**Propeller 2 P2X8C4M64P**

**Hardware Manual**

Nov 1, 2022 Release

#### COPYRIGHTS AND TRADEMARKS

This documentation is copyright © 2022 by Parallax Inc. By downloading or obtaining a printed copy of this documentation or software you agree that it is to be used with, or with products containing, the Parallax Propeller 2 P2X8C4M64P microcontroller. Any other uses are not permitted and may represent a violation of Parallax copyrights, legally punishable according to Federal copyright or intellectual property laws. Any duplication of this documentation for commercial uses is expressly prohibited by Parallax Inc. Duplication for educational use is permitted, subject to the following Conditions of Duplication: Parallax Inc. grants the user a conditional right to download, duplicate, and distribute this text without Parallax's permission. This right is based on the following conditions: the text, or any portion thereof, may not be duplicated for commercial use; it may be duplicated only for educational purposes when used solely in conjunction with Parallax products, and the user may recover from the student only the cost of duplication.

Parallax, Propeller Spin, and the Parallax logos are trademarks of Parallax Inc. If you decide to use any trademarks of Parallax Inc. on your web page or in printed material, you must state that (trademark) is a trademark of Parallax Inc.” upon the first appearance of the trademark name in each printed document or web page. Other brand and product names herein are trademarks or registered trademarks of their respective holders.

#### DISCLAIMER OF LIABILITY

Parallax, Inc. makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Parallax, Inc. assume any liability arising out of the application or use of any product, and specifically disclaims any and all liability, including without limitation consequential or incidental damages even if Parallax, Inc. has been advised of the possibility of such damages.

#### INTERNET DISCUSSION LISTS

We maintain active web-based discussion forums for people interested in Parallax Propeller products, at [forums.parallax.com](https://forums.parallax.com/).

#### ERRATA

While great effort is made to assure the accuracy of our texts, errors may still exist. If you find an error, please let us know by commenting/suggesting on live documentation, or by sending an email to editor@parallax.com. We continually strive to improve all of our educational materials and documentation, and frequently revise our texts. Occasionally, an errata sheet with a list of known errors and corrections for a given text will be posted to our website, [www.parallax.com](https://www.parallax.com). Please check the individual product page’s free downloads for an errata file.

#### SUPPORTED HARDWARE AND FIRMWARE

This manual is valid with the following hardware and firmware versions:

| Hardware | Firmware |
| --- | --- |
| P2X8C4M64P | Rev B/C |

#### CREDITS

Authorship: Jeff Martin • Format & Editing: Stephanie Lindsay • Technical Graphics: Michael Mulholland

With many thanks to everyone in the Propeller Community and staff at Parallax Inc.

# 

# TABLE OF CONTENTS

[**PREFACE**](#_cxrs8kmh359b)[**7**](#_cxrs8kmh359b)

[**CONVENTIONS**](#_12fh0v8mc8fz)[**7**](#_12fh0v8mc8fz)

[**OVERVIEW**](#_jlu3igid4m1y)[**8**](#_jlu3igid4m1y)

[Specifications](#_dai42k23lnm8) [9](#_dai42k23lnm8)

[Package Description](#_gt5bx8t7k3jo) [9](#_gt5bx8t7k3jo)

[Hardware Connections](#_2hz2drntravb) [11](#_2hz2drntravb)

[Operation](#_9egcmolt3hr9) [11](#_9egcmolt3hr9)

[Boot Up](#_k13wh1l3u0i1) [11](#_k13wh1l3u0i1)

[Runtime](#_b4czqsuc63ti) [12](#_b4czqsuc63ti)

[Shutdown](#_41et468s49cq) [12](#_41et468s49cq)

[Rebooting](#_96ldkh1txyzb) [13](#_96ldkh1txyzb)

[System Clock](#_fzfqcrgwor5b) [13](#_fzfqcrgwor5b)

[Memory](#_kbd2rb54dim) [14](#_kbd2rb54dim)

[**COGS (PROCESSORS)**](#_g0pfct476flb)[**14**](#_g0pfct476flb)

[Cog Memory](#_2jerhz1pjpim) [15](#_2jerhz1pjpim)

[Register RAM](#_t3mdwg53x91p) [15](#_t3mdwg53x91p)

[General Purpose Registers](#_at1mpulv79us) [15](#_at1mpulv79us)

[Dual-Purpose Registers](#_7abpwopg91ym) [15](#_7abpwopg91ym)

[Special-Purpose Registers](#_1shqk4lnz3zl) [16](#_1shqk4lnz3zl)

[Lookup RAM](#_g01pdnt3dk21) [16](#_g01pdnt3dk21)

[Scratch Space](#_73jkkdy9xtrr) [16](#_73jkkdy9xtrr)

[Paired-Cog Communication Mechanism](#_tld16mvlxxxc) [16](#_tld16mvlxxxc)

[Instruction Pipeline](#_d6a33r7wy3h8) [16](#_d6a33r7wy3h8)

[Instruction Stages](#_5foqcyrb9yic) [17](#_5foqcyrb9yic)

[Pipeline](#_v21lql3bhoqu) [18](#_v21lql3bhoqu)

[Wait (Pipeline Stall)](#_79rk4wtiqx9i) [19](#_79rk4wtiqx9i)

[Branch (Pipeline Flush)](#_8kf9cu23m0zh) [20](#_8kf9cu23m0zh)

[Execution](#_2na7xdhgylzo) [21](#_2na7xdhgylzo)

[Register Execution](#_unkhs3434cx4) [21](#_unkhs3434cx4)

[Lookup Execution](#_sip3znh0gf5f) [21](#_sip3znh0gf5f)

[Hub Execution](#_jqdnlg7uy7i0) [21](#_jqdnlg7uy7i0)

[Starting And Stopping Cogs](#_da1uvnijjbyr) [21](#_da1uvnijjbyr)

[Cog Attention](#_vew0a4demzg8) [22](#_vew0a4demzg8)

[System Counter](#_p9jolctm7tnz) [22](#_p9jolctm7tnz)

[Pseudo-Random Number Generator](#_w0d21bfvu5vv) [23](#_w0d21bfvu5vv)

[**HUB**](#_ffgg94dq5jmx)[**24**](#_ffgg94dq5jmx)

[Hub RAM](#_3kmq5bt13sms) [24](#_3kmq5bt13sms)

[Random Access](#_w7cn8cwckqht) [25](#_w7cn8cwckqht)

[Sequential Access](#_575uxxbguubi) [25](#_575uxxbguubi)

[Protected RAM](#_zb1tt9xbffh0) [25](#_zb1tt9xbffh0)

[System Clock Configuration](#_15fgkp930ean) [26](#_15fgkp930ean)

[PLL Example](#_rtz6oinudn49) [27](#_rtz6oinudn49)

[Locks (Semaphores)](#_5ra3z1f8dfp8) [28](#_5ra3z1f8dfp8)

[Lock Usage](#_qm1a3yc11h3i) [28](#_qm1a3yc11h3i)

[CORDIC Solver](#_iuvmkq25g2a2) [29](#_iuvmkq25g2a2)

[Multiply](#_nl9e8nr2tade) [29](#_nl9e8nr2tade)

[Divide](#_kx8d8cnmoiaw) [29](#_kx8d8cnmoiaw)

[Square Root](#_kntpsgy4veqt) [30](#_kntpsgy4veqt)

[Rotation](#_m9zo223xec2s) [30](#_m9zo223xec2s)

[Cartesian to Polar](#_ggvu5xig3t9a) [30](#_ggvu5xig3t9a)

[Polar to Cartesian](#_h0cx7gb9ecfq) [30](#_h0cx7gb9ecfq)

[Integer to Logarithm](#_atz9l1jtixiu) [30](#_atz9l1jtixiu)

[Logarithm to Integer](#_twkpx8ws37vd) [31](#_twkpx8ws37vd)

[**SMART I/O PINS**](#_whuv85gp7nps)[**31**](#_whuv85gp7nps)

[I/O Pin Circuit](#_t13e9n7z5wp7) [31](#_t13e9n7z5wp7)

[Direction and State](#_p2sdunz9fxpm) [32](#_p2sdunz9fxpm)

[Pin Modes](#_jxahsscq2nua) [32](#_jxahsscq2nua)

[Equivalent Schematics for Each Unique I/O Pin Configuration](#_6z9hr7z553r3) [36](#_6z9hr7z553r3)

[I/O Pin Timing](#_drp9s9vw79ai) [42](#_drp9s9vw79ai)

[Smart Modes](#_msbxl3vp75z) [43](#_msbxl3vp75z)

[Smart Pin Off; Default (%00000)](#_uz8yatawtj0x) [45](#_uz8yatawtj0x)

[Long Repository (%00001..%00011 and not DAC\_MODE)](#_389j2sifyw3d) [45](#_389j2sifyw3d)

[DAC Noise (%00001 and DAC\_MODE)](#_pldewc5frxnp) [46](#_pldewc5frxnp)

[DAC 16-Bit With Noise Dither (%00010 and DAC\_MODE)](#_ccftirya86pz) [46](#_ccftirya86pz)

[DAC 16-Bit With PWM dither (%00011 and DAC\_MODE)](#_ujo617c8c6ef) [46](#_ujo617c8c6ef)

[Pulse/Cycle Output (%00100)](#_pyq9vdjyr3nf) [46](#_pyq9vdjyr3nf)

[Transition Output (%00101)](#_m2xocuv28l0p) [47](#_m2xocuv28l0p)

[NCO Frequency (%00110)](#_5089qnhgckp9) [47](#_5089qnhgckp9)

[NCO Duty (%00111)](#_tes64dpabufd) [47](#_tes64dpabufd)

[PWM Triangle (%01000)](#_khuaicke4gpr) [47](#_khuaicke4gpr)

[PWM Sawtooth (%01001)](#_ojj92ylxos1v) [48](#_ojj92ylxos1v)

[PWM Switch-Mode Power Supply With Voltage And Current Feedback (%01010)](#_iyi7bgk5e0w2) [48](#_iyi7bgk5e0w2)

[A/B-Input Quadrature Encoder (%01011)](#_vqmzbery6ad9) [49](#_vqmzbery6ad9)

[Count A-Input Positive Edges When B-Input Is High (%01100)](#_6zttwyip9utx) [49](#_6zttwyip9utx)

[Count A-Input Positive Edges; Increment w/B-Input = 1, Decrement w/B-Input = 0 (%01101)](#_k8acld6ipfl6) [49](#_k8acld6ipfl6)

[Count A-Input Positive Edges (%01110 AND !Y[0])](#_o39n85ho5pd9) [49](#_o39n85ho5pd9)

[Increment w/A-Input Positive Edge, Decrement w/B-Input Positive Edge (%01110 AND Y[0])](#_h9vrm8biinn0) [49](#_h9vrm8biinn0)

[Count A-Input Highs (%01111 AND !Y[0])](#_ssvd8afsp4fl) [50](#_ssvd8afsp4fl)

[Increment w/A-Input High, Decrement w/B-Input High (%01111 AND Y[0])](#_groh7zsh3g6n) [50](#_groh7zsh3g6n)

[Time A-Input States (%10000)](#_wjx73ki24dvv) [50](#_wjx73ki24dvv)

[Time A-Input High States (%10001)](#_ryn87kyb46wk) [50](#_ryn87kyb46wk)

[Time X A-Input Highs/Rises/Edges (%10010 AND !Y[2])](#_xqx7qqsbo9xv) [50](#_xqx7qqsbo9xv)

[Timeout on X Clocks Of Missing A-Input High/Rise/Edge (%10010 AND Y[2])](#_hevxvj4egl1u) [50](#_hevxvj4egl1u)

[Count Time For X Periods (%10011)](#_w31f0bw5a40y) [51](#_w31f0bw5a40y)

[Count State For X Periods (%10100)](#_rfhshokjxp9e) [51](#_rfhshokjxp9e)

[Count Time For Periods In X+ Clock Cycles ( %10101)](#_anpxnz68bcpx) [51](#_anpxnz68bcpx)

[Count States For Periods In X+ Clock Cycles (%10110)](#_mfniwooksdz8) [51](#_mfniwooksdz8)

[Count Periods For Periods In X+ Clock Cycles (%10111)](#_d06gvojuosri) [51](#_d06gvojuosri)

[ADC Sample/Filter/Capture, Internally Clocked (%11000)](#_gwuggjybik6u) [52](#_gwuggjybik6u)

[ADC Sample/Filter/Capture, Externally Clocked (%11001)](#_whxbrxc3tgf7) [52](#_whxbrxc3tgf7)

[About SINC2 and SINC3 filtering](#_459yzhwhu1dr) [52](#_459yzhwhu1dr)

[SINC2 Sampling Mode (%00)](#_3zzx6pjibz0h) [53](#_3zzx6pjibz0h)

[SINC2 Filtering Mode (%01)](#_u6f562dnkly8) [54](#_u6f562dnkly8)

[SINC3 Filtering Mode (%10)](#_oke1ofm1w3r2) [54](#_oke1ofm1w3r2)

[Bitstream Capturing Mode (%11)](#_2ma4ebs403ko) [55](#_2ma4ebs403ko)

[ADC Scope With Trigger (%11010)](#_g52hccxy3lz3) [55](#_g52hccxy3lz3)

[SCOPE Data Pipe](#_22eo5uny6kuj) [56](#_22eo5uny6kuj)

[USB Host/Device (%11011)](#_4oyjpcfjd50a) [57](#_4oyjpcfjd50a)

[Synchronous Serial Transmit (%11100)](#_8klbwkvu2qes) [58](#_8klbwkvu2qes)

[Synchronous Serial Receive (%11101)](#_yilm9fy7a2dt) [59](#_yilm9fy7a2dt)

[Asynchronous Serial Transmit (%11110)](#_xdd3ort80he8) [59](#_xdd3ort80he8)

[Asynchronous Serial Receive (%11111)](#_ak5mbeig8bh1) [60](#_ak5mbeig8bh1)

[**HOST COMMUNICATION**](#_int6wjg53kqw)[**60**](#_int6wjg53kqw)

[Download Propeller Application](#_xpdnr1lhl1pk) [61](#_xpdnr1lhl1pk)

[Multiprogramming](#_kuk4h22mihvn) [62](#_kuk4h22mihvn)

[Loader Parsing Notes](#_vtd61rra3vl8) [62](#_vtd61rra3vl8)

[Prop\_Chk](#_j9c32mb0pw0f) [62](#_j9c32mb0pw0f)

[Prop\_Clk](#_optttfuwpnwi) [62](#_optttfuwpnwi)

[Prop\_Hex](#_dak6znq1xdte) [63](#_dak6znq1xdte)

[Prop\_Txt](#_3s74wjh4k44b) [64](#_3s74wjh4k44b)

[Interactive Mode](#_cqcwjw6x1xsb) [65](#_cqcwjw6x1xsb)

[P2 Monitor](#_ahde1qb9oo0g) [65](#_ahde1qb9oo0g)

[TAQOZ](#_5jrg8inp5nbv) [66](#_5jrg8inp5nbv)

[**PROPELLER 2 RESERVED WORDS (SPIN2 + PASM2)**](#_hg33jkjv6j2l)[**67**](#_hg33jkjv6j2l)

[**GENERAL PURPOSE I/O PIN EXCEPTIONS**](#_7na05pbcjb)[**69**](#_7na05pbcjb)

[**CHANGE LOG**](#_swmvo3n67qaa)[**70**](#_swmvo3n67qaa)

[**PARALLAX INCORPORATED**](#_cenlv5sy0ptb)[**70**](#_cenlv5sy0ptb)

# 

# PREFACE

Thank you for exploring the Propeller microcontroller! Here you will learn about the second member in the Propeller family– the Propeller 2 (P2X8C4M64P)

The Propeller 2 is the culmination of countless ideas, wishes, suggestions, and intense work by Parallax's Chip Gracey and the dedicated Propeller Community of engineers and makers. It is perhaps the most openly-designed microcontroller; constantly revised and discussed on the [public Propeller forums](https://forums.parallax.com/categories/propeller-2-multicore-microcontroller), with interim designs released as FPGA-runnable images and direct community efforts making it all the way to final silicon available today.

This manual is an in-depth description of the concepts, features, and hardware of the Propeller 2 multicore microcontroller. It serves as a reference beyond that of the Propeller 2 Datasheet. Wherever code is needed to demonstrate a hardware feature, this manual will only use PASM2 (the core language). Other languages like Spin2 and C have similar capabilities— refer to the desired language documentation as needed.

For additional documentation and resources, including programming tools, visit [www.parallax.com/P2](http://www.parallax.com/P2). The latest version of this manual, along with links to a commentable Google Doc version, are available from the Documentation section. In addition, there are links to more in-depth references for the Propeller 2 and its Spin2 and PASM2 languages, which may include commentable Google Docs.

# CONVENTIONS

* % - indicates a binary number (containing the digits 0 and 1, and underscore "\_" characters)
  + ex: %0101 and %11000111
* $ - indicates a hexadecimal number (containing the digits 0–9, A–F, and underscore "\_" characters)
  + ex: $2AF and $0D816
* \_ - (underscore) is a visual separator in numbers and in certain symbols. Occasionally they are used as the leading or trailing character of a symbol to make it unique while keeping it similar to a same-named symbol that may already exist, or may be used in a syntax description to separate words of an all-lowercase phrase meant to be replaced by the reader when typing. In numbers (i.e. decimal, hexadecimal, and binary values) they are group indicators that may separate natural boundaries (like groups of 3 digits in decimal or 8 bits in binary) or may separate context-specific fields of a value (like smart mode bits and drive level bits in an I/O pin configuration value)
  + ex: %00111010\_01011101 and $4C1F\_0D816 and 2\_328\_476
  + ex: <low\_byte> and IF\_C\_AND\_Z and ZSTR\_
* x - indicates a group of symbols where the x-part can vary
  + ex: INx means both INA and INB
  + ex: RDxxxx / WRxxx means RDBYTE, RDWORD, etc. and WRBYTE, WRWORD, etc.

-- or --

x - indicates a *don't care* bit in a binary number (a bit that can be 0 or 1 without affecting the execution context nor the explanation)

* + ex: %x\_1\_xxx0 and %xxxx10
* ALL CAPS or ALL\_CAPS - indicates the item has special meaning, such as a reference to a label in a diagram, or a component in the Propeller 2, or is a predefined symbol (reserved word) such as a command, condition, register name, etc.
* <all\_lowercase> - indicates an item meant to be completely replaced by the user when typing code; the phrase in brackets "<>" describes the intent of the specific item.

# OVERVIEW

The Propeller microcontroller family provides high-speed processing for embedded systems while maintaining low current consumption and a small physical footprint. The Propeller provides flexibility and power through its multiple processors (called cogs) that perform simultaneous, independent or cooperative tasks, all while being easy to learn and utilize.

The Propeller 2 frees application developers from common complexities of embedded systems programming:

* Application design is flexible. Every processor (cog) has the same capabilities; no special-use cases.
* Most I/O pins have the same strengths; there are few location limits. Prototype with convenience, then produce with convenience, swapping pin responsibilities at-will with ease.
* Asynchronous events are easy to handle. Assign a dedicated cog to handle such an event, leaving other cog(s) free to perform synchronous or independent processes, or split a cog's responsibilities between synchronous tasks and asynchronous events using interrupts.
* Propeller Assembly language features conditional execution, optional result writing, loop-optimized timing, and runtime-selectable instruction skipping to provide fast, consistent timing and tight, multipurpose code that is capable of jitter-free event handling.

The Propeller 2 (P2X8C4M64P) microcontroller architecture consists of 8 identical 32-bit processors (cogs), each with their own RAM, which connect to a common hub and I/O pins. The hub provides 512 KB of shared RAM, a CORDIC math solver, and housekeeping facilities. The architecture includes 64 smart I/O pins, each capable of many autonomous analog and digital functions.

The Propeller 2's assembly language (PASM2) features per-instruction conditional execution, special looping mechanisms, and pattern-based instruction skipping to encourage fast, compact code.

The Propeller 2's high-level language (Spin2) provides fast, interpreted code that builds upon the power of the hardware and PASM2. The Spin2 language naturally encourages structured, yet terse constructs organized as methods grouped within reusable objects. Spin2 objects include other existing objects to extend their own functionality through the reuse of proven, open-source code.

Propeller 2 Applications, when developed using Spin2 and/or PASM2, integrate one or more objects into a complete program. Applications perform their duty by executing designated assignments within at least one cog (or across as many as eight); starting and stopping multiple cogs, utilizing raw I/O and Smart I/O, digital and analog signaling, streamers, and other built-in hardware as-needed to get the job done. This is how Parallax tools construct and use Propeller 2 Applications— there are other options made available by Propeller Community members featuring different languages, concepts, tools, and build techniques. Parallax lists other options in the Programming Tools section of the [Propeller 2 website](https://www.parallax.com/p2).

Each cog sits dormant until called into action—referred to as "launching a cog"—at which point the cog executes its given code independently from, and in parallel alongside, other active cogs. If needed, cogs may share information and coordinate actions together, either indirectly via shared [Hub RAM](#_3kmq5bt13sms), [locks](#_5ra3z1f8dfp8), or [I/O pins](#_whuv85gp7nps), or directly via paired [Lookup RAM](#_g01pdnt3dk21) or [cog attention](#_vew0a4demzg8) signals. Cogs may monitor and control I/O pins directly or can choose to employ smart modes (automated state machines) for sophisticated signaling. Each cog has independent access to every I/O pin at all times, though the cog collective ultimately decides the direction and state of each.

The Hub provides coordinated access to Hub RAM, the CORDIC math solver, and system management and configuration features. It serves a traffic management role, ensuring that any given unique element of any exclusive resource is not accessed simultaneously by multiple cogs. In most cases, cogs must wait their turn (once every eight clock cycles) to read or modify a shared element; however, even faster access to Hub RAM and the CORDIC solver are possible using a special feature.

## Specifications

| Propeller 2 Specifications | |
| --- | --- |
| **Feature** | **Specification** |
| Model | P2X8C4M64P |
| Power | 1.8 V Core, 3.3 V I/O |
| Internal Oscillator | ~24 MHz or ~20 kHz |
| External Clock | 10 - 20 MHz crystal (P2 Clock PLL enabled) or 0 to 180 MHz (nominal) clock oscillator |
| Nominal System Clock Speed | 180 MHz @ 105 ℃ |
| Number of clock modes | 6 + PLL ÷/× & 2 OSC load options |
| Cogs (cores) | 8 identical |
| Internal execution speed | 0 to 720 MIPS (90 MIPS/cog) @ 180 MHz |
| Cog RAM | 512 longs (Register RAM) + 512 longs (Lookup RAM) |
| Hub RAM | 512 KB (byte/word/long-addressable) |
| ROM | 16 KB (Bootloader, P2 Monitor debug interface, and TAQOZ (Forth) command interface) |
| I/O Pins | 64; each featuring digital and analog signalling plus internal smart circuits |
| Max current per I/O | +/- 30mA |
| Inter-cog communication | Hub RAM, Lookup RAM, Attention Signal, or External I/O |
| Assembly language (PASM2) | 358 instructions |
| Interpreted languages | Spin2 (Propeller Tool), TAQOZ (built-in), MicroPython, or Forth via community tools |
| Compiled languages | Spin (FlexGUI community Tool), BASIC, C/C++ (community tools) |
| PASM2 execution memory | Register RAM + Lookup RAM + Hub RAM |
| Spin2 execution memory | Hub RAM |

For feature highlights, see also the [Propeller 2 website](https://www.parallax.com/p2) and P2 Datasheet for feature highlights.

## Package Description

The P2X8C4M64P microcontroller is an exposed-pad TQFP-100 package. It contains 8 cogs, 512 KB of Hub RAM, and 64 Smart I/O pins. Refer to the P2 Datasheet for package dimensions.

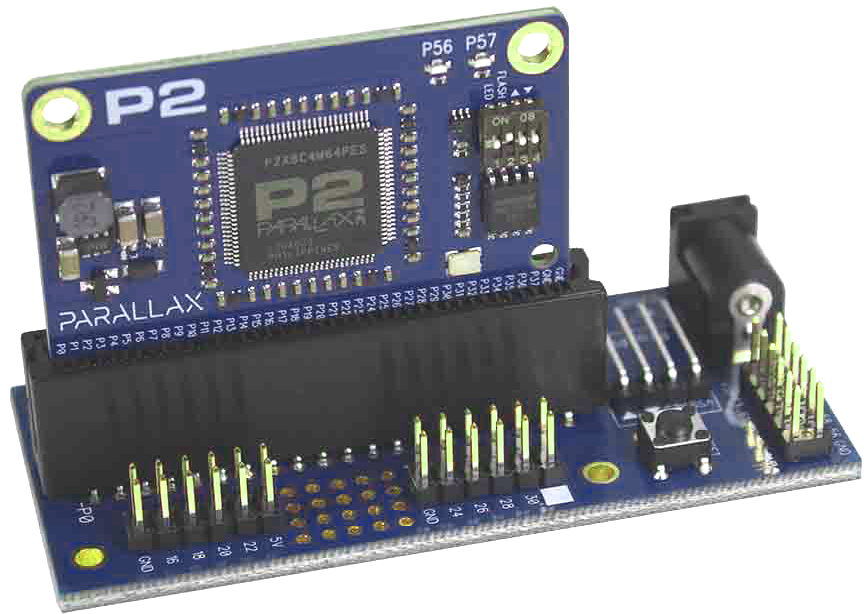
| Part Number Legend | | | |
| --- | --- | --- | --- |
| **P2X** | **8C** | **4M** | **64P** |
| Propeller 2 | 8 cogs (processors) | 4 Mbit Hub RAM (512 KB) | 64 smart I/O pins |

| Pin Descriptions | | | |
| --- | --- | --- | --- |
| **Pin Name** | **Direction** | **V (typ)** | **Description** |
| GND  [not shown] | - | 0 | Ground for core and smart pins; [not shown here] internally connected to underside exposed pad. Connect to ground plane for thermal dissipation. |
| TEST | I | 0 | Tied to ground |
| VDD | - | 1.8 | Core power |
| P0-63 | I/O | 0 to 3.3 | Smart pins; P58-P63 serve in [the boot process](#_9egcmolt3hr9), then general purpose after. |
| Vxxyy | - | 3.3 | Power for smart pins in groups of 4: Pxx through Pyy |
| XO | O | - | Crystal Output. Provides feedback for an external crystal, or may be left disconnected depending on CLK Register settings. No external resistors or capacitors are required. |
| XI | I | - | Crystal Input. Can be connected to output of crystal/oscillator pack (with XO left disconnected), or to one leg of crystal (with XO connected to the other leg of crystal or resonator) depending on CLK Register settings. No external resistors or capacitors are required. |
| RESN | I | 0 | Reset (active low). When low, resets the Propeller: all cogs disabled and I/O pins floating. Propeller restarts 3 ms after RESn transitions from low to high. Connect to a resistor to pull up to 3.3 V. |

The Propeller 2's I/O pins (P0-P63) are general purpose; however, some exceptions may apply to certain applications. See [General Purpose I/O Pin Exceptions](#_7na05pbcjb) for more information.

## Hardware Connections

Parallax offers pre-built P2 boards where vital connections are already made for you. Examples include the P2 Edge Module (#P2-EC) and P2 Mini Breakout Board (#64019) pictured here.



Schematics are available for download from the respective product pages. Visit the [Propeller 2 > Hardware](https://www.parallax.com/p2) section of the Parallax online store for development tools. For additional P2 connection details including boot options, see the P2 Datasheet in the Documentation section.

## Operation

There are three states that characterize the Propeller 2 general operation: Boot Up, Runtime, and Shutdown. Each is conceptually distinct, though their behaviors may overlap.

### Boot Up

Upon any power-up, reset (RESn) pin low-to-high, or software reset event:

1. The Propeller 2 delays for 3 ms, engages the fast clock, then loads up Cog 0 with the ROM-resident Bootloader within 2 ms.
2. The Bootloader executes, checks the Boot Pattern on pins P59-P61, and performs the prescribed boot process which may include:
   * interacting with, or receiving a Propeller application from, a host (ex: PC) over serial,
   * fast booting from a connected SPI-based flash chip, or
   * booting from a connected SD card.

| Boot Pattern | | | |
| --- | --- | --- | --- |
| **Set by floating connection 'ƒ', pull-up resistor '⇧', or pull-down resistor '⇩'. Don't care is '⨯'.** | | | |
| **P611** | **P601** | **P591** | **Procedure** |
| **ƒ** | **ƒ** | **ƒ** | Program from serial within 60 s window |
| **⨯** | **⨯** | **⇧** | Program from serial within 60 s window; no flash or microSD card boot |
| **⇧** | **⨯** | **ƒ** | Program from serial within 100 ms or boot from flash. If fails, program from serial within 60 s window. |
| **⇧** | **⨯** | **⇩** | Fast boot from flash; no serial. If it fails, shutdown. |
| **ƒ / ⇩** | **⇧2** | **ƒ** | Boot from microSD card. If fails, program from serial within 60 s window. |
| **ƒ / ⇩** | **⇧2** | **⇩** | Boot from microSD card. If it fails, shutdown. |

1 Development boards with switchable settings may show P61 as "FLASH," P60 omitted, and P59 as "△" and "▽"

2 Built into microSD card

The following connections must be made when a host system, flash memory and/or microSD Card memory is intended for programming or boot up purposes:

| Host Serial and Boot Memory Connections | | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Type** | **P63 (in)** | **P62 (out)** | **P61 (out)** | **P60 (out)** | **P59 (out)** | **P58 (in)** |
| Host Serial | TX (out) | RX (in) |  |  |  |  |
| Flash SPI |  |  | CSn (in) | CLK (in) | DI (in) | DO (out) |
| SD SPI |  |  | CLK (in) | CSn (in) | DI (in) | DO (out) |

### Runtime

In typical operation (above), the Propeller 2 will boot up and run a user's pre-written application in Cog 0. At this point, all further activity is defined by the application where there is complete control over internal clock speed, I/O pin usage and behavior, mix of cogs running, and more. At runtime, Propeller applications have the flexibility to execute full-speed at all times or to carefully manage processing speed, system functions, and current consumption dynamically—they can even willfully shut down partially or completely.

If the boot process didn't result in running a user application, the loader may still operate for some time, waiting for host communication. Most boot patterns on pins P59-P61 feature a serial communication window, allowing the boot process to talk to a host computer over pins P62 and P63. This communication window is used to load new applications or to run interactive sessions with the Propeller 2's built-in systems. See [Host Communication](#_h2tqlzwp68vp).

