Propeller 2 Assembly Programming

A beginner's guide

Written by K. L. Wanamaker

Copyright © 2026 K. L. Wanamaker
All rights reserved.

Table of Contents

Propeller 2 Assembly Programming....1
Chapter 0: Getting Started with the Propeller 2....5
What is P2ASM?....6
Chapter 1: Getting started....9
Comments....9
Tips for Writing Good Comments....9
The CON Section....10
Clock Frequency....10
Baud Rate....11
Baud Mode....11
Transmission Pin....13
The DAT Section....13
Origin....14
Real vs. Simulated Clock....14
Configuring a SmartPin....15
SmartPin speed....16
Setting the output....16
Sending Data....17
Program: tx.spin2....18
Exercise....19
Chapter 2: Data Movement....20
What is a Variable....20
Declaring a Variable....20
Storing a Value in a Variable....20
Using the Variable....21
The MOV Instruction....21
Exercise....23
Chapter 3: Adding, Subtracting and Multiplying....24
Why Math Matters....24
The ADD instruction....24
The SUB Instruction....26
The MUL Instruction....27
Exercise....29
Chapter 4: Bitwise Operations....30
The AND Instruction....30
The OR Instruction....31
The XOR Instruction....33
The NOT Instruction....34
Chapter 5: Shifting....37
SHL — Shift Left....37
SHR — Shift Right....38
SAL — Shift Arithmetic Left....39
SAR — Shift Arithmetic Right....41
Exercise....42

Chapter 6: Loops....43
JMP — Jump (Unconditional)....43
DJNZ — Decrement and Jump if Not Zero....45
TJNZ — Test and Jump if Not Zero....46
TJZ — Test and Jump if Zero....48
Exercise....49
Chapter 7: Decision Making (Conditional Execution)....51
CMP — Compare....52
IF_Z — Execute If Zero....53
IF_NZ — Execute If Not Zero....54
CMP + IF_Z / IF_NZ — Conditional Execution Based on Comparison....55
CMP + IF_C — Conditional Execution Based on Carry (Unsigned Comparison)....57
CMP + IF_NC — Conditional Execution Based on Not Carry (Unsigned Greater or Equal)....58
CMP + IF_C / IF_NC — Unsigned Conditional Execution....60
CMP + IF_BE — Unsigned Below or Equal Conditional Execution....61
CMP + IF_NC / IF_NZ — Unsigned Greater than Conditional Execution....62
Chapter 8: Subroutines....64
JMP revisited Instruction....64
CALL Instruction....66
Global labels and Local labels....67
Exercise....70
Chapter 9: Timing & Waits....71
WAITX instruction....71
Instruction: NOP (No Operation)....73
GETCT/WAITCT1/ADDCT1 instruction....74
POLLCT1 instruction (Non-blocking Timer Poll)....77
WAITX vs NOP....78
Exercise....80
Chapter 10: Hardware Functions....81
QMUL — Unsigned Multiplication Using CORDIC....81
QDIV — Unsigned Division Using CORDIC....83
Exercise....85
Chapter 11: Hub Memory Data Access....86
Variables in P2ASM....86
RDBYTE instruction....88
WRBYTE instruction....89
RDBYTE looping....91
RDBYTE for Strings....92
RDWORD instruction....94
WRWORD instruction....95
RDLONG instruction....97
WRLONG instruction....98
Exercise....100
Chapter 12 — How Numbers Work in P2 Assembly....101
Printing Numbers Using SUB....102
Printing Numbers Using QDIV....104
Chapter 13 — Cog Control....107

Chapter 14: String Manipulation....114
Print a String....114
String Length....114
String Number Length....116
String compare....117
String Number Compare....119
String copy....121
String number copy....123
String concatenate....125
String number concatenate....127
String to decimal....130
Decimal to character....131
Decimal to string....133
Decimal to string revisited....135
Chapter 15: SmartPin Setup Instructions (for UART)....139
Basic UART Echo with SmartPins....141
Program: echo.spin2....141
Single Digit Input and Check (0–9)....142
Multi-character input....144
Line Editing with CRLF....146
Read Line....148
Send string....151
Tokenized Echo Command....153
Chapter 16: Introduction to SPI (Serial Peripheral Interface)....158
Reading the JEDEC ID from SPI Flash....159
Reading from SPI Flash....163
Writing to SPI Flash....166
Chapter 17 – Practice Programs....172
Looking Ahead: From Learning Examples to Production-Ready Code....172

Chapter 0: Getting Started with the Propeller 2

We are beginning our journey with the **Propeller P2** microcontroller. Unlike many microcontrollers, which usually have a single processor, the P2 is a **multicore** microcontroller. This means it contains **8 identical 32-bit processors**, called "**cogs**," that can all run code at the same time.

