GETQX   low             ' Lower 32 bits
        GETQY   high            ' Upper 32 bits
        ' Extract middle 32 bits for 16.16 result:
        SHL     high, #16       ' Upper 16 bits of result
        SHR     low, #16        ' Lower 16 bits of result
        OR      a, low          ' Combine for 16.16 format
        OR      a, high
```

The Medicine Cabinet

Quick math reference:

- **MUL** D, S — unsigned 16×16→32
- **MULS** D, S — signed 16×16→32
- **QMUL** D, S — full 32×32→64 (read via **GETQX**/**GETQY** after 55 clocks)
- **QDIV** D, S — full 32-bit divide (read via **GETQX** quotient / **GETQY** remainder after 55 clocks)
- **QFRAC** D, S — fractional divide (returns 32-bit fraction in **GETQX**)
- 64-bit add: **ADD** + **ADDX** chained with **WC**

For everyday integer work, **MUL**/**MULS** are 2 clocks and you're done. For precision (full 64-bit results, fixed-point math, signed division), **QMUL**/**QDIV** route through the CORDIC and pay 55 clocks — but they don't block the COG, so you can interleave other work.

Your Turn: Experiments

Stretch your math muscles:

1. **Compute the average:** Read 8 longs from hub, sum them with **ADD**, then divide by 8 using a **SHR** (or **QDIV** if you want exact). Compare both approaches.
2. **Fractional reciprocal:** Use **QFRAC** to compute `2^32 / x` for various x values. You've just built a hardware reciprocal table.
3. **Pipeline overlap:** Start a **QMUL**, do 30+ clocks of other work (compute something else, update a counter), then **GETQX**/**GETQY**. Measure total cycles vs. doing the multiply blocking-style.
4. **64-bit counter:** Build a 64-bit increment loop using **ADD** + **ADDX**. Add to it in a tight loop and watch the high **WORD** advance when the low one wraps.

Common Gotchas

The math instructions hide a few traps:

1. **MUL is unsigned, MULS is signed** — Using the wrong one on negative values gives spectacular nonsense. When in doubt, **MULS**.
2. **MUL gives only low 32 bits** — For the full 64-bit result, you must use **QMUL** + **GETQX**/**GETQY**. The high **WORD** is silently discarded by **MUL**.
3. **GETQX/GETQY block until ready** — They wait for the CORDIC. If you **CALL** them too early, your COG stalls. If you **CALL** them later than necessary, you've wasted cycles. The sweet spot is starting the CORDIC, doing exactly 55 clocks of other work, then reading.

4. **Don't queue a new CORDIC op while one is pending** — A second **QMUL/QDIV/QFRAC** before reading results overwrites the queue. Use **GETQX/GETQY** first.

What We've Learned

A whole arithmetic library, in your back pocket:

- Hardware multiply (**MUL**, **MULS**) in just 2 clocks
- Division and modulo via **QDIV** + **GETQX/GETQY**
- Fractional division via **QFRAC** for fast reciprocals
- 64-bit chains using **ADD/ADDX**, **SUB/SUBX** with **WC**
- Fixed-point patterns for 16.16 math

You no longer have to write multiply-by-shift-and-add. Those days really are over.

Coming Up Next

In Chapter 6 we'll meet the P2's most quietly powerful feature: **conditional execution**. Every instruction can be conditional — no branches needed, deterministic timing preserved. We'll cover:

- The C and Z flags and how to update them
- Building complex conditions with **WC**, **WZ**, **WCZ** combined with **IF_x** prefixes
- **SKIP** and **SKIPF** for skipping patterns of instructions

If conditional execution doesn't change how you think about flow control, nothing will.

Have Fun! And remember — the CORDIC is a coprocessor with infinite patience. Use it.

Chapter 6: Flags and Decisions

Making choices at machine speed

The Hook: Any Instruction Can Be Conditional

```
        CMP     x, y wcz        ' Compare x and y
if_a    MOV     result, x       ' If x > y, result = x
if_be   MOV     result, y       ' If x <= y, result = y
        ' Max function in 3 instructions!
```

The C and Z Flags

```
' Z Flag - Zero detection
        SUB     x, y wz         ' Z=1 if x equals y
if_z    JMP     #equal          ' Jump if equal

' C Flag - Carry/Borrow
        ADD     x, y wc         ' C=1 if overflow
if_c    JMP     #overflow       ' Handle overflow
```

Complex Conditions

```
' Combining flags
        CMP     x, y wcz
if_a    JMP     #greater        ' x > y (unsigned)
if_b    JMP     #less           ' x < y (unsigned)
if_z    JMP     #equal          ' x == y

' Signed comparisons
        CMPS    x, y wcz        ' Signed compare
if_gt   JMP     #greater        ' x > y (signed)
if_lt   JMP     #less           ' x < y (signed)
```

Skip Patterns - Conditional Blocks

```
        SKIPF   pattern          ' Set skip pattern
        ADD     x, #1           ' Maybe executed
        SUB     y, #1           ' Maybe executed
        MOV     z, #0           ' Maybe executed
        ' Pattern determines what runs!
```

The Medicine Cabinet

Conditional execution quick reference:

- **WC** — write the C flag with the instruction's carry-out
- **WZ** — write the Z flag based on result == 0
- **WCZ** — write both flags
- **IF_C**, **IF_NC** — execute next instruction only if C set / clear
- **IF_Z**, **IF_NZ** — same for Z
- **IF_A**, **IF_B**, **IF_AE**, **IF_BE** — unsigned comparisons after **CMP**
- **IF_GT**, **IF_LT**, **IF_GE**, **IF_LE** — signed comparisons after **CMPS**
- **IF_E**, **IF_NE** — equal / not equal (same as **IF_Z**/**IF_NZ** after compare)

Any instruction can carry a condition prefix and still take exactly 2 clocks. No branches, no pipeline flush, no surprise.

Your Turn: Experiments

Practice the conditional mindset:

1. **Branchless absolute value:** Compute `result = ABS(x)` using **ABS** — then prove to yourself the same thing works branchless using **CMPS** + a conditional **NEG**.
2. **Saturating add:** Add two unsigned values; if the result overflows (**WC** sets C), clamp to `$FFFF_FFFF`. No jumps.
3. **Min/max:** Build a min function using **CMP** + conditional **MOV**. Then build max. Three instructions each.
4. **Skip-pattern selector:** Use **SKIPF** with a runtime bit pattern to selectively execute 4 different instructions based on a state **BYTE**. This is the building block for hand-coded jump tables.

Common Gotchas

The conditional mindset has its own traps:

1. **Forgetting the effect** — `CMP x, y` *with no flag effect* doesn't update C or Z. You need **WC**, **WZ**, or **WCZ**. The default for **CMP** is no update.
2. **Signed vs. unsigned comparison** — **CMP** sets flags for unsigned semantics (**IF_A**/**IF_B**). **CMPS** sets them for signed (**IF_GT**/**IF_LT**). Mixing them produces baffling bugs near zero and at the high bit.
3. **One instruction follows the prefix** — `IF_C MOV x, y` conditionally runs *only* the **MOV**. The instruction after that runs unconditionally. To skip a block, use **SKIPF**.
4. **C flag inversion on subtract** — **SUB** sets C on *borrow*, not carry. So after `SUB a, b wc`, C=1 means `a < b` (unsigned). Many P1 programmers expect the opposite — re-check.
5. **Skip patterns are LSB-first** — In **SKIPF**, bit 0 controls the next instruction, bit 1 the one after, etc. The numbering can feel backward; sketch it out.

What We've Learned

You can now think like a P2 programmer:

- Every instruction can carry a condition prefix
- Flags are updated only when you ask (**WC**, **WZ**, **WCZ**)
- Unsigned and signed comparisons use different instructions (**CMP** vs. **CMPS**)
- Complex multi-way logic without a single jump using **IF_x** prefixes
- **SKIP** and **SKIPF** for skipping arbitrary patterns of instructions

You've learned to write branchless code that preserves deterministic timing. That's the deepest part of P2 thinking.

Coming Up Next

Chapter 7 reveals the CORDIC coprocessor in full — the math beast you glimpsed in Chapter 5. Trigonometry at the speed of logic gates:

- Rotate points in three instructions
- Polar Cartesian conversion
- Hardware square root and natural log
- Pipelined operations for graphics and signal processing

If you thought hardware multiply was nice, wait until you see what the CORDIC does.

Have Fun! And remember — every `JMP` you avoid is a pipeline flush you didn't pay for.

Chapter 7: CORDIC Magic

Trigonometry at the speed of logic gates

The Hook: Rotate a Point in 3 Lines

Let me show you something that, on most processors, would take a coffee break of instructions, a math library, and probably a tear or two:

```
' Rotate point (x,y) by angle - that's it!
        SETQ    y_coord         ' Set Y coordinate
        QROTATE x_coord, angle  ' Start rotation by angle
        GETQX   new_x          ' Get rotated X (55 clocks later)
        GETQY   new_y          ' Get rotated Y
```

Read that again. *Four lines*, and a 2D rotation is done. No floating-point library. No lookup tables. No iterative approximation. The P2 has a dedicated trigonometric coprocessor sitting next to every COG, just waiting for you to wake it up. You're about to learn how.

Let me show you something even more impressive:

```
' Calculate sine and cosine simultaneously
        QROTATE ##$7FFF_FFFF, angle  ' D=radius (max), S=angle
        GETQX   cosine               ' cos(angle) in 2.30 fixed pt
        GETQY   sine                 ' sin(angle) in 2.30 fixed pt
        ' Both trig functions in 55 clocks total!
```

What Just Happened?

CORDIC stands for COordinate Rotation DIgital Computer. It's a method invented in 1959 for calculating trigonometric functions using only shifts and **ADDS** - no multiplies needed! Each P2 COG has its own dedicated CORDIC unit built into the hardware.

Think of CORDIC as your mathematical co-processor that can:

- Rotate points around the origin
- Convert between rectangular and polar coordinates

- Calculate sine, cosine, tangent
- Compute square roots and magnitudes

- Find arctangent (angle between points)
- Even do logarithms and exponentials!

All of this in exactly 55 clock cycles. Every time. No variation.

The CORDIC Pipeline - Your Mathematical Assembly Line

Here's the beautiful part: CORDIC operations are pipelined. While one calculation is running, you can start another. Let's see what that buys us:

```
' Generate sine wave samples rapid-fire
        MOV     angle, #0
        MOV     count, #256

generate
        QROTATE ##$7FFF_FFFF, angle   ' D=radius, S=angle
        ADD     angle, ##$0100_0000   ' Increment angle (no wait!)

        ' Do other work while CORDIC calculates
        ADD     sample_ptr, #4
        SUB     count, #1

        GETQY   sample                ' Get sine result
        WRLONG  sample, sample_ptr    ' Store it

        TJNZ    count, #generate      ' Test-jump-not-zero
        ' Generated 256 samples with perfect overlap!
```

The pipeline means you're not really waiting 55 clocks - you're getting useful work done while CORDIC churns away in the background. Uff! That's free math!

Core CORDIC Operations

QROTATE - The Rotation Engine

Here's a subtle detail: CORDIC operations work on 2D coordinates (X, Y), but the **QROTATE** instruction only takes one coordinate directly. The solution? **SETQ** loads the Y coordinate into the Q register, then **QROTATE** takes X from its first operand. It's a two-instruction dance that becomes second nature:

```
' Basic rotation: rotate point (x,y) by angle
        SETQ    y               ' First: load Y into Q register
        QROTATE x, angle        ' Then: X from operand, Y from Q
        GETQX   new_x           ' Result: X' = X*cos( ) - Y*sin( )
        GETQY   new_y           ' Result: Y' = X*sin( ) + Y*cos( )
```

The angle format is special: it's a 32-bit unsigned value where:

- $0000_0000 = 0 degrees
- $4000_0000 = 90 degrees

- $8000_0000 = 180 degrees
- $C000_0000 = 270 degrees
- $FFFF_FFFF = just under 360 degrees

This makes angle math incredibly easy - just use regular addition and subtraction!

QVECTOR - From Rectangular to Polar

```
' Convert (x,y) to polar (radius, angle)
        QVECTOR x, y            ' X in D, Y in S (no SETQ for QVECTOR)
        GETQX   radius          ' sqrt(x² + y²)
        GETQY   angle           ' atan2(y, x)
```

Perfect for:

- Finding distances between points
- Converting joystick input to angle/magnitude
- Radar and sonar applications

The Power of 32-Bit Precision

CORDIC uses 32-bit precision throughout:

- Angles: 32 bits (0.00000008 degree resolution!)
- Coordinates: 32 bits signed
- Results: Full 32-bit or 64-bit when needed

Real-World Example: Spinning a Sprite

Let's rotate a sprite around its center:

```
' Rotate sprite vertices around center
' (PTRA for vertex iteration - only PTRA/PTRB support ++ post-increment.)
rotate_sprite
        MOV     ptra, ##sprite_data
        MOV     vertex_count, #4        ' 4 corners

next_vertex
        RDLONG  x, ptra++               ' Get X coordinate, advance pointer
        RDLONG  y, ptra++               ' Get Y coordinate, advance pointer

        ' Center sprite at origin
        SUB     x, center_x
        SUB     y, center_y

        ' Rotate by current angle
        SETQ    y
        QROTATE x, rotation_angle

        ' While waiting, we can prepare
        MOV     temp_x, center_x
        MOV     temp_y, center_y

        ' Get rotated coordinates
        GETQX   x
        GETQY   y

        ' Translate back to position
        ADD     x, temp_x
        ADD     y, temp_y

        ' Store rotated vertex (back up PTRA 8 to overwrite source longs)
        SUB     ptra, #8
        WRLONG  x, ptra++
        WRLONG  y, ptra++

        DJNZ    vertex_count, #next_vertex

```

continues on next page →

↪ continued from previous page

```
38          ' Increment rotation for animation
39          ADD     rotation_angle, ##$0100_0000  ' ~1.4 deg (1/256 rotation)
```

Your Turn: CORDIC Experiments

Your Turn

Your Turn: Create a circular motion pattern
Starting code:

```
1           ORG     0
2
3           MOV     angle, #0
4           MOV     radius, ##100           ' 100 pixel radius
5
6   .loop   QROTATE radius, angle          ' D=X (radius), S=angle
7           ' Add code to:
8           ' 1. Get X,Y coordinates
9           ' 2. Add screen center offset
10          ' 3. Draw pixel at that position
11          ' 4. Increment angle
12          ' 5. Loop back to .loop
```

Goal: Make a dot trace a perfect circle on screen Hint: After **QROTATE**, use **GETQX**/getqy to get coordinates Success Check: Smooth circular motion, no gaps

Your Turn

Your Turn: Distance calculator
Starting code:

continues on next page →

↪ *continued from previous page*

```
' Calculate distance between two points
        MOV     x1, #10
        MOV     y1, #20
        MOV     x2, #40
        MOV     y2, #60

        ' Calculate differences
        SUB     x2, x1          ' dx
        SUB     y2, y1          ' dy

        ' Your code here: use qvector to find distance
```

Goal: Calculate the distance between the two points Hint: **QVECTOR** dx, dy gives you radius (distance) in QX Success Check: Distance should be 50 units

The Medicine Cabinet

Feeling overwhelmed by all this trigonometry? Here's your simplified prescription:
Too Complex? Just remember these three patterns:
Pattern 1: Get sine/cosine

```
        QROTATE ##$7FFF_FFFF, angle     ' D=radius, S=angle
        GETQX   cos_value
        GETQY   sin_value
```

Pattern 2: Rotate a point