### Shutdown

Most applications run continuously until power is shut off, though there are cases that are considered to be a *powered shutdown* state. During powered shutdowns, no cogs are running and all I/O pins become high-impedance inputs. Powered shutdown occurs when the power supply remains stable while one of the following happens:

1. the RESn pin goes low, or
2. the boot process fails to load an application or connect to a host, or
3. the user application terminates the last running cog, or
4. the user application requests a reboot (momentary shutdown)

A powered shutdown state may last indefinitely in all but the last case. Boot up begins again when the RESn pin transitions from low to high (case 1), when the Propeller 2 is power-cycled (case 2 and #3), or after an application requests reboot (case 4).

If the boot process is about to shut down the Propeller 2, it first switches to the RCSLOW (~20 kHz) clock, then terminates cog 0 (the only cog that was running), for the lowest powered shutdown state. User applications may not switch to the slowest clock source before terminating the last cog, thus they will have a higher powered shutdown current draw.

Note that Smart I/O operates independent of the cogs and continues while its associated DIR bit is high; however, with all the cogs terminated, all I/O pin DIR bits are naturally low (i.e. set to input) which puts every Smart I/O into its reset state.

### Rebooting

While normally powered, the Propeller 2 reboots if it receives a low pulse on the RESn pin or executes a HUBSET #$1000\_0000 instruction. Both reset methods, external (via RESn pin) and internal (via HUBSET), behave the same; however, the internal reset is not detectable externally using the RESn pin.

**Shared Resources**

The interaction between each cog and the Hub is vital for sharing resources in the Propeller 2. At any given time, the Hub gives a specific cog momentary exclusive access to certain shared resources such as a region of Hub RAM and system configuration settings. This happens for each cog in a “round robin” fashion– timing is consistent regardless of how many cogs are running. Cogs can choose to use or ignore those resources depending on their current needs; often processing internally (in Cog RAM) in parallel and only accessing exclusive resources in bursts.

There are two types of shared resources in the Propeller 2: 1) common, and 2) exclusive. Common resources can be accessed at any time by any number of cogs; theyclude Smart I/O Pins, the System Counter, and the Pseudo-Random Number Generator results. Exclusive resources can also be accessed by each cog, but only by one cog at a time; they include Hub RAM, the CORDIC solver, Lock bits and the seeder functionality for the Pseudo-Random Number Generator. The Hub helps govern access to exclusive elements by granting each cog a turn to use it, one at a time, facilitating atomic operations without any contention. For cases involving multiple elements (ex: a block of Hub RAM locations) where an atomic operation is not intrinsically possible, lock bits can be used to cooperatively share access between cogs. See the [Hub](#_qsf6j4yr0nx4) section for more information.

## System Clock

The System Clock is the central clock source for nearly every component of the Propeller 2. All cogs and I/O pins perform their next step upon the next System Clock's clock edge. The System Clock itself is driven from one of three selectable sources: 1) the Internal RC Oscillator, 2) the Phase-Locked Loop (PLL), or 3) the Crystal Oscillator (an internal circuit that operates an external crystal or receives an external oscillator signal). The PLL uses the Crystal Oscillator as its reference clock input. The System Clock source is selected by the CLK register setting, which is configurable both at compile time and at run time.

The System Clock speed chosen for any Propeller application is of vital importance to timing calculations in code. If coded properly via the clock setting constants (\_clkfreq, \_xinfreq, \_xtlfreq, \_rcslow, or \_rcfast) the compiled clock mode is reflected in clkfreq\_ and clkmode\_. When set via the HUBSET or ASMCLK instructions, the run time CLKFREQ and CLKMODE values reflect the current System Clock speed.

See [System Clock Configuration](#_jk1aj35i4rah) for more information.

## Memory

The Propeller 2 has three memory regions: Register RAM, Lookup RAM, and Hub RAM. Each cog has its own Register RAM and Lookup RAM (collectively called Cog RAM), while the Hub RAM is shared by all cogs.

| Propeller 2 (P2X8C4M64P) RAM Memory Configuration | | | | | |
| --- | --- | --- | --- | --- | --- |
| **Region** | **Depth** | **Width** | **Address Range**  **(Hex)** | **PASM Instruction D/S**  **Address Range (Hex)** | **PC Increment 1** |
| Cog "Register" RAM | 512 | 32 bits | $00000..$001FF | $000..$1FF | 1 |
| Cog "Lookup" RAM | 512 | 32 bits | $00200..$003FF | $000..$1FF | 1 |
| Hub RAM | 524,288 | 8 bits | $00400..$7FFFF | $00000..$7FFFF | 4 |

1 PC is the Program Counter for PASM execution; incrementing relative to width to retrieve 32-bit instructions.

# COGS (PROCESSORS)

The Propeller 2 contains eight (8) processors, called cogs, numbered 0 to 7. Each cog contains the same components, including a Processor block, Cog RAM, Event Tracker, Cog Attention strobes, Streamer, Colorspace Converter, Pixel Mixer, DAC Channels, an I/O Output Register, and an I/O Direction Register. Each cog is designed exactly the same and can run tasks independently from the others.

All eight cogs are driven from the same clock source, the [System Clock](#_fzfqcrgwor5b), so they each maintain the same time reference and all active cogs execute instructions simultaneously. They also all have access to the same [shared resources](#_oiu4it94h7rh), like I/O pins, Hub RAM, the System Counter, and CORDIC math solver.

Cogs can be started and stopped at-will, performing independent or cooperative tasks simultaneously. Regardless of the nature of their use, the Propeller application developer has full control over how and when each cog is employed; there is no compiler-driven or operating system-driven splitting of tasks between multiple cogs. This empowers the developer to deliver absolutely deterministic timing, power consumption, and response to the embedded application.

## 

## Cog Memory

Each cog has its own internal RAM that it uses to execute code and to store and manipulate data independent of every other cog. This internal RAM is organized into two contiguous blocks of 512 longs (512 x 32), called Register RAM and Lookup RAM, each with special attributes. See [RAM Memory Configuration](#_h78p5tez6kv0).

#### 

Note that $1FE (INA) and $1FF (INB) are also the debug interrupt call address and return address, respectively.

### Register RAM

Each cog's primary 512 x 32-bit dual-port Register RAM (Reg RAM for short) provides for code execution, fast direct register access, and special use. It is read and written as longs (4 bytes) and contains general purpose, dual-purpose, and special-purpose registers.

#### General Purpose Registers

Register RAM locations $000 through $1EF are general-purpose registers for code and data usage.

#### Dual-Purpose Registers

Register RAM locations $1F0 through $1F7 may either be used as general-purpose registers, or may be used as special-purpose registers if their associated functions are enabled.

| **Address** | **Name** | **Purpose** |
| --- | --- | --- |
| $1F0  $1F1  $1F2  $1F3  $1F4  $1F5  $1F6  $1F7 | RAM / IJMP3  RAM / IRET3  RAM / IJMP2  RAM / IRET2  RAM / IJMP1  RAM / IRET1  RAM / PA  RAM / PB | Interrupt call address for INT3  Interrupt return address for INT3  Interrupt call address for INT2  Interrupt return address for INT2  Interrupt call address for INT1  Interrupt return address for INT1  CALLD-imm return, CALLPA parameter, or LOC address  CALLD-imm return, CALLPB parameter, or LOC address |

#### 

#### Special-Purpose Registers

RAM registers $1F8 through $1FF give mapped access to eight special-purpose functions. In general, when specifying an address between $1F8 and $1FF, the PASM2 instruction accesses a special-purpose register, *not* just the underlying RAM.

| **Address** | **Name** | **Purpose** |
| --- | --- | --- |
| $1F8  $1F9  $1FA  $1FB  $1FC  $1FD  $1FE  $1FF | PTRA  PTRB  DIRA  DIRB  OUTA  OUTB  INA1  INB2 | Pointer A to Hub RAM  Pointer B to Hub RAM  Output enables (direction bits) for P31..P0  Output enables (direction bits) for P63..P32  Output states for P31..P0  Output states for P63..P32  Input states for P31..P0  Input states for P63..P32 |

1 Also debug interrupt call address

2 Also debug interrupt return address

### Lookup RAM

Each cog's secondary 512 x 32-bit dual-port Lookup RAM (LUT RAM for short) is read and written as longs (4 bytes). It is useful for:

* Scratch space
* Streamer access
* Bytecode execution lookup table
* Smart pin data source
* Paired-Cog communication mechanism
* Code execution

#### Scratch Space

In contrast to Register RAM, the cog cannot directly reference Lookup RAM locations in the majority of its PASM2 instructions. Instead, the desired location(s) must be read or written between Lookup RAM and Register RAM using the RDLUT and WRLUT instructions, respectively. This is synonymous with other hardware architecture's scratch storage using "LOAD" and "STORE" instructions. When using the RDLUT and WRLUT instructions, the Lookup RAM's locations $200..$3FF are addressable as $000..$1FF.

#### Paired-Cog Communication Mechanism

Adjacent cogs whose ID numbers differ by only the LSB (cogs 0 and 1, 2 and 3, etc.) can each allow their Lookup RAMs to be written by the other cog via its local Lookup RAM writes. This allows adjacent cogs to share data very quickly through their Lookup RAMs.

*Warning****:*** Lookup RAM writes from the adjacent cog are implemented on the Lookup RAM's 2nd port. The 2nd port is also shared by the streamer in DDS/LUT modes. If an external write occurs on the same clock as a streamer read, the external write gets priority. It is not intended that external writes would be enabled at the same time the streamer is in DDS/LUT mode.

## Instruction Pipeline

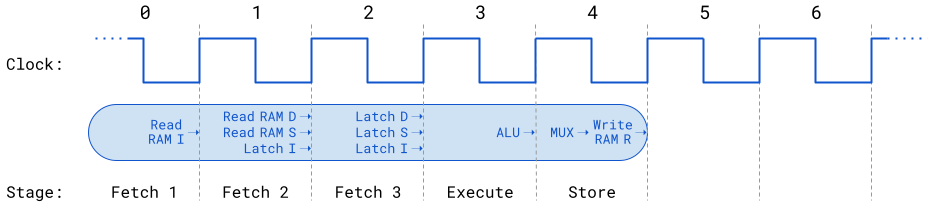
To optimize execution speed, cogs employ a pipelined execution architecture for PASM2. The nature of the pipeline is summarized by these attributes:

* There are five stages of processing per instruction, performed in a minimum of five clock cycles
* Instructions are overlapped to effectively execute in as little as two clock cycles when the pipeline is full
* Branch instructions cause the pipeline to be flushed; the first instruction following the branch will take at least five clock cycles (13 or 14 if branching to a hub address) since the pipeline is refilling
* Any instruction that is conditionally canceled will not execute but will still take effectively two clocks (or at least five clocks, if following a branch) to pass through the pipeline
* If an instruction must wait for a resource, all the following instructions in the pipeline also wait

### Instruction Stages

To understand the pipeline, first consider the process of executing a single PASM2 instruction. An instruction's five stages of processing are illustrated below. Every PASM2 instruction is processed this way— five stages taking at least five clock cycles total. Each stage is completed upon the rising edge of the following clock signal.

###### Isolated Instruction Processing



I = instruction opcode • D = destination operand • S = source operand • ALU = arithmetic logic unit

MUX = result multiplexer • R = result of instruction execution; ALU output value, or Hub result, plus C and Z flags

The first three stages (Fetches) involve reading the 32-bit PASM2 instruction (I) opcode from RAM, latching (saving) the instruction opcode for decoding, and reading/latching the instruction's source (S) and destination (D) values (32-bits each). The final two stages (Execute and Store) perform the instruction's intent by using the arithmetic logic unit (ALU) and writing the resulting 32-bit value and the carry and zero flags if required. At that point (five clock cycles in this case) the instruction is fully executed.

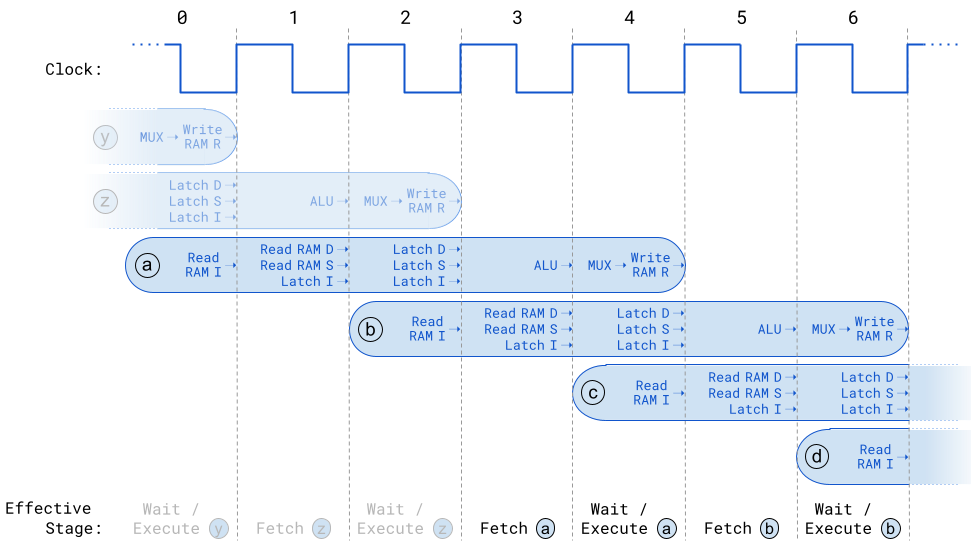
* The MUX gathers all possible results (from the ALU, Hub RAM, etc.) and delivers only what is appropriate for the Write RAM R operation
* The final result value is written and the carry and zero flags are either written or discarded, depending on the specific instruction and given effects (WC / WZ / WCZ)
* As needed for proper processing, an instruction may wait (one or more extra clock cycles are inserted, without any stage advancement) immediately before the final stage

### 

### Pipeline

In the pipeline, instructions are overlapped by three stages, resulting in an effective two-stage execution per instruction known as Fetch and Execute (or Wait). Compare the single instruction illustration above with the multi-instruction pipeline flow below— the instruction above appears in the next illustration with the prefix "a" while others use prefixes "b", "c", etc. This seven-cycle slice of time is processing six contiguous instructions.

###### Instruction Pipeline Flow



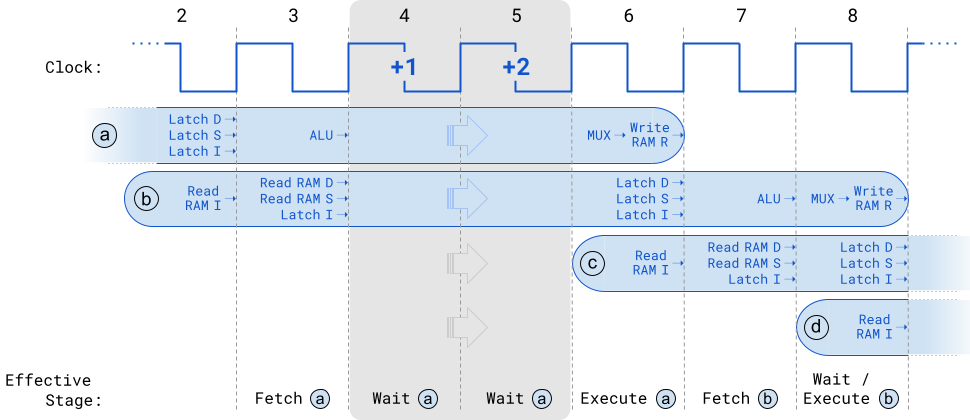
As the pipeline fills (or is full, as shown here) instructions overlap in time by three stages. Each cycle of the instruction pipeline simultaneously processes different stages of two or three contiguous instructions. Here, *instruction a* is read (in the first cycle) at the same moment its two previous instructions (*y* and *z*) write results and latch for later ALU operation, respectively.

### 

### Wait (Pipeline Stall)

When an instruction requires a resource that is not yet available (such as Hub RAM), the whole pipeline delays for additional cycles ahead of the instruction's execute stage. These extra cycles align the target instruction's MUX-update to the delayed resource's moment-of-result while performing no operation in any other instruction. When the resource is ready, processing continues again for all instructions in the pipeline. For example, if *instruction a* needs to wait 2 extra cycles to execute properly, the pipeline flow (above) would be stretched starting at cycle 4; appearing like this:

###### Pipeline Stall



*Instruction a* is delayed 2 cycles (shown in gray) to wait for the result from a non-immediate resource; all further instructions are equally delayed. Hollow arrows represent a shift in time by 2 clocks.

Instead of *Execute a* in cycle 4, two *Wait* cycles occur, delaying the *Write* operation (until the *MUX* has valid results) as well as the latching and reading operations of *instructions b*, *c*, and *d*.

### 

### Branch (Pipeline Flush)

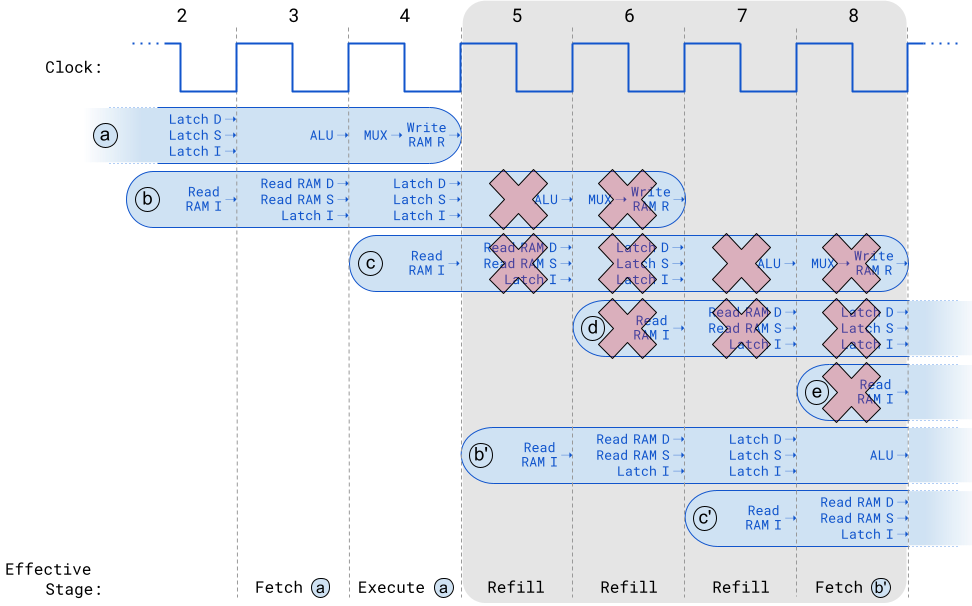
Branch instructions potentially change the path of code execution. A "branch" instruction may be one of a few static operations (such as JMP or CALL) or among many dynamic operations (such as a conditionally-executed JMP or CALL, or a modify/test-and-branch instruction like DJZ or TJNZ). When processing a branch instruction, one of two things will happen.

1. Branch not taken? Execution continues as normal with the instruction following the branch instruction
2. Branch is taken? Execution is diverted (the program counter is immediately changed to point to the branch's target destination), all remaining instruction stages are flushed from the pipeline, and new instructions from the destination begin to refill the pipeline

Due to the pipeline flush and the need to refill it, the instruction following an executed branch will take at least five clock cycles to execute (13 or 14 if branching to a hub address).

In the illustration below, *instruction a* is a "branch taken." Immediately after the *Execute a* stage — all remaining stages from the next four contiguous instructions are flushed (each marked by a red 'X') and new instructions refill those empty pipeline positions.

###### Pipeline Flush



*Instruction a* is an executed branch— changing the program counter at the end of cycle 4, flushing further instruction stages from the pipeline (shown in gray), and refilling the pipeline with instructions from the new (branched to) location. *Instructions b* through *e* (which physically followed *instruction a*) are effectively "canceled" from the pipeline and replaced by *instructions b'* and *c'* from the new code location.

## Execution

Cogs use 20-bit addresses for their program counters (PC); the upper bit is a "don't care" bit - this affords an execution space of up to 512 KB. Depending on the value of a cog's PC, an instruction will be fetched from either its Register RAM, its Lookup RAM, or the Hub RAM. See [RAM Memory Configuration](#_h78p5tez6kv0).

### Register Execution

When the PC is in the range of $00000 to $001FF, the cog fetches instructions from Cog Register RAM. This is referred to as "cog execution mode." There are no special considerations when branching to a cog register address.

### Lookup Execution

When the PC is in the range of $00200 to $003FF, the cog fetches instructions from Cog Lookup RAM. This is referred to as "lut execution mode." There are no special considerations when branching to a cog lookup address.

### Hub Execution

When the PC is in the range of $00400 to $7FFFF, the cog fetches instructions from Hub RAM. This is referred to as "hub execution mode." Special considerations are involved with hub execution.

1. The PC rolling beyond $003FF will not initiate hub execution (it will just wrap back to $00000); a branch must occur to get from register or lookup execution to hub execution.
2. Branching to a hub address takes a minimum of 13 clock cycles. If the instruction being branched to is not long-aligned, one additional clock cycle is required.
3. When executing from Hub RAM, the cog employs the FIFO hardware to spool up instructions so that a stream of instructions will be available for continuous execution. This means the FIFO cannot be used for anything else. So, during hub execution these instructions cannot be used:

RDFAST / WRFAST / FBLOCK

RFBYTE / RFWORD / RFLONG / RFVAR / RFVARS

WFBYTE / WFWORD / WFLONG

XINIT / XZERO / XCONT - when the streamer mode engages the FIFO

It is not possible to execute code from hub addresses $00000 through $003FF, as the cog will instead read instructions from the cog's Register RAM or Lookup RAM as indicated above.

## Starting And Stopping Cogs

Any cog can start or stop any other cog, or restart or stop itself. Each cog has a unique ID which can be used to start or stop it. It is also possible to start free (stopped or never started) cogs, without needing to know their IDs. This way, applications can simply start free cogs, as needed, and as those cogs retire by stopping themselves or getting stopped by others, they return to the pool of free cogs to become available again for restarting.

To start a free cog:

COGINIT id, addr WC '(id=$30) start a free cog at addr, C=0 and id=Cog ID if okay

To (re)start a specific cog:

COGINIT #1, #$100 'load and start cog 1 from hub address $100

To start a cog, passing in a pointer or 32-bit value:

SETQ ptra\_val 'ptra\_val will go into target cog's PTRA register

COGINIT #%0\_1\_0000, addr 'load and start a free cog at addr

To retrieve this cog's ID:

COGID myID 'my cog ID is written to myID

To stop this cog:

COGID myID 'get my ID

COGSTOP myID 'halt myself

## Cog Attention

Each cog can request the attention of other cogs by using the COGATN instruction. One or more of the D operand's lower 8 bits may be set high (1) to signal the corresponding cog or cogs.

COGATN #00001100 'Get attention of cogs 2 and 3

For each high bit, the matching cog sees an *attention* event for POLLATN / WAITATN / JATN / JNATN and for interrupt use. The attention strobe outputs from all cogs are OR'd together to form a composite set of 8 strobes from which each cog receives its particular strobe.

Examples:

POLLATN WC 'has attention been requested?

WAITATN 'wait for attention request

JATN addr 'jump to addr if attention requested

JNATN addr 'jump to addr if attention not requested

In the intended use case, the cog receiving an attention request knows which other cog is strobing it and how to respond. In cases where multiple cogs may request the attention of a single cog, some messaging structure may need to be implemented in Hub RAM to differentiate requests.

## System Counter

The System Counter is a 64-bit free-running counter that increments upon every clock cycle. It is a shared resource, accessible by all cogs at any time, and it's lower 32 bits (usually referred to as CT) serves as the official time reference for many instructions and events. It is often used for brief, relative time measurements; however, since it is cleared to zero upon every power-up/reset, it is also a *system up time* reference.

To read the current System Counter value:

GETCT X 'read lower 32-bits (CT) of system counter into X register

--or--

GETCT X WC 'read upper 32-bits (CTH) of system counter into X register

GETCT Y 'read lower 32-bits (CT) of system counter into Y register

Note: PASM2 instructions that deal with the System Counter only use its lower 32 bits (known as CT)— with the one exception being GETCT. To get the full 64-bit System Counter value (CTH:CT), it is important to read the upper 32-bits first (as shown above) and immediately read the lower 32-bits second. This sequence employs a special mechanism that avoids phase issues; CT (the lower 32-bits) is returned exactly as it was back at the moment in which the upper 32-bits (CTH) had been read. This 64-bit value is handy for calculation of long time frame events (>23.8s @ 180 MHz).

For event handling, there are three hidden registers (CT1, CT2, and CT3) dedicated to short timeframe monitoring (< 23.9s @ 180 MHz). These represent a target moment in time (future 32-bit value; CT), settable via the ADDCTx instructions and used internally by many event instructions.

To mark a moment in time to wait for, use GETCT with ADDCTx and WAITCTx:

GETCT x 'get current CT

ADDCT1 x,#500 'make target 500 cycles later (CT1)

WAITCT1 'wait for CT to pass CT1 target

This can easily be extended to create a 500-cycle activity-loop instead. See ADDCTx / WAITCTx in the [PASM2 Manual](https://www.parallax.com/propeller-2/documentation/).

The event-timing instructions that utilize the System Counter are: ADDCTx, POLLCTx, WAITCTx, JCTx, and JNCTx. In addition, by including a SETQ right before any WAITxxx instruction, a 32-bit *timeout* is created to abort the *wait* in case the target event never arrives.

## Pseudo-Random Number Generator

The Propeller 2 features a pseudo-random number generator (PRNG) based on the Xoroshiro128\*\* algorithm. Note that the "\*\*" is part of the name, indicating the exact variation of the Xoroshiro128 algorithm used.

The Xoroshiro128\*\* PRNG iterates on every clock cycle, generating 64 fresh bits which get spread among all cogs and smart pins. From this 64-bit pool, upon every clock cycle, each cog receives a unique set of 32 different bits (in a scrambled arrangement with some bits inverted) and each smart pin receives a similarly-unique set of 8 different bits. Cogs can read their current 32-bit *pseudo-random* value using the GETRND instruction and directly apply them using the BITRND and DRVRND instructions. Smart pins utilize their 8 bits as noise sources for DAC dithering and noise output.

After reset, the bootloader seeds the Xoroshiro128\*\* PRNG fifty times, each time with 31 bits of thermal noise gleaned from pin 63 while in ADC calibration mode. This establishes a very random seed which the PRNG iterates from, thereafter. There is no need to do this again, but here is how you would do it if 'x' contained a seed value:

SETB x,#31 'set the MSB of x to make a PRNG seed command

HUBSET x 'seed 32 bits of the Xoroshiro128\*\* state

Note: using HUBSET, with D's MSB set, will seed the 128-bit PRNG. This will write all bits of D into 32 bits of the PRNG, affecting 1/4th of its total state. The required high MSB bit in D ensures that the overall state will not go to zero. Because the PRNG's 128 state bits rotate, shift, and XOR against each other, they are thoroughly spread around within a few clocks, so seeding from a fixed set of 32 bits should not pose a limitation on seeding quality.

Note there is also another pseudo-random number feature, accessed via the XORO32 instruction; however it doesn't use the Xoroshiro128\*\* PRNG— instead, it iterates a register value to make a relatively good PRNG sequence under software control.

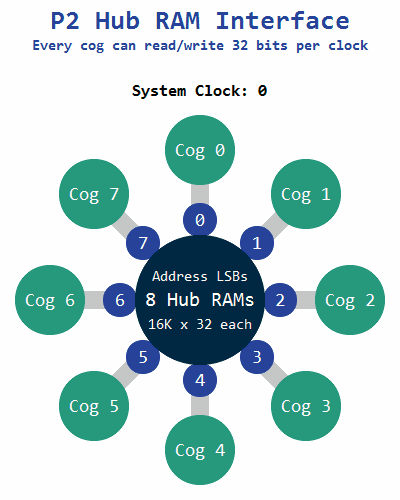
# 

# HUB

While common shared resources, such as I/O Pins, are simultaneously accessible by every cog, exclusive shared resources, such as an individual Hub RAM location, can not be simultaneously accessed without causing problems. The Hub governs access to exclusive resources, giving each cog a turn to safely perform an atomic operation on that resource. On the Propeller 2 (P2X8C4M64P), a given cog receives such access once every eight clock cycles. This moment of access is known as the Hub Access Window.

Atomic operations are elemental in nature (indivisible); each performed in one step with guaranteed isolation from other operations happening at the same time. In any given Hub Access Window, a cog can perform a single atomic operation on a single entity, such as writing a value to one Hub RAM location or setting the system clock speed.

The time-sliced nature of the Hub Access Window applies to every exclusive resource and results in a kind of speed limit for accessing them– each cog must wait its turn among the collective of cogs. Luckily, there are some natural use cases where the Propeller 2 provides optimized access as well; namely, sequential location access of Hub RAM and successive CORDIC math engine use.



## Hub RAM

Hub RAM consists of long elements (32-bits wide) that can be read and written as bytes, words, and longs, in little-endian format. Each cog has access to the entire Hub RAM, though any given RAM location must be used in an orderly, time-sliced fashion. Hub addresses are always byte-oriented and there are no special alignment rules for words and longs in Hub RAM. Cogs can read and write bytes, words, and longs at any hub address, as well as execute instruction longs from any hub address starting at $400 (see [Hub Execution](#_jqdnlg7uy7i0)).

On the Propeller 2 (P2X8C4M64P), the Hub RAM is split into eight slices that are multiplexed among all cogs. Each RAM slice holds every 8th long of the composite Hub RAM. Upon every clock cycle, each cog can access the *next* RAM slice, allowing all cogs simultaneous access to some part of Hub RAM. The Hub RAM Interface diagram illustrates this process conceptually as the collective of RAM slices rotates around, each facing a new cog every clock cycle.

For any target Hub RAM location, the lower 3 bits of its address determines the slice ID (0 through 7) that it resides within. If that slice doesn't happen to be aligned with the cog at the moment it is executing the memory read/write instruction (RDxxxx / WRxxxx), the cog will automatically wait for that slice to come around.

### 

### Random Access

Random accesses of Hub RAM (i.e. using non-contiguous locations) must always align to the Hub Access Window of the RAM slice in question. This means each individual random access (RDxxxx / WRxxxx instruction) may take as many as 16 clock cycles to complete. It is safe for any number of cogs to begin a random access read or write operation on any clock cycle— the Hub will automatically delay each of them individually until they line up with their target addresses' Hub Access Window.

### Sequential Access

By default, sequential access of Hub RAM operates the same as with random access; however, when using either the cog's Hub FIFO interface or fast block move feature, subsequent addresses in the series are available immediately when called for.

Each cog has a Hub FIFO interface which can be set for Hub-RAM-read or Hub-RAM-write operation. This interface allows the cog to either sequentially read or sequentially write the Hub RAM in units of bytes, words, or longs, at any rate up to full speed; one long (32-bits) per clock. Regardless of the transfer frequency or the word size, the FIFO will ensure that the cog's reads or writes are all properly conducted from/to the composite Hub RAM.