Having multiple processors makes the P2 very flexible and powerful because each processor can handle a different task independently. For example, one processor could control an LED display while another reads input from a sensor, all without slowing each other down.

All 8 processors are connected through a **central hub**, which acts as the brain that coordinates access to shared resources. The hub contains shared RAM that all processors can read from and write to, enabling fast data exchange between cogs. There is a built-in hardware math engine that accelerates complex operations such as trigonometry, vector math, and graphics calculations. The hub also contains housekeeping hardware that manages system timing, synchronization, and inter-processor communication to keep all cogs operating smoothly.

One of the most unique features of the P2 is its **64 smart Input/Output(I/O) pins**. These pins are much more than simple digital inputs or outputs. Each pin can be programmed to perform complex analog or digital functions independently. The pins can generate waveforms, count pulses, communicating with devices, or even acting as a small ADC (analog-to-digital converter).

The P2 is designed so every processor can access every I/O pin, giving us maximum flexibility. The pins are powered at 3.3 volts in groups of four, which improves their analog performance.

Overall, the Propeller P2 is a versatile microcontroller capable of handling multiple tasks simultaneously, performing complex calculations efficiently, and controlling a wide range of input/output devices. For beginners, it's exciting because it opens the door to experimenting with advanced electronics while still being approachable.

The Propeller 2 was developed by **Parallax, Inc.**, a company known for creating innovative educational and hobbyist microcontrollers. Its design builds upon the success of the original Propeller chip, offering more memory, faster processors, and more flexible I/O capabilities. The P2 was first publicly announced around **2020**, with development starting several years earlier to address the needs of modern embedded electronics.

What is P2ASM?

As we continue our journey, we come across the question: **how do we tell the P2 what to do?** This is where **P2ASM** comes in.

P2ASM stands for **Propeller 2 Assembler**, and it is the programming language that speaks directly to the P2's processors, or "cogs." Unlike high-level languages like C, Basic or Python, which need to be translated into machine instructions, P2ASM lets us write instructions that are very close to the hardware. Think of it like giving the P2 **step-by-step directions in a language it naturally understands**.

P2ASM allows us to control each processor individually, directly manage the smart I/O pins, and write highly efficient code with minimal overhead. This makes P2ASM ideal for tasks that require precise timing, low latency, and maximum performance on the Propeller 2.

P2ASM gives full control of the Propeller 2 hardware, letting programs run much faster than high-level languages and typically occupy less memory. Its precise timing makes it ideal for time-critical tasks, advanced I/O, and projects involving multiple processors. With its clear and structured commands, beginners can start with simple tasks and quickly progress to more complex projects.

This text is a practical guide to programming the Propeller 2 using P2ASM. It focuses on teaching full control of the hardware, writing efficient code, and achieving precise timing. It is designed to help you start with simple tasks and gradually progress to more complex projects, including multiple processors, smart I/O, and graphics.

This book is not a complete reference for every Propeller 2 instruction or hardware feature. It does not focus on high-level languages, but on low-level

programming with P2ASM. It is also not intended to overwhelm beginners; concepts are introduced step by step with clear explanations and practical examples.

P2ASM (Propeller 2 Assembly Language) is included as part of the official **Parallax Propeller 2 software tools**. It is not a separate download.

To get P2ASM software tools:

- Visit the **Parallax Inc. website**
- Navigate to the **Propeller 2** section
- Download the current **official development tools** provided by Parallax (these include the assembler, compiler, and supporting utilities)
- You can also download third-party development tools.

Tool setup steps and filenames may change over time. Always follow the instructions provided by Parallax or third-parties for installation and updates.