```
        SETQ    y
        QROTATE x, angle
        GETQX   new_x
        GETQY   new_y
```

Pattern 3: Get distance

```
        QVECTOR dx, dy
        GETQX   distance
```

Master these three and you can do 90% of what you need!

Advanced CORDIC: The Pipeline Dance

Here's where CORDIC gets really powerful - overlapping operations:

```
' Process multiple points while calculating
process_points
        MOV     count, #16
        MOV     ptra, ##point_array

        ' Start first calculation
        RDLONG  x, ptra++
        RDLONG  y, ptra++
        SETQ    y
        QROTATE x, angle

process_loop
        ' Start next calculation immediately
        RDLONG  x, ptra++ wz    ' Z flag tells us if done
   if_nz RDLONG  y, ptra++
   if_nz SETQ    y
   if_nz QROTATE x, angle        ' New calculation starts

        ' Get previous result
        GETQX   prev_x
        GETQY   prev_y

        ' Store previous result
        WRLONG  prev_x, ptrb++
        WRLONG  prev_y, ptrb++

        DJNZ    count, #process_loop

        ' Don't forget last result!
        GETQX   prev_x
        GETQY   prev_y
        WRLONG  prev_x, ptrb++
        WRLONG  prev_y, ptrb++
```

See what happened? We started each new CORDIC operation immediately after the previous one, then retrieved results later. This pipeline approach means we're effectively getting one rotation every few instructions instead of waiting 55 clocks each time!

CORDIC for Graphics

Want to draw a spiral? CORDIC makes it trivial:

```
' Expanding spiral generator
spiral
        MOV     angle, #0
        MOV     radius, #1

draw_spiral
        QROTATE radius, angle         ' D=X (radius), S=angle
        GETQX   x
        GETQY   y

        ' Convert to screen coordinates
        SAR     x, #16          ' Scale down
        SAR     y, #16
        ADD     x, #320         ' Center X
        ADD     y, #240         ' Center Y

        ' Plot pixel (simplified)
        CALL    #plot_pixel

        ' Expand spiral
        ADD     angle, ##$0400_0000   ' Rotate 5.625 degrees (1/64 turn)
        ADD     radius, ##100         ' Expand slowly

        CMP     radius, ##30000 wcz
    if_b JMP    #draw_spiral
```

CORDIC for Audio

Generate perfect sine waves for audio:

```
' Audio tone generator using CORDIC
tone_generator
        MOV     phase, #0
        MOV     frequency, ##$0100_0000  ' ~1.4 deg/sample (1/256 rot)

sample_loop
```

continues on next page →

↪ *continued from previous page*

```
        QROTATE ##$7FFF_FFFF, phase     ' D=radius, S=angle
        ADD     phase, frequency        ' Increment phase

        ' Do other audio processing while waiting
        RDLONG  volume, ##volume_addr

        GETQY   sample                  ' Get sine value
        SAR     sample, #16             ' Scale to 16-bit
        MULS    sample, volume          ' Apply volume

        ' Output to DAC
        WYPIN   sample, #AUDIO_PIN

        ' Wait for sample period (48kHz)
        WAITX   ##4166                  ' 200MHz / 48kHz

        JMP     #sample_loop
```

Common CORDIC Gotchas

Before you pull your hair out debugging, know these:

1. **One result at a time** - Each COG has its own CORDIC, but starting a new operation before retrieving your result overwrites it!
2. **55 clocks is exact** - Not 54, not 56. Always exactly 55 clocks from operation start to result ready.
3. **Don't forget SETQ** - For two-operand operations (QROTATE with X,Y), you must load Y into Q first.
4. **Results are scaled** - When rotating by unit circle ($7FFF_FFFF), results are in 2.30 fixed point format.
5. **Angles wrap naturally** - Adding $1_0000_0000 to an angle is the same as adding 0. Use this!

What About QLOG, QEXP?

Don't worry, we won't leave logarithms and exponentials behind. CORDIC handles those too:

```
' Base-2 logarithm
        QLOG    value
        GETQX   result          ' log2(value) in 5.27 fixed point

' Base-2 exponential (2^x)
        QEXP    value
        GETQX   result          ' 2^value
```

These are less commonly used but incredibly powerful for DSP and scientific calculations.

Interlude

Jack Volder's Gift to Computing

In 1959, Jack Volder was working on navigation computers for aircraft. He needed to calculate trigonometric functions, but the computers of the day couldn't handle the complex math quickly enough.

His insight? Any angle can be decomposed into a sequence of smaller, fixed angles. By choosing these angles cleverly (arctan of powers of 2), he could rotate vectors using only shifts and **ADDS** - no multiplication needed!

The B-58 bomber's navigation computer was the first to use CORDIC. Today, it's in your P2, calculating sines and cosines faster than those room-sized computers could add two numbers.

From military navigation to your LED projects - quite a journey for an algorithm!

What We've Learned

Let's celebrate your new CORDIC powers:

- Understood CORDIC's rotate and vector operations
- Generated sine and cosine values
- Calculated distances and angles
- Learned the pipeline technique for speed
- Created rotating graphics
- Built an audio tone generator

That's serious mathematical muscle!

Coming Up Next

Chapter 8 brings us back to Earth with "Basic I/O" - the fundamental pin operations that make the real world connection. We'll save Smart Pins for another manual and focus on the essentials: making pins go high and low, reading buttons, and basic timing.

But first, take a moment to appreciate what you just learned. CORDIC is unique to the Propeller 2 - most microcontrollers would need extensive software libraries to do what you just did in three instructions!

Have Fun! And remember - with CORDIC, you're not just calculating trigonometry, you're doing it at hardware speed. That's magical!

Chapter 8: Basic I/O

Making the real world connection

The Hook: One Pin, Three Instructions, Infinite Possibilities

Watch this:

```
' Complete button-and-LED program
.loop   TESTP   #BUTTON_PIN wc  ' Read button into C flag
    if_c DRVH   #LED_PIN        ' If pressed, LED on
   if_nc DRVL   #LED_PIN        ' If not pressed, LED off
         JMP    #.loop          ' Repeat forever
```

Four lines. Complete input/output program. No configuration registers, no data direction setup, no port manipulation. Just pure, simple I/O.

But wait, let me show you the same thing with even more elegance:

```
' Even simpler - button controls LED directly
.loop   TESTP   #BUTTON_PIN wc  ' Read button
         DRVC   #LED_PIN        ' Drive LED from C flag!
         JMP    #.loop
```

Three lines! The **DRVC** instruction drives the pin to match the C flag. Input becomes output. Simple becomes simpler.

Understanding P2 Pins

Let's unpack what makes those three lines so short. Every P2 pin is bidirectional and incredibly capable. Unlike older microcontrollers where you set data direction registers (remember TRIS bits? DDRA? Yeah, we don't miss those either), P2 pins change direction on the fly based on the instruction you use.

Here's the mental model:

- **Output instructions** automatically make the pin an output
- **Input instructions** automatically make the pin an input

- **Float instructions** make the pin high-impedance
- No setup required!

Digital Output: Making Things Happen

The Fundamental Four

```
        DRVH    #56             ' Drive pin 56 HIGH (3.3V)
        DRVL    #56             ' Drive pin 56 LOW (0V)
        DRVNOT  #56             ' Toggle pin 56
        FLTL    #56             ' Float pin 56 (high-Z)
```

That's it. Four instructions, 90% of your output needs covered. We'll meet a few specialized cousins below, but if you only remember these four, you'll get a lot done.

Conditional Driving

Here's where P2 gets clever:

```
        DRVC    #56             ' Drive pin to match C flag
        DRVNC   #56             ' Drive pin to NOT C flag
        DRVZ    #56             ' Drive pin to match Z flag
        DRVNZ   #56             ' Drive pin to NOT Z flag
```

And the really clever one:

```
        DRVNOT  #56 wcz         ' Toggle pin AND read old state to C
        ' C now contains what the pin WAS before toggling
```

Random and Pattern Outputs

```
        DRVRND  #56             ' Drive pin to a random level (hardware PRNG)
        OUTL    #56             ' Set OUT bit low (dir unchanged)
        OUTH    #56             ' Set OUT bit high (dir unchanged)
```

Digital Input: Reading the World

Basic Pin Reading

```
        TESTP   #BUTTON_PIN wc ' Read pin into C flag
 if_c   JMP     #pressed       ' Branch if high
if_nc   JMP     #not_pressed   ' Branch if low
```

Or read into Z flag for zero/non-zero testing:

```
        TESTP   #SENSOR_PIN wz ' Read pin into Z flag
 if_z   JMP     #sensor_low    ' Jump if pin low (Z=1 when pin=0)
if_nz   JMP     #sensor_high   ' Jump if pin is high
```

Reading Multiple Pins

```
' Read 8 pins at once (pins 0-7)
        MOV     mask, #$FF     ' Pins 0-7
        TESTB   ina, #0 wc     ' Test pin 0
        RCL     result, #1     ' Rotate C into result
        TESTB   ina, #1 wc     ' Test pin 1
        RCL     result, #1
        ' ... continue for all 8 pins
```

Pin Timing: When Things Happen

Waiting for Pin Changes

```
' Wait for pin to go high
wait_high
        TESTP   #SIGNAL_PIN wc
 if_nc  JMP     #wait_high

' Wait for pin to go low
wait_low
        TESTP   #SIGNAL_PIN wc
 if_c   JMP     #wait_low
```

But there's a better way - hardware-assisted waiting with Smart Events:

```
        WAITSE1                 ' Wait for event 1
        WAITSE2                 ' Wait for event 2
        ' Configure events to watch pins - super efficient!
```

Real-World Example: Button Debouncing

Mechanical buttons bounce. Here's how to handle it:

```
' Debounced button reader
read_button
        MOV     debounce, #0

check_button
        TESTP   #BUTTON_PIN wc
    if_c ADD    debounce, #1    ' Count high readings
   if_nc MOV    debounce, #0    ' Reset on any low

        CMP     debounce, #10 wcz ' Need 10 consecutive highs
   if_ae JMP    #button_confirmed

        WAITX   ##200_000        ' Wait 1ms at 200MHz
        JMP     #check_button

button_confirmed
        ' Button definitely pressed
        DRVH    #LED_PIN
```

Bit-Banged Serial (The Basics)

Yes, you'll usually reach for Smart Pins for serial — but it's worth seeing how to do it the hard way once, just so you appreciate what Smart Pins are doing for you. Here's how:

```
' Bit-bang serial transmit at 115200 baud
tx_byte
        OR      data, ##$100    ' Add stop bit
        SHL     data, #1        ' Add start bit (0)
        MOV     bits, #10       ' 1 start + 8 data + 1 stop
```

continues on next page →

↪ continued from previous page

```
6
7         GETCT   time            ' Get current time
8
9  tx_loop
10        SHR     data, #1 wc     ' Get next bit into C
11        DRVC    #TX_PIN         ' Output bit
12
13        ADDCT1  time, bit_time  ' Next bit time
14        WAITCT1                 ' Wait for it
15
16        DJNZ    bits, #tx_loop
17        RET
18
19 bit_time LONG   100_000_000 / 115200  ' Clock cycles per bit
```

Your Turn: I/O Experiments

Your Turn

Your Turn: Create a light chaser
Starting code:

```
1         ORG     0
2
3         MOV     pattern, #1     ' Start with one LED
4
5  .loop  MOV     pins, pattern   ' Your code here
6         ' Make pattern rotate through pins 56-63
7         ' Add delay between changes
8         ' Wrap around at the end, then jmp #.loop
```

Goal: Create a rotating light pattern on LEDs Hint: Use **SHL** and check for overflow Success Check: Single lit LED rotating through all positions

Your Turn

Your Turn: Reaction timer
Starting code:

continues on next page →

↪ continued from previous page

```
        ORG     0

        ' Turn on LED after random delay
        GETRND  delay
        AND     delay, ##$3FFF_FFFF  ' Limit range
        WAITX   delay
        DRVH    #LED_PIN

        GETCT   start_time
        ' Your code: wait for button press
        ' Calculate reaction time
```

Goal: Measure reaction time between LED and button press Hint: Use **GETCT** after button detection Success Check: Time measured in clock cycles

The Medicine Cabinet

Feeling overwhelmed by all these pin operations? Here's the simplified prescription:
Just need something working? Remember these patterns:
Output pattern:

```
        DRVH    #PIN    ' Make it high
        DRVL    #PIN    ' Make it low
        DRVNOT  #PIN    ' Toggle it
```

Input pattern:

```
        TESTP   #PIN wc ' Read it
    if_c JMP    #high   ' It's high
   if_nc JMP    #low    ' It's low
```

Timed pattern:

```
.loop   DRVNOT  #LED
        WAITX   ##50_000_000
        JMP     #.loop
```

That's 80% of all I/O right there!

Advanced Pin Control

Pin Groups

You can control multiple pins at once:

```
        DRVH    #LED_BASE addpins 3  ' Drive 4 pins high (base+3)
        DRVL    #LED_BASE addpins 7  ' Drive 8 pins low
```

Direct Pin Manipulation

For when you need absolute control:

```
        MOV     outa, pattern    ' Set output register directly
        MOV     dira, ##$FF      ' Set direction reg (rare in P2!)
```

But honestly? You'll rarely need these. The individual pin instructions are cleaner and clearer.

Common I/O Gotchas

Before you pull your hair out wondering why a pin "won't work," save yourself debugging time and skim these:

1. **Pin numbers are 0-63** - Not port.bit notation like other MCUs
2. **No pullup/pulldown by default** - Use external resistors or configure Smart Pin modes (advanced topic)
3. **Pins float on reset** - All pins start as inputs (floating)
4. **Reading output pins** - You CAN read a pin you're driving (reads the actual pin state)
5. **3.3V logic levels** - P2 is 3.3V, not 5V tolerant!

Timing Is Everything

Now here's something we'll keep coming back to: P2 I/O is deterministic. When you execute:

```
        DRVH    #56
        DRVL    #57
```

Pin 56 goes high and pin 57 goes low at EXACTLY the same clock cycle. No skew, no uncertainty. Uff! Try doing that on an interrupt-driven MCU. This determinism is what makes P2 perfect for precise timing applications.

Real-World Example: Servo Control

Even without Smart Pins, controlling a servo is easy:

```
' Standard servo control (1-2ms pulse every 20ms)
servo_control
        MOV     position, ##300_000    ' 1.5ms = center at 200MHz

servo_loop
        DRVH    #SERVO_PIN
        WAITX   position              ' 1-2ms high pulse
        DRVL    #SERVO_PIN
        WAITX   ##4_000_000          ' Rest of 20ms at 200MHz

        ' Adjust position as needed
        RDLONG  position, ##position_addr
        FLE     position, ##200_000   ' Limit to 1ms min
        FGE     position, ##400_000   ' Limit to 2ms max