Fast block moves can also read/write a sequential series of values (longs only) at one per clock cycle. Either the cog's Reg RAM or LUT RAM may be used as the destination/source. This is achieved by preceding a RDLONG / WRLONG / WMLONG instruction with a SETQ or SETQ2 instruction to specify the number of longs to move using Reg RAM or LUT RAM, respectively.

### Protected RAM

Upon startup/reset, the internal 16 KB ROM contents are copied to the last 16 KB of Hub RAM to bring that code image into addressable memory space. At that point, the ROM code image is accessible in Hub RAM at $7C000–$7FFFF and also outside the 512 KB range, at $FC000–$FFFFF. It is readable and writable in both locations (changes in one also appear in the other), though it can be write-protected using the HUBSET instruction When write-protection is enabled, the image in RAM at $7C000–$7FFFF is hidden (becomes all zeros) and is in-tact but read-only for the running user application in the range $FC000–$FFFFF. The debug mechanism (code that runs during a debug interrupt) uses this area to perform its tasks and is the only code that can continue to read and write that memory despite the engaged write-protection.

## 

## System Clock Configuration

The system clock is the time base for all internal components and can be configured for several modes.

* Direct from internal slow clock (RCSLOW); a ~20 kHz oscillator is intended for low-power operation
* Direct from internal fast clock (RCFAST); a 20 MHz+ oscillator designed for minimum 20 MHz operation
* Direct from XI pin; driven externally via a clock oscillator or a crystal oscillator (w/crystal-feedback on XO)
* Enhanced from XI pin; driven externally via a clock oscillator or a crystal oscillator (w/crystal-feedback on XO) and the XI signal internally modified by the PLL (phase-locked loop), often to accelerate the frequency

If the XI and XO pins (eXternal Input/Output) are used for clocking via an attached 10 – 20 MHz crystal oscillator, internal loading caps can also be enabled on XI and XO for crystal impedance matching.

If the XI pin is used as a clock input or crystal oscillator input, its frequency can be modified through the internal phase-locked loop (PLL). The PLL divides the XI pin frequency from 1 to 64, then multiplies the resulting frequency from 1 to 1024 in the voltage-controlled oscillator (VCO). The VCO frequency can either be used directly (i.e. divided by 1) or divided down by any *even* value from 2 to 30 to get the final PLL clock frequency which becomes the system clock frequency.

The system clock is configured by the running Propeller 2 application using the HUBSET instruction in the following format. The four LSBs are all that are needed to switch among clock sources and select all but the PLL settings.

HUBSET ##%0000\_000E\_DDDD\_DDMM\_MMMM\_MMMM\_PPPP\_CCSS 'set clock mode

The bit fields (E, D, M, P, C, and S) are described in the following tables.

| **PLL Setting** | **Value** | **Effect** | **Notes** |
| --- | --- | --- | --- |
| %E | 0/1 | PLL off/on | XI input must be enabled by %CC. Allow 10 ms for crystal+PLL to stabilize before switching over to PLL clock source. |
| %DDDDDD | 0..63 | 1..64 division of XI pin frequency | This divided XI frequency feeds into the phase-frequency comparator's 'reference' input. |
| %MMMMMMMMMM | 0..1023 | 1..1024 division of VCO frequency | This divided VCO frequency feeds into the phase-frequency comparator's 'feedback' input. This frequency division has the effect of *multiplying* the divided XI frequency (per %DDDDDD) inside the VCO. The VCO frequency should be kept within 100 MHz to 350 MHz. |
| %PPPP | 0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | VCO / 2  VCO / 4  VCO / 6  VCO / 8  VCO / 10  VCO / 12  VCO / 14  VCO / 16  VCO / 18  VCO / 20  VCO / 22  VCO / 24  VCO / 26  VCO / 28  VCO / 30  VCO / 1 | This divided VCO frequency is selectable as the system clock when SS = %11. |

| **%CC** | **XI status** | **XO status** | **XI / XO impedance** | **XI / XO loading caps** |
| --- | --- | --- | --- | --- |
| %00 | ignored | float | Hi-Z | OFF |
| %01 | input | 600-ohm drive | 1M-ohm | OFF |
| %10 | input | 600-ohm drive | 1M-ohm | 15pF per pin |
| %11 | input | 600-ohm drive | 1M-ohm | 30pF per pin |

| **%SS** | **Clock Source** | **Notes** |
| --- | --- | --- |
| %11 | PLL | CC != %00 and E=1, allow 10ms for crystal+PLL to stabilize before switching to PLL |
| %10 | XI | CC != %00, allow 5ms for crystal to stabilize before switching to XI pin |
| %01 | RCSLOW | ~20 kHz, can be switched to at any time, low-power |
| %00 | RCFAST | 20 MHz+1, can be switched to at any time, used on boot up |

1 Designed to run a least 20 MHz, worst case, to accommodate 2 MBaud serial loading during boot

**WARNING**: Incorrectly switching away from the PLL setting (%SS = %11) can cause a glitch which will hang the clock circuit. To safely switch, always start by switching to an internal oscillator using either HUBSET #$F0 (for RCFAST) or HUBSET #$F1 (for RCSLOW).

### PLL Example

The PLL divides the XI pin frequency from 1 to 64, then multiplies the resulting frequency from 1 to 1024 in the VCO. The VCO frequency can be used directly, or divided by 2, 4, 6, ...30, to get the final PLL clock frequency which can be used as the system clock.

The PLL's VCO is designed to run between 100 MHz and 200 MHz and should be kept within that range.

|  |
| --- |

Let's say you have a 20 MHz crystal attached to XI and XO and you want to run the Prop2 at 148.5 MHz. You could divide the crystal by 40 (%DDDDDD = 39) to get a 500 kHz reference, then multiply that by 297 (%MMMMMMMMMM = 296) in the VCO to get 148.5 MHz. You would set %PPPP to %1111 to use the VCO output directly. The configuration value would be %1\_100111\_0100101000\_1111\_10\_11. The last two 2-bit fields select 15 pf crystal mode and the PLL. In order to realize this clock setting, though, it must be done over a few steps:

HUBSET #$F0 'set 20 MHz+ (RCFAST) mode

HUBSET ##%1\_100111\_0100101000\_1111\_10\_00 'enable crystal+PLL, stay in RCFAST mode

WAITX ##20\_000\_000/100 'wait ~10ms for crystal+PLL to stabilize

HUBSET ##%1\_100111\_0100101000\_1111\_10\_11 'now switch to PLL running at 148.5 MHz

The clock selector controlled by the %SS bits has a deglitching circuit which waits for a positive edge on the old clock source before disengaging, holding its output high, and then waiting for a positive edge on the new clock source before switching over to it. It is necessary to select mode %00 or %01 while waiting for the crystal and/or PLL to settle into operation, before switching over to either.

## Locks (Semaphores)

For application-defined cog coordination, the hub provides a pool of 16 semaphore bits, called locks. Cogs may use locks, for example, to manage exclusive access of a resource or to represent an exclusive state, shared among multiple cogs. What a lock represents is completely up to the application using it; they are a means of allowing one cog at a time the exclusive status of 'owner' of a particular lock ID. In order to be useful, all participant cogs must agree on a lock's ID and what purpose it serves.

The LOCK instructions are:

LOCKNEW D {WC}  
LOCKRET {#}D  
LOCKTRY {#}D {WC}  
LOCKREL {#}D {WC}

### Lock Usage

In order to use a lock, one cog must first allocate a lock from the lock pool with LOCKNEW and communicate that lock's ID with other cooperative cogs. If successful, LOCKNEW returns the lock ID in D and, if WC is given, will clear C (0) if a lock was available or set C (1) if all locks were already allocated. A cog may allocate more than one lock if needed.

Cooperative cogs then use LOCKTRY to *take* ownership of the state which that lock represents. The Hub arbitrates lock ownership in a round-robin fashion (as with all exclusive resources) so any cog waiting to take ownership of a lock will get its fair turn and only one will be awarded ownership at any given time. Here's an example of looping until ownership of a lock is successful:

'Keep trying to capture lock until successful

.try LOCKTRY write\_lock WC

IF\_NC JMP #.try

Once lock ownership is successful, the cog should perform the task the lock was designed to protect while all other cogs in this cooperative arrangement should be busy with other tasks or waiting for lock ownership approval in a loop similar to the above. It is recommended that lock-protected steps be intentionally swift so as not to hold up other cogs waiting for ownership to perform their lock-protected counter steps.

After the designated task is performed, the cog must immediately use LOCKREL to *release* ownership of the lock; allowing other cogs potential ownership of the lock. Only the cog that has taken ownership of the lock can release it; however, a lock will also be implicitly released if the cog that's holding ownership is stopped (COGSTOP), restarted (COGINIT), or if LOCKRET is executed for that lock.

If the lock is no longer needed by the application (i.e. no cogs need it for the designed purpose), it may be returned to the unallocated lock pool by executing LOCKRET. Any cog can return a lock, even if it wasn't the cog that allocated it with LOCKNEW.

## 

## CORDIC Solver

The Hub contains a 54-stage pipelined CORDIC solver (Coordinate Rotation Digital Computer) that can compute the following functions for all cogs:

* [**Multiply**](#_nl9e8nr2tade)**:** 32 x 32 unsigned multiply with 64-bit product
* [**Divide**](#_kx8d8cnmoiaw)**:** 64 / 32 unsigned divide with 32-bit quotient and 32-bit remainder
* [**Square Root**](#_kntpsgy4veqt)**:** root of 64-bit unsigned value with 32-bit result
* [**Rotation**](#_m9zo223xec2s)**:** 32-bit signed (X, Y) rotation around (0, 0) by a 32-bit angle with 32-bit signed (X, Y) results
* [**Cartesian to Polar**](#_ggvu5xig3t9a)**:** 32-bit signed (X, Y) to 32-bit (length, angle) cartesian to polar operation
* [**Polar to Cartesian**](#_h0cx7gb9ecfq)**:** 32-bit (length, angle) to 32-bit signed (X, Y) polar to cartesian operation
* [**Integer to Logarithm**](#_atz9l1jtixiu)**:** 32-bit unsigned integer to 5:27-bit logarithm
* [**Logarithm to Integer**](#_twkpx8ws37vd)**:** 5:27-bit logarithm to 32-bit unsigned integer

Each cog can issue one CORDIC instruction per its hub access window, which occurs once every eight clocks, and retrieve the result 55 clocks later via the GETQX and GETQY instructions. For faster throughput, cogs can take advantage of the hub access window and CORDIC pipeline relationship to issue a stream of CORDIC instructions interleaved with retrieving corresponding results, achieving up to one CORDIC result every eight clocks. Each cog's active CORDIC instructions and forthcoming results are completely isolated from each other, as well as from other cogs; however, each result must be retrieved on time or else it will be overwritten by the following result, if any. As soon as a CORDIC result is calculated, it is automatically *deposited* into the cog's hidden "X" and "Y" registers, ready to be picked up at any moment by the GETQX and GETQY instructions (independent of the hub access window).

### Multiply

Use the QMUL instruction to multiply two unsigned 32-bit numbers together and retrieve the CORDIC result with the GETQX and GETQY instructions (for lower and upper long, respectively). QMUL will wait for the hub access window and GETQX / GETQY will wait for the CORDIC results.

QMUL D/#,S/# - Multiply D by S

To get the results (these instructions wait for the CORDIC results):

GETQX lower\_long

GETQY upper\_long

### Divide

Use the QDIV or QFRAC instruction (either with optional preceding SETQ instruction) to divide a 64-bit numerator by a 32-bit denominator, then retrieve the CORDIC results with the GETQX and GETQY instructions (f$or quotient and remainder, respectively). QDIV / QFRAC will wait for the hub access window and GETQX / GETQY will wait for the CORDIC results.

QDIV D/#,S,# - Divide {$00000000:D} by S

...or...

SETQ Q/# - Set top part of numerator

QDIV D/#,S,# - Divide {Q:D} by S

...or...

QFRAC D/#,S,# - Divide {D:$00000000} by S

...or...

SETQ Q/# - Set bottom part of numerator

QFRAC D/#,S,# - Divide {D:Q} by S

...and to get the results:

GETQX quotient

GETQY remainder

### Square Root

Use the QSQRT instruction on a 64-bit number and retrieve the square root CORDIC result with the GETQX instruction. QSQRT will wait for the hub access window and GETQX will wait for the CORDIC results.

QSQRT D/#,S,# - Compute square root of {S:D}

GETQX root

### Rotation

Use the SETQ instruction followed by the QROTATE instruction to rotate a 32-bit signed Y and X point pair by an unsigned 32-bit angle and retrieve the CORDIC results with the GETQX and GETQY instructions for X and Y, respectively. For the angle (in S), $00000000..$FFFFFFFF = 0..359.9999999 degrees. QROTATE will wait for the hub access window and GETQX / GETQY will wait for the CORDIC results.

SETQ Q/# - Set Y

QROTATE D/#,S,# - Rotate (D,Q) by S

GETQX X

GETQY Y

### Cartesian to Polar

Use the QVECTOR instruction to convert a (X, Y) cartesian coordinate into (length, angle) polar coordinate and retrieve the CORDIC results with the GETQX and GETQY instructions (for length and angle, respectively). QVECTOR will wait for the hub access window and GETQX / GETQY will wait for the CORDIC results.

QVECTOR D/#,S,# - (X=D,Y=S) cartesian into (length,angle) polar

GETQX length

GETQY angle

### Polar to Cartesian

Use the QROTATE instruction to convert a (length, angle) polar coordinate into (X, Y) cartesian coordinate and retrieve the CORDIC results with the GETQX and GETQY instructions (for X and Y, respectively). For the angle (in S), $00000000..$FFFFFFFF = 0..359.9999999 degrees. QROTATE will wait for the hub access window and GETQX / GETQY will wait for the CORDIC results.

QROTATE D/#,S,# - Rotate (D,$00000000) by S

GETQX X

GETQY Y

Note this is just like an [X,Y Rotation](#_m9zo223xec2s), but with Y set to 0 (by omitting the leading SETQ).

### Integer to Logarithm

Use the QLOG instruction on an unsigned 32-bit integer and retrieve the 5:27-bit logarithm CORDIC result (5-bit exponent and 27-bit mantissa) with the GETQX instruction. QLOG will wait for the hub access window and GETQX will wait for the CORDIC results.

QLOG D/# - Compute log base 2 of D

GETQX logarithm

### Logarithm to Integer

Use the QEXP instruction on a 5:27-bit logarithm and retrieve the unsigned 32-bit integer CORDIC result with the GETQX instruction. QEXP will wait for the hub access window and GETQX will wait for the CORDIC results.

QEXP D/# - Compute 2 to the power of D

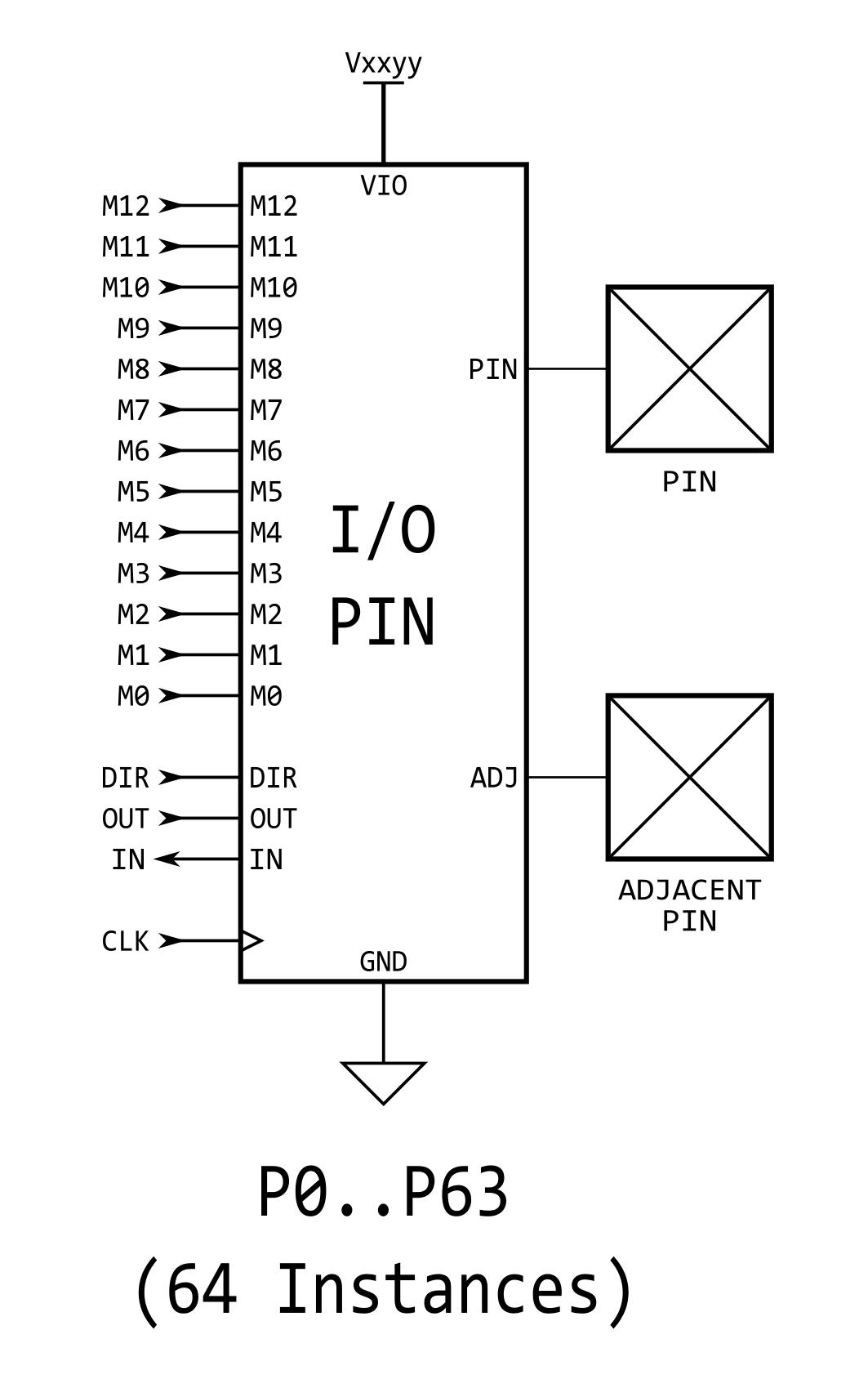
GETQX integer

# SMART I/O PINS

Every I/O pin features versatile digital and analog capabilities as well as autonomous state machine functions that would otherwise require processor time to perform. The combination of built-in circuitry plus configurable *pin* modes and *smart* modes provides adept functionality for application design, increasing the Propeller 2 potential beyond what multicore architecture alone provides. There are 24 low-level pin modes and 34 high-level smart modes.

## I/O Pin Circuit

Here is an illustration of a single I/O pin circuit which is powered from its local 3.3V supply pin (Vxxyy). There are 64 such circuits in the P2X8C4M64P, each connecting to its own physical pin (PIN), as well as its adjacent odd or even pin (ADJ). I/O Pins P0 and P1 see each other as adjacent pins, as do P2 and P3, etc.



Each I/O pin's behavior is described by the combination of four settings: 1) direction (input/output), 2) state (output drive / input sense), 3) pin mode, and 4) smart mode (optional). The first three of these activate or utilize special internal circuitry in the I/O Pin block itself (via DIR, OUT/IN, Mxx signals) and the fourth provides optional state-machine control outside of the circuit shown here. See [Equivalent Schematics](#_6z9hr7z553r3) to visualize each pin mode's effect. Each of the four settings is described below.

## Direction and State

In simplest form, I/O pins are controlled via dedicated cog registers and the instructions that affect them.

| I/O Pin Registers | | |
| --- | --- | --- |
| **Register** | **Cog Address** | **Purpose** |
| DIRA | $1FA | Output enable bits for P0..P31 (active high) |
| DIRB | $1FB | Output enable bits for P32..P63 (active high) |
| OUTA | $1FC | Output state bits for P0..P31 (corresponding DIRA bit must be high to enable output) |
| OUTB | $1FD | Output state bits for P32..P63 (corresponding DIRB bit must be high to enable output) |
| INA | $1FE | Input state bits for P0..P31 |
| INB | $1FF | Input state bits for P32..P63 |

General-purpose and special pin instructions can write to DIRA / DIRB / OUTA / OUTB to affect pin input/output behavior and can read from INA / INB to retrieve pin states. General-purpose instructions operate on the entire 32-bit register (all pins) while the special pin instructions operate on a single bit (pin) within them.

| Special Pin Instructions | |
| --- | --- |
| **Instructions** | **Purpose** |
| DIRL / DIRH / DIRC / DIRNC / DIRZ / DIRNZ / DIRRND / DIRNOT {#}D | Affect pin D bit in DIRx |
| OUTL / OUTH / OUTC / OUTNC / OUTZ / OUTNZ / OUTRND / OUTNOT {#}D | Affect pin D bit in OUTx |
| FLTL / FLTH / FLTC / FLTNC / FLTZ / FLTNZ / FLTRND / FLTNOT {#}D | Affect pin D bit in OUTx, clear bit in DIRx |
| DRVL / DRVH / DRVC / DRVNC / DRVZ / DRVNZ / DRVRND / DRVNOT {#}D | Affect pin D bit in OUTx, set bit in DIRx |
| TESTP {#}D WC / WZ / ANDC / ANDZ / ORC / ORZ / XORC / XORZ | Read pin D bit in INx and affect C or Z |
| TESTPN {#}D WC / WZ / ANDC / ANDZ / ORC / ORZ / XORC / XORZ | Read pin D bit in !INx and affect C or Z |

The selected pin mode and smart mode (if other than the default) may override some of the above, as described in their respective sections, later.

## Pin Modes

Each I/O pin has 13 low-level pin mode configuration bits which determine the mode of operation (1 of 24) for its 3.3 V circuit. The pin mode is set using the WRPIN instruction, where the 13 'M' bits within the instruction's D operand specifies the pin mode configuration. Note that in some smart pin modes, the configuration bits are partially overwritten to set things like DAC values.

The format of the WRPIN's D operand value is:

%AAAA\_BBBB\_FFF\_MMMMMMMMMMMMM\_TT\_SSSSS\_0

* A = PIN input selector
* B = ADJ input selector
* F = PIN and ADJ input logic/filtering (applied to result of PIN and ADJ input selectors)
* M = pin mode
* T = pin DIR / OUT control (default = %00)
* S = smart mode

| (A) PIN or (B) ADJ Input Selector | |
| --- | --- |
| **%AAAA**  **%BBBB** | **Selection** |
| 0xxx | true (default) |
| 1xxx | inverted |
| x000 | this pin's read state (default) |
| x001 | relative +1 pin's read state |
| x010 | relative +2 pin's read state |
| x011 | relative +3 pin's read state |
| x100 | this pin's OUT bit from cogs |
| x101 | relative -3 pin's read state |
| x110 | relative -2 pin's read state |
| x111 | relative -1 pin's read state |

| (F) PIN and ADJ Logic/Filtering | |
| --- | --- |
| **%FFF** | **Logic/Filter** |
| 000 | A, B (default) |
| 001 | A AND B, B |
| 010 | A OR B, B |
| 011 | A XOR B, B |
| 100 | A, B, both filtered using global filt0 settings |
| 101 | A, B, both filtered using global filt1 settings |
| 110 | A, B, both filtered using global filt2 settings |
| 111 | A, B, both filtered using global filt3 settings |

The resultant 'A' will drive the IN signal in non-smart-pin modes.

| (M) Pin Mode | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **WRPIN D[20:8] Configuration** | | |  | **Resulting Internal Configuration** | | | | | |
| **M[12:0]** | **Input** | **Pin Output1** |  | **CIOHHHLLL** | **OE2** | **DAC** | **ADC** | **ADC Mode** | **Comparator** |
| 0000\_CIOHHHLLL  0001\_CIOHHHLLL  0010\_CIOHHHLLL  0011\_CIOHHHLLL  0100\_CIOHHHLLL  0101\_CIOHHHLLL  0110\_CIOHHHLLL  0111\_CIOHHHLLL | Pin Logic  Pin Logic  Adj Logic  Pin Schmitt  Pin Schmitt  Adj Schmitt  Pin > Adj  Pin > Adj | OUT  Input  Input  OUT  Input  Input  OUT  Input |  | CIOHHHLLL  CIOHHHLLL  CIOHHHLLL  CIOHHHLLL  CIOHHHLLL  CIOHHHLLL  CIOHHHLLL  CIOHHHLLL | DIR  DIR  DIR  DIR  DIR  DIR  DIR  DIR | 0  0  0  0  0  0  0  0 | 0  0  0  0  0  0  0  0 |  | 0  0  0  0  0  0  Pin > Adj  Pin > Adj |
| 100000\_OHHHLLL  100001\_OHHHLLL  100010\_OHHHLLL  100011\_OHHHLLL  100100\_OHHHLLL  100101\_OHHHLLL  100110\_OHHHLLL  100111\_OHHHLLL | ADC, GND  ADC, Vxxyy  ADC, float  ADC, Pin 1x  ADC, Pin 3.16x  ADC, Pin 10x  ADC, Pin 31.6x  ADC, Pin 100x | OUT  OUT  OUT  OUT  OUT  OUT  OUT  OUT |  | 10OHHHLLL  10OHHHLLL  10OHHHLLL  10OHHHLLL  10OHHHLLL  10OHHHLLL  10OHHHLLL  10OHHHLLL | DIR  DIR  DIR  DIR  DIR  DIR  DIR  DIR | 0  0  0  0  0  0  0  0 | 1  1  1  1  1  1  1  1 | 000  001  010  011  100  101  110  111 | 0  0  0  0  0  0  0  0 |
| 10100\_DDDDDDDD  10101\_DDDDDDDD  10110\_DDDDDDDD  10111\_DDDDDDDD | ADC, Pin 1x 3  ADC, Pin 1x 3  ADC, Pin 1x 3  ADC, Pin 1x 3 | DAC 990 Ω, 3.3 V  DAC 600 Ω, 2.0 V  DAC 123.75 Ω, 3.3 V  DAC 75 Ω, 2.0 V |  | 10xxxxxxx  10xxxxxxx  10xxxxxxx  10xxxxxxx | 0  0  0  0 | DIR  DIR  DIR  DIR | OUT  OUT  OUT  OUT | 011  011  011  011 | 0  0  0  0 |
| 1100\_CDDDDDDDD  1101\_CDDDDDDDD  1110\_CDDDDDDDD  1111\_CDDDDDDDD | Pin > D  Pin > D  Adj > D  Adj > D | OUT, 1.5 kΩ  !Input, 1.5 kΩ  Input, 1.5 kΩ  !Input, 1.5 kΩ |  | C00001001  C01001001  C00001001  C01001001 | DIR  DIR  DIR  DIR | 0  0  0  0 | 0  0  0  0 |  | Pin > D  Pin > D  Adj > D  Adj > D |
| 1 OUT means output latch bit drives output; Input means the 'Input' column's item drives output  2 OE is digital logic output enable only; analog output is indicated in the DAC column  3 if OUT bit = 1 | | | | | | | | | |
| **Pin Mode Legend** | | | | | | | | | |
| | | **C** | **IN** **/** **OUT** | | --- | --- | | 0  1 | Live 1  Clocked 2 | |  |  | | **I** | **IN** | | 0  1 | True  Not (inverted) | |  |  | | **O** | **Output** | | 0  1 | True  Not (inverted) | | | **HHH**  **LLL** | **Drive** | | --- | --- | | 000  001  010  011  100  101  110  111 | Fast  1.5 kΩ  15 kΩ  150 kΩ  1 mA  100 µA  10 µA  Float | |  |  | | | OE = digital output enable (when DIR bit high) | | --- | | DAC = digital to analog converter enable (when DIR bit high) | | ADC = analog to digital converter enable (fixed, or when OUT bit high) | | OUT = output latch bit; 0: low, 1: high.  Exception: DAC modes use OUT as 0: disable, 1: enable. | | DIR = direction bit; 0: input (float), 1: output (drive)  Exception: DAC modes use DIR as 0: disable, 1: enable. | | DDDDDDDD and D = DAC Level | | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | | | | | | | | | |

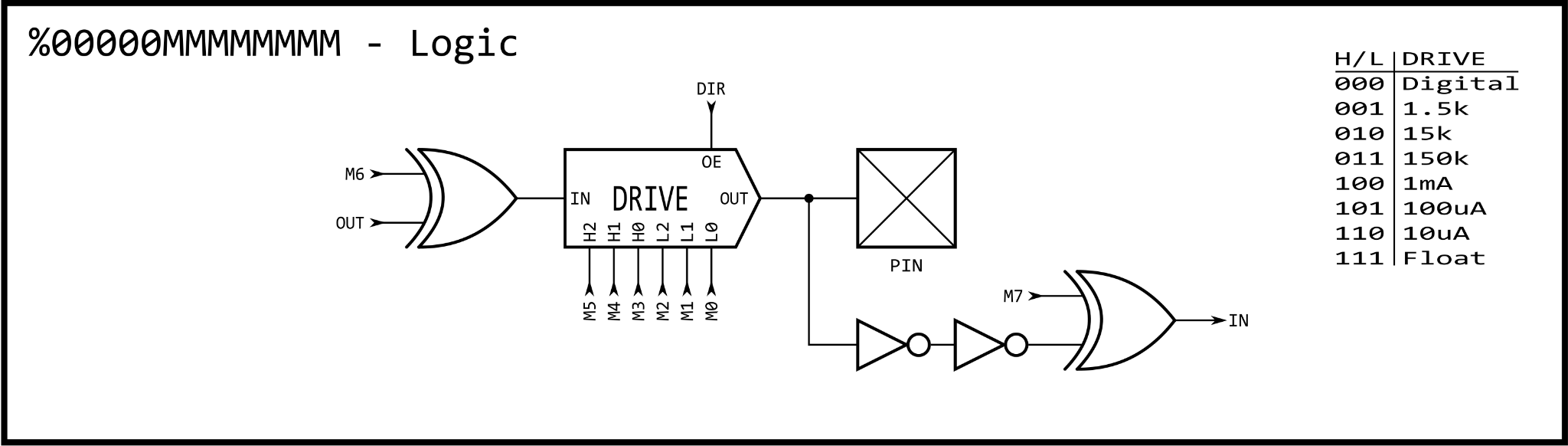
1 used for feedback operations; provides continuous (non-clocked) signal

