P2 Documentation and Code Project

You are viewing draft content created with the use of Claude.AI

P2 Streamer Programming Guide

Comprehensive Reference for Propeller 2 Streamer Hardware

June 2026

Version 1.0.1

Guide Organization

High-Speed I/O, Video, and Signal Processing with the P2 Streamer

Part I: Streamer Fundamentals

- Understanding the Streamer
- Architecture
- NCO and Timing
- Command Structure

Part II: Mode Reference

- Immediate Modes
- RDFAST Modes
- RGB Video Modes
- WRFAST Input Modes
- ADC Sampling Modes
- DDS/Goertzel Mode

Part III: Configuration Reference

- DAC Channel Configuration
- Pin Selection and Control
- Programming Constants
- Events and Synchronization

Part IV: Applications

- Video Output (VGA, HDMI, Composite)
- High-Speed Serial (SPI)
- Signal Processing
- Integration Patterns

Part V: Appendices

- Complete Mode Encoding Table
- Symbol Quick Reference
- Frequency Calculation Tables
- Troubleshooting Guide
- Index

Iron Sheep Productions, LLC

P2 Knowledge Base Project

Contents

Copyright and License **7**

Acknowledgments . . . 7

Sources . . . 7

How to Use This Guide . . . 8

Document Conventions . . . 8

Enhancement Markers . . . 8

Part I: Streamer Fundamentals **9**

Chapter 1: Understanding the Streamer **9**

1.1 What the streamer is . . . 9

1.2 Why the streamer exists . . . 10

1.3 A mental model: a paced, configurable pipe . . . 10

1.4 Two directions: output and input . . . 11

1.5 Why there are so many modes — and why they differ . . . 11

1.6 The special capabilities, in plain words . . . 11

1.7 If you're building… . . . 12

1.8 What each COG actually has . . . 12

Chapter 2: Architecture **14**

2.1 Block Diagram . . . 14

2.2 Data Flow Paths . . . 15

2.3 Component Functions . . . 15

Chapter 3: NCO and Timing **17**

3.1 NCO Operation . . . 17

3.2 Frequency Calculation . . . 17

3.3 Setting NCO Frequency . . . 18

3.4 Choosing a Pixel Rate . . . 19

3.5 Clock Accuracy and Jitter . . . 20

Chapter 4: Command Structure **22**

4.1 Command Word Format . . . 22

4.2 Mode Field D[31:28] . . . 22

4.3 DAC Routing Field D[27:24] . . . 23

4.4 Enable/Write Field D[23] . . . 23
4.5 Pin Group Field D[22:20] . . . 23
4.6 Count Field D[15:0] . . . 23
4.7 Streamer Instructions . . . 24

Part II: Mode Reference **25**

Chapter 5: Immediate Modes **25**
5.1 Immediate → LUT → Pins/DACs . . . 25
5.2 Immediate → Pins/DACs . . . 26

Chapter 6: RDFAST Modes **27**
6.1 RDFAST → LUT → Pins/DACs . . . 27
6.2 RDFAST → Pins/DACs . . . 28

Chapter 7: RGB Video Modes **29**
7.1 RGB Format Modes . . . 29
7.2 Color Format Details . . . 29
7.3 RGB Mode Example . . . 30

Chapter 8: WRFAST Input Modes **31**
8.1 Pin Capture Modes . . . 31

Chapter 9: ADC Sampling Modes **33**
9.1 ADC Capture Modes . . . 33
9.2 ADC Configuration Example . . . 33

Chapter 10: DDS/Goertzel Mode **34**
10.1 Mode Variants . . . 34
10.2 Operation . . . 34
10.3 LUT Setup . . . 35
10.4 SINC1 vs SINC2 . . . 35
10.5 Reading Results . . . 36
10.6 Frequency Calculation . . . 36

Part III: Configuration Reference **37**

Chapter 11: DAC Channel Configuration **37**
11.1 DAC Routing Table . . . 38
11.2 DAC Pin Mapping . . . 38
11.3 Common DAC Configurations . . . 39

Chapter 12: Pin Selection and Control **40**
12.1 Pin Group Selection . . . 40
12.2 Sub-Pin Selection . . . 41

12.3 Enable Control . . . 41
12.4 Alternate Bit Order . . . 42

Chapter 13: Programming Constants **43**
13.1 Mode Symbols . . . 43
13.2 Control Symbols . . . 44
13.3 DAC Symbols . . . 45
13.4 Symbol Composition . . . 45

Chapter 14: Events and Synchronization **46**
14.1 Streamer Events . . . 46
14.2 Event Instructions . . . 46
14.3 Event Clearing . . . 47
14.4 Synchronization Patterns . . . 47

Part IV: Applications **49**

Chapter 15: Video Output **49**
15.1 VGA Output . . . 49
15.2 HDMI/DVI Output . . . 52
15.3 Composite Video . . . 52

Chapter 16: High-Speed Serial (SPI) **54**
16.1 SPI Output with Streamer . . . 54
16.2 Coordinating with WAITXFI . . . 55

Chapter 17: Signal Processing **56**
17.1 Goertzel Frequency Detection . . . 56
17.2 DDS Waveform Generation . . . 57

Chapter 18: Integration Patterns **60**
18.1 Double Buffering . . . 60
18.2 Multi-COG Video . . . 60
18.3 Streamer + Smart Pin Coordination . . . 61

Part V: Appendices **62**

Appendix A: Complete Mode Encoding Table **62**

Appendix B: Symbol Quick Reference **65**
Mode Symbols . . . 65
Control Symbols . . . 65
DAC Symbols . . . 65

Appendix C: Frequency Calculation Tables **66**

NCO Frequency Values . . . 66
Common Video Pixel Rates . . . 66

Appendix D: Troubleshooting Guide **67**
Symptom: No Output on Pins . . . 67
Symptom: Corrupted Data from RDFAST . . . 67
Symptom: Streamer Stops Unexpectedly . . . 67
Symptom: Phase Drift in Video . . . 67
Symptom: DAC Output Incorrect . . . 67
Symptom: Goertzel Results Invalid . . . 68

Index **69**

List of Figures

2.1 Streamer data path 14
2.2 Streamer data-flow paths 15

3.1 NCO phase accumulation and rollover 17

4.1 Command word: D operand of XINIT / XCONT / XZERO 22

7.1 RGB source-format bit packing 29

10.1 DDS output with Goertzel accumulation (every clock) 35

15.1 VGA 640×480 horizontal line timing 50

Copyright and License

Copyright © 2026 Iron Sheep Productions, LLC and Parallax Inc.

This work is licensed under the Creative Commons Attribution–NonCommercial–NoDerivatives 4.0 International License (CC BY-NC-ND 4.0).

You are free to:

- **Share** — copy and redistribute the material in any medium or format

Under the following terms:

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made (for example, formatting or excerpting).
- **NonCommercial** — You may not use the material for commercial purposes.
- **NoDerivatives** — If you remix, transform, translate, or build upon the material, you may not distribute the modified material.

Commercial use: For uses that may be commercial (including paid courses, kits, or redistribution with products), please contact Iron Sheep Productions, LLC and Parallax Inc. (info@ironsheep.biz) for separate permission.

To view the full license, visit: https://creativecommons.org/licenses/by-nc-nd/4.0/

Trademarks

Parallax, Propeller, Spin, and the Parallax logo are trademarks of Parallax Inc.

Acknowledgments

This guide would not exist without the contributions of many individuals and organizations:

Parallax Inc. for creating the Propeller 2 microcontroller and providing the comprehensive reference documentation that forms the foundation of this work.

Chip Gracey for the design of the P2 streamer, NCO, and colorspace converter, and for maintaining the detailed silicon documentation that defines their behavior.

The P2 Community for the video, audio, and signal-processing drivers whose real-world usage informed the examples and patterns in this guide.

Sources

This guide draws on the following primary and community sources:

- **P2 Silicon Documentation v35** (Chip Gracey, Parallax Inc.) — streamer architecture, mode encodings, NCO and DDS/Goertzel behavior

- **Spin2 Documentation v51** (Parallax Inc.) — built-in streamer symbols and language integration
- **P2 Flash Loader source** (official P2 ROM) — verified instruction usage
- **Community video and Goertzel drivers** (Parallax OBEX) — application patterns

How to Use This Guide

This reference serves developers implementing high-speed I/O on the Propeller 2. It assumes familiarity with P2 COG/Hub architecture, basic PASM2 instructions, and **RDFAST**/WRFAST FIFO operations.

Structure:

- **Part I (Fundamentals)** establishes the mental model — read this first to understand how the streamer operates
- **Part II (Mode Reference)** documents all streamer modes — use for specific mode lookup
- **Part III (Configuration)** covers cross-cutting concerns — DAC routing, pin selection, symbols
- **Part IV (Applications)** provides implementation patterns — video, SPI, signal processing
- **Appendices** contain quick-reference tables and troubleshooting guidance

Document Conventions

Element	Format	Example
Instructions	Bold uppercase	**XINIT**, **XCONT**
Symbols	Monospace	`X_RFWORD_RGB16`
Bit fields	Brackets	D[31:28], S[19:16]
Binary	Percent + underscores	`%1011_0000`
Hexadecimal	Dollar prefix	`$B085_0000`

Enhancement Markers

Three colored callout boxes set "things to know" apart from the running text:

- **CAUTION** (amber) — common mistakes with non-obvious consequences
- **TIP** (teal) — non-obvious techniques or optimizations
- **HARDWARE** (graphite) — silicon-level details affecting usage

Part I: Streamer Fundamentals

The streamer rewards a little grounding before the encodings. This Part builds the mental model — what the streamer is, how its NCO sets the pace, and the command vocabulary you will use everywhere else — so the mode tables in Part II read as deliberate choices rather than magic.

Chapter 1: Understanding the Streamer

Before the mode tables and bit fields, it helps to know what the streamer *is*, why the P2 has one, and how to think about it when you sit down to build something. This chapter builds that picture. The rest of the guide is the detailed reference; this chapter is the map.

1.1 What the streamer is

Every COG on the P2 has its own **streamer**: a small, tireless engine that moves data between hub memory and the outside world — the pins, the DAC channels, the ADC inputs — entirely on its own, at a rate you choose. Once you start it, it runs without the COG's help. Your code can compute, make decisions, or sleep while the streamer keeps feeding pixels to a display or pulling samples off a wire.

The detail that makes the streamer special is that it carries **its own clock — and you set its rate**. A piece of hardware called the NCO (Numerically-Controlled Oscillator) is the streamer's adjustable metronome: it ticks at whatever rate your application needs, and the streamer moves one piece of data on each tick. You dial that rate in directly — a ~25 MHz pixel rate for VGA, a 48 kHz sample rate for audio, or anything else — and it stays rock-steady and exact. That precise, self-kept timing is what lets a single COG produce a clean video picture or an unwavering audio stream without ever having to babysit the timing in software.

And because the streamer lives *inside* the COG, **each COG has its own streamer and its own NCO** — so the eight streamers are clocked independently. They do not share a rate. One COG can push video pixels at 25 MHz while another streams audio at 48 kHz and a third samples an ADC at some other rate entirely, all at the same moment, each running at exactly the rate its job requires.

> **If you've used DMA before:** the streamer is a close cousin of a DMA channel, with two important additions. First, it has that built-in metronome, so it does *paced*

transfers at an exact sample rate rather than "as fast as the bus allows." Second, it reshapes data as it moves — packing bits, expanding through a palette, converting color formats — instead of copying bytes verbatim. If you have never met DMA, don't worry: everything below stands on its own.

1.2 Why the streamer exists

Generating precise, fast, repetitive signals in software is brutally expensive. Imagine driving a video display by hand: your code would have to write a new color to the pins every forty nanoseconds, forever, without ever slipping. A single loop like that would consume an entire COG and still stutter the moment anything interrupted it. The streamer exists so that this relentless, timed, repetitive work happens in hardware, leaving the COG free for the *interesting* part — drawing the next frame, decoding the next packet, running the game.

So why is it so complicated? Because one engine has to serve very different jobs: pushing video to a screen, playing audio through a DAC, capturing pins like a logic analyzer, sampling an analog input like an oscilloscope, generating tones, even detecting a specific frequency in an incoming signal. Rather than give each COG six narrow peripherals, the P2 gives it **one highly configurable engine**. The configurability is the complexity — and it is also the payoff. Learn the handful of knobs once, and the same engine does all of those jobs.