Chapter 1: Getting started

Comments

Comments are notes written in your code that the assembler ignores when generating machine code. They are meant for humans, helping explain what the code does, why certain instructions are used, and how different sections work. Using comments effectively makes your code easier to understand, maintain, and debug.

In P2ASM, comments begin with a single quote (`'`), and everything after it on that line is ignored by the assembler. This lets you add explanations, reminders, and documentation—such as your logic or hardware connections—without affecting how the program runs, turning your code into a clear, readable guide instead of just a list of instructions.

Tips for Writing Good Comments

Keep comments short and clear, focusing on **why** the code does something rather than just what it does. Always update comments if the code changes, since outdated comments can be misleading. For complex calculations or bitwise operations, step-by-step explanations in comments are especially helpful, making it easier to understand and maintain your code over time.

Example

```
mov A, #5           ' Put the number 5 into register A
add B, A            ' Add the value in A to register B
wypin A, #62        ' Send the value in A to I/O pin 62

' This section initializes the TX pin for serial output
wrpin ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
mov number1, #2     ' Assign 2 to number1
```

The CON Section

The CON section in P2ASM is used to define constants — values that do not change while your program is running. Constants make your code easier to read and maintain, allowing you to give meaningful names to important values such as the system clock frequency (`_CLKFREQ`), the baud rate (`BAUD_RATE`), or pin numbers like `TRANSMISSION_PIN`. Unlike variables, constants are fixed and defined before the program starts running. Using descriptive names instead of raw numbers throughout your code improves readability and makes it easier to update values in one place without changing multiple lines of code.

Example

```
con
_CLKFREQ = 180_000_000   ‘System clock in Hz
BAUD_RATE = 2_000_000    ‘Serial speed in bits per second
TRANSMISSION_PIN = 62    ‘Pin used for transmitting serial data
```

Clock Frequency

Before communicating with other devices we need a reliable sense of time. The system clock, `_CLKFREQ`, acts like the heartbeat of the Propeller 2 chip, telling the chip when to execute instructions, toggle pins, or measure intervals. This main internal clock drives instruction execution, SmartPin timing, and all time-sensitive hardware, ensuring that every operation occurs with precise and predictable timing.

Example

```
_CLKFREQ = 180_000_000
```

At a system clock frequency of 180 MHz, the Propeller 2 executes one instruction every 5.56 nanoseconds (`1 ÷ 180,000,000 s`). This incredibly fast timing allows precise control over hardware, ensuring that the chip can handle high-speed tasks accurately.

Baud Rate

The baud rate determines how fast data is sent over a serial connection and is measured in bits per second (bps). Choosing the correct baud rate is essential to ensure that both the sender and receiver communicate reliably. If the baud rates don't match, data can be lost or misinterpreted, leading to communication errors.

Example

```
BAUD_RATE = 2_000_000    ' Set the serial communication speed to
                         ' 2,000,000 bits per second
```

Baud Mode

When sending serial data with a SmartPin, the Propeller 2 needs to know how long to hold each bit and how many bits are in a frame. This information is defined using `BAUD_MODE`, which calculates the timing for each bit and ensures that data is transmitted accurately at the desired baud rate.

The first step in setting up serial communication is calculating how long each bit should last. This is done by dividing the system clock frequency (`_CLKFREQ`) by the desired baud rate (`BAUD_RATE`). For example, if `_CLKFREQ` is 180,000,000 Hz and you want to send data at 2,000,000 bits per second, this calculation determines the exact duration of each bit, ensuring that the Propeller 2 transmits data at the correct speed.

Example

```
BIT_PERIOD = (_CLKFREQ / BAUD_RATE)
```

This calculation means that each bit will last 90 clock cycles (`180,000,000 ÷ 2,000,000 = 90`). You can think of it like setting a timer: every 90 “ticks” of the Propeller 2 system clock, the SmartPin moves on to the next bit. This ensures that each bit is held for the correct duration, allowing the receiver to read the data accurately.