2 signal updates on clock edge only

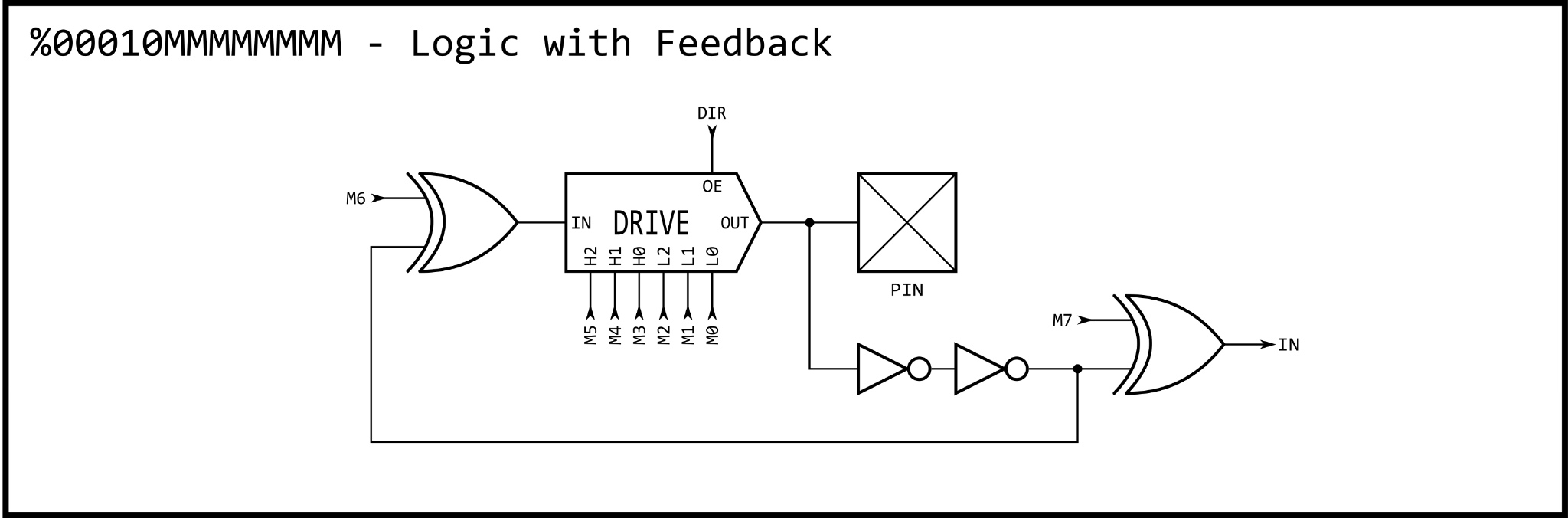
| (T) Pin DIR/OUT Control | | |
| --- | --- | --- |
| **Default (%TT = 00)** | | |
| for odd pins | | 'OTHER' = even pin's NOT (inverted) output state (diff source) |
| for even pins | | 'OTHER' = unique pseudo-random bit (noise source) |
| for all pins | | 'SMART' = smart pin output which overrides OUT / OTHER |
| 'DAC\_MODE' is enabled when M[12:10] = %101 | | |
| 'BIT\_DAC' outputs {2{M[7:4]}} for 'high' or {2{M[3:0]}} for 'low' in DAC\_MODE | | |
| **for smart pin mode "off" (%SSSSS = %00000)** | | |
|  | DIR enables output | |
| **for non-DAC\_MODE** | |
| 0x | OUT drives output |
| 1x | OTHER drives output |
| **for DAC\_MODE** | |
| 00 | DIR enables DAC, M[7:0] sets DAC level |
| 01 | OUT enables ADC, M[3:0] selects cog DAC channel |
| 10 | OUT drives BIT\_DAC |
| 11 | OTHER drives BIT\_DAC |
| **for smart pin mode "on" (%SSSSS > %00000)** | | |
|  | x0 | output disabled, regardless of DIR |
| x1 | output enabled, regardless of DIR |
| **for DAC smart pin modes (%SSSSS = %00001..%00011)** | | |
|  | 0x | OUT enables DAC in DAC\_MODE, M[7:0] overridden |
| 1x | OTHER enables DAC in DAC\_MODE, M[7:0] overridden |
| **for non-DAC smart pin modes (%SSSSS = %00100..%11111)** | | |
|  | 0x | SMART / OUT drives output, or BIT\_DAC if DAC\_MODE |
| 1x | SMART / OTHER drives output, or BIT\_DAC if DAC\_MODE |

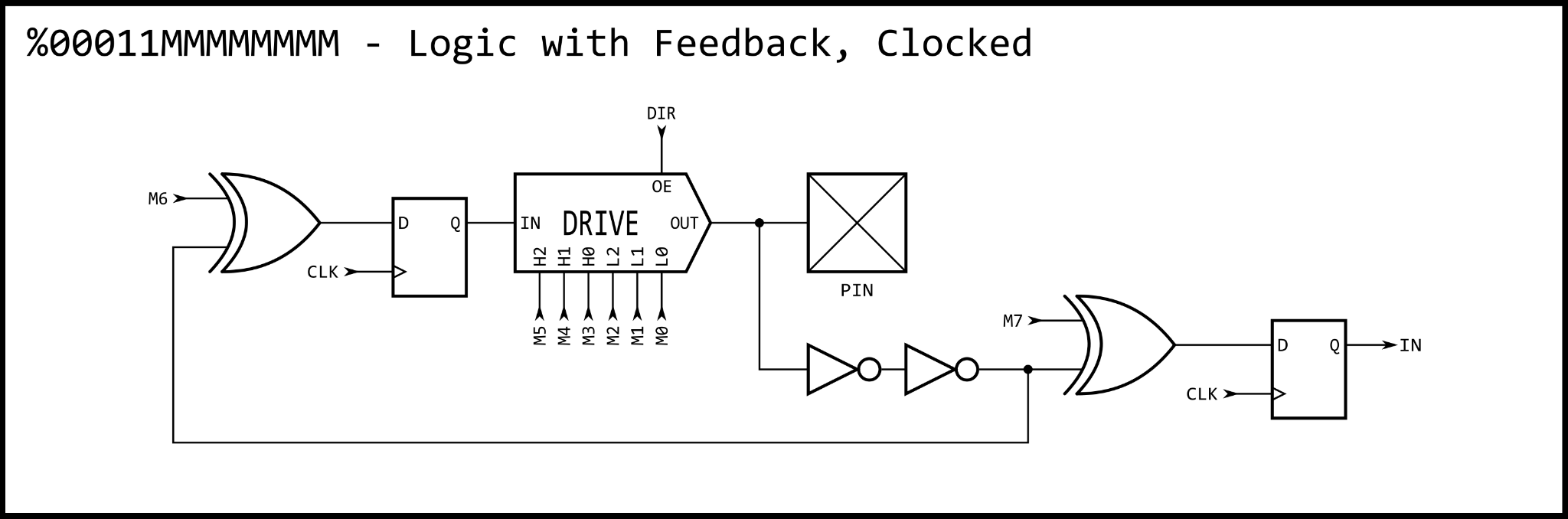
NOTE: The (S) smart modes are listed and described in their own section, [Smart Modes](#_msbxl3vp75z).

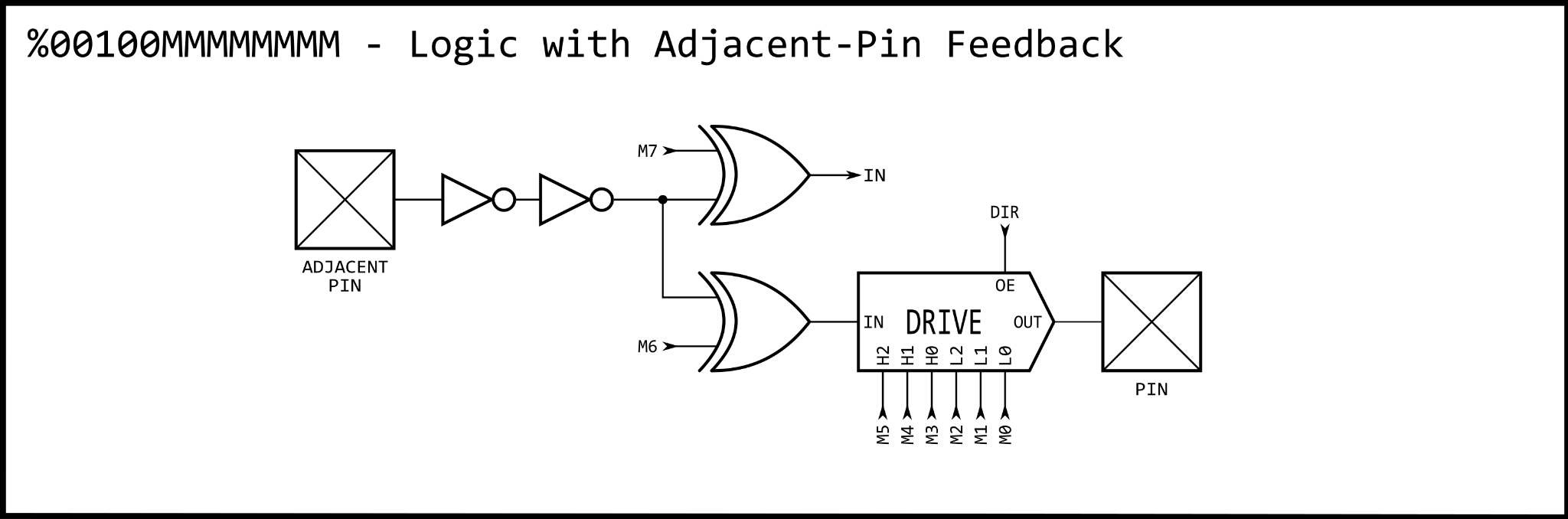
## Equivalent Schematics for Each Unique I/O Pin Configuration