1.3 A mental model: a paced, configurable pipe

Picture a pipe running from hub memory to the pins, with an adjustable shaper in the middle and a metronome setting the pace. Every streamer command you issue is just a way of answering four questions about that pipe:

1. **Where does the data come from?** An immediate value you supply, or a stream pulled from hub memory through the FIFO.
2. **Where does it go?** Out to the pins, out to the DAC channels — or, when capturing, back into hub memory.
3. **How is it shaped along the way?** Sliced into 1-, 2-, 4-, 8-, 16-, or 32-bit pieces; expanded through a lookup table (a palette); or converted into a video color format.
4. **How fast?** The NCO's beat.

That is the whole idea. The long list of modes in Part II is nothing more than the *useful combinations* of those four answers. Once you read a mode as "hub data, to the pins, one byte at a time, at the pixel rate," the names stop looking like alphabet soup.

1.4 Two directions: output and input

The streamer works in two directions, and they are different enough that it helps to think about them separately.

Output modes drive the world — pixels to a screen, samples to a speaker, bits down a serial line. Here the interesting questions are *where the data comes from* (a buffer in memory, or a small repeating table) and *how it is shaped* (raw bytes, palette lookup, RGB color) before it reaches the pins.

Input modes capture the world — pin states recorded into memory (a logic analyzer); ADC readings recorded into memory (an oscilloscope). Here the interesting question is *how the captured data is packed* as it is written back to hub.

Knowing which direction you are working in immediately cuts the mode list roughly in half — you only ever care about one side at a time.

1.5 Why there are so many modes — and why they differ

It is tempting to see the mode tables as a random pile of similar-looking options. They are not. They vary along just a few axes, and each combination earns its place by doing a *distinct* job. Two quick contrasts make the point:

- **Playing a recording vs. synthesizing a tone.** Both end at the DAC. But playing a recorded sound streams a long buffer out of hub memory, while synthesizing a tone loops a *tiny* table — one cycle of a sine wave — over and over, with the NCO setting the pitch. Same destination, completely different source strategy. That difference is exactly what separates the two modes.
- **One channel vs. multichannel audio.** These are not different engines at all — they are one routing knob (the DAC-routing field) deciding how many of the four DAC channels get fed. "Stereo" is a setting, not a separate mode.

So when two mode names sound alike, the question to ask is: *which of the four pipe-questions do they answer differently?* That is always where the real distinction lives.

1.6 The special capabilities, in plain words

A few streamer features have intimidating names. Here is what they actually mean.

Video (RGB and colorspace conversion). The streamer can pull pixels from a framebuffer in memory and push them out as an analog VGA signal or a digital HDMI signal, translating color formats as it goes. You provide the picture; the streamer handles the relentless pixel timing. *(Chapters 7 and 15.)*

DDS — Direct Digital Synthesis. A grand name for a simple trick: generate a repeating waveform — a sine, a tone, any shape you like — by stepping through a small table at a precise

rate. The NCO sets the frequency, so the same table produces any pitch. This is how you build a function generator, an audio tone, or a modulated carrier. *(Chapters 10 and 17.)*

Goertzel frequency analysis. This one is never obvious from its name, so plainly: it is a way to ask, in hardware, *"how much of one specific frequency is present in this incoming signal?"* You tell it the frequency you care about, and it reports how strongly that tone is there. The textbook use is decoding telephone touch-tones — the beeps of a phone keypad, known as DTMF — but the same trick detects whistles and alarm tones, measures distance with ultrasound (send a ping, listen for its echo), and builds simple receivers. It is cheap precisely because it checks only the *one* frequency you ask about, instead of computing a whole spectrum. *(Chapters 10 and 17.)*

1.7 If you're building...

You usually arrive at the streamer with an application already in mind. Find it below; the chapters on the right are the ones worth reading closely first. The rest of the reference is there for when you need the exact bits.

If you're thinking about...	The streamer gives you...	Start at
Video (VGA, HDMI, composite)	color pixels from a framebuffer	Ch. 7, Ch. 15
Audio / sound output	DAC samples from a buffer	Ch. 5–6, Ch. 11
Generating tones or waveforms	DDS from a small table	Ch. 10, Ch. 17
A logic analyzer (capturing pins)	pin states written to memory	Ch. 8
Sampling an analog signal	ADC readings written to memory	Ch. 9
Detecting a specific tone or frequency	Goertzel analysis	Ch. 10, Ch. 17
High-speed serial (fast SPI)	timed bit output, clocked by a smart pin	Ch. 16

1.8 What each COG actually has

For all that capability, the hardware budget is modest. Each COG contains exactly one streamer, with:

- one 32-bit NCO (the metronome / phase accumulator);
- one command buffer (it holds one queued command, so commands can hand off without a gap);
- four 8-bit DAC channels (X0, X1, X2, X3);
- one Goertzel analyzer;
- access to the COG's LUT RAM (used as a palette or a waveform table).

And it leans on a few neighboring P2 subsystems:

Subsystem	What it provides
Hub FIFO	the data source / sink, via RDFAST / WRFAST
LUT RAM	palette lookups, sine/cosine tables
DAC channels	analog output for video and audio
Colorspace converter	HDMI encoding and composite video
Smart pins	clock generation and timing synchronization

With that picture in place, Chapter 2 opens up the engine itself — the data paths inside the streamer and how the pieces connect.

Chapter 2: Architecture

Chapter 1 described the streamer as a paced pipe from memory to the pins. This chapter opens that pipe up — the pieces inside, how data flows through them, and how the NCO drives the whole thing. You do not need this depth to *use* the streamer, but it makes the mode choices in Part II feel inevitable rather than arbitrary.

2.1 Block Diagram

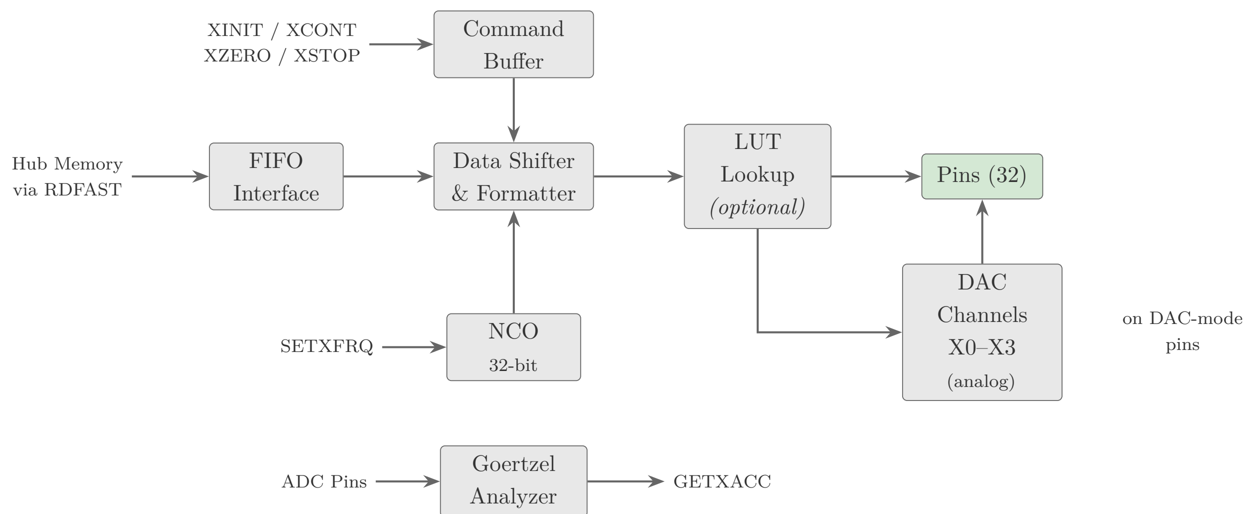

Figure 2.1. Streamer data path

What the diagram does not show is *which physical pins* DAC0–DAC3 — and the 32 output pins — actually land on, and you will care about that the moment you wire something up. The mapping is not arbitrary: each DAC channel can only drive pins whose number ends in its own two bits — DAC0 drives pins ending in `%00` (pins 0, 4, 8, …), DAC1 those ending in `%01`, and so on. The complete channel-to-pin mapping is in Chapter 11, and choosing which 32-pin group a command targets is Chapter 12. (Setting a pin up to *act* as an analog/DAC output is a pin-configuration topic in its own right — see the *P2 I/O & Smart Pins User Guide*.) For now, just note that the diagram's "DAC0–DAC3" and "Pins" become specific pin numbers once you choose them.

2.2 Data Flow Paths

The streamer supports multiple data flow configurations:

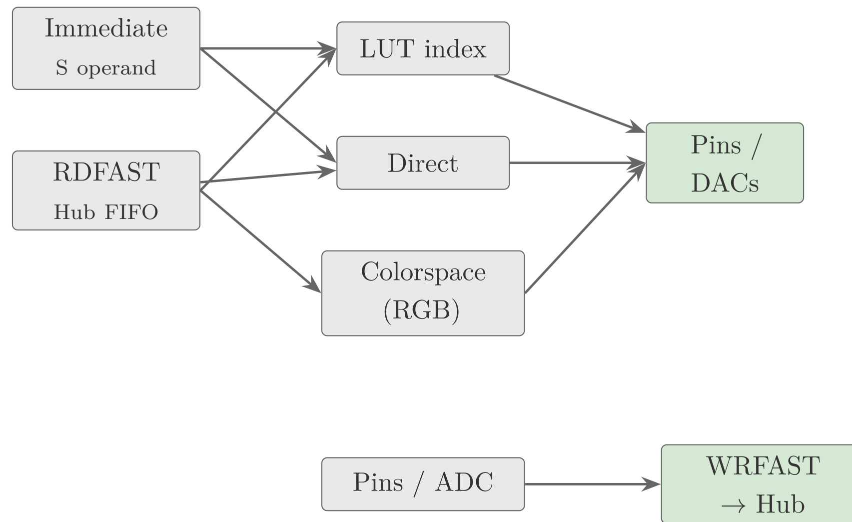

Figure 2.2. Streamer data-flow paths

Output Paths (Hub → Pins/DACs):

1. **Immediate → LUT → Pins/DACs**: S operand indexes LUT; LUT data drives output
2. **Immediate → Pins/DACs**: S operand drives output directly
3. **RDFAST → LUT → Pins/DACs**: Hub data indexes LUT; LUT data drives output
4. **RDFAST → Pins/DACs**: Hub data drives output directly
5. **RDFAST → RGB → Pins/DACs**: Hub data passes through colorspace converter

Input Paths (Pins → Hub):

1. **Pins → DACs/WRFAST**: Pin states written to Hub
2. **ADC → DACs/WRFAST**: ADC readings written to Hub

Special Path:

1. **DDS/Goertzel**: LUT-based synthesis with simultaneous frequency analysis

2.3 Component Functions

NCO (Numerically-Controlled Oscillator): Times all streamer operations. On each clock, adds frequency value to phase accumulator. Rollover (MSB set) triggers data advancement.

Command Buffer: Holds one pending command. Enables seamless transitions between streamer operations without gaps.

Data Shifter: Handles data widths from 1-bit to 32-bit. Extracts and formats data according to mode.

LUT Interface: Reads COG LUT RAM for palette expansion or waveform generation. 512 entries $\times$ 32 bits.

DAC Channels: Four 8-bit channels (X0-X3) map to pins based on pin number LSBs. Configurable routing allows stereo, differential, or independent operation.

Goertzel Analyzer: Hardware frequency detection using Goertzel algorithm. Accumulates sine and cosine products for magnitude/phase extraction.

Chapter 3: NCO and Timing

The NCO is the streamer's metronome, and it is the single most important thing to understand about the streamer's timing. Everything the streamer does happens on the NCO's beat, so setting its rate correctly is the difference between a steady picture and a rolling one. This chapter shows how the NCO produces that beat and how to compute the value you need for a given rate.