        JMP     #servo_loop
```

The Power of Simple

Here's something beautiful about P2's I/O philosophy: it's transparent. Unlike microcontrollers with complex GPIO configurations, port multiplexing, and alternate functions, P2 pins just… work.

Want an output? Drive it. Want an input? Read it. Want it to float? Float it.

No setup, no configuration, no confusion.

What We've Learned

Look at your new I/O skills:

- Understood P2's automatic pin direction
- Mastered the four fundamental output instructions
- Learned pin reading and conditional testing
- Created debounced inputs
- Built bit-banged serial
- Discovered deterministic timing

A Note About Smart Pins

You might wonder - if basic I/O is this simple, why do we need Smart Pins?

Well, while you CAN bit-bang serial at 115200 baud, or generate PWM, or measure frequencies using the techniques in this chapter, Smart Pins do all of this in hardware, freeing your COG for more important work.

For Smart Pin details: See the dedicated "P2 Smart Pins Manual" which covers all 64 modes, from simple PWM to complex protocol generation. Smart Pins deserve their own complete treatment!

Coming Up Next

Chapter 9 takes us into "Streaming Data" - the P2's incredible FIFO system that can move megabytes of data without breaking a sweat. We'll see how to stream video, audio, and massive data blocks at maximum speed.

Have Fun! Remember, every embedded system ultimately comes down to pins going high and low. You've just mastered the fundamentals that everything else builds upon!

Chapter 9: Streaming Data

Moving mountains of data without breaking a sweat

The Hook: 4KB in 4 Instructions

Watch this data transfer magic:

```
' Copy 1000 longs (4KB) at maximum speed
        SETQ    ##1000-1        ' Setup for 1000 longs
        RDLONG  buffer, source  ' Read them all!
        SETQ    ##1000-1        ' Setup for 1000 longs
        WRLONG  buffer, dest    ' Write them all!
        ' 4KB moved in microseconds!
```

Four instructions. Four kilobytes. Faster than DMA on most processors. And we're just getting started…

Block Transfers: The Power Move

The **SETQ** instruction is your gateway to block transfers:

```
' Basic block read
        SETQ    #16-1           ' Transfer 16 longs (minus 1!)
        RDLONG  buffer, hubaddr ' Reads 16 consecutive longs

' Basic block write
        SETQ    #16-1           ' Transfer 16 longs
        WRLONG  buffer, hubaddr ' Writes 16 consecutive longs
```

Here's the trick: **SETQ** tells the next hub instruction how many longs to transfer. The "-1" is because it's a count from 0 (yes, we'll trip over that off-by-one at least once — everyone does).

The FIFO: Your Streaming Pipeline

The FIFO (First In, First Out) is P2's streaming engine. Think of it as a conveyor belt between hub memory and your COG:

```
' Start FIFO reading
        RDFAST  #0, ##data_start  ' Start fast read at data_start

' Now read data at maximum speed
stream_loop
        RFLONG  value            ' Read from FIFO (no waiting!)
        ' Process value here
        ADD     accumulator, value
        DJNZ    count, #stream_loop

' The FIFO keeps feeding data automatically
```

The beauty? The FIFO reads ahead automatically. While you're processing one value, it's already fetching the next. No hub timing slots to worry about!

Writing Through the FIFO

Reading was nice — writing is symmetric. We'll use **WRFAST** instead of **RDFAST**, and **WFLONG** instead of **RFLONG**, and that's it:

```
' Start FIFO writing
        WRFAST  #0, ##dest_buffer

' Stream data out
write_loop
        ' Generate or process data
        MOV     value, calculation
        WFLONG  value            ' Write to FIFO
        DJNZ    count, #write_loop

' Data streams to hub automatically
```

Real-World Example: Screen Buffer Clear

Let's clear a 320x240x4 **BYTE** screen buffer (~307KB - fits in hub!):

```
' Ultra-fast screen clear
clear_screen
        MOV     color, ##$00_00_00_00    ' Black (4 bytes per pixel)
```

continues on next page →

↪ continued from previous page

```
        MOV     pixels, ##320*240        ' 76,800 pixels

        WRFAST  #0, ##screen_buffer      ' Start FIFO write

clear_loop
        WFLONG  color                    ' Write 4-byte pixel
        DJNZ    pixels, #clear_loop

        ' 307KB cleared at maximum hub speed!
        ' Note: Hub RAM is 512KB - plan buffer sizes accordingly
```

Streaming with the Streamer

The Streamer is different from the FIFO - it's a dedicated DMA engine that can move data between hub memory and pins:

```
' Configure streamer for video output
        SETXFRQ ##PIXEL_FREQ    ' Set the pixel (NCO) output rate

' Start streaming video data to pins
        XINIT   ##STREAM_CMD, #0  ' Start streamer
        ' Data flows from hub to pins automatically!
```

FIFO and COG Execution

Here's something amazing - you can execute code from hub through the FIFO:

```
' Execute large program from hub
        ORGH    $1000           ' Code in hub memory

hub_code
        ' This code is in hub but executes like it's in COG
        ADD     x, y
        SUB     a, b
        ' Can be megabytes of code!
```

When you **CALL** or jump to hub code, the FIFO automatically feeds instructions to the COG. It's like having unlimited code space!

The Medicine Cabinet

Feeling overwhelmed by all this streaming? Here's your prescription:

Just need to move data? Use these simple patterns:

Block read pattern:

```
        SETQ    #SIZE-1
        RDLONG  buffer, source
```

Block write pattern:

```
        SETQ    #SIZE-1
        WRLONG  buffer, dest
```

FIFO read pattern:

```
        RDFAST  #0, ##source
.loop   RFLONG  value
        ' Process value
        DJNZ    count, #.loop
```

That's 90% of streaming right there!

Advanced Streaming Techniques

Circular Buffers with FIFO

```
' Circular buffer reading
        RDFAST  ##BUF_BLOCKS, ##buffer  ' D[13:0]=block count; auto-wraps

circular_loop
        RFLONG  value                   ' Read from FIFO
        ' Process value
        ' FIFO automatically wraps at buffer end!
        JMP     #circular_loop
```

Processing Pipeline with FIFO

```
' Read data with FIFO, process, write via PTRA
        RDFAST  #0, ##source    ' Set up FIFO for reading
        MOV     dest_ptr, ##dest

pipeline
        RFLONG  input           ' Get next input from FIFO

        ' Scale using 16x16 multiply (result in input)
        MUL     input, #SCALE_FACTOR

        WRLONG  input, dest_ptr ' Write result via PTRA
        ADD     dest_ptr, #4
        DJNZ    count, #pipeline
```

Note: FIFO can only read OR write at a time, not both. Use PTRA/PTRB for the other direction.

Your Turn: Streaming Experiments

Your Turn

Your Turn: Fast memory fill
Starting code:

```
        ORG     0

        MOV     pattern, ##$DEADBEEF
        MOV     dest, ##$1000
        MOV     count, #256

        ' Your code here: Fill 256 longs with pattern
        ' Use SETQ and WRLONG
```

Goal: Fill memory with pattern using block transfer Hint: You'll need **SETQ** #255 (not #256) Success Check: Memory filled in one operation

Your Turn

Your Turn: Data filter pipeline
Starting code:

```
        ORG     0

        RDFAST  #0, ##input_data        ' FIFO for reading
        MOV     ptra, ##output_data     ' PTRA for writing
        MOV     count, #100

filter_loop
        RFLONG  value                   ' Read from FIFO
        ' Your code: Simple filter
        ' Maybe average with previous value?
        WRLONG  result, ptra++          ' Write via PTRA
        DJNZ    count, #filter_loop
```

Goal: Process streaming data through simple filter Hint: Keep previous value in a register Success Check: Output is filtered version of input

Common Streaming Gotchas

Before you pull your hair out wondering why your transfer is one **LONG** short, or why your FIFO won't cooperate, skim these:

1. **SETQ uses count-1** - For 16 longs, use `SETQ #15`, not `SETQ #16`
2. **FIFO is shared per COG** - Can't use FIFO for both code execution and data streaming simultaneously
3. **Write synchronization** - **WRFAST** writes complete in the background. To force a flush, issue the next **RDFAST**/WRFAST with D[31]=0 (it waits for the prior **WRFAST** to finish) rather than relying on a fixed delay
4. **Hub alignment** - Block transfers work best with long-aligned addresses
5. **FIFO depth** - The FIFO holds (cogs+11) = 19 longs. It refills automatically, so you rarely outrun it.

Performance Numbers

Let's talk speed:

- **Block transfer**: Up to 1 **LONG** per clock (at 200MHz = 800MB/s!)
- **FIFO streaming**: Up to 1 **LONG** per clock sustained
- **Random hub access**: 9-16 clocks per access
- **Streamer to pins**: Up to sysclock/1 rate

Uff! That's seriously fast. Most microcontrollers need dedicated DMA controllers, peripheral coprocessors, and a stack of config registers to achieve what P2 does with two instructions. You're not paying for that DMA controller — it's already in the silicon.

Real-World Example: Audio Buffer

Stream audio samples through processing:

```
' Audio processing pipeline
audio_process
        RDFAST  #0, ##input_buffer      ' FIFO for reading input
        MOV     ptra, ##output_buffer   ' PTRA for writing output
        MOV     samples, ##BUFFER_SIZE

process_loop
        RFLONG  left_sample             ' Read left from FIFO
        RFLONG  right_sample            ' Read right from FIFO

        ' Apply simple low-pass filter
        ADD     left_filtered, left_sample
        SHR     left_filtered, #1       ' Average with previous

        ADD     right_filtered, right_sample
        SHR     right_filtered, #1

        ' Apply volume
        MULS    left_filtered, volume
        MULS    right_filtered, volume

        ' Output processed samples via PTRA
        WRLONG  left_filtered, ptra++
        WRLONG  right_filtered, ptra++

        DJNZ    samples, #process_loop
```

What We've Learned

Your streaming skills now include:

- Block transfers with **SETQ**
- FIFO reading and writing
- Streaming pipeline concepts
- Circular buffer techniques
- Parallel processing while streaming
- Real-world applications

Coming Up Next

Chapter 10 explores "Hub Execution" - how to break free from the 496-instruction limit and run massive programs directly from hub memory. It's like having your cake and eating it too!

Have Fun! Remember, streaming is about throughput, not just speed. It's the difference between carrying one brick at a time and using a wheelbarrow!

Chapter 10: Hub Execution

Breaking free from the 496-instruction limit

The Hook: Unlimited Code Space

Remember fretting about fitting your code into 496 COG instructions? Watch this:

```
        ORGH    $400            ' Place code in hub memory

        ' This can be thousands of instructions!
main    MOV     x, #0
        MOV     y, #0
        CALL    #huge_function  ' Can be massive
        CALL    #another_big_one
        CALL    #yet_another
        ' Keep going... no limit!

huge_function
        ' 1000 instructions? No problem!
        ' 10000 instructions? Still fine!
        RET
```

Your code now lives in hub memory's 512KB instead of COG memory's 2KB. That's 256 times more space!

COG vs Hub Execution: The Trade-offs

Let's be honest about the differences:

COG Execution (traditional):

- Fast: exactly 2 clocks per instruction
- Deterministic: perfect for real-time
- Limited: only 496 instructions
- Self-contained: runs independently

Hub Execution (the new way):

- Fast sequential: 2 clocks per instruction (same as cog-exec, thanks to the 19-stage FIFO prefetch)

- Slower on branches: minimum 13 clocks per branch (the FIFO refill cost; +1 if target isn't long-aligned)
- Unlimited: 512KB of code space!
- Flexible: can call COG routines

The beauty? You can mix both in the same program! Sequential code in hub runs at full speed — only branches show the hub-execution penalty.

How Hub Execution Works

Let's peek behind the curtain. When the processor encounters a jump or call to a hub address ($400), it automatically switches to hub execution mode. The FIFO starts streaming instructions from hub memory:

```
        ORG     0               ' Start in COG

cog_code
        ' This runs from COG RAM
        CALL    #hub_function   ' Call into hub
        ' Back in COG mode here

        ORGH    $1000           ' Switch to hub addresses

hub_function
        ' This runs from hub RAM via FIFO
        ' Can be huge!
        RET                     ' Returns to COG code
```

The magic happens automatically. No mode switching instructions needed!

Real-World Example: Menu System

Here's something that would never fit in COG RAM:

```
        ORGH    $2000

menu_system
        CALL    #draw_menu_frame
        CALL    #display_options
        CALL    #get_user_input
```

continues on next page →

↪ *continued from previous page*

```
        CALL    #process_selection

        CMP     selection, #1 wcz
   if_e CALL    #option_1_handler
        CMP     selection, #2 wcz
   if_e CALL    #option_2_handler
        CMP     selection, #3 wcz
   if_e CALL    #option_3_handler
        ' ... many more options

        JMP     #menu_system

draw_menu_frame
        ' 200 instructions for fancy graphics
        RET

display_options
        ' 300 instructions for text rendering
        RET

option_1_handler
        ' 500 instructions for configuration
        RET

' Thousands of instructions total - no problem!
```

The Hub Execution FIFO

You met the FIFO in the previous chapter as a data-streaming engine. Same hardware, different job here: it reads ahead, keeping a buffer of upcoming *instructions* ready for the COG. Think of it as a moving sidewalk for your code:

```
' The FIFO maintains performance by reading ahead
hub_loop
        ADD     x, y            ' FIFO has next instructions ready
        SUB     a, b            ' No waiting for hub access
        MUL     c, d            ' Instructions stream smoothly
        ' FIFO automatically refills as needed
```

This read-ahead behavior is the whole reason sequential hub code matches cog-exec speed — the FIFO is doing the waiting for you, in parallel with the COG running instructions it already prefetched.

Mixing COG and Hub Code

Here's the real power - combining both modes:

```
        ORG     0

' Critical timing code in COG
critical_loop
        TESTP   #TRIGGER_PIN wz       ' Test trigger pin
  if_nz JMP     #critical_loop        ' Loop until triggered
        DRVH    #CRITICAL_PIN         ' Immediate response!
        CALL    #hub_process    ' Do complex processing
        JMP     #critical_loop

        ORGH    $4000

' Complex processing in hub
hub_process
        ' Hundreds of instructions for data analysis
        ' Not time-critical, so hub execution is fine
        RET
```

Time-critical code stays in COG RAM for deterministic timing. Complex code lives in hub RAM for space.

Your Turn: Hub Execution Experiments

Your Turn

Your Turn: Build a simple calculator
Starting code:

```
        ORG     0
        JMP     #calculator    ' Jump to hub code

        ORGH    $1000
calculator
        ' Your code here:
        ' 1. Display menu
        ' 2. Get operation choice
        ' 3. Get two numbers
        ' 4. Call appropriate function
        ' 5. Display result

add_function
        ' Addition code
        RET

subtract_function
        ' Subtraction code
        RET

' Add more functions - no size limit!
```

Goal: Create a multi-function calculator Hint: Each function can be as complex as needed Success Check: Multiple operations working

The Medicine Cabinet

Overwhelmed by execution modes? Here's the simple version:

Keep it simple:

1. **Small, time-critical code** → Put in COG (org 0)
2. **Large, complex code** → Put in hub (orgh $400+)
3. **Don't overthink it** → The processor handles the switch

Basic pattern:

```
        ORG     0
        JMP     #main       ' Jump to hub

        ORGH    $400
main    ' Your big program here
```

That's it. Let the processor worry about the details!

Advanced Hub Execution

Long Jumps and Calls

Hub addresses need 20 bits, so jumping far requires special handling:

```
' Jump to distant hub code
        JMP     ##far_away      ' ## forces 32-bit immediate

        ORGH    $40000          ' Far away in hub
far_away
        ' Code here
```

Hub Data Access from Hub Code

When executing from hub, you can still access hub data:

```
        ORGH    $1000

hub_code
        RDLONG  value, ##hub_data   ' Read hub data
```

continues on next page →

↪ continued from previous page

```
        ADD     value, #1
        WRLONG  value, ##hub_data  ' Write back

        ORGH    $8000
hub_data
        LONG    $12345678
```

Performance Optimization

To maximize hub execution speed:

```
' Align branch targets to 8-byte boundaries
        ALIGNL                  ' Align to long boundary
loop_start
        ' Loop code here
        DJNZ    count, #loop_start