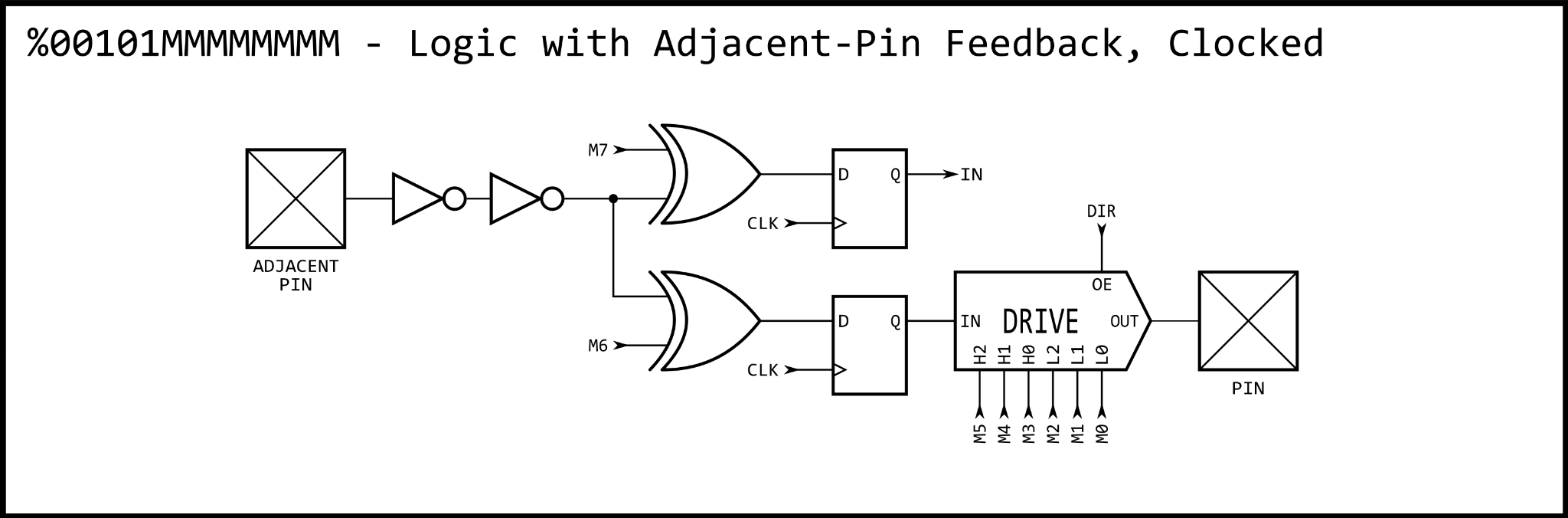


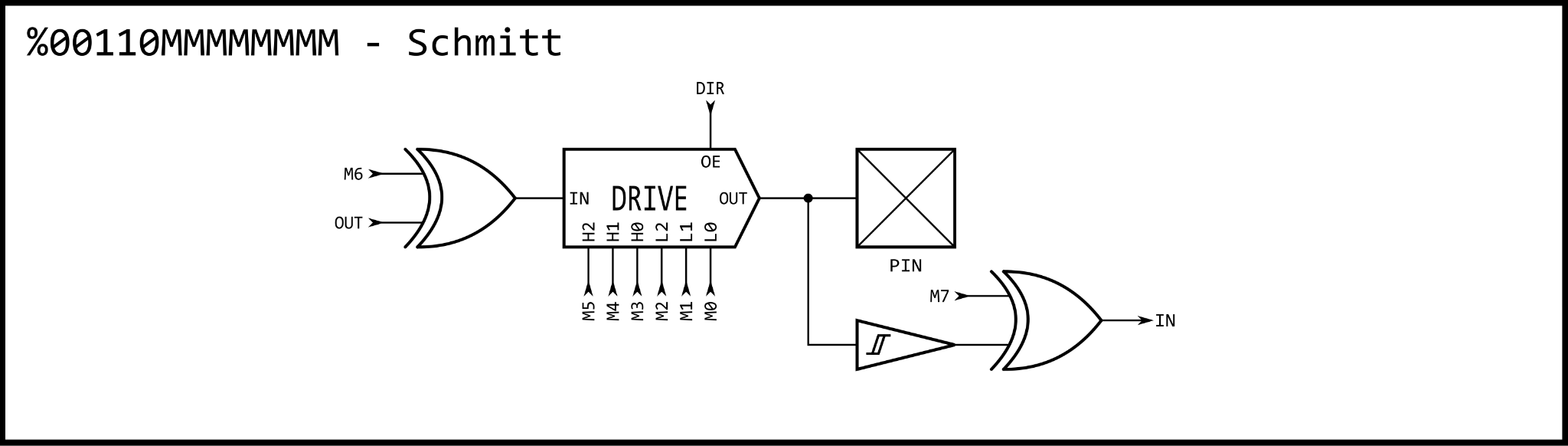


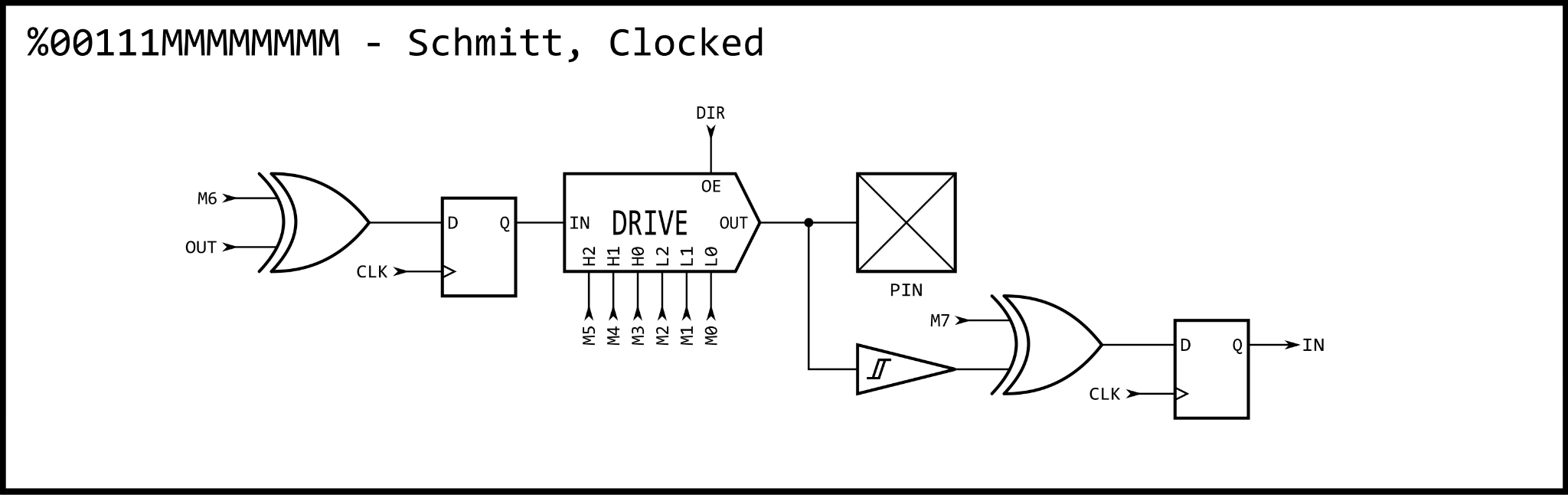


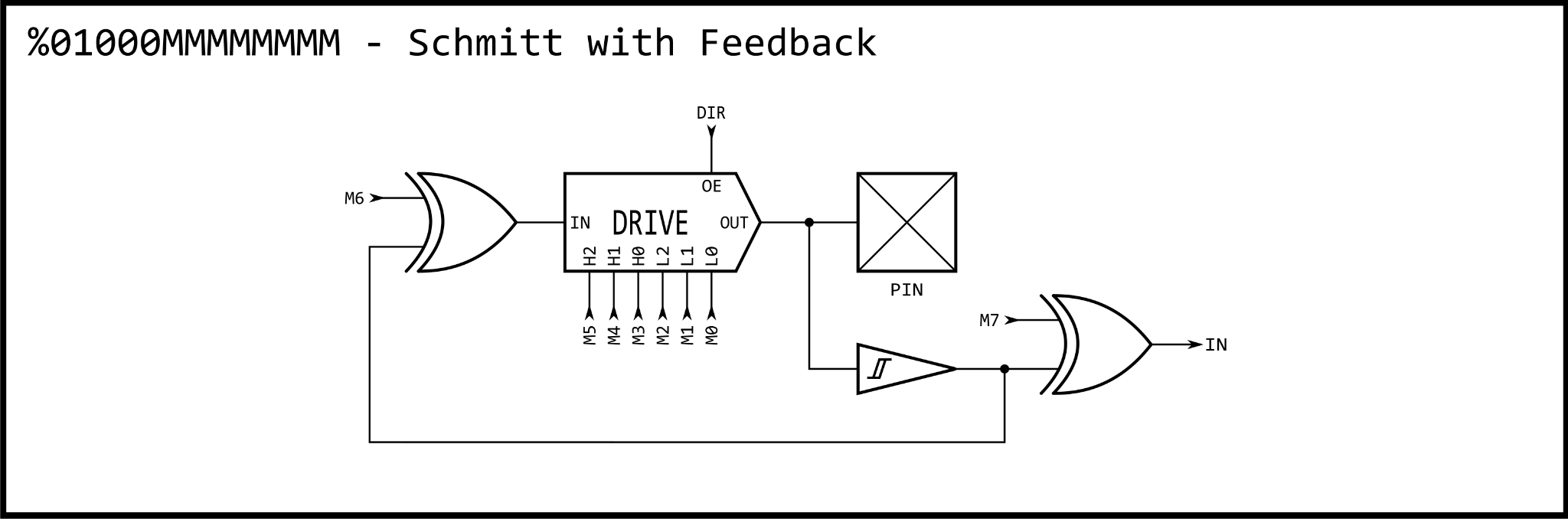


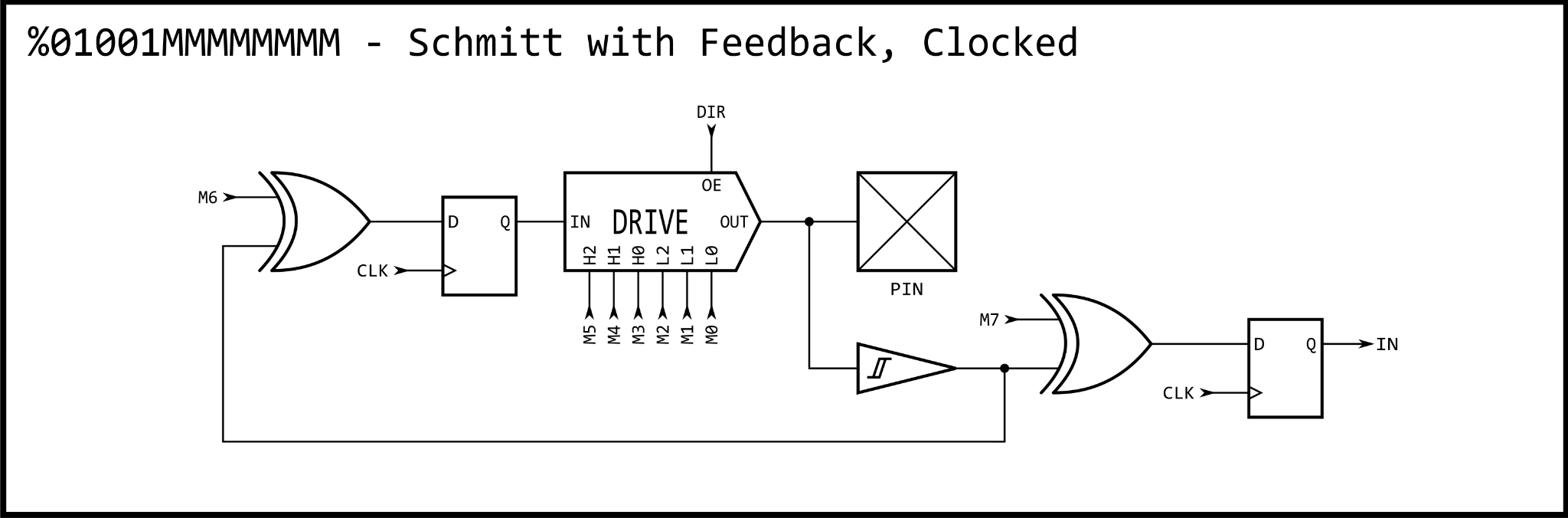


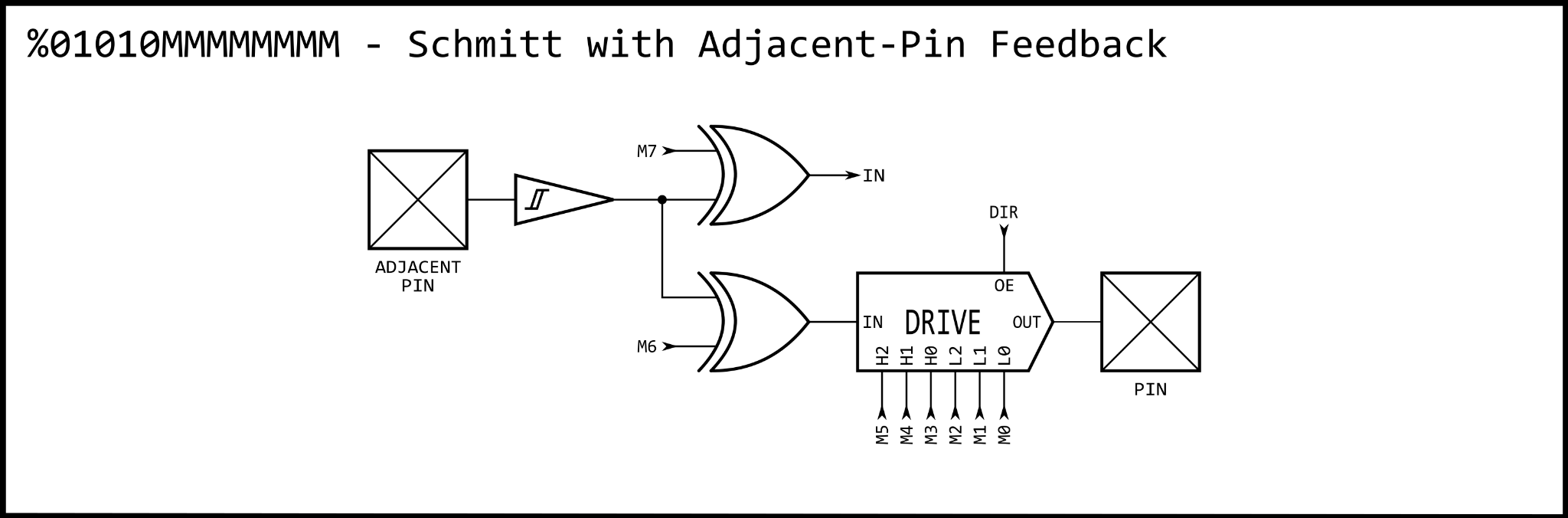


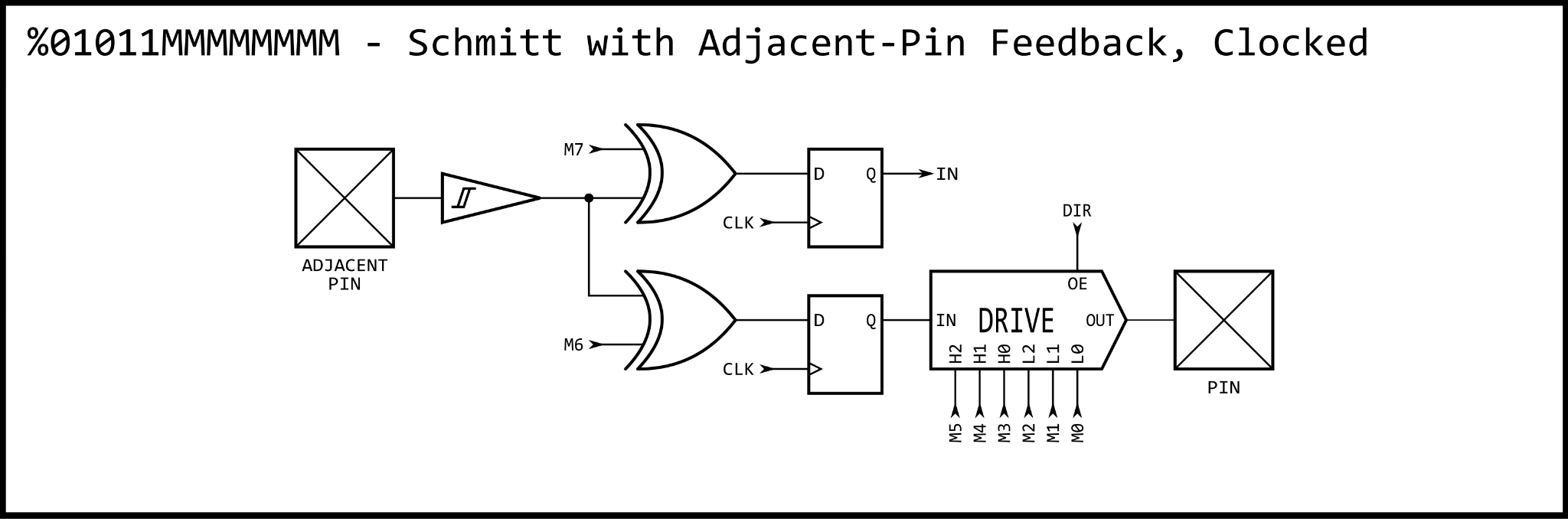


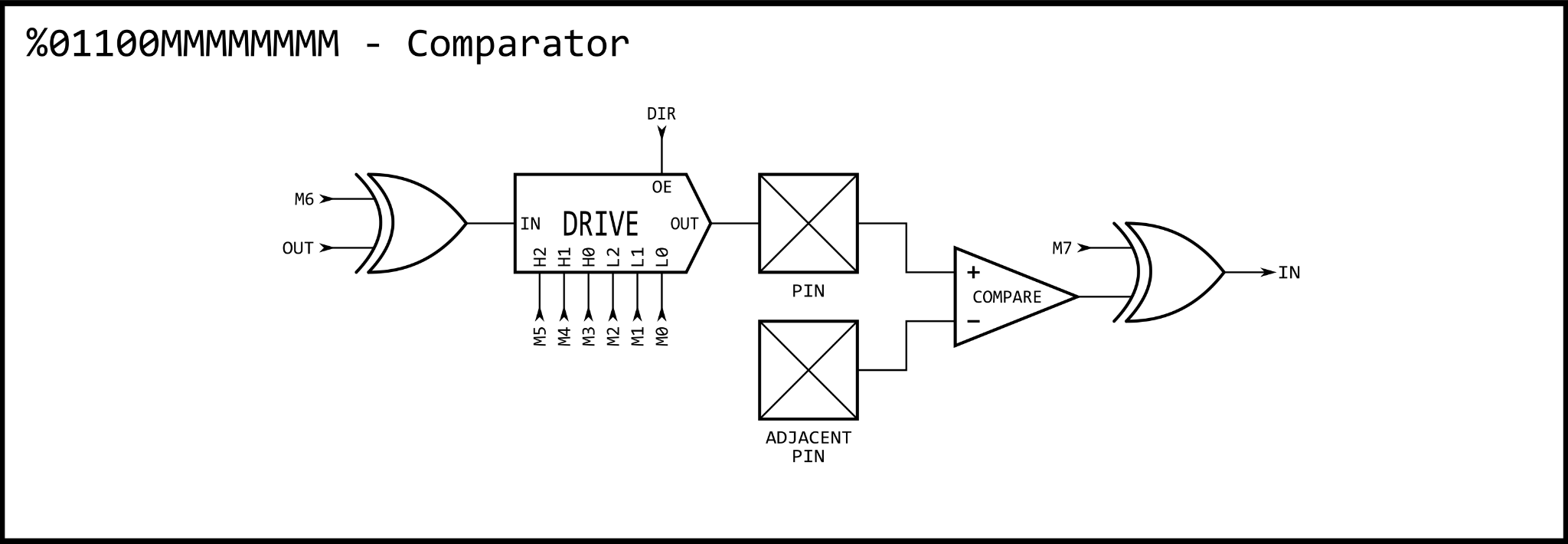


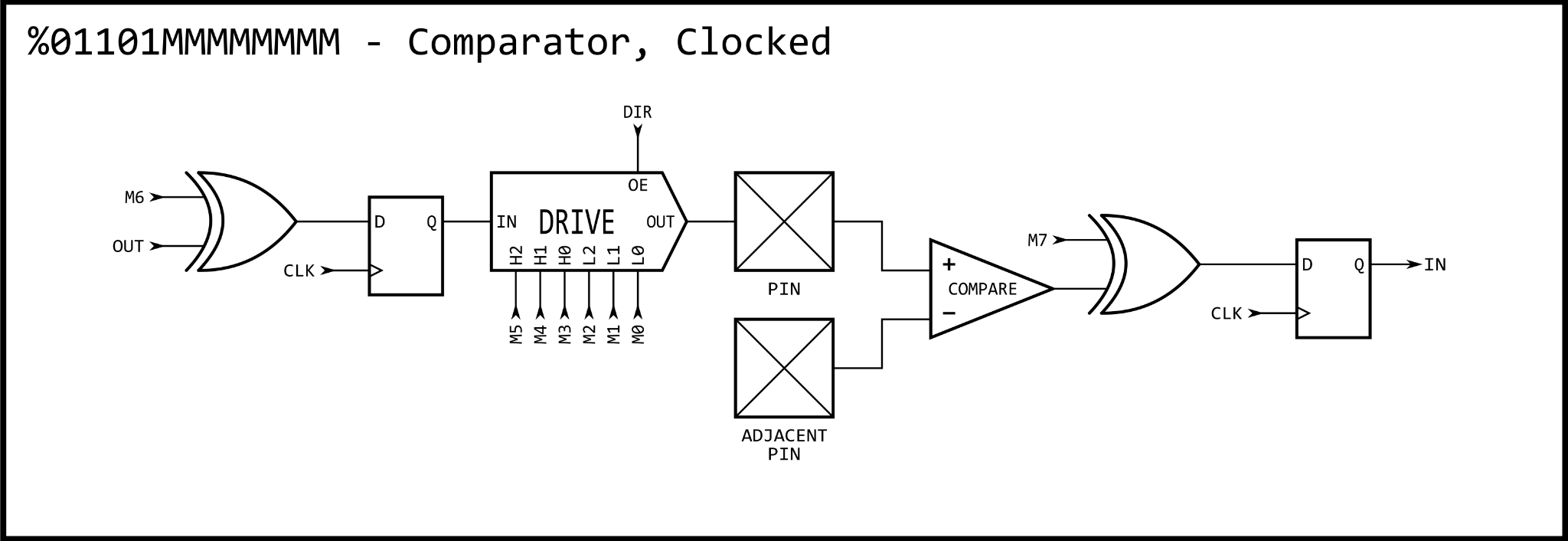


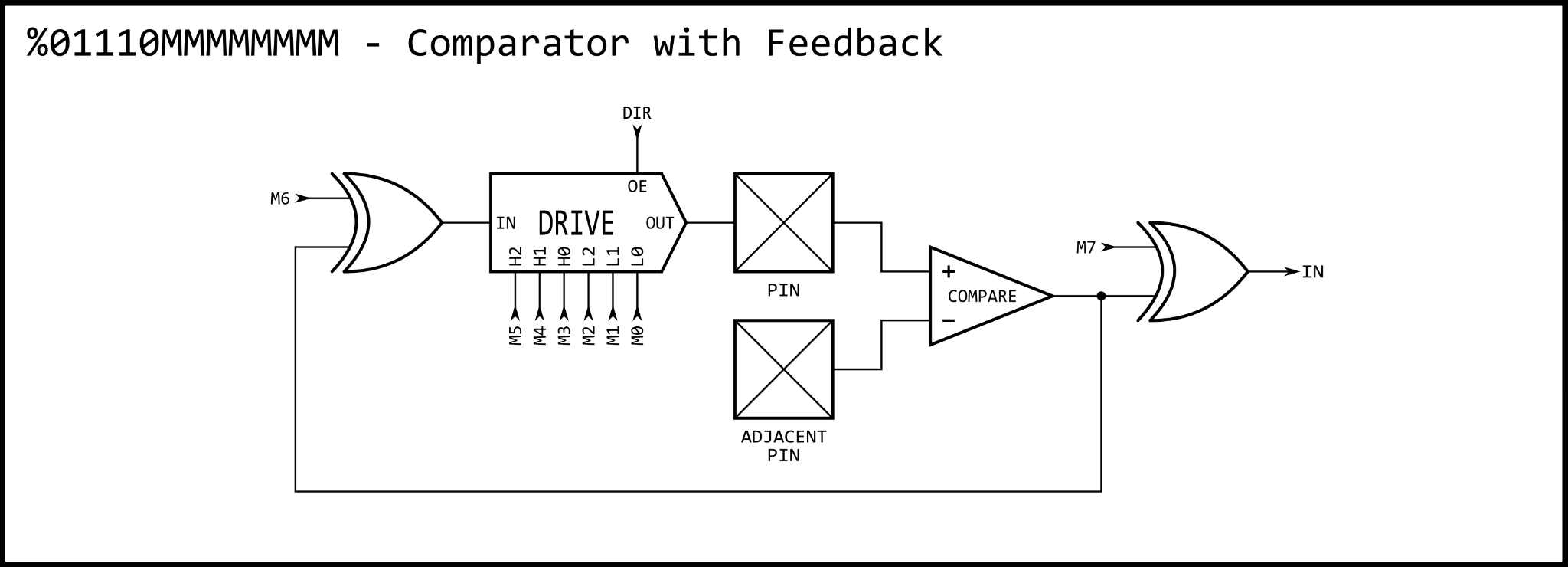


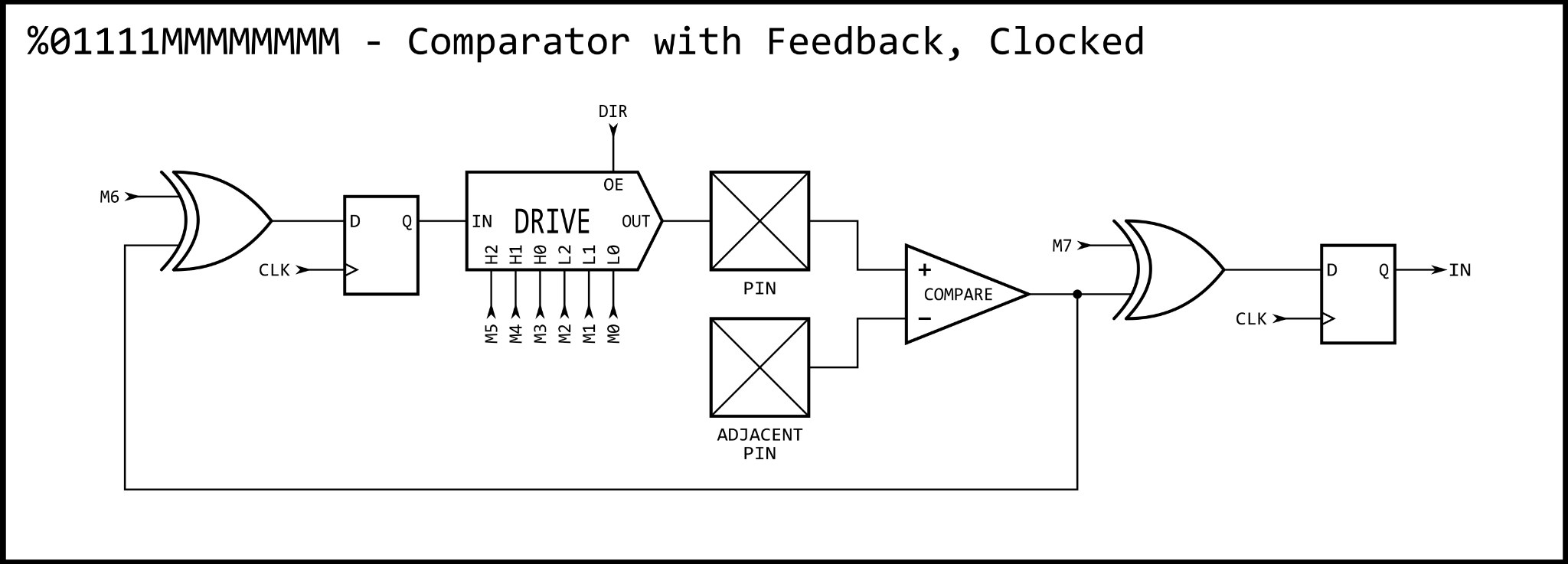


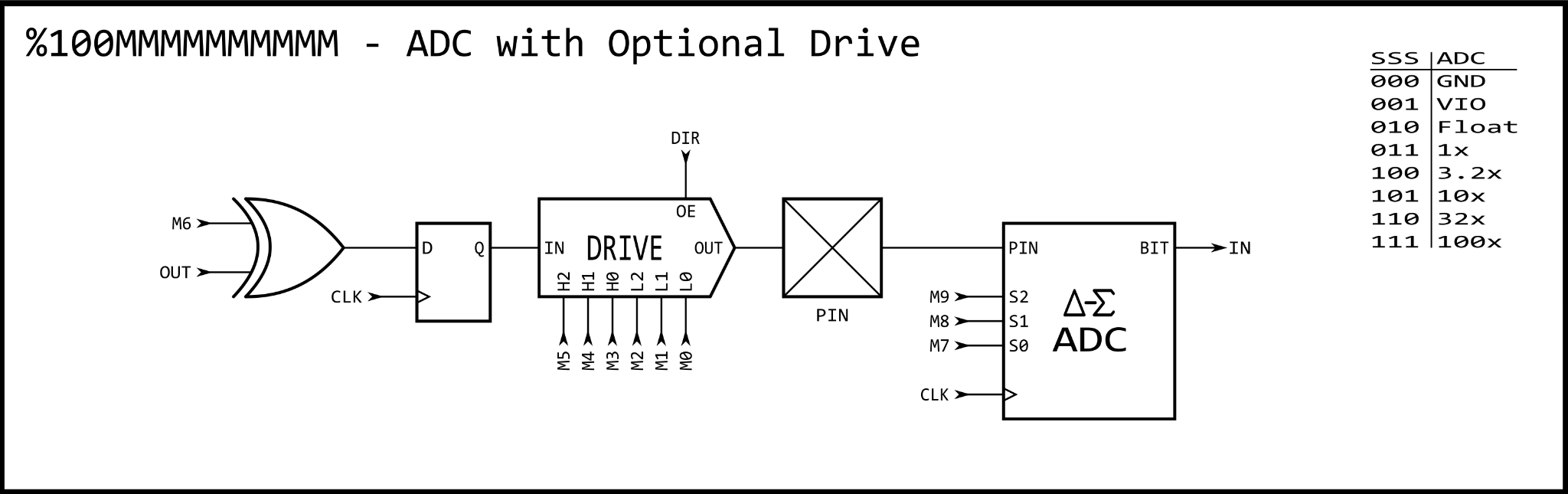


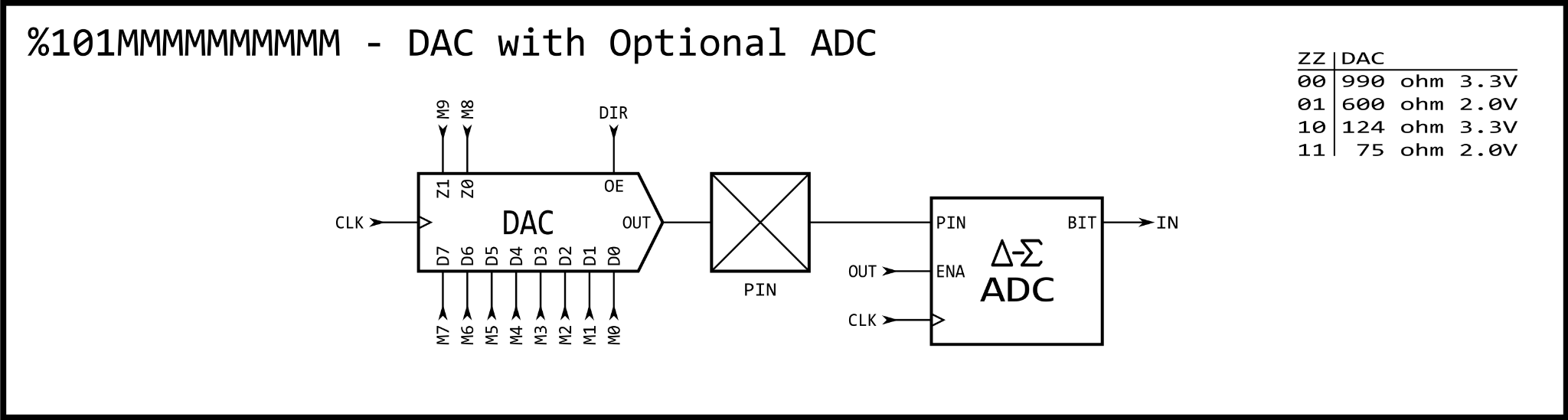


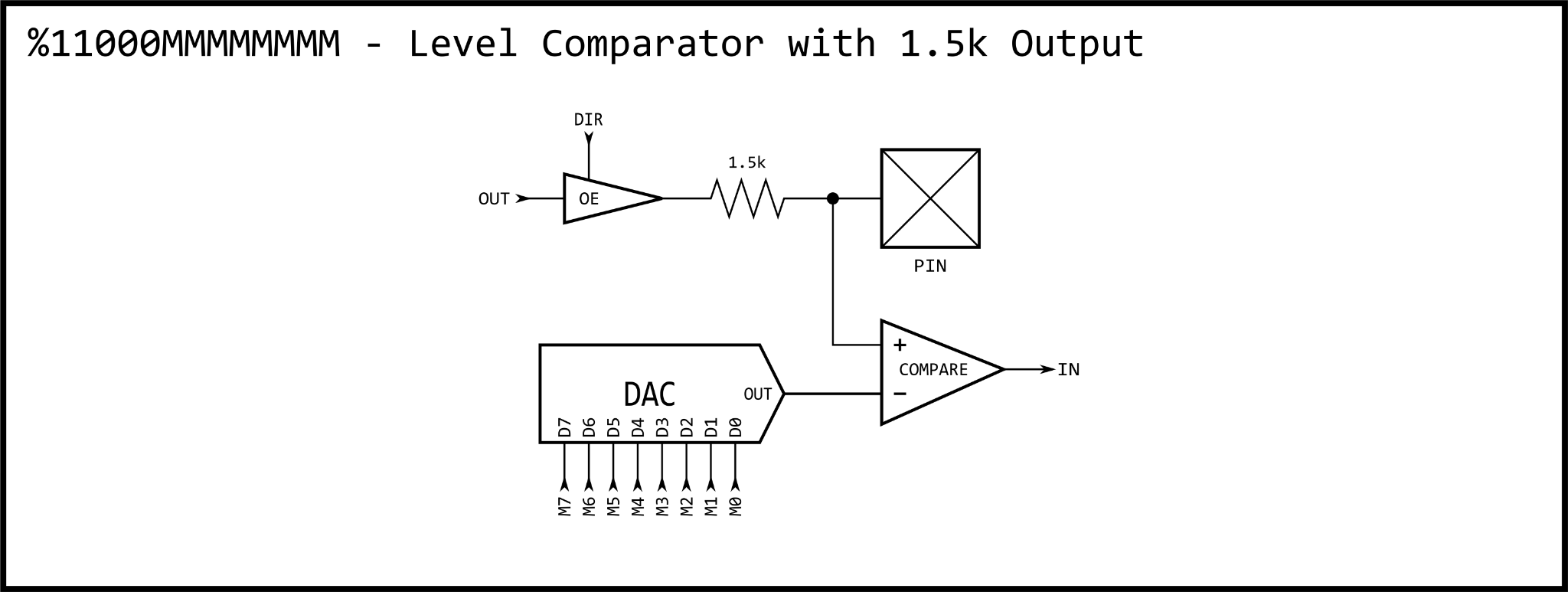


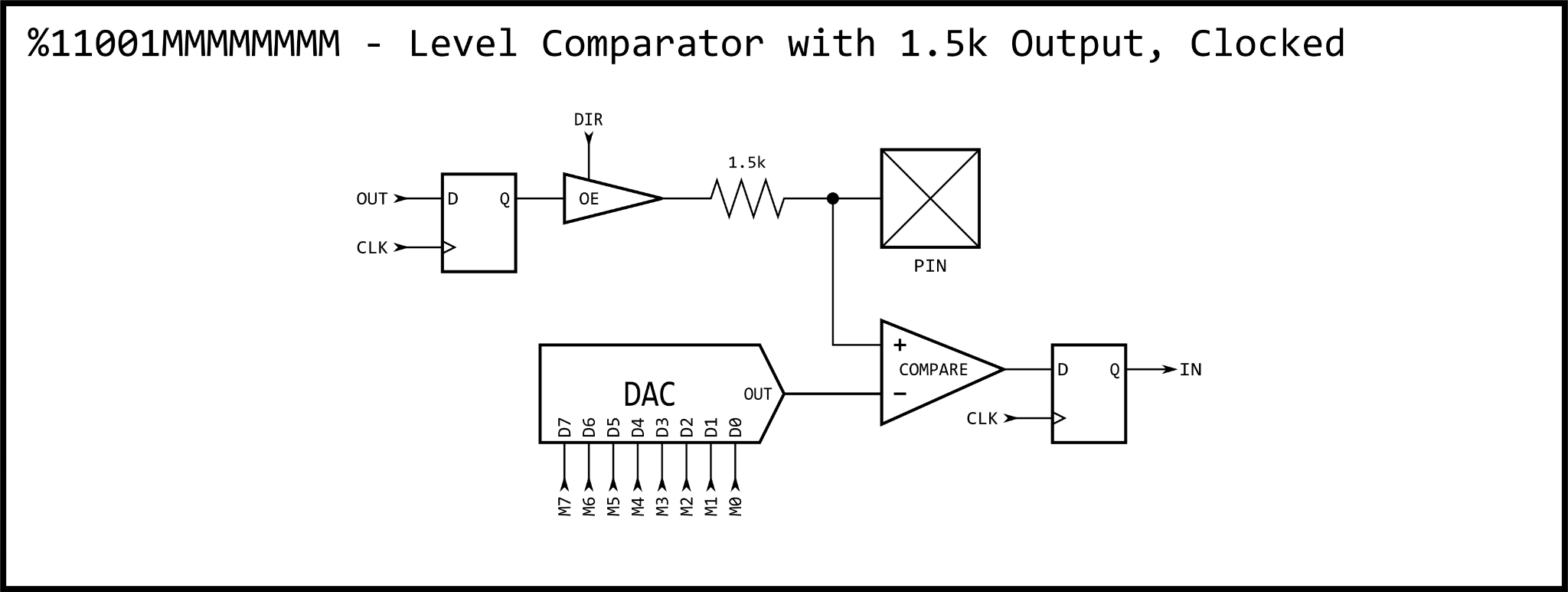


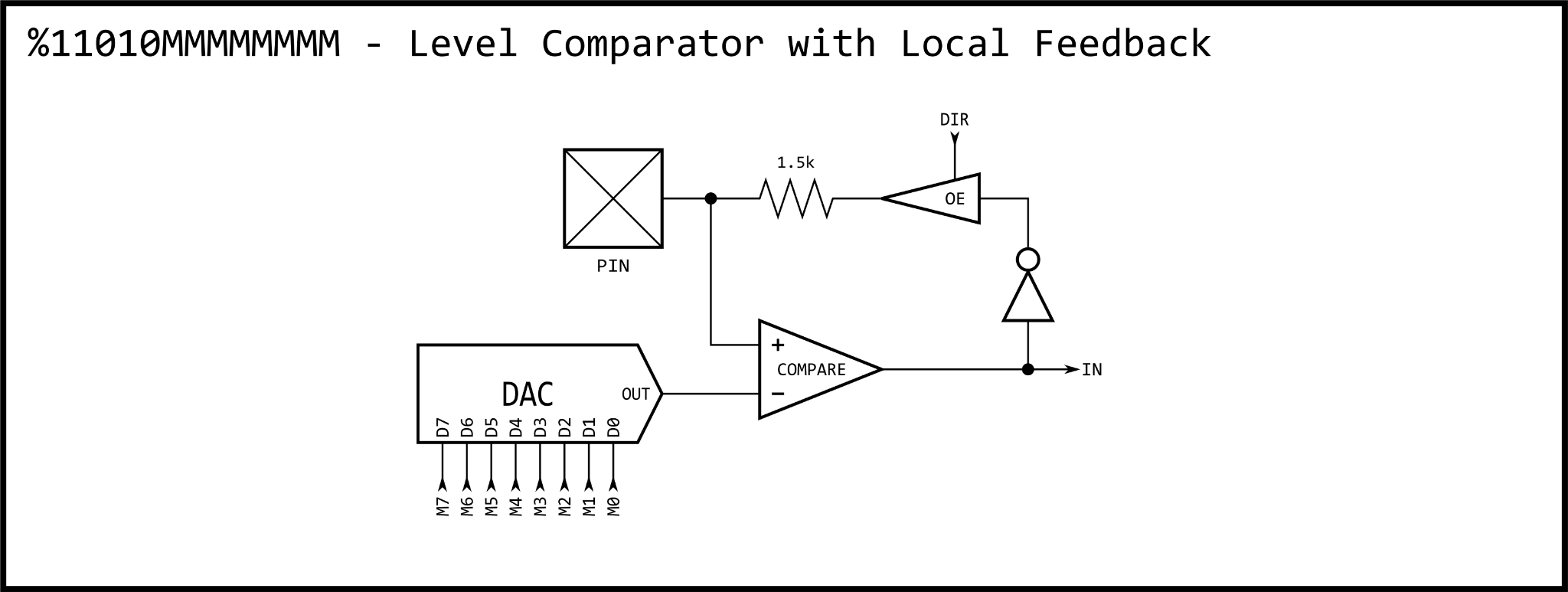


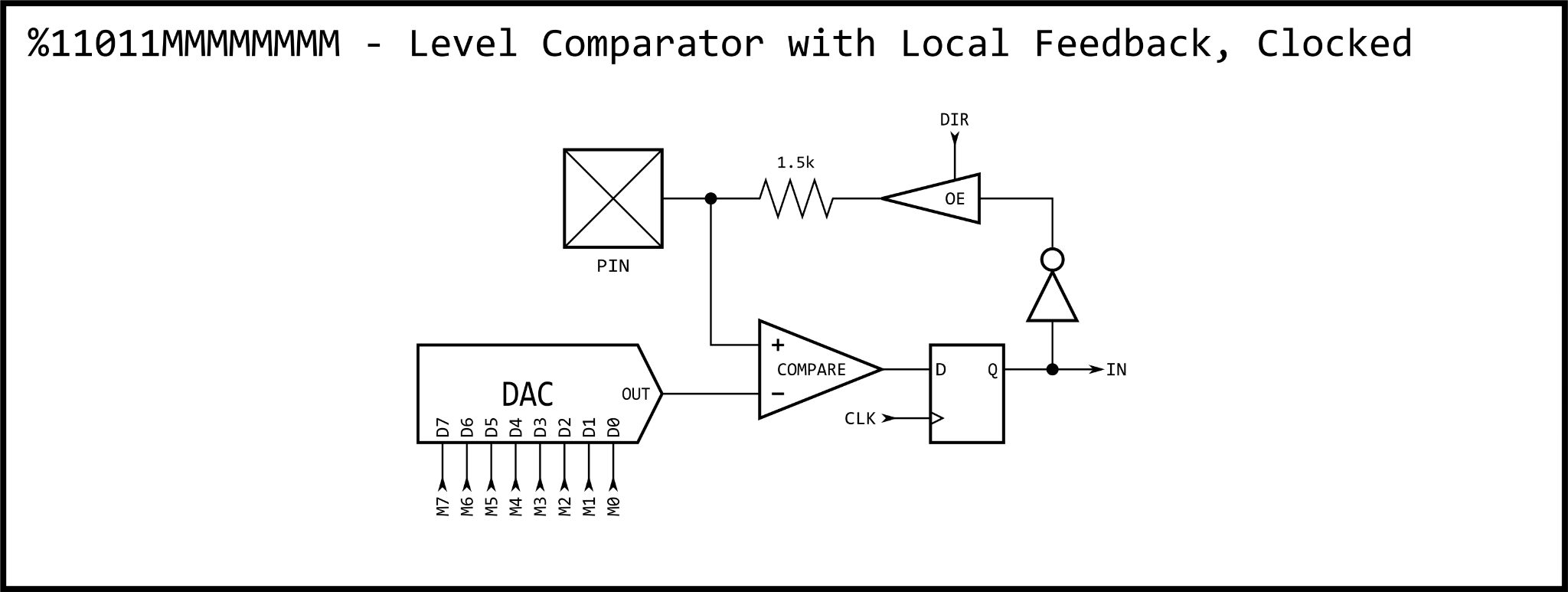


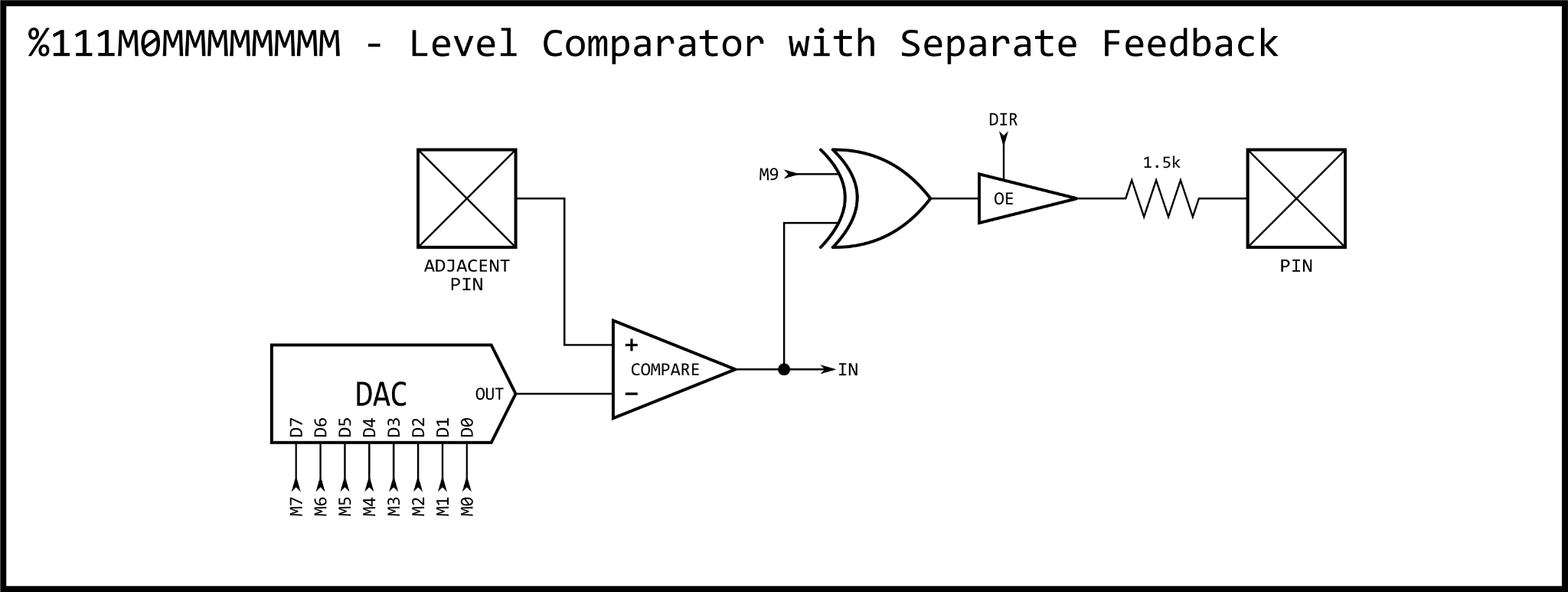


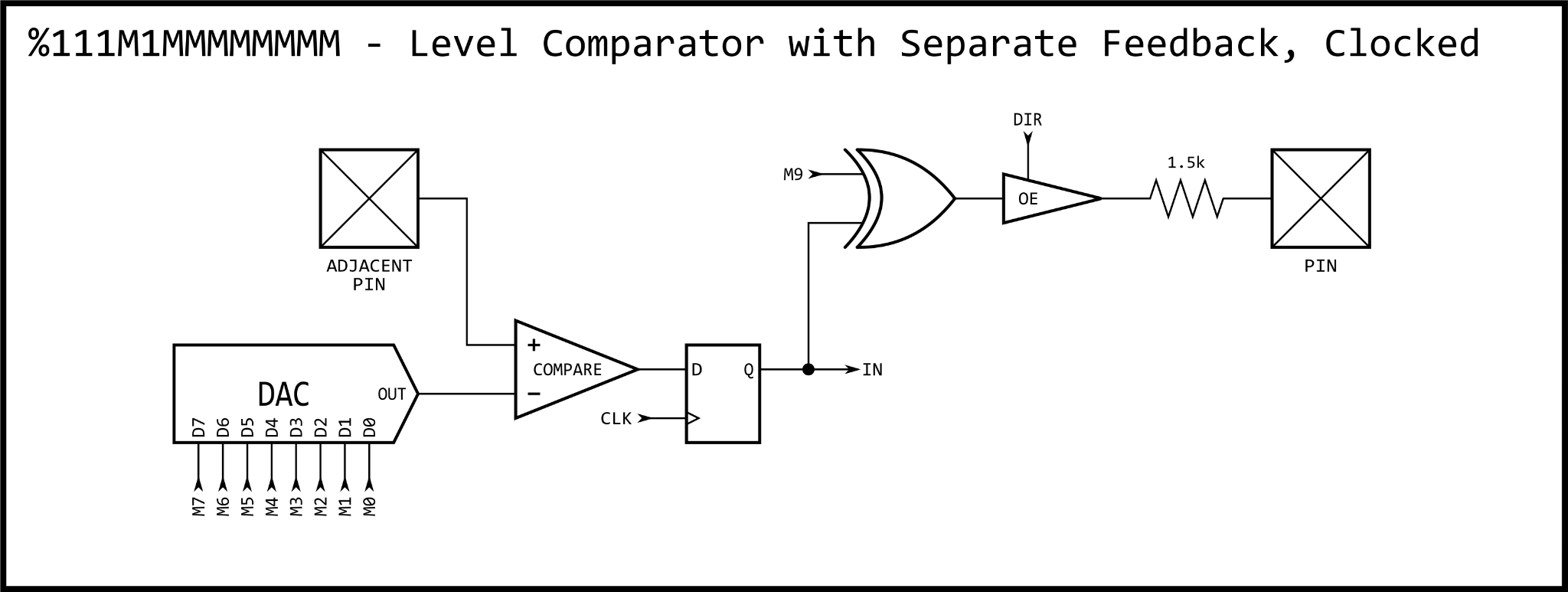








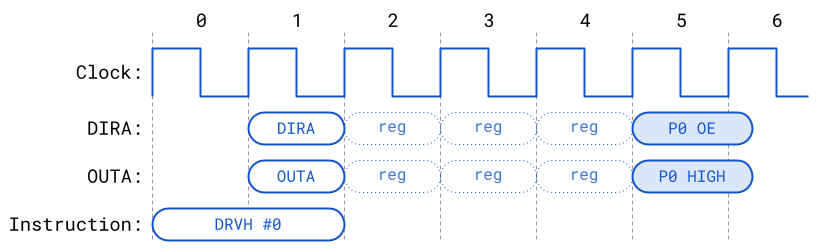




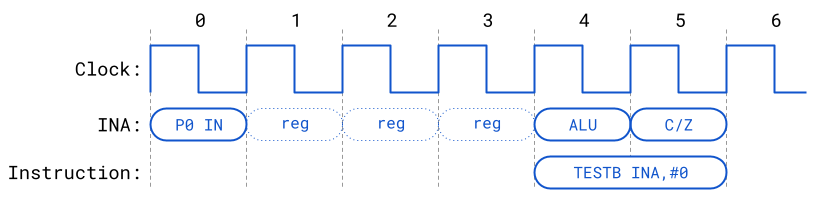
## I/O Pin Timing

Between each physical I/O pin and the cog(s) controlling them, there is a chain of three single-bit registers (reg). The live signal (input or output) traverses through this chain on the way to its destination as described below.