Once you’ve calculated the bit duration, it needs to be loaded into the upper 16 bits of the `BAUD_MODE` value. This tells the SmartPin exactly how long to hold each bit during transmission. By placing the bit duration in the upper half, the Propeller 2 can use it to control timing precisely while the lower 16 bits of `BAUD_MODE` are used for other settings, such as the frame size or number of bits per transmission.

Example

```
<< 16
```

The SmartPin uses a 32-bit configuration value to control serial communication. The upper 16 bits store the bit duration, or how long each bit lasts. By shifting our calculated value (`90 << 16`), we move the bit duration into the correct position, ensuring the SmartPin holds each bit for exactly the right number of clock cycles.

A serial frame usually contains 8 data bits. When configuring a SmartPin, the lower 16 bits of `BAUD_MODE` store the frame size **minus one**, so the SmartPin knows how many bits to send per frame. For an 8-bit frame, this means `8 - 1 = 7`, so we place the value 7 in the lower 16 bits. Combined with the upper 16 bits that hold the bit duration, this fully defines the timing and length of each serial frame for accurate transmission.

Example

```
| (8 – 1)
```

Think of `BAUD_MODE` like labeling a box: the upper 16 bits indicate how long to hold each bit, while the lower 16 bits specify how many bits to send in a frame. This clear "labeling" tells the SmartPin both the timing and size of each serial frame, ensuring accurate data transmission.

The upper 16 bits hold the bit duration and the lower 16 bits hold the frame size. Using the bitwise OR operator (`|`), we merge these two values into a single 32-bit `BAUD_MODE` setting for the SmartPin. This combined value tells the SmartPin both how long to hold each bit and how many bits to send per frame, completing the configuration needed for accurate serial transmission.

Example

```
BAUD_MODE = (BIT_PERIOD << 16) | (8 - 1)
```

Now the SmartPin knows exactly how long to hold each bit and how many bits to send per frame. With the 32-bit `BAUD_MODE` value properly configured, it can transmit serial data accurately, ensuring that each frame is timed correctly and received reliably by other devices.

In summary the upper 16 bits of `BAUD_MODE` store the bit duration, specifying how many clock cycles each bit lasts. The lower 16 bits store the frame size minus one, indicating how many bits the SmartPin should send per frame. Combined into a single 32-bit value, `BAUD_MODE` allows the SmartPin to transmit serial data at the correct speed reliably, ensuring accurate timing and proper communication with other devices.

Transmission Pin

The transmission pin is the Propeller 2 I/O pin that actually sends the data. For this example, we'll use pin 62. This pin must be properly configured with `wrpin`, `wxpin`, and `dirh` so it knows its role, timing and direction. This ensures that data is transmitted correctly.

Example

```
TRANSMISSION_PIN = 62    'Use pin 62 to send serial data
```

The DAT Section

The DAT section is where the actual program instructions live, containing the code that the Propeller 2 executes. Anything that will run—whether setting up pins, sending data, or blinking an LED—goes here. You can think of DAT as the engine of your program, providing the step-by-step instructions the Propeller 2 follows to perform tasks. Constants defined in the CON section are often used in DAT to make the code cleaner and easier to read. Without the DAT section, the Propeller 2 has no instructions to execute, so your program wouldn't do anything.

Example

```
dat
org 0
asmclk
wrpin    ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
wxpin    ##BAUD_MODE, #TRANSMISSION_PIN
wypin    #1, #TRANSMISSION_PIN
dirh     #TRANSMISSION_PIN
wypin   #"0", #TRANSMISSION_PIN
```

Origin

In Propeller 2 assembly (P2ASM), the origin (`ORG 0`) is used to tell the assembler that the program should begin at address 0 within a cog's local memory. Each cog in the Parallax Propeller 2 has its own 512-long (2 KB) memory space, and when a cog starts executing, it begins at location 0 by default. The `ORG` directive sets the assembly origin — meaning it defines the starting address where instructions will be placed in cog RAM. By writing `ORG 0`, you ensure your first instruction is assembled at the exact location where the

cog expects to begin execution. Without it, code might assemble at a different address, which could cause incorrect behavior if execution doesn't line up with where the instructions were placed. In short, `ORG 0` aligns your program's starting point with the cog's execution entry point.