' Keep critical loops small
' Consider moving inner loops to COG RAM
```

Common Hub Execution Gotchas

Before you cram everything into hub and call it a day, know these:

1. **Speed variation** - Don't use hub execution for precise timing
2. **FIFO conflicts** - Can't stream data while executing from hub
3. **Address confusion** - Remember: <$200 is COG, $200-$3FF is LUT, $400 is hub
4. **Stack depth** - Still limited to 8-level hardware stack
5. **Relative jumps** - Work differently in hub mode

Real-World Example: Command Parser

```
        ORGH    $2000

command_parser
        CALL    #get_command_line
        CALL    #tokenize

```

continues on next page →

↪ *continued from previous page*

```
        ' Compare against commands
        MOV     ptra, #command_string
        MOV     ptrb, ##cmd_help
        CALL    #string_compare
   if_z JMP     #help_command

        MOV     ptrb, ##cmd_run
        CALL    #string_compare
   if_z JMP     #run_command

        ' Many more commands...

help_command
        ' 500 instructions of help text display
        RET

run_command
        ' 1000 instructions of program execution
        RET

string_compare
        ' 50 instructions of string comparison
        RET

' Thousands of instructions total
' Would need multiple COGs without hub execution!
```

When to Use Hub Execution

Perfect for:

- User interfaces and menus
- Command processors
- Complex algorithms
- String manipulation
- Protocol handlers
- Error handling and recovery

Avoid for:

- Interrupt handlers (if you use them)

- Precise timing loops
- Bit-banged protocols
- Real-time control loops

What We've Learned

You've mastered hub execution:

- Understanding COG vs hub trade-offs
- Automatic mode switching
- Mixing COG and hub code
- FIFO streaming of instructions
- When to use each mode
- Real-world applications

Coming Up Next

Chapter 11 tackles the controversial topic: "Why No Interrupts?" We'll explore why the Propeller philosophy says you don't need them, and why that's actually a good thing!

Have Fun! Hub execution is like having a sports car that can also carry furniture - you get both speed and capacity when you need them!

Chapter 11: Why No Interrupts?

The most controversial P2 feature explained

The Hook: Interrupts Without Interrupts

Here's a traditional interrupt-driven button handler:

```
' Traditional approach (not P2!) - pseudocode for the interrupt-driven style
ISR(BUTTON_INTERRUPT)
    ' Interrupt service routine
    buttonPressed = true
    ' Return to interrupted code
END_ISR
```

And here's the P2 way:

```
' Dedicated COG watching button
button_watcher
         TESTP   #BUTTON_PIN wc
    if_c WRLONG  ##1, ##button_flag
         JMP     #button_watcher

' Main COG doing important work
main_code
         ' Never interrupted!
         ' Checks button_flag when convenient
```

No interrupt latency. No context switching. No priority inversion. No critical sections. Just clean, deterministic, parallel processing.

The Interrupt Problem

Let me tell you a story. You're concentrating on a complex problem when someone taps your shoulder. You stop, handle their request, then try to remember where you were. Now imagine this happening randomly, unpredictably, dozens of times per second.

That's interrupts.

Traditional processors need interrupts because they only have one processor. Something important happens? Stop everything and handle it! But this causes:

- **Latency**: Time to save context and jump to handler
- **Jitter**: Variable response time depending on what was interrupted
- **Priority inversion**: Low-priority task blocks high-priority
- **Race conditions**: Shared data access problems
- **Debugging nightmares**: Non-reproducible timing bugs

The Propeller Solution

Eight processors. No sharing required.

```
' COG 0: Main application
main_app
        ' Complex calculations
        ' Never interrupted
        RDLONG  command, ##mailbox wz
   if_nz CALL   #process_command
        JMP     #main_app

' COG 1: Serial port handler
serial_handler
        ' Continuously monitors serial
        TESTP   #RX_PIN wc
   if_c CALL    #receive_byte
        JMP     #serial_handler

' COG 2: Motor control
motor_control
        ' Precise timing loops
        ' Never disrupted
        WAITCNT motor_time
        DRVNOT  #STEP_PIN
        JMP     #motor_control

' COG 3: Sensor monitor
sensor_monitor
        ' Watches multiple sensors
        ' Responds instantly
        ' ... and so on
```

Each COG does one thing perfectly. No interruptions. No conflicts. Just pure, focused execution.

Real-World Example: Perfect Servo Control

With interrupts, servo pulses jitter. With dedicated COGs, they're perfect:

```
' COG dedicated to servo control
servo_cog
        GETCT   pulse_time

servo_loop
        ' Generate 8 servo pulses simultaneously
        MOV     servo_mask, ##$FF      ' 8 servos
        OR      outa, servo_mask       ' All high

        MOV     index, #0
check_servos
        RDLONG  width, ptra[index]     ' Get pulse width
        ADDCT1  pulse_time, width      ' Set compare time

        WAITCT1                        ' Wait for exact time
        BITL    outa, index            ' Turn off this servo

        INCMOD  index, #7  wc          ' C set when index wraps 7->0
  if_nc JMP     #check_servos          ' Loop until all 8 servos done

        ' Wait for 20ms frame
        WAITX   ##4_000_000
        JMP     #servo_loop

' Result: 8 servos with ZERO jitter!
```

Try that with interrupts. I'll wait. Actually, I won't - it's impossible to achieve this precision with interrupts.

"But P2 HAS Interrupts!"

Yes, it does. And you probably shouldn't use them.

Well, let me be more nuanced. P2 has interrupts for those rare cases where you absolutely need them:

```
' Setting up an interrupt (not recommended!)
        SETSE1  #%001<<6 + PANIC_BUTTON   ' SE1 triggers when pin goes high
        SETINT1 #EVENT_SE1                ' Enable INT1 on SE1 event

int1_handler
        ' Interrupt code here
        RETI1
```

When might you use them?

- Porting legacy code that requires interrupts
- Ultra-low-power designs where COGs must sleep
- Theoretical minimum latency response (but dedicated COG is usually faster!)

Uff! Even writing interrupt code feels wrong on a Propeller!

The Medicine Cabinet

Still thinking you need interrupts? Here's your medicine:

Think you need an interrupt for...

Fast response?

```
' Dedicated COG responds in ~4 clocks
watcher
        TESTP   #INPUT_PIN wz         ' Test pin state
  if_nz JMP     #watcher              ' Loop until pin high
        DRVH    #RESPONSE_PIN         ' Instant response!
```

Multiple events?

```
' One COG watches everything
monitor
        TEST    sensors, #SENSOR1 wz
   if_nz CALL   #handle_sensor1
        TEST    sensors, #SENSOR2 wz
   if_nz CALL   #handle_sensor2
        ' Check all sensors every loop
```

continues on next page →

↪ *continued from previous page*

Periodic tasks?

```
' Perfect timing without interrupts
        GETCT   next_time
.loop   ADDCT1  next_time, ##PERIOD
        WAITCT1                 ' Exact timing
        CALL    #periodic_task
        JMP     #.loop
```

See? No interrupts needed!

The Event System: Better Than Interrupts

P2 has something better than interrupts - events:

```
' Configure event to watch pin
        SETSE1  #%01_000000 | BUTTON_PIN  ' Rising edge event

' Main code runs normally
main_loop
        ' Do work...
        POLLSE1 wc              ' Check if event occurred
    if_c CALL   #handle_button  ' Handle when convenient
        ' Continue work...
        JMP     #main_loop
```

Events are like interrupts that wait politely for you to check them. No rudeness!

Interrupt Horror Stories

Let me share why we avoid interrupts:

Story 1: The Jittery Display

Approach	Problem	Result
With Interrupts	Display updates interrupted by serial	Visible glitches, tearing, inconsistent timing
With COGs	Display COG runs uninterrupted	Perfect, smooth, glitch-free display

Story 2: The Missed Pulse

Approach	Problem	Result
With Interrupts	Motor step interrupted by sensor read	Missed step, motor stalls, position lost
With COGs	Motor COG never misses a beat	Perfect positioning, no lost steps

Story 3: The Debugging Nightmare

Approach	Problem	Result
With Interrupts	Bug only appears under specific timing	Days of debugging, hair loss, coffee overdose
With COGs	Deterministic timing, reproducible behavior	Bug found in minutes, sanity preserved

Your Turn: COG vs Interrupt Challenge

Your Turn

Your Turn: Build a reaction timer without interrupts
Starting code:

continues on next page →

↪ *continued from previous page*

```
        ORG     0

' COG 0: Main game logic
        SETQ    ##button_flag           ' PTRA for new COG
        COGINIT #1, @button_watcher

game_loop
        ' Random delay
        GETRND  delay
        WAITX   delay

        DRVH    #LED_PIN
        WRLONG  #0, ##button_flag
        GETCT   start_time

' Wait for button (no interrupt!)
wait_press
        RDLONG  pressed, ##button_flag wz
   if_z JMP     #wait_press

        GETCT   end_time
        SUB     end_time, start_time
        ' Display reaction time

' COG 1: Button watcher
        ORGH    $400
button_watcher
        ' Your code here
```

Goal: Implement button watcher COG Hint: Continuously monitor and set flag Success Check: Perfect timing without interrupts

The Philosophy Deep Dive

The Propeller philosophy is about **determinism over responsiveness**.

Traditional processors optimize for average-case performance:

- Interrupts handle rare events
- Most code runs uninterrupted

- When events happen, everything stops

Propeller optimizes for worst-case determinism:

- Every COG runs predictably
- No surprises, ever
- Timing is guaranteed

It's the difference between a talented soloist who might miss a note and an orchestra where everyone plays their part perfectly.

When Interrupts Actually Make Sense

I'll admit it - there are rare cases where interrupts are appropriate:

1. **Power-critical applications** where COGs must sleep
2. **Legacy code ports** that fundamentally require interrupts
3. **Single-COG designs** (but why waste the P2's power?)

But in 15 years of Propeller programming, I've needed interrupts exactly… never.

Common "But What About..." Questions

Q: "But what about interrupt priority?" A: COGs don't have priority. They're all equal. Design your system accordingly.

Q: "How do I handle critical events?" A: Dedicate a COG to critical events. It will respond faster than any interrupt.

Q: "Isn't dedicating a whole COG wasteful?" A: You have eight! And a focused COG is simpler than interrupt-riddled code.

Q: "What about power consumption?" A: Use WAITSE/WAITCT for low-power waiting. COG sleeps until event.

What We've Learned

You now understand the Propeller way:

- Why interrupts cause problems
- How COGs eliminate interrupt need
- Event system as polite alternative
- Real-world benefits of no interrupts
- Rare cases where interrupts might be used

- The philosophy of determinism

Coming Up Next

Chapter 12 shows you "Optimization Mastery" - how to make your PASM2 code blazingly fast. We'll explore the pipeline, instruction pairing, and timing tricks that squeeze every drop of performance from the P2.

Have Fun! And remember - in a world of interruptions, be a COG: focused, deterministic, and uninterruptible!

Chapter 12: Optimization Mastery

Making the fast faster

The Hook: Double Your Speed with One Change

Let me show you a loop that looks fine — until you realize you're paying for the same thing twice. Watch:

```
' Before optimization: 13 clocks
.loop   RDLONG  value, ptra      ' 9-16 (cog-exec) / 9-26 (hub-exec)
        ADD     value, #1        ' 2 clocks
        WRLONG  value, ptra      ' 3-10 (cog-exec) / 3-20 (hub-exec)
        ADD     ptra, #4         ' 2 clocks
        DJNZ    count, #.loop    ' 2/4 (cog-exec) / 2/13-20 (hub-exec)

' After optimization using PTR expressions:
.loop   RDLONG  value, ptra      ' Read from current address
        ADD     value, #1        ' Process
        WRLONG  value, ptra++    ' Write and increment in one!
        DJNZ    count, #.loop    ' Saved the ADD instruction
```

Almost twice as fast! The secret? Understanding how P2 really works.

Understanding the Pipeline

Before we hunt for clocks to save, let's understand where they go in the first place. P2 has a 2-stage pipeline:

1. **Fetch** - Get next instruction
2. **Execute** - Do the work

This means while one instruction executes, the next is already being fetched:

```
        ADD     x, y      ' Executing while next inst fetches
        SUB     a, b      ' Fetching while previous executes
        ' Perfect overlap = maximum throughput
```

Instruction Timing Basics

Not all instructions are created equal:

```
' 2-clock instructions (most ALU operations)
        ADD     x, y            ' 2 clocks
        MOV     a, b            ' 2 clocks
        AND     c, d            ' 2 clocks

' Variable timing (hub access)
        RDLONG  value, hubaddr  ' 9-16 clocks (hub slot wait)
        WRLONG  value, hubaddr  ' 3-10 clocks (variable)

' Long operations (CORDIC)
        QROTATE x, angle        ' 2 clocks to start
        GETQX   result          ' 2 clocks (but wait 55 for result)

' Special cases
        MUL     x, y            ' 2 clocks
        QDIV    x, y            ' 2 clocks to start
        GETQX   result          ' 2 clocks (but wait 30 for result)
```

REP: The Speed Loop

REP creates hardware-accelerated loops with zero overhead:

```
' Traditional loop: overhead per iteration
.loop   ADD     sum, value      ' 2 clocks
        ADD     ptr, #4         ' 2 clocks
        DJNZ    count, #.loop   ' 2 or 4 clocks (4 if branch taken)

' REP loop: 0 clocks overhead!
        REP     #2, count       ' Repeat next 2 instructions
        ADD     sum, value      ' 2 clocks
        ADD     ptr, #4         ' 2 clocks = 4 total!
```

That's 33% faster just by using **REP**!

Hub-Exec Note: REP works in hub-exec too, but each iteration executes a hidden jump to loop back — and that hidden jump pays the 13+ clock hub-branch cost per iteration. So a 2-instruction **REP** loop that takes 4 clocks in cog-exec balloons to ~17+ clocks per iteration in hub-exec. For

time-critical inner loops, keep **REP** in COG or LUT memory. Hub-exec **REP** works correctly; it just isn't zero-overhead there.

SKIP: Conditional Execution on Steroids

SKIP and **SKIPF** let you conditionally execute patterns of instructions:

```
' Traditional: multiple jumps
        CMP     x, #5 wcz
if_a    JMP     #greater
if_b    JMP     #less
        JMP     #equal

' With SKIP: cancel optional steps per a precomputed pattern
'   bit N (LSB first) skips the Nth instruction after SKIP
        SKIP    config_mask     ' e.g. %010 runs steps 0 and 2, skips step 1
        CALL    #setup_uart     ' Step 0
        CALL    #setup_spi      ' Step 1
        CALL    #setup_timer    ' Step 2
        ' One pattern picks which steps run - no jumps, no stalls!
```

Hub Access Optimization

Hub timing is critical for performance:

```
' Unaligned hub access: variable timing
        RDLONG  v1, ##$1001     ' Not long-aligned, slower

' Aligned hub access: predictable timing
        RDLONG  v1, ##$1000     ' Long-aligned, faster

' Sequential access: maximum speed
        RDLONG  v1, ptra++      ' Hardware manages sequence
        RDLONG  v2, ptra++      ' Optimal hub slot usage
        RDLONG  v3, ptra++      ' Maximum throughput
```

The FIFO Fast Path

For ultimate speed, use the FIFO:

```
' Traditional hub reading: ~6 clocks average per long
.loop   RDLONG  value, ptra++
        ADD     sum, value
        DJNZ    count, #.loop