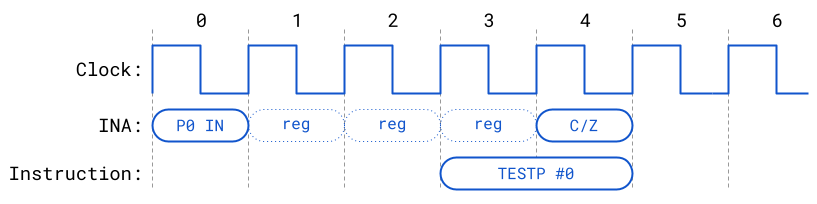
When a DIRx / OUTx bit is changed by any instruction, it takes *three* additional clocks after the instruction before the pin starts transitioning to the new state. Here this delay is demonstrated using DRVH to set I/O pin P0's output enable (OE) and drive P0's output latch high.

 *P0 OE* and *P0 HIGH* begin their transition on the rising edge of clock 5; however, the duration until complete depends on clock frequency and circuit load. The I/O pads are asynchronous (not tied strictly to the clock) so with a slow operating frequency, the transition may complete within 1 clock, whereas with higher frequencies it may take multiple clocks to complete.

When an INx register is read by an instruction, it will reflect the state of the pins registered *three* clocks before the start of the instruction. Here this delay is demonstrated using TESTB:



When a TESTP / TESTPN instruction is used to read a pin, the value read will reflect the state of the pin registered *two* clocks before the start of the instruction. Effectively, TESTP / TESTPN get fresher INx data than is available via the INx registers:



## Smart Modes

Each I/O pin has built-in 'smart pin' circuitry which (when enabled) performs one of 34 different autonomous functions on the pin. Smart pins free the cogs from the need to micromanage many I/O operations by providing high-bandwidth concurrent hardware functions that cogs could otherwise not perform as well through I/O pin manipulating instructions.

In normal operation, an I/O pin's output enable is controlled by its DIR bit, its output state is controlled by its OUT bit, and its IN bit returns the pin's read state. With smart pin mode enabled, its DIR bit is used as an active-low reset signal to the smart pin circuitry, while the output enable state is controlled by a configuration bit. In some modes, the smart pin circuit takes over driving the output state, in which case the OUT bit gets ignored. Its IN bit serves as a flag to indicate to the cog(s) that the smart pin has completed some function or an event has occurred, and acknowledgment is perhaps needed.

To configure a smart pin, first set its DIR bit to low (holding it in reset) then use WRPIN, WXPIN, and WYPIN to establish the mode and related parameters. Once configured, DIR can be raised high and the smart pin will begin operating. After that, depending on the mode, you may feed it new data via WXPIN / WYPIN or retrieve results using RDPIN / RQPIN. These activities are usually coordinated with the IN signal going high; explained later.

Note that while a smart pin is configured, the %TT bits (of the WRPIN instruction's D operand) will govern the pin's output enable, regardless of the DIR state.

Smart pins have four 32-bit registers inside of them:

| **Smart Pin Registers** | |
| --- | --- |
| **32-bit Register** | **Purpose** |
| Mode | smart pin mode, as well as low-level I/O pin mode (write-only) |
| X | mode-specific parameter (write-only) |
| Y | mode-specific parameter (write-only) |
| Z | mode-specific result (read-only) |

These four registers are written and read via the following 2-clock instructions, in which S/# is used to select the pin number (0..63) and D/# is the 32-bit data conduit:

WRPIN D/#,S/# - Set smart pin S/# mode to D/#, ack pin

WXPIN D/#,S/# - Set smart pin S/# parameter X to D/#, ack pin

WYPIN D/#,S/# - Set smart pin S/# parameter Y to D/#, ack pin

RDPIN D,S/# {WC} - Get smart pin S/# result Z into D, flag into C, ack pin

RQPIN D,S/# {WC} - Get smart pin S/# result Z into D, flag into C, don't ack pin

AKPIN S/# - Acknowledge pin S/#

Each smart pin has a 34-bit input bus and a 33-bit output bus that connect it to the cogs.

To configure and control smart pins, each cog writes data and acknowledgement signals to the smart pin input bus. Each smart pin OR's all incoming 34-bit buses from the collective of cogs in the same way DIR and OUT bits are OR'd before going to the pins. Therefore, if you intend to have multiple cogs execute WRPIN / WXPIN / WYPIN / RDPIN / AKPIN instructions on the same smart pin, you must be sure that they do so at different times, in order to avoid clobbering each other's bus data. Reading a smart pin with RDPIN can cause the same conflict; however, any number of cogs can read a smart pin simultaneously without bus conflict by using RQPIN ('read quiet'), since it does not utilize the smart pin input bus for acknowledgement signaling (like RDPIN does).

Each smart pin writes to its output bus to convey its Z result and a special flag. The RDPIN and RQPIN multiplex and read these buses, so that a pin's Z result is read into D and its special flag can be read into C. C will be either a mode-related flag or the MSB of the Z result.

When a mode-related event occurs in a smart pin, it raises its IN signal to alert the cog(s) that new data is ready, new data can be loaded, or some process has finished. A cog can test for this signal via the TESTP instruction and can acknowledge a smart pin by executing a WRPIN, WXPIN, WYPIN, RDPIN, or AKPIN instruction for it. This acknowledgement causes the smart pin to lower its IN signal so that it can be raised again on the next event. After a WRPIN / WXPIN / WYPIN / RDPIN / AKPIN, it takes two clocks for IN to drop, before it can be polled again.

Example:

WRPIN 'acknowledge smart pin, releases IN from high

NOP 'elapse 2 clocks (or more)

TESTP pin WC 'IN can now be polled again

A smart pin can be reset at any time, without the need to reconfigure it, by clearing and then setting its DIR bit.

To return a pin to normal mode, do a 'WRPIN #0,pin'.

| (S) Smart Pin Modes | | |
| --- | --- | --- |
| **%SSSSS** | **Mode** | **Note** |
| 00000 | [smart pin off; normal operation (default)](#_uz8yatawtj0x) |  |
| 00001 | [long repository](#_389j2sifyw3d) | M[12:10] != %101 (not DAC\_MODE) |
| 00010 | [long repository](#_389j2sifyw3d) | M[12:10] != %101 (not DAC\_MODE) |
| 00011 | [long repository](#_389j2sifyw3d) | M[12:10] != %101 (not DAC\_MODE) |
| 00001 | [DAC noise](#_vq8qbvdsgtmy) | M[12:10] = %101 (DAC\_MODE) |
| 00010 | [DAC 16-bit dither, noise](#_ccftirya86pz) | M[12:10] = %101 (DAC\_MODE) |
| 00011 | [DAC 16-bit dither, PWM](#_ujo617c8c6ef) | M[12:10] = %101 (DAC\_MODE) |
| 001001 | [pulse/cycle output](#_pyq9vdjyr3nf) |  |
| 001011 | [transition output](#_m2xocuv28l0p) |  |
| 001101 | [NCO frequency](#_5089qnhgckp9) |  |
| 001111 | [NCO duty](#_tes64dpabufd) |  |
| 010001 | [PWM triangle](#_khuaicke4gpr) |  |
| 010011 | [PWM sawtooth](#_ojj92ylxos1v) |  |
| 010101 | [PWM switch-mode power supply, V and I feedback](#_iyi7bgk5e0w2) |  |
| 01011 | [periodic/continuous: A-B quadrature encoder](#_31fs6vsrpp1i) |  |
| 01100 | [periodic/continuous: inc on A-rise & B-high](#_6zttwyip9utx) |  |
| 01101 | [periodic/continuous: inc on A-rise & B-high / dec on A-rise & B-low](#_k8acld6ipfl6) |  |
| 01110 | [periodic/continuous: inc on A-rise](#_o39n85ho5pd9) {[/ dec on B-rise](#_h9vrm8biinn0)} |  |
| 01111 | [periodic/continuous: inc on A-high](#_ssvd8afsp4fl) {[/ dec on B-high](#_groh7zsh3g6n)} |  |
| 10000 | [time A-states](#_wjx73ki24dvv) |  |
| 10001 | [time A-highs](#_ryn87kyb46wk) |  |
| 10010 | [time X A-highs/rises/edges](#_xqx7qqsbo9xv) -or- [timeout on X A-high/rise/edge](#_hevxvj4egl1u) |  |
| 10011 | [count time for X periods](#_w31f0bw5a40y) |  |
| 10100 | [count state for X periods](#_rfhshokjxp9e) |  |
| 10101 | [count time for periods in X+ clocks](#_anpxnz68bcpx) |  |
| 10110 | [count states for periods in X+ clocks](#_mfniwooksdz8) |  |
| 10111 | [count periods for periods in X+ clocks](#_d06gvojuosri) |  |
| 11000 | [ADC sample/filter/capture, internally clocked](#_gwuggjybik6u) |  |
| 11001 | [ADC sample/filter/capture, externally clocked](#_whxbrxc3tgf7) |  |
| 11010 | [ADC scope with trigger](#_g52hccxy3lz3) |  |
| 110111 | [USB host/device](#_4oyjpcfjd50a) | even/odd pin pair = DM/DP |
| 111001 | [sync serial transmit](#_8klbwkvu2qes) | A-data, B-clock |
| 11101 | [sync serial receive](#_yilm9fy7a2dt) | A-data, B-clock |
| 111101 | [async serial transmit](#_xdd3ort80he8) | baud rate |
| 11111 | [async serial receive](#_ak5mbeig8bh1) | baud rate |

1 OUT signal overridden

Each mode from the [Smart Pin Modes](#_1hv5dzbv2qd1) table is described below. The Smart Pin Mode is set (along with Pin Mode) using the WRPIN instruction ([see the %SSSSS bit field](#_jxahsscq2nua)).

### Smart Pin Off; Default (%00000)

Normal operation, without any smart pin functionality.

### Long Repository (%00001..%00011 and not DAC\_MODE)

Turns the smart pin into a long repository, where WXPIN writes the long and RDPIN / RQPIN can read the long.

* Upon each WXPIN, IN is raised.
* During reset (DIR=0), IN is low.

### 

### DAC Noise (%00001 and DAC\_MODE)

* Overrides M[7:0] to feed the pin's 8-bit DAC unique pseudo-random data on every clock. M[12:10] must be set to %101 to configure the low-level pin for DAC output.
* X[15:0] can be set to a sample period, in clock cycles, in case you want to mark time with IN raising at each period completion. If a sample period is not wanted, set X[15:0] to zero (65,536 clocks), in order to maximize the unused sample period, thereby reducing switching power.
* RDPIN / RQPIN can be used to retrieve the 16-bit ADC accumulation from the last sample period.
* During reset (DIR=0), IN is low.

### DAC 16-Bit With Noise Dither (%00010 and DAC\_MODE)

* Overrides M[7:0] to feed the pin's 8-bit DAC with pseudo-randomly-dithered data on every clock. M[12:10] must be set to %101 to configure the low-level pin for DAC output.
* X[15:0] establishes the sample period in clock cycles.
* Y[15:0] establishes the DAC output value which gets captured at each sample period and used for its duration.
* On completion of each sample period, Y[15:0] is captured for the next output value and IN is raised. Therefore, you would coordinate updating Y[15:0] with IN going high.
* Pseudo-random dithering does not require any kind of fixed period, as it randomly dithers the 8-bit DAC between adjacent levels, in order to achieve 16-bit DAC output, averaged over time. So, if you would like to be able to update the output value at any time and have it take immediate effect, set X[15:0] to one (IN will stay high).
* Y[15:0] values larger than $FF00 are invalid and will be clipped down to $FF00.
* If OUT is high, the ADC will be enabled and RDPIN / RQPIN can be used to retrieve the 16-bit ADC accumulation from the last sample period. This can be used to measure loading on the DAC pin.
* During reset (DIR=0), IN is low and Y[15:0] is captured.

### DAC 16-Bit With PWM dither (%00011 and DAC\_MODE)

* Overrides MP[7:0] to feed the pin's 8-bit DAC with PWM-dithered data on every clock. M[12:10] must be set to %101 to configure the low-level pin for DAC output.
* X[15:0] establishes the sample period in clock cycles. The sample period must be a multiple of 256 (X[7:0]=0), so that an integral number of 256 steps are afforded the PWM, which dithers the DAC between adjacent 8-bit levels.
* Y[15:0] establishes the DAC output value which gets captured at each sample period and used for its duration.
* On completion of each sample period, Y[15:0] is captured for the next output value and IN is raised. Therefore, you would coordinate updating Y[15:0] with IN going high.
* PWM dithering will give better dynamic range than pseudo-random dithering, since a maximum of only two transitions occur for every 256 clocks. This means, though, that a frequency of Fclock/256 will be present in the output at -48dB.
* Y[15:0] values larger than $FF00 are invalid and will be clipped down to $FF00.
* If OUT is high, the ADC will be enabled and RDPIN / RQPIN can be used to retrieve the 16-bit ADC accumulation from the last sample period. This can be used to measure loading on the DAC pin.
* During reset (DIR=0), IN is low and Y[15:0] is captured.

### Pulse/Cycle Output (%00100)

* Overrides OUT to control the pin output state.
* X[15:0] establishes a base period in clock cycles which forms the empirical high-time and low-time units.
* X[31:16] establishes a value to which the base period counter will be compared to on each clock cycle, as it counts from X[15:0] down to 1, before starting over at X[15:0] if decremented Y > 0. On each clock, if the base period counter > X[31:16] and Y > 0, the output will be high (else low).
* Whenever Y[31:0] is written with a non-zero value, the pin will begin outputting a high pulse or cycles, starting at the next base period. After each pulse, Y is decremented by one, until it reaches zero, at which the output will remain low. Examples:
  + If X[31:16] is set to 0, the output will be high for the duration of Y > 0.
  + If X[15:0] is set to 3 and X[31:16] is set to 2, the output will be 0-0-1 (repeat) for duration of Y > 0.
* IN will be raised and the pin will revert to low output when the pulse or cycles complete, meaning Y has been decremented to zero.
* During reset (DIR=0), IN is low, the output is low, and Y is set to zero.

### Transition Output (%00101)

* Overrides OUT to control the pin output state.
* X[15:0] establishes a base period in clock cycles which forms the empirical high-time and low-time units.
* Whenever Y[31:0] is written with a non-zero value, the pin will begin toggling for Y transitions at each base period, starting at the next base period.
* IN will be raised when the transitions complete, with the pin remaining in its current output state.
* During reset (DIR=0), IN is low, the output is low, and Y is set to zero.

### NCO Frequency (%00110)

* Overrides OUT to control the pin output state.
* X[15:0] establishes a base period in clock cycles which forms the empirical high-time and low-time units.
* Upon WXPIN, X[31:16] is written to Z[31:16] to allow phase setting.
* Y[31:0] will be added into Z[31:0] at each base period.
* The pin output will reflect Z[31].
* IN will be raised whenever Z overflows.
* During reset (DIR=0), IN is low, the output is low, and Z is set to zero.

### NCO Duty (%00111)

* Overrides OUT to control the pin output state.
* X[15:0] establishes a base period in clock cycles which forms the empirical high-time and low-time units.
* Upon WXPIN, X[31:16] is written to Z[31:16] to allow phase setting.
* Y[31:0] will be added into Z[31:0] at each base period.
* The pin output will reflect Z overflow.
* IN will be raised whenever Z overflows.
* During reset (DIR=0), IN is low, the output is low, and Z is set to zero.

### PWM Triangle (%01000)

* Overrides OUT to control the pin output state.
* X[15:0] establishes a base period in clock cycles which forms the empirical high-time and low-time units.
* X[31:16] establishes a PWM frame period in terms of base periods.
* Y[15:0] establishes the PWM output value which gets captured at each frame start and used for its duration. It should range from zero to the frame period.
* A counter, updating at each base period, counts from the frame period down to one, then from one back up to the frame period. Then, Y[15:0] is captured, IN is raised, and the process repeats.
* At each base period, the captured output value is compared to the counter. If it is equal or greater, a high is output. If it is less, a low is output. Therefore, a zero will always output a low and the frame period value will always output a high.
* During reset (DIR=0), IN is low, the output is low, and Y[15:0] is captured.

### PWM Sawtooth (%01001)

* Overrides OUT to control the pin output state.
* X[15:0] establishes a base period in clock cycles which forms the empirical high-time and low-time units.
* X[31:16] establishes a PWM frame period in terms of base periods.
* Y[15:0] establishes the PWM output value which gets captured at each frame start and used for its duration. It should range from zero to the frame period.
* A counter, updating at each base period, counts from one up to the frame period. Then, Y[15:0] is captured, IN is raised, and the process repeats.
* At each base period, the captured output value is compared to the counter. If it is equal or greater, a high is output. If it is less, a low is output. Therefore, a zero will always output a low and the frame period value will always output a high.
* During reset (DIR=0), IN is low, the output is low, and Y[15:0] is captured.

### PWM Switch-Mode Power Supply With Voltage And Current Feedback (%01010)

* Overrides OUT to control the pin output state.
* X[15:0] establishes a base period in clock cycles which forms the empirical high-time and low-time units.
* X[31:16] establishes a PWM frame period in terms of base periods.
* Y[15:0] establishes the PWM output value which gets captured at each frame start and used for its duration. It should range from zero to the frame period.
* A counter, updating at each base period, counts from one up to the frame period. Then, the 'A' input is sampled at each base period until it reads low. After 'A' reads low, Y[15:0] is captured, IN is raised, and the process repeats.
* At each base period, the captured output value is compared to the counter. If it is equal or greater, a high is output. If it is less, a low is output. If, at any time during the cycle, the 'B' input goes high, the output will be low for the rest of that cycle.
* Due to the nature of switch-mode power supplies, it may be appropriate to just set Y[15:0] once and let it repeat indefinitely.
* During reset (DIR=0), IN is low, the output is low, and Y[15:0] is captured.
  + WXPIN is used to set the base period (X[15:0]) and the PWM frame count (X[31:16]). The base period is the number of clocks which makes a base unit of time. The frame count is the number of base units that make up a PWM cycle.
  + WYPIN is used to set the output value (Y[15:0]), which is internally captured at the start of every PWM frame and compared to the frame counter upon completion of each base unit of time. If the output value is greater than or equal to the frame counter, the pin outputs a high, else a low. This is intended to drive the gate of the switcher FET.
  + The "A" input is the voltage detector for the SMPS output. This could be an adjacent pin using the internal-DAC-comparison mode to observe the center tap of a voltage divider which is fed by the final SMPS output. When "A" is low, a PWM cycle is performed because the final output voltage has sagged below the requirement and it's time to do another pulse.
  + The "B" input is the over-current detector which, if ever high during the PWM cycle, immediately forces the output low for the rest of that PWM cycle. This could be an adjacent pin using the internal-DAC-comparison mode to observe a shunt resistor between GND and the FET source. When the shunt voltage gets too high, too much current is flowing (or the desired amount of current is flowing), so the output goes low to turn off the FET and allow the inductor connected to its drain to shoot high, creating a power pulse to be captured by a diode and dumped into a cap, which is the SMPS final output.

### 

### A/B-Input Quadrature Encoder (%01011)

* X[31:0] establishes a measurement period in clock cycles.
* If zero is used for the period, the measurement operation will not be periodic, but continuous, like a totalizer, and the current 32-bit quadrature step count can always be read via RDPIN / RQPIN.
* If a non-zero value is used for the period, quadrature steps will be counted for that many clock cycles and then the result will be placed in Z while the accumulator will be set to the 0/1/-1 value that would have otherwise been added into it. This way, all quadrature steps get counted across measurements. At the end of each period, IN will be raised and RDPIN / RQPIN can be used to retrieve the last 32-bit measurement.
* It may be useful to configure both 'A' and 'B' smart pins to quadrature mode, with one being continuous (X=0) for absolute position tracking and the other being periodic (x<>0) for velocity measurement.
* The quadrature encoder can be "zeroed" by pulsing DIR low at any time; no need to do another WXPIN.
* During reset (DIR=0), IN is low and Z is set to the adder value (0/1/-1).

### Count A-Input Positive Edges When B-Input Is High (%01100)

* X[31:0] establishes a measurement period in clock cycles.
* If zero is used for the period, the measurement operation will not be periodic, but continuous, like a totalizer, and the current 32-bit high count can always be read via RDPIN / RQPIN.
* If a non-zero value is used for the period, events will be counted for that many clock cycles and then the result will be placed in Z, while the accumulator will be set to the 0/1 value that would have otherwise been added into it, beginning a new measurement. This way, all events get counted across measurements. At the end of each period, IN will be raised and RDPIN / RQPIN can be used to retrieve the 32-bit measurement.
* During reset (DIR=0), IN is low and Z is set to the adder value (0/1).

### Count A-Input Positive Edges; Increment w/B-Input = 1, Decrement w/B-Input = 0 (%01101)

* X[31:0] establishes a measurement period in clock cycles.
* If zero is used for the period, the measurement operation will not be periodic, but continuous, like a totalizer, and the current 32-bit high count can always be read via RDPIN / RQPIN.
* If a non-zero value is used for the period, events will be counted for that many clock cycles and then the result will be placed in Z, while the accumulator will be set to the 0/1/-1 value that would have otherwise been added into it, beginning a new measurement. This way, all events get counted across measurements. At the end of each period, IN will be raised and RDPIN / RQPIN can be used to retrieve the 32-bit measurement.
* During reset (DIR=0), IN is low and Z is set to the adder value (0/1/-1).

### Count A-Input Positive Edges (%01110 AND !Y[0])

### Increment w/A-Input Positive Edge, Decrement w/B-Input Positive Edge (%01110 AND Y[0])

* X[31:0] establishes a measurement period in clock cycles. Y[0] establishes whether to just count A-input positive edges (=0), or to increment on A-input positive edge and decrement on B-input positive edge (=1).
* If zero is used for the period, the measurement operation will not be periodic, but continuous, like a totalizer, and the current 32-bit high count can always be read via RDPIN / RQPIN.
* If a non-zero value is used for the period, events will be counted for that many clock cycles and then the result will be placed in Z, while the accumulator will be set to the 0/1/-1 value that would have otherwise been added into it, beginning a new measurement. This way, all events get counted across measurements. At the end of each period, IN will be raised and RDPIN / RQPIN can be used to retrieve the 32-bit measurement.
* During reset (DIR=0), IN is low and Z is set to the adder value (0/1/-1).

### Count A-Input Highs (%01111 AND !Y[0])

### Increment w/A-Input High, Decrement w/B-Input High (%01111 AND Y[0])

* X[31:0] establishes a measurement period in clock cycles. Y[0] establishes whether to just count A-input highs (=0), or to increment on A-input high and decrement on B-input high (=1).
* If zero is used for the period, the measurement operation will not be periodic, but continuous, like a totalizer, and the current 32-bit high count can always be read via RDPIN / RQPIN.
* If a non-zero value is used for the period, events will be counted for that many clock cycles and then the result will be placed in Z, while the accumulator will be set to the 0/1/-1 value that would have otherwise been added into it, beginning a new measurement. This way, all events get counted across measurements. At the end of each period, IN will be raised and RDPIN / RQPIN can be used to retrieve the 32-bit measurement.
* During reset (DIR=0), IN is low and Z is set to the adder value (0/1/-1).

### Time A-Input States (%10000)

* Continuous states are counted in clock cycles.
* Upon each state change, the prior state is placed in the C-flag buffer, the prior state's duration count is placed in Z, and IN is raised. RDPIN / RQPIN can then be used to retrieve the measurement. Z will be limited to $80000000.
* If states change faster than the cog is able to retrieve measurements, the measurements will effectively be lost, as old ones will be overwritten with new ones. This may be gotten around by using two smart pins to time highs, with one pin inverting its 'A' input. Then, you could capture both states, as long as the sum of the states' durations didn't exceed the cog's ability to retrieve both results. This would help in cases where one of the states was very short in duration, but the other wasn't.
* During reset (DIR=0), IN is low and Z is set to $00000001.

### Time A-Input High States (%10001)

* Continuous high states are counted in clock cycles.
* Upon each high-to-low transition, the previous high duration count is placed in Z, and IN is raised. RDPIN/RQPIN can then be used to retrieve the measurement. Z will be limited to $80000000.
* During reset (DIR=0), IN is low and Z is set to $00000001.

### Time X A-Input Highs/Rises/Edges (%10010 AND !Y[2])

* Time is measured until X A-input highs/rises/edges are accumulated.
* X[31:0] establishes how many A-input highs/rises/edges are to be accumulated.
* Y[1:0] establishes A-input high/rise/edge sensitivity:
  + %00 = A-input high
  + %01 = A-input rise
  + %1x = A-input edge
* Time is measured in clock cycles until X highs/rises/edges are accumulated from the A-input. The measurement is then placed in Z, and IN is raised. RDPIN / RQPIN can then be used to retrieve the measurement. Z will be limited to $80000000.
* During reset (DIR=0), IN is low and Z is set to $00000001.

### Timeout on X Clocks Of Missing A-Input High/Rise/Edge (%10010 AND Y[2])

* If no A-input high/rise/edge occurs within X clocks, IN is raised, a new timeout period of X clocks begins, and Z maintains a running count of how many clocks have elapsed since the last A-input high/rise/edge. Z will be limited to $80000000 and can be read any time via RDPIN / RQPIN.
* If an A-input high/rise/edge does occur within X clocks, a new timeout period of X clocks begins and Z is reset to $00000001.
* X[31:0] establishes how many clocks before a timeout due to no A-input high/rise/edge occurring.
* Y[1:0] establishes A-input high/rise/edge sensitivity:
  + %00 = A-input high
  + %01 = A-input rise
  + %1x = A-input edge
* During reset (DIR=0), IN is low and Z is set to $00000001.

### Count Time For X Periods (%10011)

### Count State For X Periods (%10100)

* X[31:0] establishes how many A-input rise/edge to B-input rise/edge periods are to be measured.
* Y[1:0] establishes A-input and B-input rise/edge sensitivity:
  + %00 = A-input rise to B-input rise
  + %01 = A-input rise to B-input edge
  + %10 = A-input edge to B-input rise
  + %11 = A-input edge to B-input edge
  + Note: The B-input can be set to the same pin as the A-input for single-pin cycle measurement.
* Clock cycles or A-input trigger states are counted from each A-input rise/edge to each B-input rise/edge for X periods. If the A-input rise/edge is ever coincident with the B-input rise/edge at the end of the period, the start of the next period is registered. Upon completion of X periods, the measurement is placed in Z, IN is raised, and a new measurement begins. RDPIN / RQPIN can then be used to retrieve the completed measurement. Z will be limited to $80000000.
* The first mode is intended to be used as an oversampling period measurement, while the second mode is a complementary duty measurement.
* During reset (DIR=0), IN is low and Z is set to $00000000.

### Count Time For Periods In X+ Clock Cycles ( %10101)

### Count States For Periods In X+ Clock Cycles (%10110)

### Count Periods For Periods In X+ Clock Cycles (%10111)

* X[31:0] establishes the minimum number of clock cycles to track periods for. Periods are A-input rise/edge to B-input rise/edge.
* Y[1:0] establishes A-input and B-input rise/edge sensitivity:
  + %00 = A-input rise to B-input rise
  + %01 = A-input rise to B-input edge
  + %10 = A-input edge to B-input rise
  + %11 = A-input edge to B-input edge
  + Note: The B-input can be set to the same pin as the A-input for single-pin cycle measurement.
* A measurement is taken across some number of A-input rise/edge to B-input rise/edge periods, until X clock cycles elapse and then any period in progress completes. If the A-input rise/edge is ever coincident with the B-input rise/edge at the end of the period, the start of the next period is registered. Upon completion, the measurement is placed in Z, IN is raised, and a new measurement begins. RDPIN / RQPIN can then be used to retrieve the completed measurement. Z will be limited to $80000000.
* The first mode accumulates time within each period, for an oversampled period measurement.
* The second mode accumulates A-input trigger states within each period, for an oversampled duty measurement.
* The third mode counts the periods.
* Knowing how many clock cycles some number of complete periods took, and what the duty was, affords a very time-efficient and precise means of determining frequency and duty cycle. At least two of these measurements must be made concurrently to get useful results.
* During reset (DIR=0), IN is low and Z is set to $00000000.

### ADC Sample/Filter/Capture, Internally Clocked (%11000)

### ADC Sample/Filter/Capture, Externally Clocked (%11001)

These modes facilitate sampling, SINC filtering, and raw capturing of ADC bitstream data.