Real vs. Simulated Clock

The Propeller 2 offers two ways to handle "time" in your programs. The `asmclk` instruction uses the actual system clock frequency (`_CLKFREQ`) for all timing calculations. For example, at 180 MHz, instructions execute every 5.56 ns, which is essential for timing-critical tasks like serial communication, PWM, video output, or SmartPin bit timing. Using `asmclk` ensures delays, pulse widths, and signal periods match the real hardware exactly.

The `simclk` instruction switches to a slower, simulated clock, allowing you to step through code for learning, testing, or debugging without running at full speed. Timing in this mode is not accurate, so delays and bit periods won't reflect real hardware behavior. Because features like SmartPins and serial communication rely on precise cycle timing, `simclk` can cause incorrect baud rates or missed bits. For any program intended to run on actual hardware, `asmclk` should be used to guarantee correct timing.

Configuring a SmartPin

The `wrpin` instruction sets the role of a Propeller `2` pin, telling it whether to send signals, receive signals, or remain idle. By defining the pin's behavior, `wrpin` prepares it for further configuration, such as timing or direction. Without `wrpin`, the pin has no defined function and cannot operate correctly in your program.

It can also configure special behaviors, such as transmitting serial data (TX), receiving serial data (RX), or performing timing-based tasks like pulse-width modulation (PWM). Think of the pin as a worker: `P_ASYNC_TX` is the instruction to "send letters (data) to the outside world," `P_OE` gives permission

to open the door (output enabled), and `wrpin` assigns the job. This is why `wrpin` is always the first step when setting up a SmartPin for UART, SPI, PWM, or similar tasks.

Syntax

```
wrpin ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
```

```
P_ASYNC_TX → sets the pin to send serial data (UART transmit)
P_OE → enables the output (so it can actually drive the signal)
| → combines multiple settings together
#TRANSMISSION_PIN → the pin number being configured
```

SmartPin speed

The `wxpin` instruction sets how fast a SmartPin sends or receives signals by defining its timing. This is especially important for serial communication, such as UART, where precise timing ensures data is transmitted and received correctly. Without `wxpin`, the pin has no concept of signal speed, which can lead to garbled data or communication errors.

Think of a SmartPin like a telegraph worker sending dots and dashes—`wxpin` tells the worker how long to hold each signal. It controls timing, not data, and is typically used after `wrpin`, since the pin must first know its role as TX, RX, or PWM. In serial setups, using `wxpin ##BAUD_MODE, #TRANSMISSION_PIN` sets the bit duration and frame format so data is sent at the correct speed.

Syntax

```
wxpin ##value, pin
value — the timing for the pin
```

- Usually calculated from the system clock (_CLKFREQ) and baud rate (BAUD_RATE).
- Example: if each bit should last 90 clock cycles, then value = 90.
- This tells the SmartPin: “Hold each bit for this many clock cycles.”

pin — which I/O pin to apply the timing to

- Example: #TRANSMISSION_PIN → the pin we are using for serial output.

Setting the output

The `dirh` instruction tells the Propeller 2 that a specific pin should function as an output, allowing it to drive a high or low voltage to the outside world. Until `dirh` is used, the pin remains in a neutral, input state and won’t actively send anything, even if it has been configured with `wrpin` or `wxpin`. In short, `dirh` turns the pin outward, making it a signal source rather than a listener.

Imagine each pin as a tiny faucet—`dirh` is turning the handle so water can flow out. Without it, the faucet stays closed and the pin can only listen. `dirh` controls direction, not timing or data, and is required whenever a pin must drive something like a UART TX line, an LED, or any SmartPin-based output. Even with perfect timing, no signal will leave the pin unless `dirh` is set.

Syntax

```
dirh #pin
```

pin — the number of the I/O pin you want to set as an output.

- Example: #TRANSMISSION_PIN sets the TX pin for sending data.

Sending Data

The `wypin` instruction is what actually sends data or a signal out through a SmartPin. Think of `wrpin`, `wxpin`, and `dirh` as the setup phase: `wrpin` assigns the pin's job, `wxpin` sets its timing or parameters, and `dirh` enables the output path. None of these steps produce an output on their own—they only prepare the pin. When you use `wypin`, the Propeller 2 takes the value you provide and delivers it to the SmartPin, causing the pin to transmit a byte, toggle a line, or perform whatever action its mode is configured for.