3.1 NCO Operation

The NCO operates on every system clock:

```
phase = (phase & $7FFF_FFFF) + frequency
```

1. The MSB is masked (cleared) before addition
2. The frequency value adds to the phase accumulator
3. If the new MSB is set, a "rollover" occurs
4. On rollover, the streamer advances to the next data element

HARDWARE

The phase accumulator is a 32-bit register. Its most-significant bit is masked off before each addition and used as the rollover flag, so 31 bits accumulate the phase. Frequency resolution is therefore `clock_frequency / 2³¹`.

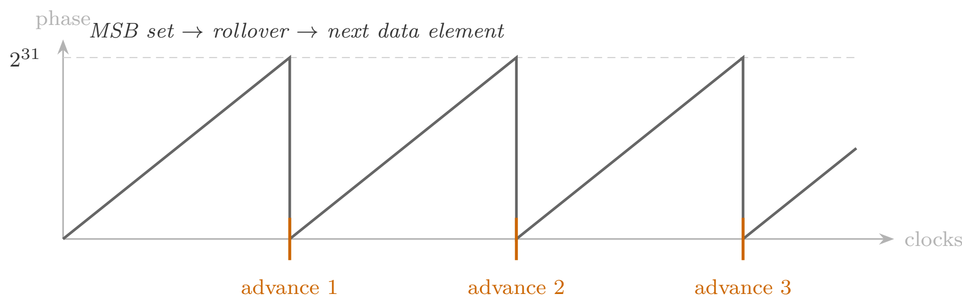

Figure 3.1. NCO phase accumulation and rollover

3.2 Frequency Calculation

Formula:

```
frequency = $8000_0000 × (desired_rate / clock_frequency)
```

Common Values:

Rate Ratio	Frequency Word	Exact?
1:1	`$8000_0000`	exact
1:2	`$4000_0000`	exact
1:4	`$2000_0000`	exact
1:8	`$1000_0000`	exact
1:3	`$2AAA_AAAB`	rounded (+1)
1:10	`$0CCC_CCCD`	rounded (+1)

A **power-of-two ratio** divides `$8000_0000` evenly, so its word is exact; every other ratio must be rounded up (the +1 convention below). This exactness is also what eliminates per-pixel jitter — see §3.4 for choosing a rate around it. Appendix C lists the full set of ratio and pixel-rate values.

CAUTION

Round up, and never let the value reach zero. Truncating `$8000_0000 × rate/clock` leaves the frequency word a hair short of a clean rollover, so the streamer's *first* rollover lands one clock late and the timing is skewed from there. Round the result up instead — or simply add 1 to a truncated value. This is the Silicon Doc's **+1 convention**: its HDMI example sets the 1/10 rate as `$0CCC_CCCC + 1`, because *"the +1 forces initial NCO rollover on the 10th clock."* The same habit guards against a second, nastier failure — a frequency word of **zero never rolls over at all, so the streamer stalls forever**. When a calculation could land low (or on zero), round up. The common-values table above already includes the +1 where the exact ratios need it.

3.3 Setting NCO Frequency

Method 1: SETXFRQ instruction

```
        ' 1/10 clock = 25 MHz; the +1 forces the first rollover
        SETXFRQ ##$0CCC_CCCC+1
```

The `+1` is the rounding-up habit from the pitfall above: `$0CCC_CCCC` is the truncated 1/10 value, and adding 1 makes the streamer roll over on the 10th clock instead of the 11th (and keeps the word off zero). You will see this `+1` throughout the examples.

Method 2: SETQ before streamer command

```
1         SETQ    ##$0CCC_CCCC+1        ' Frequency in Q
2         XINIT   mode, data          ' Uses Q as frequency
```

The **SETQ** method allows changing frequency atomically with a new command.

3.4 Choosing a Pixel Rate

Two facts decide this, and the first is counter-intuitive:

1. **The average pixel clock is essentially exact at any sysclk.** The NCO word is 31-bit, so its resolution is `sysclk / 2`^{31} 0.12 Hz at 250 MHz. The error in the *average* output rate is under ~0.01 ppm — far below any monitor's tolerance. Frequency accuracy is **not** what you tune.
2. **Per-pixel jitter is what varies.** Each pixel lasts a whole number of sysclk cycles. When `sysclk ÷ pixel_clock` is an integer, every pixel is identical — **no jitter**. When it is not, pixel widths swing by ±1 sysclk cycle around the ideal (the average still comes out exact). So the rule is: **pick a sysclk that is an integer multiple of the pixel clock.**

The jitter-free sysclks below are the integer multiples the P2 PLL can actually produce from a 20 MHz crystal. The last column is the penalty for ignoring the rule and just running 250 MHz.

Mode	Pixel clock	Jitter-free sysclks (× pixel clock)	At 250 MHz
VGA 640×480	25.175 MHz	none on a 20 MHz crystal — see note	use 25.0 MHz pixel → 10.000 cyc/px, no jitter
480p 720×480	27.0 MHz	270 (×10), 297 (×11), 324 (×12)	9.26 cyc/px → ~11% jitter
SVGA 800×600	40.0 MHz	160 / 200 / 240 / 280 / 320 (×4–×8)	6.25 cyc/px → ~16% jitter
XGA 1024×768	65.0 MHz	130 / 195 / 260 / 325 (×2–×5)	3.85 cyc/px → ~26% jitter
720p 1280×720	74.25 MHz	148.5 (×2), 297 (×4)	3.37 cyc/px → ~30% jitter

> **VGA note:** 25.175 MHz's exact multiples (201.4, 251.75 MHz) cannot be produced by the P2 PLL from a 20 MHz crystal. Standard practice is a **25.0 MHz pixel clock at 250 MHz sysclk** — exactly 10 cycles per pixel (jitter-free), with the clock 0.7% slow, which monitors absorb. DVI/HDMI tops out near this rate; 1080p needs a 1.485 GHz serial clock and is out of the streamer's reach.

The **SETXFRQ** word for any combination is `round($8000_0000 × pixel_clock / sysclk)` — the lookup tables in Appendix C list common values. Worked both ways:

```
Example 1 - integer ratio: SVGA 800×600, 40 MHz pixel @ 320 MHz
  WORD     = round($8000_0000 × 40 / 320) = $8000_0000/8 = $1000_0000
  achieved = 320 × $1000_0000 / $8000_0000 = 40.000 MHz   (exact)
  cyc/px   = 320 / 40 = 8.000   -> no jitter

Example 2 - non-integer ratio: VGA 640×480, 25.175 MHz @ 250 MHz
  WORD     = round($8000_0000 × 25.175 / 250) = $0CE3_BCD3
  achieved = 250 × $0CE3_BCD3 / $8000_0000 = 25.175 MHz   (<0.01 ppm)
  cyc/px   = 250 / 25.175 = 9.93   -> ±1-cycle jitter (~10% of pixel)
  remedy   = 25.0 MHz @ 250 = 10.000 cyc/px   -> no jitter
```

TIP

The achieved pixel clock is essentially exact at **any** sysclk — what you manage is per-pixel jitter, not frequency error. Make `sysclk ÷ pixel_clock` a whole number and every pixel is the same width.

3.5 Clock Accuracy and Jitter

Accuracy and jitter are independent. The jitter in §3.4 comes from the sysclk-to-pixel *ratio* and is the same no matter how precise your oscillator is. **Absolute accuracy** — how close the pixel clock sits to its exact nominal frequency, and how steadily it holds — is set entirely by the reference crystal: the PLL only multiplies it (`sysclk = crystal × M / (D×P)`) and the NCO adds under 0.01 ppm, so your pixel clock is **no more accurate than your crystal**.

If you build on a Parallax **P2 Edge module**, this is already handled for you.

HARDWARE

The P2 Edge modules carry a 20 MHz TCXO rated ±0.5 ppm (temperature-compensated). Every clock the P2 derives is referenced to it, so your pixel clocks, sample rates, and color subcarriers come out accurate to ~0.5 ppm and hold that across temperature — with no effort on your part. The module guide calls it "higher precision than most applications require," which makes the Edge a safe default for accurate-timing work as well as general projects.

On an **externally designed board** the P2 runs from whatever crystal you fit, and the PLL can only multiply that reference — it cannot make the clock more accurate than its source. A general-purpose crystal is typically tens of ppm, with additional drift over temperature, and that error flows straight through to every streamer rate.

For most video this is a non-issue: monitors absorb thousands of ppm of pixel-clock error — the same tolerance that makes §3.4's 25.0-for-25.175 substitution invisible. It bites only when the *absolute* frequency is the deliverable: NTSC/PAL composite **colorburst** (a few ppm before the hue drifts), precise audio sample rates, or long-running timing. **So if you are designing a custom board for video — composite especially — choose the crystal with this in mind, or fit a TCXO; an Edge module gives you that precision out of the box.**

Chapter 4: Command Structure

A streamer command is a single value — the D operand — that packs together every choice from Chapter 1's four questions: what mode, where the data goes, which pins, and how long to run. This chapter lays that packed word out field by field, then introduces the small set of instructions (XINIT, **XCONT**, **XZERO**) that start and chain commands.

4.1 Command Word Format

The D operand to **XINIT**, **XCONT**, and **XZERO** contains:

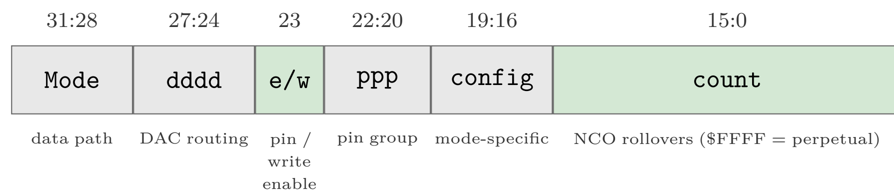

Figure 4.1. Command word: D operand of XINIT / XCONT / XZERO

4.2 Mode Field D[31:28]

D[31:28]	Category	Data Source	Data Destination
`%0000-%0011`	IMM→LUT	S operand	LUT → Pins/DACs
`%0100-%0111`	IMM→Direct	S operand	Pins/DACs
`%0111`	RF→LUT	RDFAST	LUT → Pins/DACs
`%1000-%1011`	RF→Direct	RDFAST	Pins/DACs
`%1011`	RF→RGB	RDFAST	Colorspace → Pins/DACs
`%1100-%1111`	Capture	Pins	DACs/WRFAST
`%1111`	ADC	ADC	DACs/WRFAST
`%1111_x111`	DDS/Goertzel	LUT	DACs + Analysis

Note: Every row except the last shows only the 4-bit **mode nibble** D[31:28]. DDS/-Goertzel is the exception: it is mode `%1111` (D[31:28]) *combined with* config field D[19:16] = `%x111`, so it is written here as `%1111_x111` to distinguish it from the other `%1111` rows. Within that config field, bit D[23] selects SINC1 (`0`) or SINC2 (`1`).

4.3 DAC Routing Field D[27:24]

The %dddd field controls DAC channel assignment. See Chapter 11 for the complete routing table.

4.4 Enable/Write Field D[23]

Mode Type	D[23] = 0	D[23] = 1
Output modes	Pins disabled	Pins enabled
Input modes	WRFAST disabled	WRFAST enabled

4.5 Pin Group Field D[22:20]

Selects which 32-pin block the streamer addresses:

%ppp	Pin Range
`%000`	Pins 31..0
`%001`	Pins 39..8
`%010`	Pins 47..16
`%011`	Pins 55..24
`%100`	Pins 63..32
`%101`	Pins 7..0, 63..40 (wrap)
`%110`	Pins 15..0, 63..48 (wrap)
`%111`	Pins 23..0, 63..56 (wrap)

4.6 Count Field D[15:0]

Specifies the number of NCO rollovers before the command completes.

- A count of 1 to 65,534 transfers that many data elements, then the command completes
- A count of `$FFFF` (65,535) streams **perpetually** — the command runs until a new command is issued or **XSTOP** stops it
- A count of 0 stops the streamer (this is exactly what **XSTOP** / `XINIT #0,#0` does)

4.7 Streamer Instructions

Instruction	Syntax	Effect
SETXFRQ	`SETXFRQ {#}D`	Set NCO frequency
XINIT	`XINIT {#}D,{#}S`	Start immediately, zero phase
XCONT	`XCONT {#}D,{#}S`	Buffer command, continue phase
XZERO	`XZERO {#}D,{#}S`	Buffer command, zero phase
XSTOP	`XSTOP`	Stop immediately (alias: `XINIT #0,#0`)
GETXACC	`GETXACC D`	Get Goertzel accumulators

XINIT starts the streamer immediately, interrupting any current operation and zeroing the NCO phase.