* For the internally-clocked mode, the A-input will be sampled on every clock and should be a pin configured for ADC operation (M[12:10] = %100). In the externally-clocked mode, the A-input will be sampled on each B-input rise, so that an external delta-sigma ADC may be employed.
* WXPIN sets the mode to X[5:4] and the sample period to POWER(2, X[3:0]). Not all mode and period combinations are useful, or even functional:

| Sample/Filter/Capture Configurations | | | | | |
| --- | --- | --- | --- | --- | --- |
|  | **X[5:4] ➡**  **Mode ➡** | **%00**  **SINC2 Sampling** | **%01**  **SINC2 Filtering** | **%10**  **SINC3 Filtering** | **%11**  **Bitstream Capturing** |
| **X[3:0]** | **Sample Period** | **Sample Resolution** | **Post-diff ENOB1** | **Post-diff ENOB1** | **(LSB = oldest bit)** |
| %0000 | 1 clock | impractical | impractical | impractical | 1 new bit |
| %0001 | 2 clocks | 2 bits | impractical | impractical | 2 new bits |
| %0010 | 4 clocks | 3 bits | impractical | impractical | 4 new bits |
| %0011 | 8 clocks | 4 bits | 4 | impractical | 8 new bits |
| %0100 | 16 clocks | 5 bits | 5 | 8 | 16 new bits |
| %0101 | 32 clocks | 6 bits | 6 | 10 | 32 new bits |
| %0110 | 64 clocks | 7 bits | 7 | 12 | overflow |
| %0111 | 128 clocks | 8 bits | 8 | 14 | overflow |
| %1000 | 256 clocks | 9 bits | 9 | 16 | overflow |
| %1001 | 512 clocks | 10 bits | 10 | 18 | overflow |
| %1010 | 1,024 clocks | 11 bits | 11 | overflow | overflow |
| %1011 | 2,048 clocks | 12 bits | 12 | overflow | overflow |
| %1100 | 4,096 clocks | 13 bits | 13 | overflow | overflow |
| %1101 | 8,192 clocks | 14 bits | 14 | overflow | overflow |
| %1110 | 16,384 clocks | overflow | overflow | overflow | overflow |
| %1111 | 32,768 clocks | overflow | overflow | overflow | overflow |

1 ENOB = Effective Number of Bits, or the sample resolution

* For modes other than SINC2 Sampling (X[5:4] > %00), WYPIN may be used after WXPIN to override the initial period established by X[3:0] and replace it with the arbitrary value in Y[13:0]. For example, if you'd like to do SINC3 filtering with a period of 320 clocks, you could follow the WXPIN with a 'WYPIN #320,adcpin'. The smart pin accumulators are 27 bits wide. This allows up to 2^(27/3), or 512, clocks per decimation in SINC3 filtering mode and up to 2^(27/2), or 11,585, clocks in SINC2 filtering mode.
* Upon completion of each sample period, the measurement is placed in Z, IN is raised, and a new measurement begins. RDPIN / RQPIN can then be used to retrieve the completed measurement.

#### About SINC2 and SINC3 filtering

SINC2 filtering works by summing the input bit into an accumulator on each clock which, in turn, is summed into another accumulator, to create a double integration. At the end of each sampling period, the difference between the new and previous second accumulator's value is the conversion sample, and the 'previous' value is updated. This process has the pleasant effect of returning an extra bit of resolution over simple bit-summing, as well as filtering away rectangular-sampling-window effects. SINC2 filtering is best for DC measurements, where precision is important. Practical measurements of 14-bit resolution can be made every 8,192 clocks using SINC2 filtering. After starting SINC2 filtering, the filter will become accurate starting on the second period.

SINC3 filtering is like SINC2, but employs an additional level of accumulation to increase sensitivity to dynamics in the input signal. SINC3 doubles the theoretical ENOB (effective number of bits) over simple bit-summing for fast signals, but it is only slightly better at DC measurements than SINC2 filtering at the same sample period. Because SINC3 takes more resources within the smart pin, it is limited to 512 samples per period, making it less practical than SINC2 for precision DC measurements, but quite ideal for tracking fast, dynamic signals. After starting SINC3 filtering, the filter will become accurate starting on the third period.

Because the accumulators are 27 bits wide, 32-bit integer adds and subtracts in software will roll over incorrectly. There are two ways to handle this:

You can either prescale the 27-bit values to 32-bit values:

RDPIN x,#adcpin 'get SINC2 accumulator

SHL x,#5 'prescale 27-bit to 32-bit

SUB x,diff 'compute sample

ADD diff,x 'update diff value

Or you can post-trim then to 27-bit values:

RDPIN x,#adcpin 'get SINC2 accumulator

SUB x,diff 'compute sample

ADD diff,x 'update diff value

ZEROX x,#26 'trim to 27-bit

#### SINC2 Sampling Mode (%00)

This mode performs complete SINC2 conversions, updating the ADC output sample at the end of each period. Once this mode is enabled, it is only necessary to do a RDPIN / RQPIN to acquire the latest ADC sample. The limitation of this mode is that it only works at power-of-2 sample periods, since that stricture afforded efficient implementation within the smart pin, making complete conversions possible without software. There is an additional SINC2 filtering mode (%01) which allows non-power-of-2 sample periods, but you must perform the difference computation in software.

To begin SINC2 sampling:

WRPIN ##%100011\_0000000\_00\_11000\_0,adcpin 'configure ADC+sample pin(s)

WXPIN #%00\_0111,adcpin 'SINC2 sampling at 8 bits

DIRH adcpin 'enable smart pin(s)

NOTE: The variable 'adcpin' could enable multiple pins by having the additional number of pins in bits 10..6. For example, if 'adcpin' held %00111\_010000, pins 16 through 23 would have been simultaneously configured by the above code.

To read the latest ADC sample, just do a RDPIN / RQPIN:

RDPIN sample,adcpin 'read sample at any time

#### 

#### SINC2 Filtering Mode (%01)

This mode performs SINC2 filtering, which requires some software interaction in order to realize ADC samples.

To begin SINC2 filtering:

WRPIN ##%100011\_0000000\_00\_11000\_0,#adcpin 'configure ADC+filter pin(s)

WXPIN #%01\_0111,#adcpin 'SINC2 filtering at 128 clocks

DIRH #adcpin 'enable smart pin(s)

Pin interaction must occur after each sample period, so it may be good to set up an event to detect the pin's IN going high:

SETSE1 #%001<<6 + adcpin 'SE1 triggers on pin high

.loop WAITSE1 'wait for sample period done

RDPIN x,#adcpin 'get SINC2 accumulator

SUB x,diff 'compute sample

ADD diff,x 'update diff value

SHR x,#6 'justify 8-bit sample

ZEROX x,#7 'trim 8-bit sample

'use x here 'use sample somehow

JMP #.loop 'loop for next period

x RES 1 'sample value

diff RES 1 'diff value

Note that it is necessary to shift the computed sample right by some number of bits to leave the ENOBs intact. For SINC2 filtering, you must shift right by LOG2(clocks per period)-1, which in this case is LOG2(128)-1 = 6.

#### SINC3 Filtering Mode (%10)

This mode performs SINC3 filtering, which requires some software interaction in order to realize ADC samples.

To begin SINC3 filtering:

WRPIN ##%100011\_0000000\_00\_11000\_0,#adcpin 'configure ADC+filter pin(s)

WXPIN #%10\_0111,#adcpin 'SINC3 filtering at 128 clocks

DIRH #adcpin 'enable smart pin(s)

Pin interaction must occur after each sample period, so it may be good to set up an event to detect the pin's IN going high:

SETSE1 #%001<<6 + adcpin 'SE1 triggers on pin high

.loop WAITSE1 'wait for sample period done

RDPIN x,#adcpin 'get SINC3 accumulator

SUB x,diff1 'compute sample

ADD diff1,x 'update diff1 value

SUB x,diff2 'compute sample

ADD diff2,x 'update diff2 value

SHR x,#7 'justify 14-bit sample

ZEROX x,#13 'trim 14-bit sample

'use x here 'use sample somehow

JMP #.loop 'loop for next period

x RES 1 'sample value

diff1 RES 1 'diff1 value

diff2 RES 1 'diff2 value

Note that it is necessary to shift the computed sample right by some number of bits to leave the ENOBs intact. For SINC3 filtering, you must shift right by LOG2(clocks per period), which in this case is LOG2(128) = 7.

#### Bitstream Capturing Mode (%11)

This mode captures the raw bitstream coming from the ADC. It buffers 32 bits and is meant to be read once every 32 clocks, in order to get contiguous snapshots of the ADC bitstream. RDPIN / RQPIN is used to read the snapshots. Bit 31 of the data will be the most recent ADC bit, while bit 0 will be from 31 clocks earlier.

To begin raw bitstream capturing:

WRPIN ##%100011\_0000000\_00\_11000\_0,adcpin 'configure ADC+sample pin(s)

WXPIN #%11\_0101,adcpin 'raw sampling every 32 clocks

DIRH adcpin 'enable smart pin(s)

To get a snapshot of the latest 32 bits of the ADC bitstream, just do a RDPIN / RQPIN:

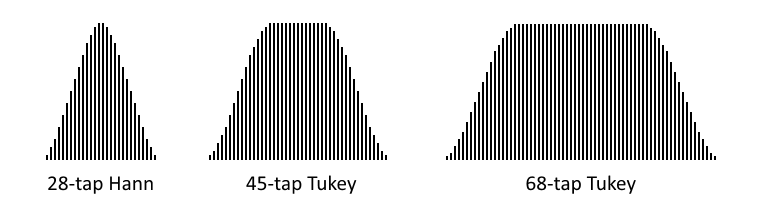
RDPIN bitstream,adcpin 'get snapshot of ADC bitstream

This mode can be used for purposes other than capturing ADC bitstreams. It is really just capturing the A-input without regard to pin configuration.

### ADC Scope With Trigger (%11010)

This mode calculates an 8-bit ADC sample and checks for hysteretic triggering on every clock, providing the basis of oscilloscope functionality. Samples from blocks of up to four pins can be grouped into a 32-bit data pipe for recording by the streamer or reading by the GETSCP instruction (see 'SCOPE Data Pipe' below).

There are three different windowed filter functions from which ADC samples can be computed. On each clock, the incoming ADC bit is shifted into a tap string and the weighted tap bits are summed together to produce the sample. The samples are normalized to 8 bits in size, but the DC dynamic range is ~5 to ~6 bits, depending on the filter length. These are plots of the actual filter shapes and sizes:



The scope trigger function is set by two 6-bit parameters, A and B, which MSB-justify to the 8-bit samples for comparison. Triggering is a two-step process of arming and then triggering, which raises the IN signal and waits for a new arming event. The relationship between A and B determine the triggering pattern:

| **A and B**  **relationship** | **Arming Event**  **(initial / after trigger)** | **Trigger Event**  **(after arming)** |
| --- | --- | --- |
| A > B | sample[7:2] => A | sample[7:2] < B |
| A <= B | sample[7:2] < A | sample[7:2] => B |

* WXPIN is used to configure this mode.
* X[15:10] sets the B trigger value.
* X[7:2] sets the A trigger value.
* X[1:0] selects the filter:
  + %00 = 68-tap Tukey filter
  + %01 = 45-tap Tukey filter
  + %1x = 28-tap Hann filter
* RDPIN / RQPIN always returns the 8-bit sample, along with the 'armed' state in the C flag.
* When 'armed' and then 'triggered', IN is raised and the 'armed' state is canceled.

#### SCOPE Data Pipe

Each cog has a 32-bit SCOPE data pipe which is intended to be used with smart pins configured to the 'scope' mode. The SCOPE data pipe continuously aggregates the lower bytes of RDPIN values from a 4-pin block, so that the streamer can record up to four time-aligned 8-bit ADC samples per clock. They can also be read at once via the GETSCP instruction.

The SETSCP instruction enables the SCOPE data pipe and selects the 4-pin block whose lower bytes of RDPIN values it will continuously carry:

SETSCP {#}D 'D[6] enables the SCOPE data pipe, D[5:2] selects the 4-pin block

The GETSCP instruction gets the SCOPE data pipe's current four bytes:

GETSCP D 'Get the lower-byte RDPIN values of four pins into the bytes of D

If the SCOPE data pipe didn't exist, the closest you could come to the GETSCP instruction would be this sequence, which would not have time-aligned samples:

RQPIN x,#pinblock | 3 'read pin3 long into x

ROLBYTE y,x 'rotate pin3 byte into y

RQPIN x,#pinblock | 2 'read pin2 long into x

ROLBYTE y,x 'rotate pin2 byte into y

RQPIN x,#pinblock | 1 'read pin1 long into x

ROLBYTE y,x 'rotate pin1 byte into y

RQPIN x,#pinblock | 0 'read pin0 long into x

ROLBYTE y,x 'rotate pin0 byte into y

The SCOPE data pipe is generic in function and may find other uses than carrying just 'scope' data.

### USB Host/Device (%11011)

This mode requires that two adjacent pins be configured together to form a USB pair, whose OUTs will be overridden to control their output states. These pins must be an even/odd pair, having only the LSB of their pin numbers different. For example: pins 0 and 1, pins 2 and 3, and pins 4 and 5 can form USB pairs. They can be configured via WRPIN with identical D data of %1\_11011\_0. Using D data of %0\_11011\_0 will disable output drive and effectively create a USB 'sniffer'. A new WRPIN can be done to effect such a change without resetting the smart pin.

WXPIN is used on the lower pin to establish the specific USB mode and set the baud rate. D[15] must be 1 for 'host' or 0 for 'device'. D[14] must be 1 for 'full-speed' or 0 for 'low-speed'. D[13:0] sets the baud rate, which is a 16-bit fraction of the system clock, whose two MSBs must be 0, necessitating that the baud rate be less than 1/4th of the system clock frequency. For example, if the main clock is 80MHz and you want a 12MHz baud rate (full-speed), use 12,000,000 / 80,000,000 \* $10000 = 9830, or $2666. To use this baud rate and select 'host' mode and 'full-speed', you could do 'WXPIN ##$E666,lowerpin'.

The upper (odd) pin is the DP pin. This pin's IN is raised whenever the output buffer empties, signaling that a new output byte can be written via WYPIN to the lower (even) pin. No WXPIN / WYPIN instructions are used for this pin.

The lower (even) pin is the DM pin. This pin's IN is raised whenever a change of status occurs in the receiver, at which point a RDPIN / RQPIN can be used on this pin to read the 16-bit status word. WXPIN is used on this pin to set the NCO baud rate.

These DP/DM electrical designations can actually be switched by swapping low-speed and full-speed modes, due to USB's complementary line signaling.

To start USB, clear the DIR bits of the intended two pins and configure them each via WRPIN. Use WXPIN on the lower pin to set the mode and baud rate. Then, set the pins' DIR bits. You are now ready to read the receiver status via RDPIN / RQPIN and set output states and send packets via WYPIN, both on the lower pin.

To affect the line states or send a packet, use WYPIN on the lower pin. Here are its D values:

0 = output IDLE - default state, float pins, except possible resistor(s) to 3.3V or GND

1 = output SE0 - drive both DP and DM low

2 = output K - drive K state onto DP and DM (opposite)

3 = output J - drive J state onto DP and DM (opposite), like IDLE, but driven

4 = output EOP - output end-of-packet: SE0, SE0, J, then IDLE

$80 = SOP - output start-of-packet, then bytes, automatic EOP when buffer runs out

To send a packet, first do a 'WYPIN #$80,lowerpin'. Then, after each IN rise on the upper pin, do a 'WYPIN byte,lowerpin' to buffer the next byte. The transmitter will automatically send an EOP when you stop giving it bytes. To keep the output buffer from overflowing, you should always verify that the upper pin's IN was raised after each WYPIN, before issuing another WYPIN, even if you are just setting a state. The reason for this is that all output activity is timed to the baud generator and even state changes must wait for the next bit period before being implemented, at which time the output buffer empties.

There are separate state machines for transmitting and receiving. Only the baud generator is common between them. The transmitter was just described above. Below, the receiver is detailed. Note that the receiver receives not just input from another host/device, but all local output, as well.

At any time, a RDPIN / RQPIN can be executed on the lower pin to read the current 16-bit status of the receiver, with the error flag going into C. The lower pin's IN will be raised whenever a change occurs in the receiver's status. This will necessitate A WRPIN / WXPIN / WYPIN / RDPIN / AKPIN before IN can be raised again, to alert of the next change in status. The receiver's status bits are as follows:

[31:16] <unused> - $0000

[15:8] byte - last byte received

[7] byte toggle - cleared on SOP, toggled on each byte received

[6] error - cleared on SOP, set on bit-unstuff error, EOP SE0 > 3 bits, or SE1

[5] EOP in - cleared on SOP or 7+ bits of J or K, set on EOP

[4] SOP in - cleared on EOP or 7+ bits of J or K, set on SOP

[3] SE1 in (illegal) - cleared on !SE1, set on 1+ bits of SE1

[2] SE0 in (RESET) - cleared on !SE0, set on 1+ bits of SE0

[1] K in (RESUME) - cleared on !K, set on 7+ bits of K

[0] J in (IDLE) - cleared on !J, set on 7+ bits of J

The result of a RDPIN/RQPIN can be bit-tested for events of interest. It can also be shifted right by 8 bits to LSB-justify the last byte received and get the byte toggle bit into C, in order to determine if you have a new byte. Assume that 'flag' is initially zero:

SHR D,#8 WC 'get byte into D, get toggle bit into C

CMPX flag,#1 WZ 'compare toggle bit to flag, new byte if Z

IF\_Z XOR flag,#1 'if new byte, toggle flag

IF\_Z <use byte> 'if new byte, do something with it

### Synchronous Serial Transmit (%11100)

* Overrides OUT to control the pin output state.
* Words of 1 to 32 bits are shifted out on the pin, LSB first, with each new bit being output two internal clock cycles after registering a positive edge on the B input. For negative-edge clocking, the B input may be inverted by setting B[3] in WRPIN's D value.
* WXPIN is used to configure the update mode and word length.
* X[5] selects the update mode:
  + X[5] = 0 sets continuous mode, where a first word is written via WYPIN during reset (DIR=0) to prime the shifter. Then, after reset (DIR=1), the second word is buffered via WYPIN and continuous clocking is started. Upon shifting each word, the buffered data written via WYPIN is advanced into the shifter and IN is raised, indicating that a new output word can be buffered via WYPIN. This mode allows steady data transmission with a continuous clock, as long as the WYPIN's after each IN-rise occur before the current word transmission is complete.
  + X[5] = 1 sets start-stop mode, where the current output word can always be updated via WYPIN before the first clock, flowing right through the buffer into the shifter. Any WYPIN issued after the first clock will be buffered and loaded into the shifter after the last clock of the current output word, at which time it could be changed again via WYPIN. This mode is useful for setting up the output word before a stream of clocks are issued to shift it out.
* X[4:0] sets the number of bits, minus 1. For example, a value of 7 will set the word size to 8 bits.
* WYPIN is used to load the output words. The words first go into a single-stage buffer before being advanced to the shifter for output. Each time the buffer is advanced into the shifter, IN is raised, indicating that a new output word can be written via WYPIN. During reset, the buffer flows straight into the shifter.
* If you intend to send MSB-first data, you must first shift and then reverse it. For example, if you had a byte in D that you wanted to send MSB-first, you would do a 'SHL D,#32-8' and then a 'REV D'.
* During reset (DIR=0) the output is held low. Upon release of reset, the output will reflect the LSB of the output word written by any WYPIN during reset.

### Synchronous Serial Receive (%11101)

* Words of 1 to 32 bits are shifted in by sampling the A input around the positive edge of the B input. For negative-edge clocking, the B input may be inverted by setting B[3] in WRPIN's D value.
* WXPIN is used to configure the sampling and word length.
* X[5] selects the A input sample position relative to the B input edge:
  + X[5] = 0 selects the A input sample just before the B input edge was registered. This requires no hold time on the part of the sender.
  + X[5] = 1 selects the sample coincident with the B edge being registered. This is useful where transmitted data remains steady after the B edge for a brief time. In the synchronous serial transmit mode, the data is steady for two internal clocks after the B edge was registered, so employing this complementary feature would enable the fastest data transmission when receiving from another smart pin in synchronous serial transmit mode.
* X[4:0] sets the number of bits, minus 1. For example, a value of 7 will set the word size to 8 bits.
* When a word is received, IN is raised and the data can then be read via RDPIN / RQPIN. The data read will be MSB-justified.
* If you received LSB-first data, it will require right-shifting, unless the word size was 32 bits. For a word size of 8 bits, you would need to do a 'SHR D,#32-8' to get the data LSB-justified.
* If you received MSB-first data, it will need to be reversed and possibly masked, unless the word size was 32 bits. For example, if you received a 9-bit word, you would do 'REV D' + 'ZEROX D,#8' to get the data LSB-justified.

### Asynchronous Serial Transmit (%11110)

* Overrides OUT to control the pin output state.
* Words from 1 to 32 bits are serially transmitted on the pin at a programmable baud rate, beginning with a low "start" bit and ending with a high "stop" bit.
* WXPIN is used to configure the baud rate and word length.
* X[31:16] establishes the number of clocks in a bit period, and in case X[31:26] is zero, X[15:10] establishes the number of fractional clocks in a bit period. The X bit period value can be simply computed as: (clocks \* $1\_0000) & $FFFFFC00. For example, 7.5 clocks would be $00078000, and 33.33 clocks would be $00215400.
* X[4:0] sets the number of bits, minus 1. For example, a value of 7 will set the word size to 8 bits.
* WYPIN is used to load the output words. The words first go into a single-stage buffer before being advanced to a shifter for output. This buffering mechanism makes it possible to keep the shifter constantly busy, so that gapless transmissions can be achieved. Any time a word is advanced from the buffer to the shifter, IN is raised, indicating that a new word can be loaded.
* Here is the internal state sequence:
  1. Wait for an output word to be buffered via WYPIN, then set the 'buffer-full' and 'busy' flags.
  2. Move the word into the shifter, clear the 'buffer-full' flag, and raise IN.
  3. Output a low for one bit period (the START bit).
  4. Output the LSB of the shifter for one bit period, shift right, and repeat until all data bits are sent.
  5. Output a high for one bit period (the STOP bit).
  6. If the 'buffer-full' flag is set due to an intervening WYPIN, loop to (b). Otherwise, clear the 'busy' flag and loop to (a).
* RDPIN / RQPIN with WC always returns the 'busy' flag into C. This is useful for knowing when a transmission has completed. The busy flag can be polled starting three clocks after the WYPIN, which loads the output words:

WYPIN x,#txpin 'load output word

WAITX #1 'wait 2+1 clocks before polling busy

wait RDPIN x,#txpin WC 'get busy flag into C

IF\_C JMP #wait 'loop until C = 0

* During reset (DIR=0) the output is held high.

### Asynchronous Serial Receive (%11111)

* Words from 1 to 32 bits are serially received on the A input at a programmable baud rate.
* WXPIN is used to configure the baud rate and word length.
* X[31:16] establishes the number of clocks in a bit period, and in case X[31:26] is zero, X[15:10] establishes the number of fractional clocks in a bit period. The X bit period value can be simply computed as: (clocks \* $1\_0000) & $FFFFFC00. For example, 7.5 clocks would be $00078000, and 33.33 clocks would be $00215400.
* X[4:0] sets the number of bits, minus 1. For example, a value of 7 will set the word size to 8 bits.
* Here is the internal state sequence:
  1. Wait for the A input to go high (idle state).
  2. Wait for the A input to go low (START bit edge).
  3. Delay for half a bit period.
  4. If the A input is no longer low, loop to (b).
  5. Delay for one bit period.
  6. Right-shift the A input into the shifter and delay for one bit period, repeat until all data bits are received.
  7. Capture the shifter into the Z register and raise IN.
  8. Loop to (a).
* RDPIN / RQPIN is used to read the received word. The word must be shifted right by 32 minus the word size. For example, to LSB-justify an 8-bit word received, you would do a 'SHR D,#32-8'.

# HOST COMMUNICATION

Normally the boot process loads a user's pre-written Propeller 2 Application and immediately executes it; however, the boot process also enables a host system to either download a new Propeller application or to begin an interactive session with the Propeller 2's built-in systems. Most boot patterns (dictated by pins P59-P61) feature a serial communication window. While configured with one of these boot patterns, a host computer's development software (such as Propeller Tool) can download new code, or instead, the user can initiate interactive mode.

## Download Propeller Application

During the boot process's serial communication window, the Propeller 2 can be loaded via asynchronous serial stream into P63, configured as 8 data bits, no parity, 1 stop bit, 9,600 to 2,000,000 baud. The signal should be inverted; start bit is low, stop bit is high, and data bits are high for 0 and low for 1.

The boot loader (or loader, for short) automatically adapts to the sender's baud rate from every greater-than ">" character ($3E) it receives. It is necessary to initially send "> " (greater-than, space; $3E, $20) before the first command, and then use ">" characters periodically throughout the data to keep the baud rate tightly calibrated to the internal RC oscillator that the loader uses during the boot process. Received ">" characters are not passed to the command parser, so they can be placed anywhere.

The loader's response messages are sent back serially over P62 at the same baud rate that the sender used. P62 is normally driven continuously during the serial protocol, but will go into open-drain mode when either the INA or INB mask of a command is non-0 (for [multiprogramming](#_kuk4h22mihvn)).

Unless forbidden by the [Boot Pattern](#_yruagv626tc7) or preempted by a program in an SPI memory chip, the serial loader becomes active within 15ms of reset being released.

Whitespace is required between command keywords and data. Each of the following characters, individually or in groups of any contiguous combination, constitute a single whitespace:

| Whitespace Characters in Serial Loading Protocol |
| --- |
| **TAB** ($09), **LF** ($0A), **CR** ($0D), **SPACE** ($20), **=** ($3D)1 |

1 The equal sign "=" may be present as padding in Base64 data

There are four commands which the sender can issue:

| Serial Loading Protocol Commands | |
| --- | --- |
| **Request Propeller Type** | |
|  | [**Prop\_Chk**](#_j9c32mb0pw0f)<INAmask> <INAdata> <INBmask> <INBdata> |
| **Change Clock Setting** | |
|  | [**Prop\_Clk**](#_optttfuwpnwi)<INAmask> <INAdata> <INBmask> <INBdata> <HUBSETclocksetting> |
| **Load and Execute Hex Data, With and Without Checksum Verification** | |
|  | [**Prop\_Hex**](#_dak6znq1xdte)<INAmask> <INAdata> <INBmask> <INBdata> <hexdatabytes> ? |
|  | [**Prop\_Hex**](#_dak6znq1xdte)<INAmask> <INAdata> <INBmask> <INBdata> <hexdatabytes> ~ |
| **Load and Execute Base64 Data, With and Without Checksum Verification** | |
|  | [**Prop\_Txt**](#_3s74wjh4k44b)<INAmask> <INAdata> <INBmask> <INBdata> <base64chrs> ? |
|  | [**Prop\_Txt**](#_3s74wjh4k44b)<INAmask> <INAdata> <INBmask> <INBdata> <base64chrs> ~ |

Each command keyword is followed by a Propeller 2 chip identifier in the form of four 32-bit hex values. The normal case is to use zeros for each of these fields to talk to any and all chips that are connected; often a single Propeller 2 chip.

### Multiprogramming

If multiple Propeller 2 chips are being loaded from the same serial line, it is often desirable to differentiate each Propeller's download by unique ID (their individual INA and INB states in the circuit). The Propeller 2's loader automatically ignores commands whose ID (INA / INB data and mask values) does not match its own INA and INB ports. All ID patterns are valid using pins P2 and beyond— the loader can not see the state of P0 and P1 since it configures them as smart pins monitoring their logical neighbor, P63, for automatic baud detection.

### Loader Parsing Notes

While waiting for a new command, Base64 data destined for another chip is safely ignored due to the fact that each command keyword contains an underscore "\_"; an illegal character in Base64.

If a character is received which does not fit expectations for that moment (ex: an "x" is received when hex digits are expected), the loader aborts the current command and waits for a new command.

### Prop\_Chk

Use the Prop\_Chk command to verify Propeller 2 connection and retrieve its version.

The response is in the form CR+LF+"Prop\_Ver"+SPACE+<version\_character>+CR+LF where <version\_character> is "A".."Z" and indicates the version of Propeller 2. The Rev B/C silicon responds with "G":

| Sender: "> Prop\_Chk 0 0 0 0"+CR  Loader: CR+LF+"Prop\_Ver G"+CR+LF |
| --- |

### Prop\_Clk

Use the Prop\_Clk command to update the P2's clock source. This is similar to executing the PASM instruction: HUBSET ##$0xxxxxxx.

The response to a valid Prop\_Clk command is a period "." character, then it performs the following steps:

1. Switches to the internal 20 MHz (fast) clock
2. Sets the desired configuration (except mode)
3. Waits ≈5 ms for the clock hardware to settle to the new configuration
4. Enables the desired clock mode

NOTE: After issuing the command, the sender should wait an additional 10ms, then send the "> " ($3E, $20) auto-baud sequence to adjust for the new clock configuration.

NOTE: If an image is loaded (see Prop\_Hex/Prop\_Txt) after switching to a PLL clock mode that is different than the mode used by that image, the uploaded image may need to issue a "HUBSET #$F0" before switching to the desired clock mode. See the warning in Configuring the Clock Generator for more details. An alternative approach is to use the same clock configuration as used by the image. This means that the image's call to HUBSET will effectively be a NOP, but always safe to perform.

**Example: Set PLL to 148.5 MHz**

To update the clock source as calculated in the PLL Example:

| Sender: "> Prop\_Clk 0 0 0 0 19D28F8"+CR  Loader: "."  Sender: (wait ≈10ms)  Sender: "> Prop\_Clk 0 0 0 0 19D28FB"+CR  Loader: "." |
| --- |

NOTE: An initial "Prop\_Clk 0 0 0 0 F0" is not required since the clock circuit starts up in this mode.

**Example: Reset to Boot Clock Configuration**

To return to the clock configuration on bootup:

| Sender: "> Prop\_Clk 0 0 0 0 F0"+CR  Loader: "." |
| --- |

### Prop\_Hex

Use the Prop\_Hex command to load code into Hub RAM, starting at $00000, and then execute it. The code must be sent as a stream of ASCII text representing bytes in hex format, separated by whitespace. Only the lower 8 bits of each value is used.

If the Prop\_Hex command is terminated with a "?" character, the loader will verify the checksum and respond before attempting to run the code. The loader responds with a period "." if the checksum was valid, or an exclamation point "!" if the checksum was invalid. When valid, the booter will perform a COGINIT #0,#0 instruction to relaunch cog 0 (currently running the loader program) with the new program starting at $00000. When invalid, the booter will wait for another command.

If the Prop\_Hex command is terminated with a "~" character, the loader will relaunch cog 0 to run the new program at $00000; skipping the validation and response steps noted above.

**Example: Loading a small program:**

Consider this small program:

DAT ORG

not dirb 'all outputs

.lp not outb 'toggle states (blinks leds on Prop123 & P2 Eval boards)

waitx ##20\_000\_000/4 'wait ¼ second

jmp #.lp 'loop

It assembles to:

00000- FB F7 23 F6 FD FB 23 F6 25 26 80 FF 1F 80 66 FD F0 FF 9F FD

and can be transmitted without a checksum like this:

| Sender: "> Prop\_Hex 0 0 0 0 FB F7 23 F6 FD FB 23 F6 25 26 80 FF 1F 80 66 FD F0 FF 9F FD ~" |
| --- |

In this example, the program image contains 5 longs (in little-endian order), the summation of which is $E6CE9A2C. To generate an embedded checksum long, you would compute:

$706F7250 ("Prop" read as a long in little-endian order)

- $E6CE9A2C (code summation)

─────────

= $89A0D824 (checksum complement)

Those resulting four bytes (the checksum complement) may be appended to the end of the data stream, transmitted as follows.

| Sender: "> Prop\_Hex 0 0 0 0 FB F7 23 F6 FD FB 23 F6 25 26 80 FF 1F 80 66 FD F0 FF 9F FD 24 D8 A0 89 ?"  Loader: "." |
| --- |

Note that for verification purposes it doesn't matter where the checksum complement long is placed– only that it be long-aligned within your data.