Using the worker analogy, `wypin` is the moment you hand the worker the data so the job actually gets done. This is when the byte, pulse, or waveform leaves the pin. Because of that, `wypin` must always come after the SmartPin has been configured, and it is essential for serial transmission, PWM generation, SPI shifting, and any other SmartPin-driven output.

Syntax

```
wypin #value, pin
     #value – the data or signal to send
     Could be a single byte for serial communication
     ("0", "A", 0x41, etc.)
     Could also be a high or low signal for controlling hardware
     pin – the number of the I/O pin that will transmit the data
     Example: #TRANSMISSION_PIN for serial output
```

Program: tx.spin2

```
'tx.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN
```

```
wypin  #"0", #TRANSMISSION_PIN

endprog
  jmp  #$
```

This is a simple example of sending a character from your Propeller 2 program to a terminal using a single pin. The process works as follows:

- **Clock and Baud Rate Setup:** The program defines the system clock (`_CLKFREQ` = 180,000,000) and the serial transmission speed (BAUD_RATE = 2,000,000). These values calculate the bit period, controlling how long each bit is transmitted.
- **Transmission Pin Setup:** Pin 62 is selected as the TRANSMISSION_PIN. The instructions configure the pin and prepare it for output:
  - wrpin ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN – sets the pin for asynchronous transmit mode and enables output.
  - wxpin ##BAUD_MODE, #TRANSMISSION_PIN – loads the baud rate and bit timing into the pin hardware.
  - wypin #1, #TRANSMISSION_PIN – sets the pin high initially (idle state for serial transmission).
  - dirh #TRANSMISSION_PIN – ensures the pin is configured as an output.
- **Sending the Character:** wypin #"0", #TRANSMISSION_PIN sends the ASCII value for "0" (decimal 48) out the pin. The hardware handles the timing and generates the correct serial waveform. The terminal displays "0", because the program automatically converts the number into the proper serial data for display.

This example demonstrates the full sequence for sending a character from the Propeller 2 to a screen using a single pin.

Exercise

Use `wypin` to send `"A"`, `"B"`, `"C"` and observe on a serial monitor.

Chapter 2: Data Movement

Moving data is essential because the Propeller 2 must remember values and send them to hardware components. Before the Propeller can perform any meaningful operation, we first need to place values in the correct location. This process is called data movement. You can think of it as transferring objects from one box to another: the **source** is where the number currently resides, and the **destination** is where we want the number to go. After the move, either both locations hold the value or only the destination does, depending on the specific instruction used. This simple yet fundamental concept underpins almost every program we write on the Propeller 2.

What is a Variable

A variable is a named location in memory that stores a value. You can think of it like a mailbox: it has a **name** so you can reference it easily in your program, a **memory location** where the value is kept, and it can hold different types of data such as numbers, letters (ASCII values), or any other information your program requires. Variables make it simple to organize, access, and manipulate data while the Propeller 2 executes instructions.

Declaring a Variable

You declare a variable by reserving space for it in memory using the `res` directive. For example, `mydata res 1` reserves one memory slot for the variable `mydata`. Initially, this memory may contain a random value, so it's important to assign a value to the variable before using it. Declaring variables ensures your program has a dedicated place to store and manage data during execution.

Storing a Value in a Variable

To store a value in a variable, use the `mov` instruction. For example, `mov mydata, #"A"` copies the ASCII value of the letter **A** (decimal 65) into the memory location reserved for `mydata`. After this instruction, `mydata` holds the value `"A"`, making it available for use elsewhere in your program.

Using the Variable

Once a variable has a value you can use it in instructions such as `wypin` to send it out, or include it in calculations. The code `wypin mydata, #TRANSMISSION_PIN` reads the value stored in `mydata` and sends it through the specified pin. The variable acts as a container for the value, allowing your code to use it dynamically without hardcoding numbers or letters directly, which makes your programs more flexible and easier to modify.

The MOV Instruction

The most basic instruction for moving data is the `mov` instruction. It copies a value from a **source** (like a number, register, or variable) into a **destination** (such as another variable or register), allowing the program to store and use data efficiently.