XCONT and **XZERO** buffer a command that executes on the final rollover of the current command. **XCONT** preserves NCO phase; **XZERO** resets it.

TIP

Use **XZERO** at video line boundaries to prevent phase drift accumulation across lines.

CAUTION

XCONT and XZERO are for seamless command-to-command continuity, not for starting the streamer. They wait for the current command's final NCO rollover; if the streamer is already idle (count = 0) there is no wait and the command runs immediately — and **XCONT** begins with whatever phase remains in the accumulator rather than a known zero. Use **XINIT** to start the streamer from a clean, phase-zeroed state.

Part II: Mode Reference

The streamer's modes are the heart of this reference. This Part documents each family in turn — immediate, hub-streamed, video, pin-capture, ADC, and the special DDS/Goertzel mode. Each chapter opens with what its modes are *for* before giving the exact encodings.

Chapter 5: Immediate Modes

Immediate modes are the simplest place to start. Instead of streaming from memory, the data you want to output is a value you hand the streamer directly, in the S operand. Reach for them when you have a small, fixed pattern to emit — a handful of pixels, a test pattern, a short bit sequence — and do not want to set up a hub buffer. The data can go straight to the pins and DACs, or pass through the LUT for palette expansion.

5.1 Immediate → LUT → Pins/DACs

The S operand provides index values into the LUT. LUT data drives pins and DACs.

Mode	Symbol	Elements	Bits/Element
`%0000`	`X_IMM_32X1_LUT`	32	1
`%0001`	`X_IMM_16X2_LUT`	16	2
`%0010`	`X_IMM_8X4_LUT`	8	4
`%0011`	`X_IMM_4X8_LUT`	4	8

D[19:16] Field: LUT base address bits [8:5] (`%bbbb` → LUT address `%bbbb_00000`)

S Operand: 32-bit immediate data containing packed index values

Operation: On each NCO rollover, the next index value selects a LUT entry. The LUT long drives all 32 pins and/or DAC channels.

Example:

```
' Output 32 1-bit pixels using 2-entry palette at LUT $000
        XINIT   ##X_IMM_32X1_LUT | X_PINS_ON + 32, ##$AAAA_5555
```

5.2 Immediate → Pins/DACs

The S operand drives pins and DACs directly without LUT lookup.

Mode	Symbol	Pins	DAC Channels	DAC Bits
%0100	X_IMM_32X1_1DAC1	1	1	1
%0101	X_IMM_16X2_2DAC1	2	2	1
%0101	X_IMM_16X2_1DAC2	2	1	2
%0110	X_IMM_8X4_4DAC1	4	4	1
%0110	X_IMM_8X4_2DAC2	4	2	2
%0110	X_IMM_8X4_1DAC4	4	1	4
%0110	X_IMM_4X8_4DAC2	8	4	2
%0110	X_IMM_4X8_2DAC4	8	2	4
%0110	X_IMM_4X8_1DAC8	8	1	8
%0110	X_IMM_2X16_4DAC4	16	4	4
%0111	X_IMM_2X16_2DAC8	16	2	8
%0111	X_IMM_1X32_4DAC8	32	4	8

D[19:16] Field: Mode variant selector

S Operand: Packed data values

Example:

```
1 ' Output 4 bytes to an 8-pin group, 8 bits each
2         XINIT   ##X_IMM_4X8_1DAC8 | X_PINS_ON + pin<<17 + 4, ##$12345678
```

Chapter 6: RDFAST Modes

RDFAST modes are the workhorse of the streamer. Where immediate modes carry a single fixed value, these stream a continuous flow of data out of hub memory — a framebuffer, an audio clip, a bitmap — onto the pins or DACs. This is what you use for anything longer than a few elements. The data arrives through the FIFO, which must be primed with **RDFAST** before the streamer command runs.

CAUTION

Run RDFAST before any RDFAST streamer command. It primes the cog's hub FIFO to *deliver* data (hub → streamer); until it does, the FIFO is not pointed at your buffer and the streamer pulls undefined data. The same FIFO handles the opposite direction via **WRFAST** (Chapter 8), so a cog streams one way at a time.

6.1 RDFAST → LUT → Pins/DACs

Hub data serves as LUT index values.

Mode	Symbol	Hub Read	Elements	Bits/Element
`%0111_001a`	`X_RFLONG_32X1_LUT`	RFLONG	32	1
`%0111_010a`	`X_RFLONG_16X2_LUT`	RFLONG	16	2
`%0111_011a`	`X_RFLONG_8X4_LUT`	RFLONG	8	4
`%0111_1000`	`X_RFLONG_4X8_LUT`	RFLONG	4	8

S[3:0]: LUT base address bits [8:5]

%a bit: Alternate bit order (0 = LSB first, 1 = MSB first)

Example:

```
' Setup FIFO
        RDFAST  #0, ##bitmap_addr

' Stream 640 pixels through 256-color palette at LUT $000
        XINIT   ##X_RFLONG_4X8_LUT | X_PINS_ON + base<<17 + 640, #0
```

6.2 RDFAST → Pins/DACs

Hub data drives pins and DACs directly.

Mode	Symbol	Hub Read	Pins	DAC Channels	DAC Bits
%1000	X_RFBYTE_1P_1DAC1	RFBYTE	1	1	1
%1001	X_RFBYTE_2P_2DAC1	RFBYTE	2	2	1
%1001	X_RFBYTE_2P_1DAC2	RFBYTE	2	1	2
%1010	X_RFBYTE_4P_4DAC1	RFBYTE	4	4	1
%1010	X_RFBYTE_4P_2DAC2	RFBYTE	4	2	2
%1010	X_RFBYTE_4P_1DAC4	RFBYTE	4	1	4
%1010	X_RFBYTE_8P_4DAC2	RFBYTE	8	4	2
%1010	X_RFBYTE_8P_2DAC4	RFBYTE	8	2	4
%1010	X_RFBYTE_8P_1DAC8	RFBYTE	8	1	8
%1010	X_RFWORD_16P_4DAC4	RFWORD	16	4	4
%1011	X_RFWORD_16P_2DAC8	RFWORD	16	2	8
%1011	X_RFLONG_32P_4DAC8	RFLONG	32	4	8

Example:

```
1 ' Stream bytes to 8 pins
2         RDFAST  #0, ##buffer
3         XINIT   ##X_RFBYTE_8P_1DAC8 | X_PINS_ON + base<<17 + 256, #0
```

Chapter 7: RGB Video Modes

Video is the streamer's headline act, and it earns its own family of modes because pixels are not just bytes. A color pixel must be unpacked into red, green, and blue and pushed out in a form a monitor understands. These RGB modes pull pixel data from a framebuffer and run it through the P2's colorspace converter on the way to the pins — so your code stores a picture and the streamer turns it into a signal. The modes differ mainly in how many bits each pixel uses, trading color depth against memory.

7.1 RGB Format Modes

Mode	Symbol	Hub Read	Format	Bytes/px
`%1011_0010`	`X_RFBYTE_LUMA8`	RFBYTE	Luminance 8	1
`%1011_0011`	`X_RFBYTE_RGBI8`	RFBYTE	RGBI 2:2:2:2	1
`%1011_0100`	`X_RFBYTE_RGB8`	RFBYTE	RGB 3:3:2	1
`%1011_0101`	`X_RFWORD_RGB16`	RFWORD	RGB 5:6:5	2
`%1011_0110`	`X_RFLONG_RGB24`	RFLONG	RGB 8:8:8	4

Memory is usually the deciding factor. A framebuffer is `width × height × bytes/px`, and it shares the P2's **512 KB hub RAM** with your code. A full 640×480 frame is **300 KB** at 1 byte/px (fits), **600 KB** at 2 bytes/px, and **1.2 MB** at 4 bytes/px (neither fits). So full-screen video on a 512 KB P2 generally uses a **1-byte format** (RGB8, RGBI8, or LUMA8); RGB16 and RGB24 suit smaller regions, sprites, or boards with external PSRAM.

7.2 Color Format Details

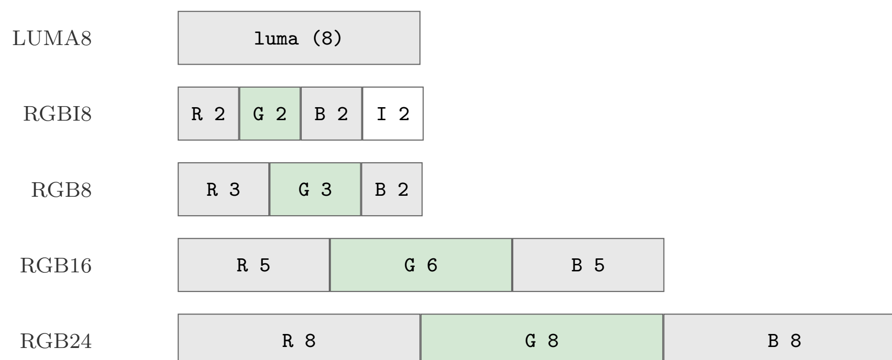

Figure 7.1. RGB source-format bit packing

LUMA8: 8-bit luminance. The `S[2:0]` field selects the output color; the byte sets its intensity.

RGBI8 (2:2:2:2): Two bits each for red, green, and blue, plus a 2-bit intensity field.

RGB8 (3:3:2): Three bits red, three bits green, two bits blue. Compact format for 256-color graphics.

RGB16 (5:6:5): Five bits red, six bits green, five bits blue. Standard 65,536-color format.

RGB24 (8:8:8): Eight bits each for R, G, B. True color, one byte wasted per pixel.

7.3 RGB Mode Example

```
' VGA 640×480 RGB16 output (assumes 250 MHz sysclk)
        RDFAST  ##640*480*2/64, ##framebuffer
        SETXFRQ ##$0CCC_CCCC+1                    ' 25 MHz pixel rate

        MOV     cmd, ##X_RFWORD_RGB16 | X_PINS_ON | X_DACS_3_2_1_0
        ADD     cmd, ##base<<17 + 640
        XCONT   cmd, #0
```

TIP

RGB16 (`X_RFWORD_RGB16`) provides the best balance of color depth and memory efficiency for most video applications.

Chapter 8: WRFAST Input Modes

Here the pipe runs the other way. Instead of driving the pins, these modes *watch* them: on every NCO beat the streamer samples a group of pins and writes the result into hub memory. That turns a COG into a logic analyzer, capturing fast digital activity that software could never sample quickly enough. The captured data flows out through the write FIFO, which — like its read counterpart — must be primed first.

CAUTION

Run WRFAST before any capture command. It primes that same hub FIFO to *receive* data (streamer → hub); until it does, captured data has no valid destination. This is the exact mirror of **RDFAST** (Chapter 6) — one FIFO, opposite direction.

8.1 Pin Capture Modes

Mode	Symbol	Pins	DAC Channels	DAC Bits	Hub Write
%1100	X_1P_1DAC1_WFBYTE	1	1	1	WFBYTE
%1101	X_2P_2DAC1_WFBYTE	2	2	1	WFBYTE
%1101	X_2P_1DAC2_WFBYTE	2	1	2	WFBYTE
%1110	X_4P_4DAC1_WFBYTE	4	4	1	WFBYTE
%1110	X_4P_2DAC2_WFBYTE	4	2	2	WFBYTE
%1110	X_4P_1DAC4_WFBYTE	4	1	4	WFBYTE
%1110	X_8P_4DAC2_WFBYTE	8	4	2	WFBYTE
%1110	X_8P_2DAC4_WFBYTE	8	2	4	WFBYTE
%1110	X_8P_1DAC8_WFBYTE	8	1	8	WFBYTE
%1110	X_16P_4DAC4_WFWORD	16	4	4	WFWORD
%1111	X_16P_2DAC8_WFWORD	16	2	8	WFWORD
%1111	X_32P_4DAC8_WFLONG	32	4	8	WFLONG

D[23] = %w: Must be 1 to enable **WRFAST** writes

Example:

```
' Capture 32 pins to Hub at 10 MHz
        WRFAST  #0, ##capture_buffer
        SETXFRQ ##$0CCC_CCCC+1

        XINIT   ##X_32P_4DAC8_WFLONG | X_WRITE_ON + base<<17 + 1000, #0
        WAITXFI
```

Chapter 9: ADC Sampling Modes

ADC modes are the analog cousin of the pin-capture modes in the previous chapter. Instead of recording whether a pin is high or low, they record *how much* — the digitized voltage on an ADC-capable pin. Streaming those readings into memory at a steady rate turns a COG into an oscilloscope or a data logger. Reach for these when you need to capture a waveform, not just a bit.