If transmitting multiple lines (blocks) of code image, it is recommended to start each Base64 data line with a greater-than ">" character to keep the baud rate tightly calibrated.

### Prop\_Txt

The Prop\_Txt command is like Prop\_Hex, but delivers Base64 data instead of hex bytes. Base64 data is a stream of text characters that convey six bits each, and is assembled into bytes as it is received. This format is 2.25x denser than hex; minimizing transmission size and time.

| Base64 Characters and Values | |
| --- | --- |
| **Characters** | **Index Values** |
| A–Z, a–z, 0–9, +, / | $00–$19, $1A–$33, $34–$3D, $3E, $3F |

Whitespaces are ignored among Base64 characters.

**Example: Loading a small program:**

The program from the Prop\_Hex example can be transmitted in Base64 form without a checksum like this:

| Sender: "> Prop\_Txt 0 0 0 0 +/cj9v37I/YlJoD/H4Bm/fD/n/0 ~" |
| --- |

With the checksum complement appended:

| Sender: "> Prop\_Txt 0 0 0 0 +/cj9v37I/YlJoD/H4Bm/fD/n/0k2KCJ ?"  Loader: "." |
| --- |

Note that the Base64 stream must be generated using the assembled data's given byte order ($FB, $F7, $23...), and from MSB to LSB within each byte (%111110, %111111, %011100, %100011...); however, the checksum is calculated on little-endian long values (as with the Prop\_Hex command) and must be appended/integrated into the assembled data before Base64 conversion.

| Assembled "Code" Data to Base64 Stream Conversion | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Code Hex** | FB | | F7 | | 23 | | ... |
| **Code (8-bit) Binary** | 11111011 | | 11110111 | | 00100011 | | ... |
| **Code (6-bit) Binary** | 111110 | 11 | 1111 | 0111 | 00 | 100011 | ... |
| **Code (6-bit) Hex** | 3E | 3F | | 1C | | 23 | ... |
| **Base64 Character** | + | / | | c | | j | ... |

To keep the baud rate tightly calibrated when transmitting multiple lines (blocks) of code image, start each Base64 data line with a greater-than ">" character.

## Interactive Mode

To enter interactive mode from a host computer:

* Run serial terminal software (like Parallax Serial Terminal, TeraTerm, or RealTerm)
* Disable character echo ("Echo On" in Parallax Serial Terminal)
* Set to any baud rate from 9600 baud to 2 Mbaud (recommended), 8 data bits, 1 stop bit, no parity
* Press and release the Propeller 2 development board's Reset button
* Type "> " (greater than followed by a space), then either Ctrl+D or the ESC key to enter P2 Monitor or TAQOZ mode, respectively

### P2 Monitor

The P2 Monitor is a built-in interactive system that allows for viewing and manipulating memory and running code. Use the P2 Monitor to explore and change current RAM contents or load and run code from microSD memory. After power-up or reset (and while preventing autorun of a flash/microSD-resident application), invoke the P2 Monitor from a terminal by typing: "> " (greater than followed by a space), then Ctrl+D.

Here is an example of listing the first 16 longs of Register RAM (in long format), by typing "000-010L":

\*000-010L

000: FF800800 FC0C003F F606C832 FCDC041F '.......?...2....'

004: FD747E40 F0A6CA01 F426CA1F FD62CA00 '.t~@.....&...b..'

008: FB6EC9FA FC0C003F FD64C428 FF0007E0 '.n.....?.d.(....'

00C: FB06012C FD655229 FF0007E1 FB0420B8 '...,.eR)...... .'

Here is a list of the first 16 bytes of Hub RAM (in long format), typing "0000-0010L":

\*0000-0010L

00000: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF '................'

For more information, see the P2 Monitor link on the Propeller 2 Documentation Page at [www.parallax.com/p2](http://www.parallax.com/p2).

To switch to TAQOZ while in P2 Monitor, type ESC followed by the Enter key.

### 

### TAQOZ

TAQOZ is a built-in interactive Forth language engine, based on Tachyon Forth. Use TAQOZ to explore "what ifs" and quickly exercise P2 hardware for testing or debugging. After power-up or reset (and while preventing autorun of a flash/microSD-resident application), invoke TAQOZ from a terminal by typing: "> " ESC (greater than followed by a space), then the Escape key.

Toggle pin 56 (ex: blink an LED on P56) by typing:

56 blink (type "56 mute" to stop toggling)

...or by typing:

begin 56 high 250 ms 56 low 250 ms key until (press any key to stop toggling)

For more information, see the TAQOZ links on the Propeller 2 Documentation Page at [www.parallax.com/p2](http://www.parallax.com/p2).

To switch to P2 Monitor while in TAQOZ, type Ctrl+D.

# 

# PROPELLER 2 RESERVED WORDS (SPIN2 + PASM2)

Predefined symbols recognized by the compiler to have special meaning.

## \_ (leading underscore)

| \_C  \_C\_AND\_NZ  \_C\_AND\_Z  \_C\_EQ\_Z | \_C\_NE\_Z  \_C\_OR\_NZ  \_C\_OR\_Z  \_CLR | \_E  \_GE  \_GT  \_LE | \_LT  \_NC  \_NC\_AND\_NZ  \_NC\_AND\_Z | \_NC\_OR\_NZ  \_NC\_OR\_Z  \_NE  \_NZ | \_NZ\_AND\_C  \_NZ\_AND\_NC  \_NZ\_OR\_C  \_NZ\_OR\_NC | \_RET\_  \_SET  \_Z  \_Z\_AND\_C | \_Z\_AND\_NC  \_Z\_EQ\_C  \_Z\_NE\_C  \_Z\_OR\_C | \_Z\_OR\_NC |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |

## A - B

| ABORT  ABS  ADD  ADDBITS  ADDCT1 | ADDCT2  ADDCT3  ADDPINS  ADDPIX  ADDS | ADDSX  ADDX  AKPIN  ALIGNL  ALIGNW | ALLOWI  ALT  ALTB  ALTD  ALTGB | ALTGN  ALTGW  ALTI  ALTR  ALTS | ALTSB  ALTSN  ALTSW  AND  ANDC | ANDN  ANDZ  ARCHIVE  ASMCLK  AUGD | AUGS  BACKCOLOR  BITC  BITH  BITL | BITMAP  BITNC  BITNOT  BITNZ  BITRND | BITZ  BLACK  BLNPIX  BLUE  BMASK | BOX  BRK  BYTE  BYTEFILL  BYTEMOVE | BYTES\_1BIT  BYTES\_2BIT  BYTES\_4BIT |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

## C - D

| CALL  CALLA  CALLB  CALLD  CALLPA  CALLPB  CARTESIAN  CASE  CASE\_FAST  CHANNEL | CIRCLE  CLEAR  CLKFREQ  CLKMODE  CLKSET  CLOSE  CMP  CMPM  CMPR  CMPS | CMPSUB  CMPSX  CMPX  COGATN  COGBRK  COGCHK  COGEXEC  COGEXEC\_NEW  COGEXEC\_NEW\_PAIR  COGID | COGINIT  COGSPIN  COGSTOP  COLOR  CON  CRCBIT  CRCNIB  CYAN  DAT  DEBUG | DEBUG\_BAUD  DEBUG\_COGS  DEBUG\_DELAY  DEBUG\_DISPLAY\_LEFT  DEBUG\_DISPLAY\_TOP  DEBUG\_HEIGHT  DEBUG\_LEFT  DEBUG\_LOG\_SIZE  DEBUG\_PIN  DEBUG\_TIMESTAMP | DEBUG\_TOP  DEBUG\_WIDTH  DEBUG\_WINDOWS\_OFF  DECMOD  DECOD  DEPTH  DEV  DIRA  DIRB  DIRC | DIRH  DIRL  DIRNC  DIRNOT  DIRNZ  DIRRND  DIRZ  DJF  DJNF  DJNZ | DJZ  DLY  DOT  DOTSIZE  DRVC  DRVH  DRVL  DRVNC  DRVNOT | DRVRND  DRVZ |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |

## E - F

| ELSE  ELSEIF  ELSEIFNOT  ENCOD  END | EVENT\_ATN  EVENT\_CT1  EVENT\_CT2  EVENT\_CT3  EVENT\_FBW | EVENT\_INT  EVENT\_PAT  EVENT\_QMT  EVENT\_SE1  EVENT\_SE2 | EVENT\_SE3  EVENT\_SE4  EVENT\_XFI  EVENT\_XMT  EVENT\_XRL | EVENT\_XRO  EXECF  FABS  FALSE  FBLOCK | FDEC  FDEC\_  FDEC\_ARRAY  FDEC\_ARRAY\_  FDEC\_REG\_ARRAY | FDEC\_REG\_ARRAY\_  FFT  FGE  FGES  FILE | FIT  FLE  FLES  FLOAT  FLTC | FLTH  FLTL  FLTNC  FLTNOT  FLTNZ | FLTRND  FLTZ  FRAC  FROM  FSQRT | FVAR  FVARS |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

## G - H

| GETBRK  GETBYTE  GETCT | GETMS  GETNIB  GETPTR | GETQX  GETQY  GETREGS | GETRND  GETSCP  GETSEC | GETWORD  GETXACC  GREEN | GREY  HIDEXY  HOLDOFF | HSV16  HSV16W  HSV16X | HSV8  HSV8W  HSV8X | HUBEXEC  HUBEXEC\_NEW  HUBEXEC\_NEW\_PAIR | HUBSET |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

## I - J

| IF  IF\_00  IF\_0000  IF\_0001  IF\_0010  IF\_0011  IF\_01  IF\_0100  IF\_0101  IF\_0110  IF\_0111  IF\_0X | IF\_10  IF\_1000  IF\_1001  IF\_1010  IF\_1011  IF\_11  IF\_1100  IF\_1101  IF\_1110  IF\_1111  IF\_1X  IF\_A | IF\_AE  IF\_ALWAYS  IF\_B  IF\_BE  IF\_C  IF\_C\_AND\_NZ  IF\_C\_AND\_Z  IF\_C\_EQ\_Z  IF\_C\_NE\_Z  IF\_C\_OR\_NZ  IF\_C\_OR\_Z  IF\_DIFF | IF\_E  IF\_GE  IF\_GT  IF\_LE  IF\_LT  IF\_NC  IF\_NC\_AND\_NZ  IF\_NC\_AND\_Z  IF\_NC\_OR\_NZ  IF\_NC\_OR\_Z  IF\_NE  IF\_NOT\_00 | IF\_NOT\_01  IF\_NOT\_10  IF\_NOT\_11  IF\_NZ  IF\_NZ\_AND\_C  IF\_NZ\_AND\_NC  IF\_NZ\_OR\_C  IF\_NZ\_OR\_NC  IF\_SAME  IF\_X0  IF\_X1  IF\_Z | IF\_Z\_AND\_C  IF\_Z\_AND\_NC  IF\_Z\_EQ\_C  IF\_Z\_NE\_C  IF\_Z\_OR\_C  IF\_Z\_OR\_NC  IFNOT  IJMP1  IJMP2  IJMP3  IJNZ  IJZ | INA  INB  INCMOD  INT\_OFF  IRET1  IRET2  IRET3  JATN  JCT1  JCT2  JCT3  JFBW | JINT  JMP  JMPREL  JNATN  JNCT1  JNCT2  JNCT3  JNFBW  JNINT  JNPAT  JNQMT  JNSE1 | JNSE2  JNSE3  JNSE4  JNXFI  JNXMT  JNXRL  JNXRO  JPAT  JQMT  JSE1  JSE2  JSE3 | JSE4  JXFI  JXMT  JXRL  JXRO |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

## L - M

| LINE  LINESIZE  LOC  LOCKCHK  LOCKNEW | LOCKREL  LOCKRET  LOCKTRY  LOGIC  LOGSCALE | LONG  LONGFILL  LONGMOVE  LONGS\_16BIT  LONGS\_1BIT | LONGS\_2BIT  LONGS\_4BIT  LONGS\_8BIT  LOOKDOWN  LOOKDOWNZ | LOOKUP  LOOKUPZ  LSTR  LSTR\_  LUMA8 | LUMA8W  LUMA8X  LUT1  LUT2  LUT4 | LUT8  LUTCOLORS  MAG  MAGENTA  MERGEB | MERGEW  MIDI  MIXPIX  MODC  MODCZ | MODZ  MOV  MOVBYTS  MUL  MULDIV64 | MULPIX  MULS  MUXC  MUXNC  MUXNIBS | MUXNITS  MUXNZ  MUXQ  MUXZ |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

## N - O

| NAN  NEG  NEGC  NEGNC | NEGNZ  NEGX  NEGZ  NEWCOG | NEXT  NIXINT1  NIXINT2  NIXINT3 | NOP  NOT  OBJ  OBOX | ONES  OPACITY  OR  ORANGE | ORC  ORG  ORGF  ORGH | ORIGIN  ORZ  OTHER  OUTA | OUTB  OUTC  OUTH  OUTL | OUTNC  OUTNOT  OUTNZ  OUTRND | OUTZ  OVAL |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

## 

## P

| P\_ADC  P\_ADC\_100X  P\_ADC\_10X  P\_ADC\_1X  P\_ADC\_30X  P\_ADC\_3X  P\_ADC\_EXT  P\_ADC\_FLOAT  P\_ADC\_GIO  P\_ADC\_SCOPE  P\_ADC\_VIO  P\_AND\_AB  P\_ASYNC\_IO  P\_ASYNC\_RX  P\_ASYNC\_TX  P\_BITDAC  P\_CHANNEL  P\_COMPARE\_AB  P\_COMPARE\_AB\_FB  P\_COUNT\_HIGHS | P\_COUNT\_RISES  P\_COUNTER\_HIGHS  P\_COUNTER\_PERIODS  P\_COUNTER\_TICKS  P\_DAC\_124R\_3V  P\_DAC\_600R\_2V  P\_DAC\_75R\_2V  P\_DAC\_990R\_3V  P\_DAC\_DITHER\_PWM  P\_DAC\_DITHER\_RND  P\_DAC\_NOISE  P\_EVENTS\_TICKS  P\_FILT0\_AB  P\_FILT1\_AB  P\_FILT2\_AB  P\_FILT3\_AB  P\_HIGH\_100UA  P\_HIGH\_10UA  P\_HIGH\_150K  P\_HIGH\_15K | P\_HIGH\_1K5  P\_HIGH\_1MA  P\_HIGH\_FAST  P\_HIGH\_FLOAT  P\_HIGH\_TICKS  P\_INVERT\_A  P\_INVERT\_B  P\_INVERT\_IN  P\_INVERT\_OUT  P\_INVERT\_OUTPUT  P\_LEVEL\_A  P\_LEVEL\_A\_FBN  P\_LEVEL\_B\_FBN  P\_LEVEL\_B\_FBP  P\_LOCAL\_A  P\_LOCAL\_B  P\_LOGIC\_A  P\_LOGIC\_A\_FB  P\_LOGIC\_B\_FB  P\_LOW\_100UA | P\_LOW\_10UA  P\_LOW\_150K  P\_LOW\_15K  P\_LOW\_1K5  P\_LOW\_1MA  P\_LOW\_FAST  P\_LOW\_FLOAT  P\_MINUS1\_A  P\_MINUS1\_B  P\_MINUS2\_A  P\_MINUS2\_B  P\_MINUS3\_A  P\_MINUS3\_B  P\_NCO\_DUTY  P\_NCO\_FREQ  P\_NORMAL  P\_OE  P\_OR\_AB  P\_OUTBIT\_A  P\_OUTBIT\_B | P\_PASS\_AB  P\_PERIODS\_HIGHS  P\_PERIODS\_TICKS  P\_PLUS1\_A  P\_PLUS1\_B  P\_PLUS2\_A  P\_PLUS2\_B  P\_PLUS3\_A  P\_PLUS3\_B  P\_PULSE  P\_PWM\_SAWTOOTH  P\_PWM\_SMPS  P\_PWM\_TRIANGLE  P\_QUADRATURE  P\_REG\_UP  P\_REG\_UP\_DOWN  P\_REPOSITORY  P\_SCHMITT\_A  P\_SCHMITT\_A\_FB  P\_SCHMITT\_B\_FB | P\_STATE\_TICKS  P\_SYNC\_IO  P\_SYNC\_RX  P\_SYNC\_TX  P\_TRANSITION  P\_TRUE\_A  P\_TRUE\_B  P\_TRUE\_IN  P\_TRUE\_OUT  P\_TRUE\_OUTPUT  P\_TT\_00  P\_TT\_01  P\_TT\_10  P\_TT\_11  P\_USB\_PAIR  P\_XOR\_AB  PA  PB  PC\_KEY  PC\_MOUSE | PI  PINCLEAR  PINF  PINFLOAT  PINH  PINHIGH  PINL  PINLOW  PINR  PINREAD  PINSTART  PINT  PINTOGGLE  PINW  PINWRITE  PLOT  POLAR  POLLATN  POLLCT  POLLCT1 | POLLCT2  POLLCT3  POLLFBW  POLLINT  POLLPAT  POLLQMT  POLLSE1  POLLSE2  POLLSE3  POLLSE4  POLLXFI  POLLXMT  POLLXRL  POLLXRO  POLXY  POP  POPA  POPB  POS  POSX | PR0  PR1  PR2  PR3  PR4  PR5  PR6  PR7  PRECISE  PRECOMPILE  PRI  PTRA  PTRB  PUB  PUSH  PUSHA  PUSHB |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |

## Q - R

| QCOS  QDIV  QEXP  QFRAC  QLOG  QMUL | QROTATE  QSIN  QSQRT  QUIT  QVECTOR  RANGE | RATE  RCL  RCR  RCZL  RCZR  RDBYTE | RDFAST  RDLONG  RDLUT  RDPIN  RDWORD  RECV | RED  REG  REGEXEC  REGLOAD  REP  REPEAT | RES  RESI0  RESI1  RESI2  RESI3  RET | RETA  RETB  RETI0  RETI1  RETI2  RETI3 | RETURN  REV  RFBYTE  RFLONG  RFVAR  RFVARS | RFWORD  RGB16  RGB24  RGB8  RGBEXP  RGBI8 | RGBI8W  RGBI8X  RGBSQZ  ROL  ROLBYTE  ROLNIB | ROLWORD  ROR  ROTXY  ROUND  RQPIN |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

## S - T

| SAL  SAMPLES  SAR  SAVE  SBIN  SBIN\_  SBIN\_BYTE\_  SBIN\_BYTE\_ARRAY  SBIN\_BYTE\_ARRAY\_  SBIN\_LONG  SBIN\_LONG\_  SBIN\_LONG\_ARRAY  SBIN\_LONG\_ARRAY\_  SBIN\_REG\_ARRAY  SBIN\_REG\_ARRAY\_  SBIN\_WORD  SBIN\_WORD\_  SBIN\_WORD\_ARRAY | SBIN\_WORD\_ARRAY\_  SCA  SCAS  SCOPE  SCOPE\_XY  SCROLL SDEC  SDEC\_  SDEC\_BYTE  SDEC\_BYTE\_  SDEC\_BYTE\_ARRAY  SDEC\_BYTE\_ARRAY\_  SDEC\_LONG  SDEC\_LONG\_  SDEC\_LONG\_ARRAY  SDEC\_LONG\_ARRAY\_  SDEC\_REG\_ARRAY  SDEC\_REG\_ARRAY\_ | SDEC\_WORD  SDEC\_WORD\_  SDEC\_WORD\_ARRAY  SDEC\_WORD\_ARRAY\_  SEND  SET  SETBYTE  SETCFRQ  SETCI  SETCMOD  SETCQ  SETCY  SETD  SETDACS  SETINT1  SETINT2  SETINT3  SETLUTS | SETNIB  SETPAT  SETPIV  SETPIX  SETQ  SETQ2  SETR  SETREGS  SETS  SETSCP  SETSE1  SETSE2  SETSE3  SETSE4  SETWORD  SETXFRQ  SEUSSF  SEUSSR | SHEX  SHEX\_  SHEX\_BYTE  SHEX\_BYTE\_  SHEX\_BYTE\_ARRAY  SHEX\_BYTE\_ARRAY\_  SHEX\_LONG  SHEX\_LONG\_  SHEX\_LONG\_ARRAY  SHEX\_LONG\_ARRAY\_  SHEX\_REG\_ARRAY  SHEX\_REG\_ARRAY\_  SHEX\_WORD  SHEX\_WORD\_  SHEX\_WORD\_ARRAY  SHEX\_WORD\_ARRAY\_  SHL  SHR | SIGNED  SIGNX  SIZE  SKIP  SKIPF  SPACING  SPECTRO  SPLITB  SPLITW  SPRITE  SPRITEDEF  SQRT  STALLI  STEP  STRCOMP  STRING  STRSIZE  SUB | SUBR  SUBS  SUBSX  SUBX  SUMC  SUMNC  SUMNZ  SUMZ  TERM  TEST  TESTB  TESTBN  TESTN  TESTP  TESTPN  TEXT  TEXTANGLE  TEXTSIZE | TEXTSTYLE  TITLE  TJF  TJNF  TJNS  TJNZ  TJS  TJV  TJZ  TO  TRACE  TRGINT1  TRGINT2  TRGINT3  TRIGGER  TRUE  TRUNC |
| --- | --- | --- | --- | --- | --- | --- | --- |

## U, V, W

| UBIN  UBIN\_  UBIN\_BYTE  UBIN\_BYTE\_  UBIN\_BYTE\_ARRAY  UBIN\_BYTE\_ARRAY\_  UBIN\_LONG  UBIN\_LONG\_  UBIN\_LONG\_ARRAY  UBIN\_LONG\_ARRAY\_  UBIN\_REG\_ARRAY  UBIN\_REG\_ARRAY\_  UBIN\_WORD  UBIN\_WORD\_  UBIN\_WORD\_ARRAY | UBIN\_WORD\_ARRAY\_  UDEC  UDEC\_  UDEC\_BYTE  UDEC\_BYTE\_  UDEC\_BYTE\_ARRAY  UDEC\_BYTE\_ARRAY\_  UDEC\_LONG  UDEC\_LONG\_  UDEC\_LONG\_ARRAY  UDEC\_LONG\_ARRAY\_  UDEC\_REG\_ARRAY  UDEC\_REG\_ARRAY\_  UDEC\_WORD  UDEC\_WORD\_ | UDEC\_WORD\_ARRAY  UDEC\_WORD\_ARRAY\_  UHEX  UHEX\_  UHEX\_BYTE  UHEX\_BYTE\_  UHEX\_BYTE\_ARRAY  UHEX\_BYTE\_ARRAY\_  UHEX\_LONG  UHEX\_LONG\_  UHEX\_LONG\_ARRAY  UHEX\_LONG\_ARRAY\_  UHEX\_REG\_ARRAY  UHEX\_REG\_ARRAY\_  UHEX\_WORD | UHEX\_WORD\_  UHEX\_WORD\_ARRAY  UHEX\_WORD\_ARRAY\_  UNTIL  UPDATE  VAR  VARBASE  WAITATN  WAITCT  WAITCT1  WAITCT2  WAITCT3  WAITFBW  WAITINT  WAITMS | WAITPAT  WAITSE1  WAITSE2  WAITSE3  WAITSE4  WAITUS  WAITX  WAITXFI  WAITXMT  WAITXRL  WAITXRO  WC  WCZ  WFBYTE  WFLONG | WFWORD  WHILE  WHITE  WINDOW  WMLONG  WORD  WORDFILL  WORDMOVE  WORDS\_1BIT  WORDS\_2BIT  WORDS\_4BIT  WORDS\_8BIT  WRBYTE  WRC  WRFAST | WRLONG  WRLUT  WRNC  WRNZ  WRPIN  WRWORD  WRZ  WXPIN  WYPIN  WZ |
| --- | --- | --- | --- | --- | --- | --- |

## X, Y, Z

| X\_16P\_2DAC8\_WFWORD  X\_16P\_4DAC4\_WFWORD  X\_1ADC8\_0P\_1DAC8\_WFBYTE  X\_1ADC8\_8P\_2DAC8\_WFWORD  X\_1P\_1DAC1\_WFBYTE  X\_2ADC8\_0P\_2DAC8\_WFWORD  X\_2ADC8\_16P\_4DAC8\_WFLONG  X\_2P\_1DAC2\_WFBYTE  X\_2P\_2DAC1\_WFBYTE  X\_32P\_4DAC8\_WFLONG  X\_4ADC8\_0P\_4DAC8\_WFLONG  X\_4P\_1DAC4\_WFBYTE  X\_4P\_2DAC2\_WFBYTE  X\_4P\_4DAC1\_WFBYTE  X\_8P\_1DAC8\_WFBYTE  X\_8P\_2DAC4\_WFBYTE | X\_8P\_4DAC2\_WFBYTE  X\_ALT\_OFF  X\_ALT\_ON  X\_DACS\_0\_0\_0\_0  X\_DACS\_0\_0\_X\_X  X\_DACS\_0\_X\_X\_X  X\_DACS\_0N0\_0N0  X\_DACS\_0N0\_X\_X  X\_DACS\_1\_0\_1\_0  X\_DACS\_1\_0\_X\_X  X\_DACS\_1N1\_0N0  X\_DACS\_3\_2\_1\_0  X\_DACS\_OFF  X\_DACS\_X\_0\_X\_X  X\_DACS\_X\_X\_0\_0  X\_DACS\_X\_X\_0\_X | X\_DACS\_X\_X\_0N0  X\_DACS\_X\_X\_1\_0  X\_DACS\_X\_X\_X\_0  X\_DDS\_GOERTZEL\_SINC1  X\_DDS\_GOERTZEL\_SINC2  X\_IMM\_16X2\_1DAC2  X\_IMM\_16X2\_2DAC1  X\_IMM\_16X2\_LUT  X\_IMM\_1X32\_4DAC8  X\_IMM\_2X16\_2DAC8  X\_IMM\_2X16\_4DAC4  X\_IMM\_32X1\_1DAC1  X\_IMM\_32X1\_LUT  X\_IMM\_4X8\_1DAC8  X\_IMM\_4X8\_2DAC4  X\_IMM\_4X8\_4DAC2 | X\_IMM\_4X8\_LUT  X\_IMM\_8X4\_1DAC4  X\_IMM\_8X4\_2DAC2  X\_IMM\_8X4\_4DAC1  X\_IMM\_8X4\_LUT  X\_PINS\_OFF  X\_PINS\_ON  X\_RFBYTE\_1P\_1DAC1  X\_RFBYTE\_2P\_1DAC2  X\_RFBYTE\_2P\_2DAC1  X\_RFBYTE\_4P\_1DAC4  X\_RFBYTE\_4P\_2DAC2  X\_RFBYTE\_4P\_4DAC1  X\_RFBYTE\_8P\_1DAC8  X\_RFBYTE\_8P\_2DAC4  X\_RFBYTE\_8P\_4DAC2 | X\_RFBYTE\_LUMA8  X\_RFBYTE\_RGB8  X\_RFBYTE\_RGBI8  X\_RFLONG\_16X2\_LUT  X\_RFLONG\_32P\_4DAC8  X\_RFLONG\_32X1\_LUT  X\_RFLONG\_4X8\_LUT  X\_RFLONG\_8X4\_LUT  X\_RFLONG\_RGB24  X\_RFWORD\_16P\_2DAC8  X\_RFWORD\_16P\_4DAC4  X\_RFWORD\_RGB16  X\_WRITE\_OFF  X\_WRITE\_ON  XCONT  XINIT | XOR  XORC  XORO32  XORZ  XSTOP  XYPOL  XZERO  YELLOW  ZEROX  ZSTR  ZSTR\_ |
| --- | --- | --- | --- | --- | --- |

# GENERAL PURPOSE I/O PIN EXCEPTIONS

All Propeller 2 I/O pins (P0–P63 on the P2X8C4M64P) share the same capabilities; however, certain applications may be sensitive to edge cases noted here. Moving certain functions to other I/O pins in sensitive hardware designs will resolve the issue.

P58–P63 : upon power-up/reset the pins have the special purpose of detecting the boot up configuration and communicating with an external flash or SD memory, or with a host system for programming. After boot up, they become general purpose for the user application (aside from that imposed by any hardware attached to them). See [Boot Up procedure](#_k13wh1l3u0i1) for more information.

P28–P31 use the same internal power rails as the XI/XO pins' clock oscillator circuitry. If P28–P31 transition simultaneously in fast digital mode, it may cause spikes on the internal power rails which can cause a slow external crystal edge (on XI) to be registered as multiple edges. To avoid this, either move such functions to other I/O pins or drive a crisp clock signal into XI (with an external clock oscillator). The P2 Edge Module Rev C employs the latter solution.

# CHANGE LOG

| **Date** | **Notes** |
| --- | --- |
| 09/09/2021 | First public draft release. |
| 10/15/2021 | Enhanced [Instruction Pipeline](#_d6a33r7wy3h8) diagrams and explanations, and added Wait and Branch examples. |
| 11/01/2022 | Added underscore, ALL CAPS and <all\_lowercase> to [Conventions](#_12fh0v8mc8fz). Clarified [CORDIC Solver](#_ihh0p7cihx2l) result availability. Clarified [System Counter](#_p9jolctm7tnz) upper and lower usage. Added floating point, sprite, and debug keyboard and mouse symbols to [Propeller 2 Reserved Words](#_hg33jkjv6j2l). |

# PARALLAX INCORPORATED

| **Parallax Inc.**  **599 Menlo Drive, Suite 100**  **Rocklin, CA 95765**  **USA** | **Office: +1 916-624-8333**  **Toll Free US: 888-512-1024** | [**sales@parallax.com**](mailto:sales@parallax.com)  [**support@parallax.com**](mailto:support@parallax.com) | [**www.parallax.com/p2**](http://www.parallax.com/p2)  [**forums.parallax.com**](https://forums.parallax.com) |
| --- | --- | --- | --- |
| **Purchase of the P2X8C4M64P does not include any license to emulate any other device nor to communicate via any specific proprietary protocol; P2X8C4M64P connectivity objects and code examples provided or referenced by Parallax, Inc. are NOT licensed and are provided for research and development purposes only; end users must seek permission to use licensed protocols for their applications and products from the protocol license holders.**  **Parallax, Inc. makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Parallax, Inc. assume any liability arising out of the application or use of any product, and specifically disclaims any and all liability, including without limitation consequential or incidental damages even if Parallax, Inc. has been advised of the possibility of such damages.**  **Copyright © 2022 Parallax, Inc. All rights are reserved. Parallax, the Parallax logo, the P2 logo, and Propeller are trademarks of Parallax, Inc.** | | | |

# 