' FIFO reading: 2 clocks per long!
        RDFAST  #0, ptra        ' Start FIFO
.loop   RFLONG  value           ' 2 clocks, always!
        ADD     sum, value      ' 2 clocks
        DJNZ    count, #.loop   ' 2 clocks
        ' 3x faster for sequential reads!
```

Parallel Operations

CORDIC operations can overlap with other work:

```
' CORDIC overlaps with other instructions
        QMUL    x, y            ' Start 32x32->64 multiply (CORDIC)
        ' 55 clocks to do other work!
        ADD     a, b            ' These execute during CORDIC
        SUB     c, d
        MOV     index, #0
        RDLONG  data, ptra++
        ' ... more work
        GETQX   low_result      ' Get CORDIC result (lower 32 bits)
        GETQY   high_result     ' Get CORDIC result (upper 32 bits)

' QROTATE overlap
        QROTATE x_coord, angle  ' Start rotation (D=X, S=angle)
        ' 55 clocks of other work!
        GETQX   new_x           ' Get rotated X
        GETQY   new_y           ' Get rotated Y
```

Note: **MUL**/MULS are 2-clock ALU instructions that complete immediately (16x16->32). Use **QMUL** for 32x32->64 with CORDIC overlap.

Real-World Example: Fast Memory Copy

Let's optimize a memory copy routine:

```
' Version 1: Basic (slow)
copy_basic
        RDLONG  temp, source
        WRLONG  temp, dest
        ADD     source, #4
        ADD     dest, #4
        DJNZ    count, #copy_basic
        ' ~13 clocks per long

' Version 2: Better pointers
copy_better
        RDLONG  temp, ptra++
        WRLONG  temp, ptrb++
        DJNZ    count, #copy_better
        ' ~8 clocks per long

' Version 3: Block transfer (ultimate)
copy_ultimate
        SUB     count, #1       ' SETQ needs count-1 (0 = 1 long)
        SETQ    count           ' Setup block transfer
        RDLONG  buffer, source  ' Read all at once
        SETQ    count           ' (count already decremented)
        WRLONG  buffer, dest    ' Write all at once
        ' ~1 clock per long for large blocks (hub maximum)!
```

The Medicine Cabinet

Optimization overwhelming you? Start with these simple improvements:

Three easy wins:

1. **Use PTRA/PTRB** instead of manual pointer math

```
' Slow
        RDLONG  x, addr
        ADD     addr, #4

' Fast
        RDLONG  x, ptra++
```

2. **Align your data** to **LONG** boundaries

```
        ALIGNL          ' Force long alignment
data    LONG    $12345678
```

3. **Use REP** for tight loops (note: hub-address expressions like `ptra++` only work with hub-access instructions, so we read first, then add)

```
        REP     @.end, count
        RDLONG  val, ptra++     ' Fetch next long from hub
        ADD     sum, val        ' Accumulate
.end
```

Just these three changes often double performance!

Your Turn: Optimization Challenges

Your Turn

Your Turn: Optimize a checksum calculator

Starting code:

continues on next page →

↪ continued from previous page

```
' Slow version
checksum_slow
        MOV     sum, #0
        MOV     addr, ##buffer
        MOV     count, #256

.loop   RDBYTE  temp, addr
        ADD     sum, temp
        ADD     addr, #1
        DJNZ    count, #.loop
```

Goal: Make it at least 4x faster Hint: Read longs instead of bytes, use FIFO Success Check: Same checksum, much faster

Advanced Techniques

Instruction Pairing

Some instruction pairs execute specially:

```
' ## syntax handles AUGS automatically
        MOV     x, ##$12345678  ' Assembler generates AUGS + MOV
        ' Same result, cleaner code!

' ALTD + instruction = indirect addressing
        ALTD    index, #array
        MOV     0-0, value      ' Stores to array[index]
```

Pipeline-Aware Coding

Avoid pipeline stalls:

```
' Bad: result needed immediately
        ADD     x, y
        CMP     x, #10 wcz      ' Stall waiting for x

' Good: interleave operations
```

continues on next page →

↪ *continued from previous page*

```
        ADD     x, y
        MOV     a, b            ' Do something else
        CMP     x, #10 wcz      ' Now x is ready
```

Unrolling Loops

Sometimes removing the loop is faster. (Remember: `ptra++` only works with hub-access instructions like **RDLONG**, so each iteration reads then **ADDS**.)

```
' Looped version
        REP     @.end, #4
        RDLONG  val, ptra++
        ADD     sum, val
.end

' Unrolled version (faster for small counts)
        RDLONG  val, ptra++
        ADD     sum, val
        RDLONG  val, ptra++
        ADD     sum, val
        RDLONG  val, ptra++
        ADD     sum, val
        RDLONG  val, ptra++
        ADD     sum, val
```

Common Optimization Gotchas

Before you rewrite everything in **REP** and SKIP, a few sanity checks:

1. **Premature optimization** - Get it working first, then optimize
2. **Over-optimizing** - Sometimes clarity is worth 2 clocks
3. **Ignoring the big picture** - Optimize the bottleneck, not everything
4. **Breaking functionality** - Fast but wrong is useless
5. **Forgetting about power** - Faster isn't always better for battery life

Profiling and Measurement

Always measure your optimizations:

```
' Time your code
        GETCT   start_time

        ' Code to measure
        CALL    #function_to_test

        GETCT   end_time
        SUB     end_time, start_time
        ' end_time now contains exact clock cycles
```

Pitfall — CT wraps: GETCT reads a 32-bit free-running counter that wraps every ~21.5 seconds at 200 MHz ($2^{32} \div 200$ MHz). For short measurements like the one above, the **SUB** trick masks the wrap correctly thanks to two's-complement arithmetic. But for a *scheduler* or *timer* running over minutes, hours, or days, you need one of two strategies:

1. **Capture the full 64-bit count.** `GETCT D WC` returns the upper 32 bits (with `WC` set); a plain `GETCT D` returns the lower 32. Read upper-then-lower-then-upper-again and retry if the upper **WORD** changed, then build a 64-bit timestamp.
2. **Work with deltas, not absolute time.** Compare `(now - start)` rather than `now > deadline`. The subtraction wraps correctly even when the counter does.

The 21.5-second wrap is *fast* — long enough that you won't see it in a basic example, short enough that real applications hit it constantly.

What We've Learned

You're now an optimization expert:

- Understanding the P2 pipeline
- Instruction timing knowledge
- **REP** and **SKIP** for zero-overhead loops
- FIFO for maximum throughput
- Parallel operation techniques
- Real-world optimization strategies

Coming Up Next

Chapters 13-15 provide quick examples of Video Generation, Serial Protocols, and Signal Processing - with references to dedicated manuals for deep dives. Think of them as appetizers showing what's possible!

Have Fun! Remember, the best optimization is often a better algorithm. But when you need every last cycle, you now know how to get them!

Chapter 13: LUT Memory - Your Private Lookup Table

512 longs of fast, deterministic storage in every COG

The Hook: A Lookup Table in 3 Cycles

Need fast data lookup without hub timing? Every COG has its own private 512-long Lookup RAM (LUT):

```
' Sine table lookup - 3 clocks, every time
get_sine
        AND     angle, #$FF      ' Mask to table index
        RDLUT   value, angle     ' Read from LUT in 3 clocks!
        RET
```

No hub timing to worry about. No waiting for the egg beater. Just 3 clock cycles, guaranteed. The LUT is like having a personal data assistant that never takes a coffee break.

Why Another Memory?

You might be thinking, "Wait, I already have COG RAM and Hub RAM - why do I need a third memory?" Excellent question!

Memory	Size per COG	Access Time	Special Features
COG RAM	512 longs	2 clocks	Instructions live here
Hub RAM	512 KB shared	2-9 clocks (hub slot wait)	Shared by all COGs
LUT RAM	512 longs	**3 clocks**	**Private, deterministic, shareable with neighbor**

The LUT fills a sweet spot: faster than hub memory, doesn't compete with your instruction space, and has a trick up its sleeve - neighboring COGs can share LUTs!

Reading and Writing the LUT

Basic LUT Access

```
' Write to LUT
        WRLUT   ##$12345678, #100 ' Write 32-bit constant to LUT[100]
        WRLUT   value, index      ' Write variable to LUT[index]

' Read from LUT
        RDLUT   result, #100      ' Read LUT[100] into result
        RDLUT   data, index       ' Read LUT[index] into data
```

Notice the operand order: **WRLUT** writes its first operand to the address in the second, while **RDLUT** reads from its second operand into the first. A bit backwards from what you might expect, but you'll get used to it.

Building a Lookup Table

Here's how to load a sine table into LUT:

```
' Copy 256-entry sine table from hub to LUT
load_sine_table
        MOV     index, #0
        LOC     ptra, #\sine_data_hub   ' Hub address of table

.loop   RDLONG  value, ptra++    ' Read from hub
        WRLUT   value, index     ' Write to LUT
        ADD     index, #1
        CMP     index, #256 wz
  if_nz JMP     #.loop
        RET

' Now lookups are fast!
get_sine
        RDLUT   sine_value, angle  ' 3 clocks!
        RET
```

Sidetrack

Bulk LUT Loading with SETQ2

For loading entire tables, **SETQ2** + **RDLONG** transfers hub data into LUT memory. The trick: when **SETQ2** is active, the **RDLONG** destination field (still 9-bit, 0–$1FF) is interpreted as a LUT offset that maps physically to $200–$3FF:

```
        SETQ2   #256-1              ' 256 longs to LUT
        RDLONG  $000, hub_table_ptr ' Dest field 0 = physical LUT $200
```

The destination operand is the LUT offset, not the absolute address — so you write `$000` (LUT base), not `$200` (which would overflow the 9-bit field). Remember the `-1` in **SETQ2** (same rule as **SETQ** for hub block transfers).

LUT Sharing Between COGs

Here's something clever: adjacent COG pairs can share LUT data! When you enable LUT sharing with **SETLUTS**, writes your neighbor makes to their LUT are automatically *copied* to your LUT too.

```
' --- COG 1 (consumer) - MUST enable sharing FIRST ---
        SETLUTS #1              ' Enable LUT write copying FROM COG 0
        ' Now when COG 0 writes to its LUT, data is COPIED to our LUT

' --- COG 0 (producer) - writes AFTER consumer enables sharing ---
        WRLUT   message, #10    ' Write MY LUT[10] (copies to COG 1)
        WRLUT   #1, #0          ' Set ready flag (copies to COG 1)

' --- COG 1 (consumer) - reads its OWN LUT (which contains copies) ---
.wait   RDLUT   flag, #0        ' Read MY LUT[0] (contains copy from COG 0)
        CMP     flag, #1 wz
  if_nz JMP     #.wait
        RDLUT   message, #10    ' Read MY LUT[10] (copied from COG 0)
```

The key instruction is:

- **SETLUTS**: Enable write copying - when neighbor writes with **WRLUT**, data is copied to YOUR LUT
- **RDLUT**: Read your own LUT (which now contains copied data)

Important: The consumer COG must enable **SETLUTS** *before* the producer writes, otherwise the writes won't be copied!

This gives you a 512-long shared buffer between COG pairs without touching hub memory. Perfect for high-bandwidth data passing!

Sidetrack

Which COGs Are Neighbors?
The LUT sharing pairs are fixed:

- COG 0 COG 1
- COG 2 COG 3
- COG 4 COG 5
- COG 6 COG 7

When you enable sharing, your odd/even companion COG's LUT *writes* are copied into your own LUT — so you read the copies from your own LUT. Sharing works only within the fixed pair; non-adjacent COGs cannot share.

Practical Examples

Enough theory — let's see what people actually use the LUT for in real code:

Fast Data Transformation

```
' Gamma correction table in LUT
' Input: 8-bit value in 'pixel'
' Output: Gamma-corrected value
gamma_correct
        AND     pixel, #$FF     ' Mask to 8 bits
        RDLUT   pixel, pixel    ' Transform via table
        RET

' Initialize gamma table (power law curve)
' Would be pre-calculated and loaded from hub
```

Circular Buffer in LUT

```
' Fast circular buffer using LUT
' 256-entry buffer at LUT addresses 0-255

buf_write_ptr   LONG    0
buf_read_ptr    LONG    0

put_byte
        WRLUT   data, buf_write_ptr
        ADD     buf_write_ptr, #1
        AND     buf_write_ptr, #$FF   ' Wrap at 256
        RET

get_byte
        RDLUT   data, buf_read_ptr
        ADD     buf_read_ptr, #1
        AND     buf_read_ptr, #$FF    ' Wrap at 256
        RET
```

Fast Stack in LUT

```
' Stack implementation in LUT
' Grows downward from $1FF
stack_ptr       LONG    $1FF

PUSH
        WRLUT   value, stack_ptr
        SUB     stack_ptr, #1
        RET

POP
        ADD     stack_ptr, #1
        RDLUT   value, stack_ptr
        RET
```

LUT with the Streamer

Here's where LUT gets really interesting. The Streamer can read directly from LUT to generate waveforms without any COG intervention:

```
' Fill LUT with waveform data
' Then let Streamer output it to DAC

load_waveform
        MOV     index, #0

.fill   MOV     value, index
        SHL     value, #24       ' Scale for DAC
        WRLUT   value, index
        ADD     index, #1
        CMP     index, #512 wz
  if_nz JMP     #.fill

' Now configure Streamer to read from LUT
' Streamer handles the rest - no COG cycles needed!
```

The Streamer configuration for LUT reading is covered in detail in the Video and Audio manuals - but the key point is that your LUT becomes a 512-sample waveform buffer that plays automatically.

Common Gotchas

WRONG: Confusing LUT addresses

```
' WRONG - This reads COG RAM, not LUT!
        MOV     value, $200     ' $200 is COG RAM address
```

RIGHT: Use RDLUT for LUT access

```
' RIGHT - RDLUT addresses the LUT space
        RDLUT   value, #0       ' LUT address 0
```

WRONG: Reading LUT before neighbor writes

```
' WRONG - No data to read yet!
        SETLUTS #1              ' Enable sharing
        RDLUT   data, #10       ' Empty - neighbor hasn't written!
```

RIGHT: Wait for neighbor's write signal

```
' RIGHT - Wait for data to be copied
        SETLUTS #1              ' Enable sharing BEFORE neighbor writes
.wait   RDLUT   ready, #0       ' Check flag in MY LUT
        TJZ     ready, #.wait   ' Wait until neighbor writes
        RDLUT   data, #10       ' Now MY LUT has copied data
```

Medicine Cabinet

The Medicine Cabinet

LUT Memory Quick Reference

Instruction	Operation	Cycles
RDLUT D, S	Read LUTS into D	3
WRLUT D, S	Write D to LUTS	2
SETLUTS D	Enable LUT write copying (D[0]=1)	2

Memory Map:

- LUT addresses: 0-511 (512 longs = 2KB)
- Neighbor pairs: 0 1, 2 3, 4 5, 6 7

Best Uses:

- Lookup tables (sine, gamma, encoding)
- Fast circular buffers
- COG-pair data sharing
- Streamer waveform source

Your Turn

Your Turn

Exercise 1: Build an 8-bit Encoder
Create a LUT-based ASCII to 7-segment display encoder. Load a 128-entry table where each entry maps an ASCII code to the 7-segment pattern for that character.

continues on next page →

↪ *continued from previous page*

```
' Your code here:
' 1. Load segment patterns into LUT
' 2. Write encode_char routine
' Hint: rdlut segment_pattern, ascii_char
```

Your Turn

Exercise 2: High-Speed COG Communication

Use LUT sharing to create a message passing system between COG 2 and COG 3:

- COG 2 writes 8-long messages
- COG 3 reads them without hub access
- Use a simple ready/ack protocol