9.1 ADC Capture Modes

Mode	Symbol	ADCs	Pins	Hub Write
`%1111_0010`	`X_1ADC8_0P_1DAC8_WFBYTE`	1	0	WFBYTE
`%1111_0011`	`X_1ADC8_8P_2DAC8_WFWORD`	1	8	WFWORD
`%1111_0100`	`X_2ADC8_0P_2DAC8_WFWORD`	2	0	WFWORD
`%1111_0101`	`X_2ADC8_16P_4DAC8_WFLONG`	2	16	WFLONG
`%1111_0110`	`X_4ADC8_0P_4DAC8_WFLONG`	4	0	WFLONG

ADC Pin Requirements: ADC-capable pins must be configured for ADC mode using **WRPIN** before sampling.

9.2 ADC Configuration Example

```
' Configure pin for ADC
        WRPIN   ##P_ADC_100X, #adc_pin
        DRVL    #adc_pin

' Capture 1024 ADC samples
        WRFAST  #0, ##adc_buffer
        MOV     cmd, ##X_1ADC8_0P_1DAC8_WFBYTE | X_WRITE_ON
        ADD     cmd, ##adc_pin<<17 + 1024
        XINIT   cmd, #0
        WAITXFI
```

HARDWARE

ADC readings are 8-bit values. For higher resolution, use smart pin ADC modes with post-processing.

Chapter 10: DDS/Goertzel Mode

This is the streamer's cleverest mode, and it does two things at once (Chapter 1 introduced both in plain terms). **DDS** *generates* a signal — it steps through a waveform table to synthesize a precise tone or arbitrary shape. **Goertzel** *measures* one — it reports how much of a single chosen frequency is present in an incoming signal, the trick behind touch-tone decoding and ultrasonic ranging. Uniquely, this mode advances on **every clock cycle**, not just on NCO rollovers, which is what gives it the resolution to do real signal processing.

10.1 Mode Variants

Mode	Symbol	Filter
`%1111_0ppp_p111`	`X_DDS_GOERTZEL_SINC1`	SINC1
`%1111_1ppp_p111`	`X_DDS_GOERTZEL_SINC2`	SINC2

Both variants share the same mode and config bits; **bit D[23]** alone selects the filter — `0` = SINC1, `1` = SINC2.

10.2 Operation

On each system clock:

1. NCO phase selects LUT entry: `LUT[NCO[30:22]]`
2. NCO phase advances: `NCO += frequency`
3. LUT bytes output to DACs (XOR `$80` for unsigned)
4. ADC input multiplied by sine/cosine from LUT
5. Products accumulated in sine/cosine registers

```
DAC3 := LUT.BYTE[3] ^ $80
DAC2 := LUT.BYTE[2] ^ $80
DAC1 := LUT.BYTE[1] ^ $80
DAC0 := LUT.BYTE[0] ^ $80

sin := LUT.BYTE[3] (sign-extended)
cos := LUT.BYTE[2] (sign-extended)
m := bitstream sum, -3 to +3 (±1 per selected ADC pin)

sin_acc += sin × m
cos_acc += cos × m
```

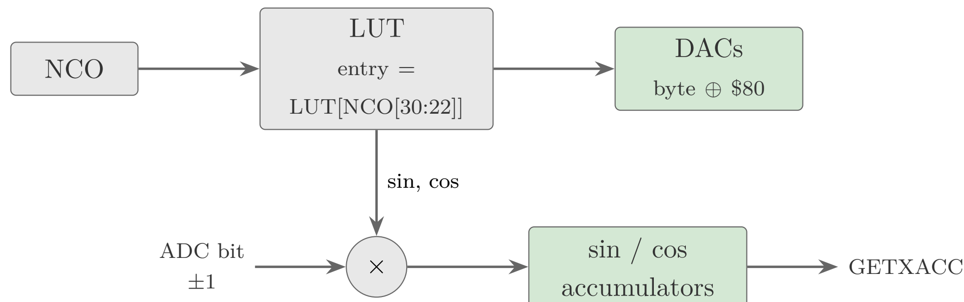

Figure 10.1. DDS output with Goertzel accumulation (every clock)

10.3 LUT Setup

The LUT must contain 512 entries with signed sine/cosine values:

```
' Build the sine/cosine table in a hub array, then bulk-load it to LUT
repeat i from 0 to 511
  cos, sin := polxy(127, i << 23)
  t.BYTE[3] := sin          ' Sine for Goertzel
  t.BYTE[2] := cos          ' Cosine for Goertzel
  t.BYTE[1] := 0            ' Unused
  t.BYTE[0] := sin          ' Optional DAC output
  sine_table[i] := t        ' loaded to LUT in §17.1 via SETQ2+RDLONG
```

10.4 SINC1 vs SINC2

Characteristic	SINC1	SINC2
Accumulation	Direct	Double integration
Q factor	Lower	Higher
Selectivity	Broader	Sharper
Max amplitude	±127 (full signed byte)	±10 (prevents overflow)

CAUTION

SINC2 double-integrates, so its accumulators grow far faster than SINC1's. Scale the LUT waveform amplitude to about ±10 (the value the Silicon Doc's Goertzel example uses for SINC2) to prevent accumulator overflow.

10.5 Reading Results

```
1         GETXACC cos_result          ' Cosine accumulator → D
2         MOV     sin_result, 0-0     ' Sine accumulator → next S
3
4         QVECTOR cos_result, sin_result
5         GETQX   magnitude
6         GETQY   phase
```

10.6 Frequency Calculation

To detect frequency F at clock rate CLK:

> `frequency = $8000_0000 × F / CLK`

HARDWARE

The Goertzel NCO uses the same SETXFRQ scaling as every streamer mode — the multiplier is `$8000_0000` (2^{31}), because the NCO masks its MSB each clock. Resolution is `clock_frequency / 2³¹`, about 0.12 Hz at 250 MHz.

Part III: Configuration Reference

These chapters cover the choices that apply across modes — where data goes among the DAC channels, which pins are driven, how commands are named, and how your code stays in step with the streamer.

Chapter 11: DAC Channel Configuration

Many modes send data to the DAC channels, but none of them say *which* channels, or *how*. That is this chapter's job. The %dddd routing field is the knob from Chapter 1's stereo example: it decides how the streamer's data spreads across the four 8-bit DAC channels — one channel, a stereo pair, a differential pair, or all four independently. The same data becomes mono, stereo, or four-channel purely by changing this field.

11.1 DAC Routing Table

%dddd	DAC3	DAC2	DAC1	DAC0	Symbol
`%0000`	–	–	–	–	`X_DACS_OFF`
`%0001`	X0	X0	X0	X0	`X_DACS_0_0_0_0`
`%0010`	–	–	X0	X0	`X_DACS_X_X_0_0`
`%0011`	X0	X0	–	–	`X_DACS_0_0_X_X`
`%0100`	–	–	–	X0	`X_DACS_X_X_X_0`
`%0101`	–	–	X0	–	`X_DACS_X_X_0_X`
`%0110`	–	X0	–	–	`X_DACS_X_0_X_X`
`%0111`	X0	–	–	–	`X_DACS_0_X_X_X`
`%1000`	!X0	X0	!X0	X0	`X_DACS_0N0_0N0`
`%1001`	–	–	!X0	X0	`X_DACS_X_X_0N0`
`%1010`	!X0	X0	–	–	`X_DACS_0N0_X_X`
`%1011`	X1	X0	X1	X0	`X_DACS_1_0_1_0`
`%1100`	–	–	X1	X0	`X_DACS_X_X_1_0`
`%1101`	X1	X0	–	–	`X_DACS_1_0_X_X`
`%1110`	!X1	X1	!X0	X0	`X_DACS_1N1_0N0`
`%1111`	X3	X2	X1	X0	`X_DACS_3_2_1_0`

Legend:

- -- = No override (SETDACS value used)
- `!` = One's complement (inverted)
- `X0-X3` = Streamer data channels

11.2 DAC Pin Mapping

DAC channels drive pins based on the pin's two LSBs:

DAC Channel	Pin Pattern	Example Pins
DAC0	`%xxxx00`	0, 4, 8, 12, 16…
DAC1	`%xxxx01`	1, 5, 9, 13, 17…
DAC2	`%xxxx10`	2, 6, 10, 14, 18…
DAC3	`%xxxx11`	3, 7, 11, 15, 19…

HARDWARE

Each DAC channel can only drive pins matching its channel number in the two LSBs. This is a silicon constraint, not a configuration option.

11.3 Common DAC Configurations

Mono Audio (single channel):

```
mode := X_RFBYTE_1P_1DAC1 | X_DACS_0_0_0_0 | X_PINS_ON + pin<<17 + count
```

Stereo Audio (two channels):

```
mode := X_RFWORD_16P_2DAC8 | X_DACS_X_X_1_0 | X_PINS_ON + pin<<17 + count
```

Differential Output (noise rejection):

```
mode := X_RFBYTE_1P_1DAC1 | X_DACS_X_X_0N0 | X_PINS_ON + pin<<17 + count
```

Four-Channel Video (RGB + sync):

```
mode := X_RFLONG_32P_4DAC8 | X_DACS_3_2_1_0 | X_PINS_ON + pin<<17 + count
```

Chapter 12: Pin Selection and Control

A streamer command also has to say *which* pins it drives or samples, and that is less obvious than it sounds: the P2 has 64 pins, but a command addresses them 32 at a time, through a window you choose. This chapter covers how to aim the streamer at the right pins, how to enable output, and a few smaller controls such as bit ordering.

12.1 Pin Group Selection

The %ppp field in D[22:20] selects the 32-pin window the streamer drives or samples. The window's **base pin is `%ppp × 8`**, and it always spans **32 consecutive pins** from there — wrapping past pin 63 back to pin 0 once the base climbs high enough.

%ppp	Base	Pin Range (always 32 pins)	Window
`%000`	0	31..0	low pins
`%001`	8	39..8	shifted up 8
`%010`	16	47..16	middle
`%011`	24	55..24	shifted up 24
`%100`	32	63..32	high pins
`%101`	40	63..40, 7..0	wrap (24 + 8)
`%110`	48	63..48, 15..0	wrap (16 + 16)
`%111`	56	63..56, 23..0	wrap (8 + 24)

The **wrap-around groups (%101–%111)** place a 32-pin window that straddles the top and bottom of the pin field — useful when the pins for one function sit at both ends of the chip (for example a peripheral wired across 63..40 and 7..0).

CAUTION

A wrap-around range is still 32 pins, not fewer. "`63..40, 7..0`" reads as two fragments but means pins 63 down to 40 (24 pins) *plus* 7 down to 0 (8 pins) = **32 total**. Don't mistake the split notation for a smaller window.

12.2 Sub-Pin Selection

For modes using fewer than 8 pins, D[19:17] refines selection within the group:

D[19:17]	1-Pin	2-Pin	4-Pin
%000	Pin 0	Pins 1..0	Pins 3..0
%001	Pin 1	Pins 3..2	Pins 7..4
%010	Pin 2	Pins 5..4	Pins 11..8
%011	Pin 3	Pins 7..6	Pins 15..12
%100	Pin 4	Pins 9..8	Pins 19..16
%101	Pin 5	Pins 11..10	Pins 23..20
%110	Pin 6	Pins 13..12	Pins 27..24
%111	Pin 7	Pins 15..14	Pins 31..28

12.3 Enable Control

Output Modes: D[23] must be 1 to drive pins

```
' Pin output enabled
mode := X_RFBYTE_8P_1DAC8 | X_PINS_ON + pin<<17 + count

' Pin output disabled (DACs only)
mode := X_RFBYTE_8P_1DAC8 | X_PINS_OFF + pin<<17 + count
```

Input Modes: D[23] must be 1 to write to Hub

```
' WRFAST enabled
mode := X_32P_4DAC8_WFLONG | X_WRITE_ON + pin<<17 + count

' WRFAST disabled (DACs only)
mode := X_32P_4DAC8_WFLONG | X_WRITE_OFF + pin<<17 + count
```

12.4 Alternate Bit Order

The %a bit in D[16] controls bit ordering for 1/2/4-bit modes:

D[16]	Order	Symbol
0	LSB first (default)	`X_ALT_OFF`
1	MSB first	`X_ALT_ON`

TIP

Use MSB-first (`X_ALT_ON`) for SPI protocols that transmit MSB first.