```
' Hint: Use LUT[0] as ready flag, LUT[1-8] as message buffer
```

What We've Learned

The LUT in your toolbox:

- 512 longs of fast, private memory in every COG
- 3-clock deterministic access via **RDLUT** / **WRLUT**
- Bulk loading via **SETQ2** + **RDLONG**
- COG-pair LUT sharing for high-bandwidth data passing
- Streamer source for waveform generation

Coming Up Next

Chapter 14 hands you the keys to 64 autonomous I/O processors — Smart Pins. We'll cover the universal configuration pattern and the modes you'll reach for most often, then point you at the dedicated Smart Pins Manual for the deep dive.

Have Fun! And remember — a 3-clock private lookup table is a luxury most chips don't give you. Use it!

Chapter 14: Smart Pins Orientation

64 autonomous I/O processors waiting to do your bidding

The Hook: A UART in 4 Lines

Remember that tedious bit-bang serial from Chapter 8? Watch this:

```
' Configure pin as UART transmitter - done!
        DIRL    #TX_PIN                 ' Reset pin first!
        WRPIN   ##P_ASYNC_TX, #TX_PIN   ' Configure as async TX
        WXPIN   ##BAUD_115200, #TX_PIN  ' Set baud rate
        DIRH    #TX_PIN                 ' Enable - runs on its own
```

That's it. The pin is now a fully autonomous UART transmitter. It handles start bits, stop bits, timing - everything. You just feed it bytes with **WYPIN** and it sends them. The pin has become a state machine.

And here's the mind-bending part: *every single one of the 64 pins can do this.* Or PWM. Or ADC. Or quadrature decoding. Or 28 other modes.

What Are Smart Pins, Really?

Each of the P2's 64 I/O pins contains its own little processor - a state machine that can operate completely independently of the COGs. This means:

- A pin configured as UART keeps sending/receiving without COG intervention
- A PWM output keeps running its duty cycle automatically
- An ADC samples continuously in the background
- A quadrature decoder tracks position even while your COG does other things

The COG only needs to configure the pin and occasionally read/write data. The pin does the rest.

The Universal Smart Pin Pattern

Every Smart Pin follows the same configuration pattern. This is **the most important thing to remember**:

```
' === THE SMART PIN RECIPE ===

```

continues on next page →

↪ *continued from previous page*

```
' Step 1: RESET the pin (CRITICAL!)
        DIRL    pin             ' Always start by resetting

' Step 2: CONFIGURE the mode
        WRPIN   mode, pin       ' What should this pin do?

' Step 3: SET parameters
        WXPIN   x_value, pin    ' Mode-specific parameter X
        WYPIN   y_value, pin    ' Mode-specific parameter Y

' Step 4: ENABLE the pin
        DIRH    pin             ' Start the magic!
```

Sidetrack

Why DIRL First?

The **DIRL** at the start isn't optional politeness - it's *required*. Smart Pins must be reset before configuration to ensure they're in a known state. **SKIP** this and you'll get unpredictable behavior as old settings conflict with new **ONES**.

Think of it like power-cycling a misbehaving device. Always start fresh.

The Core Instructions

Configuration Instructions

Instruction	Purpose
WRPIN mode, pin	Set the operating mode
WXPIN value, pin	Set X parameter (mode-specific)
WYPIN value, pin	Set Y parameter (mode-specific)
DIRH pin	Enable the Smart Pin
DIRL pin	Disable/reset the Smart Pin

Data Instructions

Instruction	Purpose
WYPIN data, pin	Write data to Smart Pin (same instruction!)
RDPIN data, pin	Read result, clear "ready" flag
RQPIN data, pin	Read result, keep "ready" flag
AKPIN pin	Acknowledge (clear "ready" flag only)

Status Instructions

Instruction	Purpose
TESTP pin WC	Check if IN flag is set (data ready)
TESTPN pin WC	Check if IN flag is clear

Understanding the IN Flag

Every Smart Pin has an IN flag that signals "something happened." What that something is depends on the mode:

- **UART TX**: IN high = ready for another **BYTE**
- **UART RX**: IN high = **BYTE** received
- **ADC**: IN high = new sample ready
- **PWM**: IN high = period complete
- **Counter**: IN high = threshold reached

You check this flag with **TESTP** and clear it by reading with **RDPIN** (or explicitly with **AKPIN**).

```
1 ' Wait for Smart Pin to be ready
2 wait_ready
3         TESTP   #PIN wc         ' Check IN flag
4   if_nc JMP     #wait_ready     ' Loop if not ready
5         RDPIN   data, #PIN      ' Read and clear flag
```

Sidetrack

Event-Driven Alternative

Instead of polling with **TESTP**, you can use the event system:

```
SETSE1  #%001<<6 + PIN    ' Event when IN rises
WAITSE1                   ' Sleep until ready - no polling!
RDPIN   result, #PIN      ' Read the result
```

This is more efficient because your COG sleeps instead of spinning. See Chapter 15 for the full event story.

Common Smart Pin Modes

Here are the modes you'll use most often:

Asynchronous Serial (UART)

```
' Transmit mode
        DIRL    #TX_PIN
        WRPIN   ##P_ASYNC_TX | P_OE, #TX_PIN
        WXPIN   ##(clkfreq/baud)<<16 | 7, #TX_PIN  ' Baud + 8 bits
        DIRH    #TX_PIN

' Send a byte
send    TESTP   #TX_PIN wc      ' Wait for ready
  if_nc JMP     #send
        WYPIN   BYTE, #TX_PIN   ' Send it
```

```
' Receive mode
        DIRL    #RX_PIN
        WRPIN   ##P_ASYNC_RX, #RX_PIN
        WXPIN   ##(clkfreq/baud)<<16 | 7, #RX_PIN
        DIRH    #RX_PIN

' Get a byte
recv    TESTP   #RX_PIN wc      ' Check for received byte
  if_nc JMP     #recv
```

continues on next page →

↪ continued from previous page

```
        RDPIN   BYTE, #RX_PIN   ' Get it
        SHR     BYTE, #24       ' Shift to low byte
```

PWM Output

```
' PWM mode - period + duty cycle
        DIRL    #PWM_PIN
        WRPIN   ##P_PWM_SAWTOOTH | P_OE, #PWM_PIN
        WXPIN   ##period, #PWM_PIN      ' Period in clocks
        WYPIN   ##duty, #PWM_PIN        ' High time in clocks
        DIRH    #PWM_PIN

' Change duty cycle on the fly
        WYPIN   ##new_duty, #PWM_PIN    ' Just update Y parameter
```

ADC Input

```
' ADC mode - continuous sampling
        DIRL    #ADC_PIN
        WRPIN   ##P_ADC | P_ADC_GIO, #ADC_PIN
        WXPIN   ##13, #ADC_PIN          ' 14-bit mode (period = 2^13 clocks)
        DIRH    #ADC_PIN

' Read ADC value
read_adc
        RDPIN   adc_value, #ADC_PIN     ' Get N-bit ADC count (LSB-aligned)
```

Quadrature Encoder

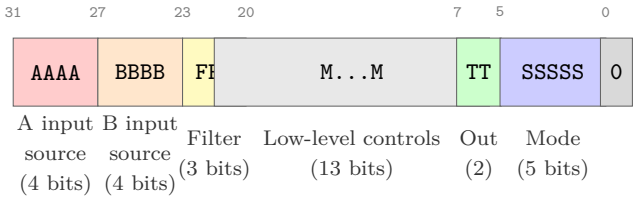
```
' Quadrature decoder - A on pin, B on pin+1
        DIRL    #ENC_PIN
        WRPIN   ##P_QUADRATURE, #ENC_PIN
        DIRH    #ENC_PIN

' Read position
        RDPIN   position, #ENC_PIN      ' Get accumulated count
```

Configuration Values Demystified

Don't worry, you don't have to memorize all 32 mode bit-patterns. The mode values like `P_ASYNC_TX` are constants defined by the assembler. But here's what's happening behind the scenes, in case you're curious:

The **WRPIN** D value is a 32-bit configuration:

WRPIN Configuration (32 bits):



For most common modes, you'll use predefined constants like `P_ASYNC_TX`, `P_PWM_SAWTOOTH`, `P_ADC`. The P2 assembler knows all of them.

Common Gotchas

WRONG: Forgetting to reset before configure

```
' WRONG - Pin may be in unknown state!
        WRPIN   ##P_PWM_SAWTOOTH, #PIN
        WXPIN   ##1000, #PIN
        DIRH    #PIN
```

RIGHT: Always DIRL first

```
' RIGHT - Start clean
        DIRL    #PIN                    ' Reset first!
        WRPIN   ##P_PWM_SAWTOOTH, #PIN
        WXPIN   ##1000, #PIN
        DIRH    #PIN
```

WRONG: Enabling before configuring

```
' WRONG - Pin enabled with partial config!
        DIRL    #PIN
        DIRH    #PIN                    ' Enabled too early!
        WRPIN   ##P_ASYNC_TX, #PIN
```

RIGHT: DIRH comes last

```
' RIGHT - Configure completely, then enable
        DIRL    #PIN
        WRPIN   ##P_ASYNC_TX, #PIN
        WXPIN   ##BAUD, #PIN
        DIRH    #PIN                    ' Enable last!
```

Medicine Cabinet

The Medicine Cabinet

Smart Pin Quick Reference

The Recipe:

1. **DIRL** pin — Reset the pin first
2. **WRPIN** mode, pin — Set the operating mode
3. **WXPIN** x, pin — Set X parameter
4. **WYPIN** y, pin — Set Y parameter
5. **DIRH** pin — Enable the Smart Pin

Common Modes:

- **UART TX**: `P_ASYNC_TX` — Serial transmit
- **UART RX**: `P_ASYNC_RX` — Serial receive
- **PWM**: `P_PWM_SAWTOOTH` — Sawtooth wave output
- **PWM**: `P_PWM_TRIANGLE` — Triangle wave output
- **ADC**: `P_ADC` — Analog input
- **Quadrature**: `P_QUADRATURE` — Encoder
- **NCO**: `P_NCO_FREQ` — Frequency output

Data Flow:

- **WYPIN** = Write data TO Smart Pin
- **RDPIN** = Read data FROM Smart Pin (clears IN)
- **TESTP** = Check if IN flag set

Golden Rule: DIRL before **WRPIN**, **DIRH** after **WXPIN**/WYPIN

Your Turn

Your Turn

Exercise 1: PWM LED Dimmer

Create a PWM output that dims an LED:

1. Configure a pin for PWM sawtooth mode
2. Set a 1 kHz period (at 160 MHz: period = 160,000)
3. Vary duty cycle from 0% to 100%

```
' Your code here:
' Hint: Change duty with WYPIN new_duty, #LED_PIN
```

Your Turn

Exercise 2: Simple Serial Echo

Set up UART at 115200 baud:

1. Configure RX on pin 63
2. Configure TX on pin 62
3. Echo every received **BYTE** back

```
' Your code here:
' At 160 MHz: baud_divisor = 160_000_000 / 115200 = 1389
' WXPIN format: (divisor << 16) | (bits - 1)
```

Going Deeper: This chapter covered the Smart Pin essentials - the configuration pattern and common modes. For complete coverage of all 32 modes, timing diagrams, and advanced techniques, see the dedicated "P2 Smart Pins Manual."

What We've Learned

Smart Pin essentials:

- Every pin contains its own state machine (32 modes available)
- The universal recipe: **DIRL** → **WRPIN** → **WXPIN** → **WYPIN** → **DIRH**
- The IN flag signals "something happened" (mode-specific)
- UART, PWM, ADC, quadrature — same configuration pattern
- Smart Pins free the COG for other work

Coming Up Next

Chapter 15 explores the event system — how to stop polling and start waiting, so your COG sleeps until something interesting happens. The companion to Smart Pins: when one tells the other a byte is ready, you want to be notified, not spinning.

Have Fun! And remember — every Smart Pin you configure is a coprocessor you don't have to babysit. That's leverage!

Chapter 15: Event-Driven Programming

Stop spinning, start waiting

The Hook: No More Polling Loops

Remember all those busy loops waiting for things to happen?

```
' OLD WAY: Spin waiting for serial data (burns CPU cycles!)
wait_rx TESTP   #RX_PIN wc      ' Check over and over
  if_nc JMP     #wait_rx        ' Spin spin spin...
        RDPIN   data, #RX_PIN

' NEW WAY: Sleep until data arrives (zero CPU cycles!)
        SETSE1  #%001<<6 + RX_PIN  ' Wake on IN rise
        WAITSE1                     ' Sleep until event
        RDPIN   data, #RX_PIN
```

The event system lets your COG sleep while waiting. When the event happens, it wakes up instantly. No cycles wasted, and you respond the moment something happens.

Why Events Matter

Polling loops have two problems:

1. **They waste cycles** - The COG spins doing nothing useful
2. **They add latency** - You check periodically, so there's delay between "thing happened" and "you noticed"

The event system solves both. Your COG *sleeps* and *wakes the instant* something happens. It's like having a personal assistant tap your shoulder instead of constantly looking up to check.

The Four Selectable Events

Let's meet the cast. Every COG has four configurable event channels: SE1, SE2, SE3, and SE4. Each can be configured to trigger on different conditions:

Event	Configuration	Wait	Poll
SE1	**SETSE1**	**WAITSE1**	**POLLSE1**
SE2	**SETSE2**	**WAITSE2**	**POLLSE2**
SE3	**SETSE3**	**WAITSE3**	**POLLSE3**
SE4	**SETSE4**	**WAITSE4**	**POLLSE4**

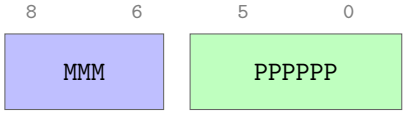
Plus there are built-in timer events:

Timer	Wait	Poll
CT1	**WAITCT1**	**POLLCT1**
CT2	**WAITCT2**	**POLLCT2**
CT3	**WAITCT3**	**POLLCT3**

Configuring an Event

The **SETSE1** through **SETSE4** instructions take a 9-bit configuration value:

SETSE1/2/3/4 Configuration (9 bits):



Event Mode (3 bits) — Pin Number (6 bits: 0–63)

%000	Disabled	%100	Pin high
%001	IN rises	%101	Pin low
%010	IN falls	%110	Pin rises
%011	IN changes	%111	Pin falls

Event Modes

Mode	Meaning
%000	Never (disabled)
%001	IN rises (Smart Pin ready)
%010	IN falls
%011	IN changes
%100	Pin is low (level)
%101	Pin is low (level)
%110	Pin is high (level)
%111	Pin is high (level)

EVENT_* Constants: When You Need Interrupts

While dedicated COGs are usually better than interrupts (see Chapter 11), sometimes you need them. The **SETINT1/2/3** instructions select which event triggers an interrupt using these constants:

Table 15.1.

Constant	Value	Description
EVENT_INT	%0000	Pin matches interrupt configuration
EVENT_CT1	%0001	CT equals CT1 (timer 1 target)
EVENT_CT2	%0010	CT equals CT2 (timer 2 target)
EVENT_CT3	%0011	CT equals CT3 (timer 3 target)
EVENT_SE1	%0100	Selectable event 1 triggered
EVENT_SE2	%0101	Selectable event 2 triggered
EVENT_SE3	%0110	Selectable event 3 triggered
EVENT_SE4	%0111	Selectable event 4 triggered
EVENT_PAT	%1000	SETPAT pattern detected
EVENT_FBW	%1001	Hub FIFO wrapped around
EVENT_XMT	%1010	Streamer needs data
EVENT_XFI	%1011	Streamer operation complete
EVENT_XRO	%1100	NCO frequency counter rolled

Continued on next page

Table 15.1. (Continued)

Constant	Value	Description
EVENT_XRL	%1101	Streamer read last LUT location ($1FF)
EVENT_ATN	%1110	Another COG signaled attention
EVENT_QMT	%1111	CORDIC/PIX math complete

Using EVENT_* with SETINT:

```
' Enable INT1 when SE1 event occurs
        SETSE1  #%001<<6 + RX_PIN       ' SE1 = IN rise on RX_PIN
        SETINT1 #EVENT_SE1              ' INT1 fires when SE1 triggers

' Enable INT2 on timer match
        ADDCT2  target, ##200_000       ' Set timer 2 target
        SETINT2 #EVENT_CT2              ' INT2 fires when CT = CT2

' Enable INT3 when another COG signals
        SETINT3 #EVENT_ATN              ' INT3 fires on COGATN
```

Pro tip: You can also use these constants with WAITSE/POLLSE by first triggering them with the appropriate hardware condition, then waiting. But for most purposes, the SETSE mode bits (the table above this one) are what you'll configure directly.

Smart Pin Events

The most common use is waiting for a Smart Pin to have data:

```
' Wait for Smart Pin on pin 15 to be ready
        SETSE1  #%001<<6 + 15   ' IN rise on pin 15
        WAITSE1                  ' Sleep until ready
        RDPIN   data, #15        ' Get the data
```

Pin Edge Events

You can also wait for raw pin edges (without Smart Pin):

```
' Wait for rising edge on pin 5
        SETSE1  #%001<<6 + 5    ' Rising edge on pin 5
        WAITSE1                  ' Sleep until edge
```

continues on next page →

↪ continued from previous page

```
        ' Edge detected!