Chapter 13: Programming Constants

You rarely build a command word bit by bit. Instead you OR together named constants — `X_RFWORD_RGB16`, `X_PINS_ON`, `X_DACS_3_2_1_0` — and the compiler assembles the value for you. This chapter is the catalog of those built-in symbols and shows how they compose. Skim it once to learn the naming pattern; after that the names read almost like sentences.

13.1 Mode Symbols

Immediate → LUT → Pins/DACs:

Symbol	Value	Description
X_IMM_32X1_LUT	%0000 << 28	32×1-bit → LUT
X_IMM_16X2_LUT	%0001 << 28	16×2-bit → LUT
X_IMM_8X4_LUT	%0010 << 28	8×4-bit → LUT
X_IMM_4X8_LUT	%0011 << 28	4×8-bit → LUT

Immediate → Pins/DACs:

Symbol	Value	Description
X_IMM_32X1_1DAC1	%0100 << 28	32×1-bit, 1-pin
X_IMM_16X2_2DAC1	%0101 << 28	16×2-bit, 2-pin
X_IMM_16X2_1DAC2	%0101 << 28 + 2<<16	16×2-bit, 2-pin
X_IMM_8X4_4DAC1	%0110 << 28	8×4-bit, 4-pin
X_IMM_8X4_2DAC2	%0110 << 28 + 2<<16	8×4-bit, 4-pin
X_IMM_8X4_1DAC4	%0110 << 28 + 4<<16	8×4-bit, 4-pin

RDFAST → Pins/DACs:

Symbol	Value	Description
X_RFBYTE_1P_1DAC1	%1000 << 28	RFBYTE, 1-pin
X_RFBYTE_2P_2DAC1	%1001 << 28	RFBYTE, 2-pin
X_RFBYTE_4P_4DAC1	%1010 << 28	RFBYTE, 4-pin
X_RFBYTE_8P_1DAC8	%1010 << 28 + $E<<16	RFBYTE, 8-pin
X_RFWORD_16P_4DAC4	%1010 << 28 + $F<<16	RFWORD, 16-pin
X_RFWORD_16P_2DAC8	%1011 << 28	RFWORD, 16-pin
X_RFLONG_32P_4DAC8	%1011 << 28 + 1<<16	RFLONG, 32-pin

RDFAST → RGB:

Symbol	Value	Description
X_RFBYTE_LUMA8	%1011 << 28 + 2<<16	8-bit grayscale
X_RFBYTE_RGBI8	%1011 << 28 + 3<<16	RGBI 2:2:2:2
X_RFBYTE_RGB8	%1011 << 28 + 4<<16	RGB 3:3:2
X_RFWORD_RGB16	%1011 << 28 + 5<<16	RGB 5:6:5
X_RFLONG_RGB24	%1011 << 28 + 6<<16	RGB 8:8:8

DDS/Goertzel:

Symbol	Value	Description
X_DDS_GOERTZEL_SINC1	%1111 << 28 + 7<<16	SINC1 filter
X_DDS_GOERTZEL_SINC2	%1111 << 28 + 7<<16 + 1<<23	SINC2 (D[23]=1)

13.2 Control Symbols

Symbol	Value	Effect
X_PINS_OFF	%0 << 23	Disable pin output
X_PINS_ON	%1 << 23	Enable pin output
X_WRITE_OFF	%0 << 23	Disable WRFAST
X_WRITE_ON	%1 << 23	Enable WRFAST
X_ALT_OFF	%0 << 16	LSB first
X_ALT_ON	%1 << 16	MSB first

13.3 DAC Symbols

Symbol	Value	Configuration
X_DACS_OFF	%0000 << 24	No DAC output
X_DACS_0_0_0_0	%0001 << 24	X0 on all channels
X_DACS_X_X_0_0	%0010 << 24	X0 on channels 0,1
X_DACS_0_0_X_X	%0011 << 24	X0 on channels 2,3
X_DACS_X_X_X_0	%0100 << 24	X0 on channel 0
X_DACS_X_X_0_X	%0101 << 24	X0 on channel 1
X_DACS_X_0_X_X	%0110 << 24	X0 on channel 2
X_DACS_0_X_X_X	%0111 << 24	X0 on channel 3
X_DACS_0N0_0N0	%1000 << 24	Differential pairs
X_DACS_X_X_0N0	%1001 << 24	Diff on 0,1
X_DACS_0N0_X_X	%1010 << 24	Diff on 2,3
X_DACS_1_0_1_0	%1011 << 24	Stereo pairs
X_DACS_X_X_1_0	%1100 << 24	Stereo on 0,1
X_DACS_1_0_X_X	%1101 << 24	Stereo on 2,3
X_DACS_1N1_0N0	%1110 << 24	Differential stereo
X_DACS_3_2_1_0	%1111 << 24	All 4 independent

13.4 Symbol Composition

Build complete commands by combining symbols:

```
' VGA 640-pixel visible line
mode := X_RFWORD_RGB16 | X_PINS_ON | X_DACS_3_2_1_0 + vga_base<<17 + 640

' SPI byte output (MSB first)
mode := X_IMM_32X1_1DAC1 | X_PINS_ON | X_ALT_ON + spi_pin<<17 + 8

' Goertzel analysis (differential DAC)
mode := X_DDS_GOERTZEL_SINC1 | X_DACS_0N0_0N0 + adc_pin<<17 + cycles
```

Chapter 14: Events and Synchronization

Because the streamer runs on its own, your code needs a way to ask *where it is up to* — is it ready for another command, has it finished, did the NCO just roll over? The streamer raises events for exactly these moments, and this chapter shows how to poll them, wait on them, or branch on them. Getting this right is how you chain commands seamlessly and keep video and audio free of glitches.

14.1 Streamer Events

Event #	Symbol	Trigger Condition
10	`EVENT_XMT`	Streamer ready for new command
11	`EVENT_XFI`	Streamer finished (no pending command)
12	`EVENT_XRO`	NCO rollover occurred
13	`EVENT_XRL`	LUT address $1FF read

14.2 Event Instructions

Polling (non-blocking):

Instruction	Effect
`POLLXMT WC`	C = 1 if ready for command
`POLLXFI WC`	C = 1 if finished
`POLLXRO WC`	C = 1 if NCO rolled over
`POLLXRL WC`	C = 1 if LUT $1FF read

Waiting (blocking):

Instruction	Effect
`WAITXMT`	Wait until ready for command
`WAITXFI`	Wait until finished
`WAITXRO`	Wait until NCO rollover
`WAITXRL`	Wait until LUT $1FF read

Conditional Jumps:

Instruction	Condition
`JXMT label`	Jump if ready
`JNXMT label`	Jump if not ready
`JXFI label`	Jump if finished
`JNXFI label`	Jump if not finished
`JXRO label`	Jump if rollover
`JNXRO label`	Jump if no rollover
`JXRL label`	Jump if LUT $1FF
`JNXRL label`	Jump if not LUT $1FF

14.3 Event Clearing

The three streamer-command events — **EVENT_XMT** (10), **EVENT_XFI** (11), and **EVENT_XRO** (12) — clear automatically on:

- **XINIT**, **XCONT**, **XZERO** execution (these instructions re-arm the events)
- **POLL**, **WAIT**, or **J** instruction execution for that event

EVENT_XRL (13, LUT address $1FF read) is the exception: it is **not** re-armed by **XINIT**/XCONT/XZERO. It clears only on cog start or on its own poll/wait/jump (POLLXRL/WAITXRL/JXRL/JNXRL).

14.4 Synchronization Patterns

Wait for completion:

```
1        XINIT   mode, data
2        WAITXFI                     ' Block until done
```

Chain commands without gaps:

```
1        XINIT   mode1, data1        ' Start first command
2        XCONT   mode2, data2        ' Queue second command
3        XCONT   mode3, data3        ' Queue third command
4        WAITXFI                     ' Wait for all to finish
```

Video line timing:

```
line    XZERO   m_sync, sync_data   ' Sync pulse (phase zeroed)
        XCONT   m_back, #0          ' Back porch
        XCONT   m_visible, #0       ' Visible pixels
        XCONT   m_front, #0         ' Front porch
        JMP     #line
```

TIP

Use **XZERO** at line start to prevent phase accumulation errors over many lines.

Part IV: Applications

Here the modes come together into the things people actually build: video, high-speed serial, signal processing, and the patterns that combine them.

Chapter 15: Video Output

Part IV puts the pieces together into real applications, beginning with the streamer's signature use: video. This chapter walks through generating VGA, HDMI, and composite signals — combining the RGB modes of Chapter 7, the NCO timing of Chapter 3, and the sync discipline that keeps a picture stable. The encoding differs by standard, but the shape is always the same: stream a framebuffer, on the beat, line after line.

15.1 VGA Output

VGA uses analog RGB on DAC channels, with **separate** horizontal and vertical sync. The streamer drives the analog levels: the horizontal-sync level rides the streamer's immediate operand, while vertical sync is a plain pin toggle.

Hardware Requirements:

- Three DAC pins for R, G, B, plus one DAC pin for the horizontal-sync level
- One additional digital pin for vertical sync (toggled directly, not streamed)
- Resistor DAC network or direct DAC output

Timing Structure (640×480 @ 60 Hz):

Element	Pixels	Duration @ 25.175 MHz
Visible	640	25.42 µs
Front porch	16	0.64 µs
Sync pulse	96	3.81 µs
Back porch	48	1.91 µs
Total line	**800**	**31.78 µs**

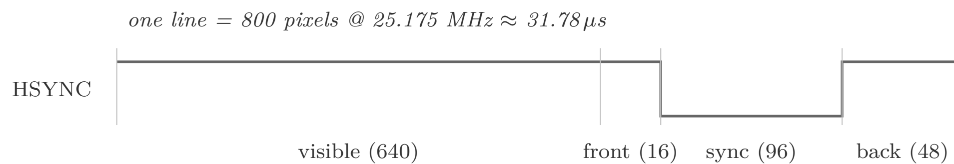

Figure 15.1. VGA 640×480 horizontal line timing

Example:

The non-visible intervals — front porch, sync, back porch, and whole blank lines — stream a **fixed DAC level** through an *immediate* mode, `X_IMM_1X32_4DAC8 | X_DACS_3_2_1_0` (`$7F01_0000`). Its S operand is the 32-bit level held across the four DAC channels for `D[15:0]` pixels, so the **horizontal-sync** level is simply a different S value during the sync interval. **Vertical sync is not streamed** — it is a separate pin toggled with `DRVNOT` around the vsync lines.

```
1  DAT             ORG
2                  SETXFRQ pixfreq                 ' 25.175 MHz pixel NCO
3                                                  ' (2^31-scaled)
4                  MOV     vsync_pin, ##VGA_BASE + 4 ' VSYNC: a separate
5                                                  ' digital pin
6                  DRVC    vsync_pin               ' establish initial
7                                                  ' VSYNC level
8
9  vfield          MOV     y, #10                  ' vertical front porch
10                                                 ' (lines)
11                 CALL    #blank
12                 RDFAST  #0, ##framebuffer       ' visible pixels stream
13                                                 ' from the FIFO
14                 MOV     y, #480                 ' visible lines
15 line            CALL    #hsync
16                 XCONT   m_visible, #0           ' 640 RGB pixels
```

continues on next page →

↪ continued from previous page

```
17                                                    ' (pipeline: Chapter 7)
18                DJNZ    y, #line
19                MOV     y, #33                      ' vertical back porch
20                                                    ' (lines)
21                CALL    #blank
22                DRVNOT  vsync_pin                   ' VSYNC active
23                MOV     y, #2                       ' vertical sync (lines)
24                CALL    #blank
25                DRVNOT  vsync_pin                   ' VSYNC inactive
26                JMP     #vfield
27
28 ' One non-visible line: horizontal sync, then a flat blank level
29 blank          CALL    #hsync
30                XCONT   m_blank, #0
31          _ret_ DJNZ    y, #blank
32
33 ' Horizontal sync drives the immediate S operand (the streamed DAC level):
34 ' #0 = blank (0 V); #1 is a PLACEHOLDER sync level - replace #0/#1 with
35 ' your hardware's calibrated blank and sync DAC values.
36 hsync          XCONT   m_front, #0                 ' 16px front porch
37                XCONT   m_sync,  #1                 ' 96px hsync pulse
38          _ret_ XCONT   m_back,  #0                 ' 48px back porch
39
40 ' Immediate level mode X_IMM_1X32_4DAC8 ($7001_0000) | X_DACS_3_2_1_0
41 ' ($0F00_0000) = $7F01_0000; D[15:0] = pixel count for the interval.
42 m_front        LONG    $7F01_0000 + 16
43 m_sync         LONG    $7F01_0000 + 96
44 m_back         LONG    $7F01_0000 + 48
45 ' whole line, no visible pixels
46 m_blank        LONG    $7F01_0000 + 800
47 ' X_RFWORD_RGB16 | X_PINS_ON
48 m_visible      LONG    $B085_0000 + 640
49
50 pixfreq        LONG    $0CE3_BCD3                  ' 25.175 MHz @ 250 MHz
51 vsync_pin      RES     1
52 y              RES     1
```

For a worked reference using this general approach, see Eric R. Smith's VGA driver (Parallax OBEX #2847), which drives sync and blanking through the streamer's DAC outputs.