' Wait for falling edge on pin 10
        SETSE2  #%010<<6 + 10   ' Falling edge on pin 10
        WAITSE2
        ' Edge detected!
```

Timer Events

For precise timing, use the counter comparison events:

```
' Wait exactly 1 millisecond (at 200 MHz)
        GETCT   target          ' Current time
        ADD     target, ##200_000  ' +1ms at 200MHz
        ADDCT1  target, #0      ' Set CT1 target
        WAITCT1                  ' Sleep until CT >= CT1

' Alternative using WAITX (simpler but less precise)
        WAITX   ##200_000       ' Wait ~1ms at 200MHz
```

The timer events are:

- **ADDCT1/ADDCT2/ADDCT3**: Set the comparison target
- **WAITCT1/WAITCT2/WAITCT3**: Wait until CT reaches target
- **POLLCT1/POLLCT2/POLLCT3**: Check (non-blocking) if target reached

Waiting vs Polling

Two ways to use events:

WAIT - Sleep Until Event

```
        WAITSE1                 ' COG sleeps here
        ' Wakes instantly when event occurs
```

- COG sleeps, uses no cycles
- Wakes immediately when event fires
- Can't do anything else while waiting

POLL - Check and Continue

```
        POLLSE1 wc              ' Check event, clear if set
  if_c  JMP     #event_handler  ' Handle if occurred
        ' Continue with other work...
```

- COG keeps running
- Checks event flag, clears it
- Returns result in C flag
- Good for servicing multiple events

Multiple Events

With four SE channels, you can monitor multiple sources:

```
' Setup multiple events
        SETSE1  #%001<<6 + RX_PIN     ' Serial data ready
        SETSE2  #%001<<6 + BUTTON_PIN ' Button pressed (rising edge)
        SETSE3  #%001<<6 + ADC_PIN    ' ADC sample ready

event_loop
        POLLSE1 wc              ' Check serial
  if_c  CALL    #handle_serial

        POLLSE2 wc              ' Check button
  if_c  CALL    #handle_button

        POLLSE3 wc              ' Check ADC
  if_c  CALL    #handle_adc

        JMP     #event_loop
```

Practical Examples

Time to put events to work. Three patterns you'll reach for again and again:

Timeout with Fallback

```
' Wait for serial data, but give up after 100ms
wait_with_timeout
        SETSE1  #%001<<6 + RX_PIN     ' Serial ready event

        GETCT   timeout
        ADD     timeout, ##16_000_000  ' 100ms at 160MHz
        ADDCT1  timeout, #0

.wait   POLLSE1 wc              ' Check serial
  if_c  JMP     #.got_data
        POLLCT1 wc              ' Check timeout
  if_c  JMP     #.timed_out
        JMP     #.wait

.got_data
        RDPIN   data, #RX_PIN
        RET

.timed_out
        MOV     data, #-1       ' Return error
        RET
```

Debounced Button Press

```
' Wait for clean button press with debounce
debounced_button
        SETSE1  #%001<<6 + BUTTON  ' Rising edge
        WAITSE1                     ' Wait for press

        WAITX   ##2_000_000        ' 10ms debounce at 200MHz

        TESTP   #BUTTON wc         ' Verify still pressed
  if_nc JMP     #debounced_button  ' Bounce - try again
        RET                         ' Clean press!
```

Precise Periodic Sampling

```
' Sample ADC at exactly 10 kHz
sample_loop
        GETCT   next_sample

.loop   ADDCT1  next_sample, ##16_000  ' 100us period
        WAITCT1                         ' Wait for next slot

        RDPIN   sample, #ADC_PIN        ' Read sample
        WRLONG  sample, buffer_ptr      ' Store it
        ADD     buffer_ptr, #4

        JMP     #.loop
```

ATN - Inter-COG Events

The ATN (attention) system lets COGs signal each other:

```
' COG 0: Signal another COG
        COGATN  #%0000_0010     ' Send ATN to COG 1

' COG 1: Wait for attention
        WAITATN                  ' Sleep until ATN received
        ' Another COG signaled us!
```

The **COGATN** instruction takes an 8-bit mask where each bit corresponds to a COG. Setting bit N sends attention to COG N.

Common Gotchas

WRONG: Forgetting to clear event flag

```
' WRONG - Event may fire before you're ready
        SETSE1  #%001<<6 + PIN
        ' ... do other stuff ...
        WAITSE1                 ' May return immediately!
```

RIGHT: Poll first to clear any pending event

```
' RIGHT - Clear any stale event
        SETSE1  #%001<<6 + PIN
        POLLSE1                 ' Clear if already set
        ' ... do other stuff ...
        WAITSE1                 ' Now wait cleanly
```

WRONG: Using WAIT when you need to handle multiple sources

```
' WRONG - Can only wait for one event at a time
        WAITSE1                 ' Stuck here until SE1
        ' SE2 might fire and be missed!
```

RIGHT: Use POLL loop for multiple events

```
' RIGHT - Check all sources
.loop   POLLSE1 wc
  if_c  CALL    #handle_se1
        POLLSE2 wc
  if_c  CALL    #handle_se2
        JMP     #.loop
```

Medicine Cabinet

The Medicine Cabinet

Event System Quick Reference
Configure Events:

```
1          SETSE1/2/3/4  #%MMM_PPPPPP    ' Mode and pin
```

Event Modes:

%MMM	Trigger
%001	IN rises (Smart Pin ready)
%010	IN falls
%011	IN changes
%10x	Pin is low (level)
%11x	Pin is high (level)

Wait (blocking):

```
1          WAITSE1/2/3/4    ' Sleep until event
2          WAITCT1/2/3      ' Sleep until timer
3          WAITATN          ' Sleep until attention
```

Poll (non-blocking):

```
1          POLLSE1/2/3/4 WC ' Check event, clear flag, C=occurred
2          POLLCT1/2/3 WC   ' Check timer, C=reached
3          POLLATN WC       ' Check attention, C=received
```

Timer Setup:

```
1          ADDCT1/2/3 target, #delta   ' Set comparison target
```

Inter-COG:

```
1          COGATN #mask    ' Signal COGs (bit per COG)
```

Your Turn

Your Turn

Exercise 1: Event-Driven Serial

Rewrite a serial receive loop to use events instead of polling:

1. Configure SE1 for UART RX Smart Pin ready
2. Use WAITSE1 instead of **TESTP** loop
3. Measure the cycle count difference

```
' Your code here:
' Hint: setse1 #%001<<6 + RX_PIN
```

Your Turn

Exercise 2: Dual Event Monitor

Create a loop that monitors both a button (pin edge event) and a timer (periodic event):

1. SE1 = button press (rising edge)
2. CT1 = 1 second heartbeat
3. On button: toggle LED
4. On timer: print timestamp

```
' Your code here:
' Use POLL for both, handle whichever fires
```

What We've Learned

The event toolkit you now command:

- Four selectable event channels (SE1-SE4) per COG
- Three timer events (CT1-CT3) for precise scheduling
- **WAIT** (sleep) vs **POLL** (check-and-continue) tradeoffs
- The ATN inter-COG signaling system
- Common patterns: timeout-with-fallback, debounce, periodic sampling

Coming Up Next

Chapter 16 brings the whole journey together — orchestrating eight COGs in parallel harmony to build complete systems. It's where the P2 philosophy really shines.

Have Fun! And remember — every spin loop you replace with **WAITSE** is CPU cycles you've handed back to your design. Be generous with events!

Chapter 16: Multi-COG Orchestration

Bringing it all together in parallel harmony

The Hook: A Complete System in 8 COGs

Watch this system architecture come alive:

```
' Main orchestrator (COG 0)
main_orchestrator
        ' Launch the orchestra (SETQ sets PTRA for new COG)
        SETQ    @sensor_params
        COGINIT #1, @sensor_cog
        SETQ    @motor_params
        COGINIT #2, @motor_cog
        SETQ    @comms_params
        COGINIT #3, @comms_cog
        SETQ    @display_params
        COGINIT #4, @display_cog
        SETQ    @safety_params
        COGINIT #5, @safety_cog
        SETQ    @logger_params
        COGINIT #6, @logger_cog
        SETQ    @debug_params
        COGINIT #7, @debug_cog

        ' Now coordinate them all
orchestrate
        RDLONG  sensor_data, ##SENSOR_MAILBOX wz
   if_nz CALL   #process_sensor_data

        RDLONG  command, ##COMMAND_MAILBOX wz
   if_nz CALL   #execute_command

        CALL    #update_system_state
        WRLONG  state, ##STATE_MAILBOX

        JMP     #orchestrate
```

Eight independent processors, each with a specific job, all working in perfect coordination. This is the true power of P2!

Communication Patterns

Eight processors running in parallel sounds wonderful — until you realize they need to talk to each other. Let's meet the three patterns you'll use 95% of the time.

The Mailbox Pattern

The simplest and most common — a single hub **LONG** that one COG writes and another reads:

```
' Producer COG
producer
        ' Generate data
        CALL    #calculate_result
        WRLONG  result, ##MAILBOX_ADDR

' Consumer COG
consumer
        RDLONG  data, ##MAILBOX_ADDR wz
   if_z JMP     #consumer              ' Wait for data
        WRLONG  #0, ##MAILBOX_ADDR     ' Clear mailbox
        CALL    #process_data
```

The Ring Buffer Pattern

For streaming data between COGs:

```
' Writer COG
writer_cog
        RDLONG  wr_ptr, ##WRITE_PTR
        WRLONG  data, wr_ptr
        ADD     wr_ptr, #4
        AND     wr_ptr, ##BUFFER_MASK  ' Wrap around
        WRLONG  wr_ptr, ##WRITE_PTR

' Reader COG
reader_cog
        RDLONG  rd_ptr, ##READ_PTR
```

continues on next page →

↪ continued from previous page

```
        RDLONG  wr_ptr, ##WRITE_PTR
        CMP     rd_ptr, wr_ptr wz
   if_z JMP     #reader_cog            ' Buffer empty

        RDLONG  data, rd_ptr
        ADD     rd_ptr, #4
        AND     rd_ptr, ##BUFFER_MASK
        WRLONG  rd_ptr, ##READ_PTR
```

The Command Queue Pattern

For sending commands between COGs:

```
' Command structure in hub
' +0: Command ID
' +4: Parameter 1
' +8: Parameter 2
' +12: Result/Status

' Commander COG
send_command
        WRLONG  cmd_id, ##CMD_BUFFER+0
        WRLONG  param1, ##CMD_BUFFER+4
        WRLONG  param2, ##CMD_BUFFER+8
        WRLONG  ##$FFFF, ##CMD_BUFFER+12  ' Mark as pending

wait_complete
        RDLONG  status, ##CMD_BUFFER+12
        CMP     status, ##$FFFF wz
   if_z JMP     #wait_complete

' Worker COG
process_commands
        RDLONG  status, ##CMD_BUFFER+12
        CMP     status, ##$FFFF wz
  if_nz JMP     #process_commands      ' No command

        RDLONG  cmd_id, ##CMD_BUFFER+0
        RDLONG  param1, ##CMD_BUFFER+4
        RDLONG  param2, ##CMD_BUFFER+8
```

continues on next page →

↪ continued from previous page

```
28
29          CALL    #execute_command
30          WRLONG  result, ##CMD_BUFFER+12   ' Signal complete
```

Synchronization Techniques

Sometimes communication isn't enough — you need two or more COGs to *agree* on what happens when. That's where synchronization comes in.

Using Locks

When multiple COGs need atomic access to the same piece of data, P2 gives you 16 hardware locks. They're tiny and they're fast:

```
1  ' Atomic increment using lock
2  atomic_increment
3           LOCKTRY #COUNTER_LOCK wc      ' C=1 if we got the lock
4    if_nc  JMP     #atomic_increment     ' Retry until we get it
5
6           RDLONG  value, ##COUNTER
7           ADD     value, #1
8           WRLONG  value, ##COUNTER
9
10          LOCKREL #COUNTER_LOCK
```

Event Synchronization

COGs waiting for specific events:

```
1  ' COG 1: Signal event
2           WRLONG  ##EVENT_FLAG, ##EVENT_ADDR
3
4  ' COG 2: Wait for event
5  wait_event
6           RDLONG  flag, ##EVENT_ADDR wz
7     if_z  JMP     #wait_event
8           WRLONG  #0, ##EVENT_ADDR      ' Clear event
```

Real-World Example: Robot Controller

Let's build a complete robot control system:

```
' COG 0: Main Controller
main_controller
        CALL    #init_system

main_loop
        ' Read sensor hub
        RDLONG  distance, ##DISTANCE_SENSOR wz
   if_z JMP     #too_close

        ' Check for commands
        RDLONG  cmd, ##SERIAL_COMMAND wz
   if_nz CALL   #process_command

        ' Update motor speeds
        CALL    #calculate_motion
        WRLONG  left_speed, ##LEFT_MOTOR
        WRLONG  right_speed, ##RIGHT_MOTOR

        JMP     #main_loop

' COG 1: Ultrasonic Sensor
sensor_cog
        ' Trigger ultrasonic pulse
        DRVH    #TRIGGER_PIN
        WAITX   ##1000
        DRVL    #TRIGGER_PIN

        ' Measure echo time - wait for rising edge
.wait_hi
        TESTP   #ECHO_PIN wz
  if_nz JMP     #.wait_hi
        GETCT   start_time
.wait_lo                              ' Wait for falling edge
        TESTP   #ECHO_PIN wz
  if_z  JMP     #.wait_lo
        GETCT   end_time

```

continues on next page →

↪ *continued from previous page*

```
        ' Calculate distance
        SUB     end_time, start_time
        ' Convert to distance...
        WRLONG  distance, ##DISTANCE_SENSOR

        WAITX   ##10_000_000         ' 50ms at 200MHz
        JMP     #sensor_cog

' COG 2: Left Motor Driver
left_motor_cog
        RDLONG  speed, ##LEFT_MOTOR wz
   if_z JMP     #left_motor_cog      ' No speed set

        ' Generate motor control signals
        ' ... PWM generation code
        JMP     #left_motor_cog

' COG 3: Right Motor Driver
' (Similar to left motor)

' COG 4: Serial Communications
serial_cog
        ' Check for incoming commands
        TESTP   #RX_PIN wc
  if_nc JMP     #serial_cog

        CALL    #receive_byte
        ' Build command...
        WRLONG  command, ##SERIAL_COMMAND
        JMP     #serial_cog

' COG 5: LED Status Display
status_cog
        RDLONG  system_state, ##STATE_MAILBOX

        ' Display state on LEDs
        CMP     system_state, #STATE_RUNNING wz
   if_z DRVH    #GREEN_LED
  if_nz DRVL    #GREEN_LED

        CMP     system_state, #STATE_ERROR wz
```

↪ continued from previous page

```
    if_z DRVH    #RED_LED
   if_nz DRVL    #RED_LED

         WAITX   ##10_000_000
         JMP     #status_cog

' COG 6: Safety Monitor
safety_cog
         ' Monitor critical systems
         RDLONG  battery, ##BATTERY_VOLTAGE
         CMP     battery, ##MIN_VOLTAGE wcz
    if_b WRLONG  ##STATE_SHUTDOWN, ##STATE_MAILBOX

         ' Check temperature
         RDLONG  temp, ##TEMPERATURE
         CMP     temp, ##MAX_TEMP wcz
    if_a WRLONG  ##STATE_OVERHEAT, ##STATE_MAILBOX

         JMP     #safety_cog

' COG 7: Debug Output
debug_cog
         ' Output system state for debugging
         ' ... debug code
```

Eight COGs, each doing one job perfectly, creating a responsive, reliable robot!

Your Turn: Multi-COG Project

Your Turn

Exercise: Traffic Light Controller

Requirements:

- COG 0: Main sequencer
- COG 1: North-South lights
- COG 2: East-West lights

- COG 3: Pedestrian button watcher
- COG 4: Timer/scheduler

Starting structure:

continues on next page →

↪ *continued from previous page*

```
        ORG     0
' COG 0: Main sequencer
        ' Launch other COGs
        ' Coordinate light changes
        ' Handle pedestrian requests

' Your implementation here
```

Goal: Working traffic light with pedestrian crossing Hint: Use mailboxes for COG communication Success Check: Lights change correctly, pedestrian button works

The Medicine Cabinet

Multi-COG systems overwhelming? Start simple:
Start with just 2 COGs:

```
' Main + Helper pattern
main    SETQ    @params                 ' PTRA for new COG
        COGINIT #1, @helper
        ' Main work

helper  ' Support work
```

Use simple mailboxes:

```
' Fixed hub addresses for communication
MAILBOX_1 = $1000
MAILBOX_2 = $1004
```

Debug one COG at a time: TEST each COG in isolation before combining!

Design Principles for Multi-COG Systems

The hardware gives you eight processors. Whether your *design* survives the journey is up to you. A few rules of thumb we've learned the hard way:

1. **Single Responsibility**: Each COG does ONE thing well
2. **Loose Coupling**: COGs communicate through hub, not direct dependencies

3. **Clear Ownership**: Each piece of data has one writer
4. **Predictable Timing**: Real-time tasks get dedicated COGs
5. **Graceful Degradation**: System continues if one COG fails

Common Multi-COG Gotchas

Before you pull your hair out wondering why the eight-COG dream turned into a debugging nightmare, skim these:

1. **Race conditions** - Use locks for shared write access
2. **Deadlocks** - Avoid circular dependencies
3. **Starvation** - Ensure all COGs get resources
4. **Communication overhead** - Don't over-communicate
5. **Debugging complexity** - Use LED indicators for each COG

What We've Learned

You've mastered multi-COG orchestration:

- Communication patterns (mailbox, ring buffer, queue)
- Synchronization techniques
- Real-world system architecture
- Design principles
- Common pitfalls and solutions

This is it - you now understand the full power of the Propeller 2!

Your Journey Continues

You've completed this manual, but your P2 journey has just begun:

1. **Build something amazing** - Put your knowledge to work
2. **Share with the community** - Your projects inspire others
3. **Explore other manuals** - Smart Pins, Video, I/O await
4. **Push boundaries** - P2 can do things we haven't imagined yet

Chapter Summary

Chapter Summary

Congratulations! You've mastered multi-COG orchestration!
You now understand:

- How to coordinate 8 parallel processors
- Communication patterns between COGs
- Synchronization techniques
- Real-world system design

You did it! You're now fluent in PASM2 and ready to build incredible parallel systems!

Have Fun!

Remember what you've learned:

- Eight COGs working together are more powerful than any interrupt-driven system
- Parallel processing isn't harder, it's different
- The P2 way is about determinism and elegance
- Every complex system is just simple parts working together

Now go forth and create something amazing with your Propeller 2!

Epilogue: The Journey Forward

Well, here we are at the end… or should I say, at the beginning?

You've traveled from blinking your first LED to orchestrating eight parallel processors. You've mastered CORDIC mathematics, tamed the FIFO, and learned why interrupts are usually the wrong answer. That's quite a journey!

But here's the secret: everything you've learned is just the foundation. The P2 community continues to discover new techniques, new optimizations, new ways to use this remarkable chip. Every project pushes the boundaries a little further.

What Makes You Different Now

You're not just another embedded programmer anymore. You think in parallel. You see solutions that others miss. When someone says "that's impossible in real-time," you know better - you just dedicate a COG to it.

The Community Awaits

The Parallax forums are filled with fellow travelers on this journey. Share your projects. Ask questions. Help newcomers. The community that inspired this manual continues to grow because people like you contribute back.

One Last Story

I remember my first P2 project. I was trying to control 16 servos with perfect timing while reading sensors and communicating over serial. On my previous microcontroller, it was a nightmare of interrupts and jitter.

On the P2? Three COGs. Clean, simple, perfect timing. That's when I truly understood - this isn't just a different processor, it's a different philosophy of computing.

Your Challenge

Build something that wouldn't be possible without parallel processing. Something that would be a nightmare of interrupts on other processors. Then share it with the world.

Show them what eight COGs can do.

Show them the Propeller way.

"The best way to predict the future is to invent it."
— Alan Kay

And with your Propeller 2, you have everything you need to invent amazing futures.

Have Fun!

— The ghosts of deSilva, the P2 community, and one very enthusiastic AI

P.S. - Don't forget to blink an LED once in a while, just for old times' sake. It's still magical, even after all you've learned.

THE END

(But really, just the beginning…)

Further Reading

This teaching manual focuses on concepts, patterns, and building your understanding. For complete technical specifications, refer to these companion documents:

Propeller 2 Assembly Language (PASM2) Manual Complete PASM2 instruction details including syntax, timing, and flag effects for all 300+ instructions. Quick lookup reference for day-to-day development.

Parallax Propeller 2 Documentation ***(v35, Rev B/C silicon, 2021-05-18)*** Official silicon documentation from Parallax covering hardware specifications, electrical characteristics, and detailed register maps.

Appendix A: Platform Comparison

How P2 compares to other microcontrollers

If you're coming from another embedded platform, this appendix shows how the P2's approach differs and when those differences matter.

The Landscape

The embedded world is dominated by a handful of architectures:

Platform	Architecture	Typical Cores	Peripherals	Timing
STM32	ARM Cortex-M	1-2	Fixed location	Cache-dependent
ESP32	Xtensa/RISC-V	2	Fixed location	FreeRTOS scheduled
Arduino/AVR	AVR	1	Fixed location	Deterministic but slow
PIC32	MIPS	1	Fixed location	Interrupt-driven
P2 Propeller	Custom	**8**	**Any pin**	**Deterministic**

What Makes P2 Different

Eight Real Processors, Not Time-Slicing

On ARM, ESP32, or PIC, you typically have 1-2 cores that share time between tasks using interrupts or an RTOS. The P2 gives you eight complete, identical processors that run truly in parallel.

Traditional approach:

```
' Everyone fights for the same CPU
ISR(TIMER1_vect) { motor_control(); }   ' Might delay...
ISR(UART_RX_vect) { serial_handler(); } ' ...this
main() { while(1) { sensor_loop(); } }  ' Hope we get time
```

P2 approach:

```
' Each task owns its own processor
COG0: COGINIT(1, @motor_control)   ' Coordinator launches workers
COG1: motor_control()              ' Dedicated - always on time
COG2: serial_handler()             ' Dedicated - never misses byte
COG3: sensor_loop()                ' Dedicated - consistent sample
COG4-7: ready for more
```

No interrupt priority juggling. No RTOS configuration. Each task owns its processor.

Smart Pins: Peripherals on Every Pin

Traditional MCUs have fixed peripheral assignments: UART1 is on PA9/PA10, SPI1 is on PB3/PB4/PB5, and if you need those pins for something else, you're stuck rerouting your PCB.

On P2, every pin contains a programmable state machine. Any pin can become a UART, SPI, PWM, ADC, quadrature decoder, or 27 other modes. The peripheral comes to your pin, not the other way around.

Deterministic Timing

ARM MCUs with cache have unpredictable timing. A memory read might take 1 cycle (cache hit) or 50+ cycles (cache miss). Even instruction timing varies—ARM instructions take 1-3+ cycles depending on the operation. This makes cycle-accurate timing extremely difficult.

P2 takes a different approach: nearly all instructions execute in exactly **2 clock cycles**. Want to know how long a code sequence takes? Count the instructions and multiply by 2. Hub memory uses round-robin access that gives every COG predictable, guaranteed access slots. Your timing loops work identically every time—no cache luck required.

Coming From ARM/STM32

You're used to configuring HAL structures, writing interrupt handlers, and managing DMA. Here's how P2 solves those problems:

Instead of...	On P2...	The Benefit
`HAL_UART_Transmit()`	Configure Smart Pin once, then **WYPIN** bytes	Pin handles all timing autonomously
`HAL_TIM_PWM_Start()`	Configure Smart Pin once, update with **WYPIN**	Pin runs independently—your COG is free
NVIC priority configuration	Nothing needed	All COGs equal, no priority inversion ever
`HAL_DMA_Start()`	Use built-in FIFO/Streamer	Simpler API, integrated into each COG
`arm_sin_f32()` library	**QROTATE** instruction	Hardware trig in exactly 55 clocks
FreeRTOS `xTaskCreate()`	**COGINIT**	True parallel execution, not scheduled

The result: Deterministic timing, zero interrupt conflicts, and I/O configuration that just works.

Coming From ESP32

You're used to WiFi/Bluetooth convenience and FreeRTOS abstractions. P2 takes a different approach:

ESP32 Way	P2 Way	The Benefit
Built-in WiFi/BT	Add WizNet or ESP module	You choose your connectivity—or skip it entirely
`xTaskCreate()`	**COGINIT**	Not scheduled—truly parallel, guaranteed timing
GPIO matrix routing	Smart Pins	32 modes per pin, far more capability
FreeRTOS timing	Deterministic hub	Cycle-accurate timing guaranteed
Arduino framework	Spin2/PASM2	Deeper control, deeper understanding

The result: 8 real cores running simultaneously, timing you can count on, I/O flexibility that eliminates peripheral conflicts.

Coming From Arduino/AVR

You'll find P2 familiar but dramatically more powerful:

Arduino Way	P2 Way	The Upgrade
`digitalWrite()`	**DRVH/DRVL** or Smart Pins	Similar syntax, vastly more capability
`delay()` blocks everything	**WAITX** or dedicated COG	Timing without blocking other tasks
One thing at a time	8 things truly parallel	Real concurrency, not fake multitasking
8-bit math limits	32-bit + hardware CORDIC	No more overflow worries, hardware trig
Libraries for everything	Growing ecosystem + OBEX	More control, deeper understanding

The result: Graduate from 8-bit limitations to 8 parallel 32-bit processors with hardware math and Smart Pins on every I/O.

When P2 Is the Right Choice

P2 excels when you need:

- **Multiple real-time tasks** running simultaneously without conflicts
- **Precise timing** that cache misses and interrupts can't disrupt
- **Video or audio generation** requiring cycle-accurate output
- **Flexible I/O** where any pin can become any peripheral
- **Hardware math** for motor control, signal processing, or robotics
- **Multiple motor/servo control** with dedicated COGs per channel
- **Protocol implementation** where Smart Pins handle timing autonomously

Platform Trade-offs

Every platform makes trade-offs. P2 optimizes for **determinism, parallelism, and flexibility** rather than:

If you need...	P2's answer
Built-in WiFi/Bluetooth	Add WizNet or ESP module—you choose connectivity
Massive library ecosystem	Growing OBEX + helpful community
Ultra-low-power sleep	External modules or different platform
Lowest unit cost at 100K+ volumes	P2 targets flexibility over commodity pricing

The honest reality: If your project is "connect to WiFi and display data," an ESP32 does that with less effort. But if your ESP32 project is fighting timing jitter, missing deadlines, or running out of peripheral pins—that's exactly what P2 solves.

Community Resources

While P2's ecosystem is smaller than ARM or Arduino, it's active and welcoming:

Parallax Forums - The heart of the P2 community. Chip Gracey (P2's designer) participates actively, answering questions and discussing design decisions. You'll find help from experienced developers who've solved problems you haven't encountered yet.

P2 Object Exchange (OBEX) - A library of reusable Spin2 and PASM2 objects covering drivers, protocols, display interfaces, and more. Before writing something from scratch, check OBEX—someone may have already done the work.

Community Support - Unlike large platforms where your question disappears in a sea of posts, the P2 community is small enough that questions get noticed and answered. Many community members have decades of Propeller experience.

Coming from Arduino's library-for-everything culture, you'll write more code yourself—but you'll understand it deeply, and help is always available when you get stuck.

The P2 Hardware Ecosystem

P2 isn't just a chip - it's a platform with expansion options:

Video & Audio:

- A/V Breakout Board: VGA, RCA, 80mW stereo, microphone input
- Digital Video Out: HDMI-type with differential signaling
- Built-in 8-bit DAC per pin (16-bit with dithering)

Connectivity:

- USB Host Board: Two USB-A ports
- USB Device Board: HID or CDC modes
- Serial Host/Device: RS-232 interfaces
- WizNet/ESP modules: Ethernet or WiFi

Development:

- P2 Eval Board: Complete development environment
- Edge Modules: 4MB or 32MB flash for embedding
- Breakout Boards: All 64 pins accessible

You add what you need - no paying for peripherals you won't use.

Summary

P2 represents a fundamentally different approach to embedded computing—one that eliminates entire categories of problems:

- **Eight processors** means your motor control never delays your serial handler
- **64 Smart Pins** means peripheral conflicts become impossible
- **Deterministic timing** means your code works the same way every time
- **Hardware CORDIC** means real-time math without floating-point libraries

Engineers who've fought interrupt priority inversions, missed timing deadlines, and PCB rework due to peripheral conflicts find P2 refreshing. You spend your time solving your actual problem, not fighting your MCU.

Welcome to the P2 community. You've got 8 processors, 64 Smart Pins, and a community that's been building amazing things since the original Propeller. Time to see what you can build.

Index

F

G

H

I

J

L

M

T

U

V

W

Z