15.2 HDMI/DVI Output

HDMI uses TMDS encoding via the colorspace converter. Requires 10× pixel clock.

Hardware Requirements:

- Eight pins in sequence for TMDS pairs
- 1 mA pin drive strength
- System clock = 10 × pixel clock

Configuration:

```
1                   ' Enable DVI forward mode
2                   SETCMOD #$100
3
4                   ' Configure HDMI pins
5                   DRVL    #7<<6 + hdmi_base
6                   WRPIN   ##%100100_00_00000_0, #7<<6 + hdmi_base
7
8                   ' NCO for 1/10 rate (TMDS serialization)
9                   SETXFRQ ##$0CCC_CCCC+1
```

HARDWARE

HDMI requires the colorspace converter in DVI mode. The converter generates TMDS encoding automatically from RGB data.

15.3 Composite Video

Composite video uses the colorspace converter to generate NTSC or PAL signals.

Configuration:

```
1                   ' Composite mode (NTSC/PAL) - CMOD[6:5] = %11
2                   SETCMOD #%11 << 5
3
4                   ' NTSC chroma carrier 3.579545 MHz at 250 MHz clock
5                   ' CFRQ = $1_0000_0000 × carrier / clkfreq
6                   SETCFRQ ##$03AA_5B33
7
8                   ' Color matrix coefficients
9                   SETCY   ##cy_ntsc
10                  SETCI   ##ci_ntsc
11                  SETCQ   ##cq_ntsc
```

Chapter 16: High-Speed Serial (SPI)

Not every streamer job is video or audio. This chapter shows the streamer as a fast, precise bit pump for serial protocols such as SPI — emitting a stream of bits from memory while a smart pin generates the matching clock. The pairing is the point: the streamer handles the data, the smart pin handles the clock, and — both being driven from the same system clock — they run at matched rates for transfers far faster than a software bit-bang.

16.1 SPI Output with Streamer

The streamer outputs SPI data while a smart pin generates the clock.

Configuration:

```
                ' Configure clock pin as transition counter
                WRPIN   ##P_TRANSITION + P_OE, #spi_clk
                WXPIN   ##2, #spi_clk           ' base period in clocks
                DRVL    #spi_clk

                ' NCO at half clock rate
                SETXFRQ ##$4000_0000
```

Single Byte Transfer:

```
spi_byte        MOV     bmode, ##X_IMM_32X1_1DAC1 | X_PINS_ON | X_ALT_ON
                ADD     bmode, ##spi_do<<17 + 8
                XINIT   bmode, pa               ' Output 8 bits
                WYPIN   #16, #spi_clk           ' 16 clock transitions
          _ret_ WAITXFI
```

Bulk Transfer:

```
spi_block       RDFAST  #0, ptra                ' Point to data
                MOV     rmode, ##X_RFBYTE_1P_1DAC1 | X_PINS_ON | X_ALT_ON
                ADD     rmode, ##spi_do<<17 + 256*8
                XINIT   rmode, #0               ' Stream 256 bytes
                WYPIN   ##256*8*2, #spi_clk     ' Clock transitions
          _ret_ WAITXFI
```

16.2 Coordinating with WAITXFI

The **WAITXFI** instruction blocks only until the streamer finishes — it has no knowledge of the smart-pin clock. Check the clock's completion separately (e.g. `TESTP` on the clock pin):

```
1                XINIT   mode, data
2                WYPIN   transitions, #clk_pin
3                WAITXFI                          ' Wait for data
4                TESTP   #clk_pin wc              ' Verify clock done
```

CAUTION

The smart pin clock and streamer operate independently. Verify both complete before starting the next transfer.

Chapter 17: Signal Processing

This chapter returns to the DDS and Goertzel capabilities of Chapter 10 and puts them to work — generating waveforms with a function generator's precision, and detecting specific frequencies for tone decoding, distance sensing, and measurement. Where Chapter 10 explained the mechanism, this chapter shows the applications.

17.1 Goertzel Frequency Detection

Goertzel analysis detects specific frequencies in ADC input.

Application: Ultrasonic distance measurement, DTMF decoding, tone detection

Setup:

```
1                  ' Load sine/cosine table to LUT
2                  SETQ2   #$200-1
3                  RDLONG  0, ##sine_table
4
5                  ' Configure ADC pin
6                  WRPIN   ##P_ADC_100X, #adc_pin
7                  DRVL    #adc_pin
8
9                  ' Configure DAC output (differential)
10                 WRPIN   ##P_DAC_124R_3V + P_CHANNEL, dac_pins
11                 DRVL    dac_pins
```

Detection Loop:

```
1  ' Calculate NCO frequency for target (2^31-scaled for the NCO)
2                  RDLONG  clkf, #$44              ' clkfreq from hub $44
3  detect
4                  QFRAC   target_freq, clkf       ' QFRAC = 2^32 × target/clk
5                  GETQX   xfrq
6                  SHR     xfrq, #1                ' halve to the NCO's
7                                                  ' 2^31 scaling
8
9                  ' Run Goertzel analysis
10                 SETWORD dds_cmd, cycles, #0
11                 SETQ    xfrq
12                 XCONT   dds_cmd, dds_s
```

continues on next page →

↪ continued from previous page

```
13
14                  ' Get result
15                  GETXACC cos_acc
16                  MOV     sin_acc, 0-0
17
18                  ' Convert to magnitude
19                  QVECTOR cos_acc, sin_acc
20                  GETQX   magnitude
21
22                  ' Check threshold
23                  CMP     magnitude, threshold wcz
24          if_a    CALL    #detected
25
26                  JMP     #detect
27
28 dds_cmd          LONG    X_DDS_GOERTZEL_SINC1 | X_DACS_ON0_ON0
```

17.2 DDS Waveform Generation

DDS synthesizes arbitrary waveforms at precise frequencies.

Applications: Function generator, audio synthesis, RF modulation

Configuration:

```
                ' Load waveform to LUT
                SETQ2   #$200-1
                RDLONG  0, ##waveform_table

                ' Set output frequency (2^31-scaled for the NCO)
                RDLONG  clkf, #$44              ' clkfreq from hub $44
                QFRAC   output_freq, clkf       ' QFRAC = 2^32 × output/clk
                GETQX   xfrq
                SHR     xfrq, #1                ' halve to the NCO's
                                                ' 2^31 scaling
                SETXFRQ xfrq

                ' Continuous output
                XINIT   dds_mode, #0
```

TIP

The LUT can contain any waveform shape—sine, square, triangle, or arbitrary samples. The NCO steps through the 512 entries at the programmed rate.

Chapter 18: Integration Patterns

The final chapter collects patterns that cut across everything above: double-buffering so display and rendering never collide, splitting work across multiple COGs, and coordinating the streamer with smart pins. These are the techniques that turn a working streamer demo into a robust system.

18.1 Double Buffering

Use two buffers to allow simultaneous rendering and display:

```
                ' Buffer addresses
                MOV     display_buf, ##buffer_a
                MOV     render_buf, ##buffer_b

frame_loop      ' Start displaying current buffer
                RDFAST  ##frame_size/64, display_buf

                ' Render to other buffer while displaying
                CALL    #render_frame

                ' Swap buffers
                MOV     temp, display_buf
                MOV     display_buf, render_buf
                MOV     render_buf, temp

                JMP     #frame_loop
```

18.2 Multi-COG Video

Complex video systems span multiple COGs:

COG	Function
0	Main application
1	Horizontal timing, pixel streaming
2	Vertical timing, frame sync
3	Sprite rendering

Synchronization via Hub flags:

```
' COG 1: Signal line complete
                WRLONG  #1, ##line_done_flag

' COG 2: Wait for line complete, then clear the flag (handshake)
wait_line       RDLONG  temp, ##line_done_flag wz
        if_z    JMP     #wait_line
                WRLONG  #0, ##line_done_flag    ' clear for next line
```

18.3 Streamer + Smart Pin Coordination

Many applications combine streamer I/O with smart pin timing:

Pattern: Streamer data with smart pin clock

```
                XINIT   data_mode, #0           ' Start data output
                WYPIN   clocks, #clk_pin        ' Start clock generation
                WAITXFI                         ' Wait for data complete
```

Pattern: Smart pin trigger for streamer

```
wait_trigger    TESTP   #trigger_pin wc         ' wait for the event
        if_nc   JMP     #wait_trigger
                AKPIN   #trigger_pin            ' acknowledge it
                XINIT   capture_mode, #0        ' Start capture
```

Part V: Appendices

The appendices are lookup material: the complete mode-encoding table, the symbol quick-reference, the frequency-calculation tables, and a troubleshooting guide. Reach for them once you know which mode you need and want the exact bits.

Appendix A: Complete Mode Encoding Table

Table A.1.

D[31:28]	D[19:16]	Mode	Symbol
%0000	%bbbb	IMM 32×1 → LUT	X_IMM_32X1_LUT
%0001	%bbbb	IMM 16×2 → LUT	X_IMM_16X2_LUT
%0010	%bbbb	IMM 8×4 → LUT	X_IMM_8X4_LUT
%0011	%bbbb	IMM 4×8 → LUT	X_IMM_4X8_LUT
%0100	%0000	IMM 32×1 → 1-pin + 1-DAC1	X_IMM_32X1_1DAC1
%0101	%0000	IMM 16×2 → 2-pin + 2-DAC1	X_IMM_16X2_2DAC1
%0101	%0010	IMM 16×2 → 2-pin + 1-DAC2	X_IMM_16X2_1DAC2
%0110	%0000	IMM 8×4 → 4-pin + 4-DAC1	X_IMM_8X4_4DAC1
%0110	%0010	IMM 8×4 → 4-pin + 2-DAC2	X_IMM_8X4_2DAC2
%0110	%0100	IMM 8×4 → 4-pin + 1-DAC4	X_IMM_8X4_1DAC4
%0110	%0110	IMM 4×8 → 8-pin + 4-DAC2	X_IMM_4X8_4DAC2
%0110	%0111	IMM 4×8 → 8-pin + 2-DAC4	X_IMM_4X8_2DAC4
%0110	%1110	IMM 4×8 → 8-pin + 1-DAC8	X_IMM_4X8_1DAC8
%0110	%1111	IMM 2×16 → 16-pin + 4-DAC4	X_IMM_2X16_4DAC4

Continued on next page

Table A.1. (Continued)

D[31:28]	D[19:16]	Mode	Symbol
%0111	%0000	IMM 2×16 → 16-pin + 2-DAC8	X_IMM_2X16_2DAC8
%0111	%0001	IMM 1×32 → 32-pin + 4-DAC8	X_IMM_1X32_4DAC8
%0111	%001a	RFLONG 32×1 → LUT	X_RFLONG_32X1_LUT
%0111	%010a	RFLONG 16×2 → LUT	X_RFLONG_16X2_LUT
%0111	%011a	RFLONG 8×4 → LUT	X_RFLONG_8X4_LUT
%0111	%1000	RFLONG 4×8 → LUT	X_RFLONG_4X8_LUT
%1000	%pppp	RFBYTE → 1-pin + 1-DAC1	X_RFBYTE_1P_1DAC1
%1001	%ppp0	RFBYTE → 2-pin + 2-DAC1	X_RFBYTE_2P_2DAC1
%1001	%ppp0+2	RFBYTE → 2-pin + 1-DAC2	X_RFBYTE_2P_1DAC2
%1010	%pp00	RFBYTE → 4-pin + 4-DAC1	X_RFBYTE_4P_4DAC1
%1010	%pp00+2	RFBYTE → 4-pin + 2-DAC2	X_RFBYTE_4P_2DAC2
%1010	%pp00+4	RFBYTE → 4-pin + 1-DAC4	X_RFBYTE_4P_1DAC4
%1010	%p000+6	RFBYTE → 8-pin + 4-DAC2	X_RFBYTE_8P_4DAC2
%1010	%p000+7	RFBYTE → 8-pin + 2-DAC4	X_RFBYTE_8P_2DAC4
%1010	%p000+$E	RFBYTE → 8-pin + 1-DAC8	X_RFBYTE_8P_1DAC8
%1010	%1111	RFWORD → 16-pin + 4-DAC4	X_RFWORD_16P_4DAC4
%1011	%0000	RFWORD → 16-pin + 2-DAC8	X_RFWORD_16P_2DAC8
%1011	%0001	RFLONG → 32-pin + 4-DAC8	X_RFLONG_32P_4DAC8
%1011	%0010	RFBYTE LUMA8	X_RFBYTE_LUMA8
%1011	%0011	RFBYTE RGBI8	X_RFBYTE_RGBI8
%1011	%0100	RFBYTE RGB8	X_RFBYTE_RGB8
%1011	%0101	RFWORD RGB16	X_RFWORD_RGB16
%1011	%0110	RFLONG RGB24	X_RFLONG_RGB24
%1100	%pppp	1-pin + 1-DAC1 → WFBYTE	X_1P_1DAC1_WFBYTE
%1101	%ppp0	2-pin + 2-DAC1 → WFBYTE	X_2P_2DAC1_WFBYTE
%1101	%ppp0+2	2-pin + 1-DAC2 → WFBYTE	X_2P_1DAC2_WFBYTE

Continued on next page

Table A.1. (Continued)

D[31:28]	D[19:16]	Mode	Symbol
%1110	%pp00	4-pin + 4-DAC1 → WFBYTE	X_4P_4DAC1_WFBYTE
%1110	%pp00+2	4-pin + 2-DAC2 → WFBYTE	X_4P_2DAC2_WFBYTE
%1110	%pp00+4	4-pin + 1-DAC4 → WFBYTE	X_4P_1DAC4_WFBYTE
%1110	%p000+6	8-pin + 4-DAC2 → WFBYTE	X_8P_4DAC2_WFBYTE
%1110	%p000+7	8-pin + 2-DAC4 → WFBYTE	X_8P_2DAC4_WFBYTE
%1110	%p000+$E	8-pin + 1-DAC8 → WFBYTE	X_8P_1DAC8_WFBYTE
%1110	%1111	16-pin + 4-DAC4 → WFWORD	X_16P_4DAC4_WFWORD
%1111	%0000	16-pin + 2-DAC8 → WFWORD	X_16P_2DAC8_WFWORD
%1111	%0001	32-pin + 4-DAC8 → WFLONG	X_32P_4DAC8_WFLONG
%1111	%0010	1 ADC → WFBYTE	X_1ADC8_0P_1DAC8_WFBYTE
%1111	%0011	1 ADC + 8-pin → WFWORD	X_1ADC8_8P_2DAC8_WFWORD
%1111	%0100	2 ADC → WFWORD	X_2ADC8_0P_2DAC8_WFWORD
%1111	%0101	2 ADC + 16-pin → WFLONG	X_2ADC8_16P_4DAC8_WFLONG
%1111	%0110	4 ADC → WFLONG	X_4ADC8_0P_4DAC8_WFLONG
%1111	%0111	DDS/Goertzel SINC1	X_DDS_GOERTZEL_SINC1
%1111	%0111 (D[23]=1)	DDS/Goertzel SINC2	X_DDS_GOERTZEL_SINC2

Appendix B: Symbol Quick Reference

Mode Symbols

```
X_IMM_32X1_LUT          X_IMM_16X2_LUT          X_IMM_8X4_LUT
X_IMM_4X8_LUT           X_IMM_32X1_1DAC1        X_IMM_16X2_2DAC1
X_IMM_16X2_1DAC2        X_IMM_8X4_4DAC1         X_IMM_8X4_2DAC2
X_IMM_8X4_1DAC4         X_IMM_4X8_4DAC2         X_IMM_4X8_2DAC4
X_IMM_4X8_1DAC8         X_IMM_2X16_4DAC4        X_IMM_2X16_2DAC8
X_IMM_1X32_4DAC8        X_RFLONG_32X1_LUT       X_RFLONG_16X2_LUT
X_RFLONG_8X4_LUT        X_RFLONG_4X8_LUT        X_RFBYTE_1P_1DAC1
X_RFBYTE_2P_2DAC1       X_RFBYTE_2P_1DAC2       X_RFBYTE_4P_4DAC1
X_RFBYTE_4P_2DAC2       X_RFBYTE_4P_1DAC4       X_RFBYTE_8P_4DAC2
X_RFBYTE_8P_2DAC4       X_RFBYTE_8P_1DAC8       X_RFWORD_16P_4DAC4
X_RFWORD_16P_2DAC8      X_RFLONG_32P_4DAC8      X_RFBYTE_LUMA8
X_RFBYTE_RGBI8          X_RFBYTE_RGB8           X_RFWORD_RGB16
X_RFLONG_RGB24          X_1P_1DAC1_WFBYTE       X_2P_2DAC1_WFBYTE
X_2P_1DAC2_WFBYTE       X_4P_4DAC1_WFBYTE       X_4P_2DAC2_WFBYTE
X_4P_1DAC4_WFBYTE       X_8P_4DAC2_WFBYTE       X_8P_2DAC4_WFBYTE
X_8P_1DAC8_WFBYTE       X_16P_4DAC4_WFWORD      X_16P_2DAC8_WFWORD
X_32P_4DAC8_WFLONG      X_1ADC8_0P_1DAC8_WFBYTE X_1ADC8_8P_2DAC8_WFWORD
X_2ADC8_0P_2DAC8_WFWORD X_2ADC8_16P_4DAC8_WFLONG X_4ADC8_0P_4DAC8_WFLONG
X_DDS_GOERTZEL_SINC1    X_DDS_GOERTZEL_SINC2
```

Control Symbols

```
X_PINS_OFF    X_PINS_ON     X_WRITE_OFF   X_WRITE_ON
X_ALT_OFF     X_ALT_ON
```

DAC Symbols

```
X_DACS_OFF       X_DACS_0_0_0_0    X_DACS_X_X_0_0    X_DACS_0_0_X_X
X_DACS_X_X_X_0   X_DACS_X_X_0_X    X_DACS_X_0_X_X    X_DACS_0_X_X_X
X_DACS_0N0_0N0   X_DACS_X_X_0N0    X_DACS_0N0_X_X    X_DACS_1_0_1_0
X_DACS_X_X_1_0   X_DACS_1_0_X_X    X_DACS_1N1_0N0    X_DACS_3_2_1_0
```

Appendix C: Frequency Calculation Tables

NCO Frequency Values

Formula: `frequency = $8000_0000 × (rate / clock)`

Rate Ratio	Value	Notes
1:1	`$8000_0000`	Every clock
1:2	`$4000_0000`	Half clock
1:3	`$2AAA_AAAB`	Add 1 for fractional
1:4	`$2000_0000`	Quarter clock
1:5	`$1999_999A`	Add 1
1:8	`$1000_0000`	Eighth clock
1:10	`$0CCC_CCCD`	Tenth clock
1:16	`$0800_0000`	Sixteenth clock

Common Video Pixel Rates

Resolution	Pixel Rate	At 250 MHz	At 300 MHz	At 320 MHz
640×480	25.175 MHz	`$0CE3_BCD3`	`$0ABD_C805`	`$0A11_EB85`
720×480	27.000 MHz	`$0DD2_F1AA`	`$0B85_1EB8`	`$0ACC_CCCD`
800×600	40.000 MHz	`$147A_E148`	`$1111_1111`	`$1000_0000`
1024×768	65.000 MHz	`$2147_AE14`	`$1BBB_BBBC`	`$1A00_0000`
1280×720	74.250 MHz	`$2604_1893`	`$1FAE_147B`	`$1DB3_3333`

Values are `round($8000_0000 × pixel_rate / clock_frequency)`.

Appendix D: Troubleshooting Guide

Symptom: No Output on Pins

Check:

1. D[23] = 1 (`X_PINS_ON` included in mode)
2. Pin group %ppp selects correct pins
3. Sub-pin selection matches target pins
4. Pins configured as outputs (DRVH/DRVL as needed)

Symptom: Corrupted Data from RDFAST

Check:

1. **RDFAST** executed before streamer command
2. Buffer address aligned to 64-byte boundary for wrap mode
3. Buffer size matches FIFO configuration
4. No other code reading from FIFO simultaneously

Symptom: Streamer Stops Unexpectedly

Check:

1. Count field D[15:0] not zero
2. Command buffer has pending command (**XCONT**/**XZERO** issued)
3. No **XINIT #0,#0** executed

Symptom: Phase Drift in Video

Check:

1. Use **XZERO** at line boundaries
2. NCO frequency matches pixel rate exactly
3. Total pixels per line equals timing specification

Symptom: DAC Output Incorrect

Check:

1. %dddd field selects correct routing

2. DAC channel matches pin LSBs
3. Pin configured for DAC mode

Symptom: Goertzel Results Invalid

Check:

1. LUT contains signed sine/cosine values
2. ADC pin configured for ADC mode
3. Sample count adequate for frequency resolution
4. SINC2 amplitude reduced to ± 10 to prevent overflow

Index

A

- ADC sampling modes: Chapter 9
- Alternate bit order: 12.4
- Architecture: Chapter 2

B

- Block diagram: 2.1

C

- Clock accuracy: 3.5
- Colorspace converter: 15.2, 15.3
- Command structure: Chapter 4
- Count field: 4.6

D

- DAC channels: Chapter 11
- DAC pin mapping: 11.2
- DAC routing table: 11.1
- DAC symbols: 13.3
- DDS mode: Chapter 10
- Double buffering: 18.1

E

- Enable control: 12.3
- Events: Chapter 14

F

- Frequency calculation: 3.2, 3.4, Appendix C

G

- **GETXACC**: 4.7, 10.5
- Goertzel mode: Chapter 10

H

- HDMI output: 15.2
- Hub FIFO: 6.1

I

- Immediate modes: Chapter 5

J

- Jitter (per-pixel): 3.4, 3.5

L

- LUT setup: 5.1, 10.3

M

- Mode encoding table: Appendix A
- Mode field: 4.2
- Mode symbols: 13.1
- Multi-COG: 18.2

N

- NCO: Chapter 3

O

- Oscillator (TCXO): 3.5

P

- Pin group selection: 12.1
- Pin selection: Chapter 12
- Pixel rate: 3.4

R

- **RDFAST** modes: Chapter 6
- RGB modes: Chapter 7

S

- **SETXFRQ**: 3.3, 4.7
- Signal processing: Chapter 17
- SINC1/SINC2: 10.4
- Smart pin coordination: 18.3
- SPI: Chapter 16
- Sub-pin selection: 12.2
- Symbol composition: 13.4
- Symbols quick reference: Appendix B

T

- TCXO: 3.5
- Troubleshooting: Appendix D

V

- VGA output: 15.1
- Video output: Chapter 15

W

- **WAITXFI**: 14.2
- **WRFAST** modes: Chapter 8

X

- **XCONT**: 4.7
- **XINIT**: 4.7
- **XSTOP**: 4.7
- **XZERO**: 4.7