Syntax

```
mov destination, {#}source
```

- **destination** = where the value will go
- **source** = where the value comes from
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - Omit # if it is a **variable or register**.

Imagine a row of mailboxes labeled A, B, C... where each mailbox can hold a value. Using `mov A, B` is like taking whatever is in mailbox B and putting a copy into mailbox A. After the move, mailbox B still retains its original value, while mailbox A now holds the same value as B. This illustrates how `mov` copies data without altering the source.

Sometimes we want to start with a specific value, such as the letter "A". We use the # symbol to indicate a **literal value**, telling the processor to use the value directly rather than treating it as a register or memory location. This lets you assign exact numbers, letters, or other constants to variables or registers.

Example

```
mov mybox, #"A"
```

This means “put the letter A into mailbox `mybox`.” After this instruction, `mybox` holds the value `"A"`. The # symbol is required whenever you use a literal value, such as a number or letter, to tell the processor it’s a constant rather than a memory location or register.

We can also copy a value from one register to another using the `mov` instruction. This transfers the value from the **source register** into the **destination register**, leaving the original value unchanged in the source. It allows registers to share or reuse data without altering the original content.

Example

```
mov B, A
```

Here’s what happens: the value in mailbox A (`"A"`) is copied into mailbox B. After this operation, both A and B hold the letter `"A"`. This is a very common practice when you want to reuse a value or prepare it for calculations without changing the original.

Visualizing MOV

Instruction	Before	After
`MOV A, #8`	A = 0	A = 8
`MOV B, A`	B = 0	B = 8
`MOV OUTA, #1`	OUTA = 0	OUTA = 1

We can also move values into **output registers**. By placing the desired value in the correct output register, the Propeller 2 sends signals to the connected hardware, allowing your program to interact with the physical world.

Writing a value to a **SmartPin** can turn an LED on or off. The concept is the same as moving data between variables or registers: you transfer the value from a register to a location that controls hardware. This allows your program to interact with external devices, making it possible to control LEDs, motors, or other components directly through code.

Program: print_character.spin2

```
'print_character.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov character, #"A"
  wypin character, #TRANSMISSION_PIN

  endprog
    jmp  #$

  character res 1
```

When this program runs, you will see the letter `"A"` appear on the terminal screen. This happens because the program first stores the ASCII value for `"A"` in the variable `character`, then sends that value out through the transmission pin. The Propeller `2` handles all the timing and signal control behind the scenes, so the letter shows up on the screen just like typing it, without you having to worry about the electrical details.

Exercise

Load the letter `"B"` into register A, then copy the value from A into register B. Verify that both registers hold the same value by printing them or sending them to an output.

Chapter 3: Adding, Subtracting and Multiplying

The Propeller 2 microcontroller can perform a wide range of mathematical operations, from simple addition and subtraction to more advanced calculations. These operations are carried out directly on **registers** or **memory locations. This** gives you precise control over how values are manipulated, stored, and used in your program. This low-level approach allows for efficient and predictable computation, which is especially useful in embedded systems programming.

Why Math Matters

Math is essential because it is used in almost every program. It helps **count loops**, **control timing**, **move objects on a screen**, and **convert numbers for display or output**. Even a simple task like blinking an LED requires math to calculate the correct delay between turning it on and off. Math lets your program make decisions, measure time, and manipulate data in meaningful ways.

The ADD instruction

The ADD instruction adds two values together. It takes a **source** (which can be a register or a literal value) and a **destination register**. The value from the source is added to the current value in the destination. The **result is stored back in the destination register**, updating it with the new total. If we can perform `add A, #5. This` adds 5 to the value already stored in register A, and `add B, A` adds the value in register A to register B.

Sometimes you need to display a number over a serial connection or on a screen. Computers use ASCII codes to represent characters, assigning a unique

number to each one. For instance, the digits `'0'` through `'9'` correspond to ASCII codes 48 to 57, so sending the value 48 to the screen will display the character `'0'`.

Since register A contains 4, adding 48 converts it to the ASCII code 52, which represents the character `'4'`. Sending this value to a terminal or display will show `'4'`. This method works well for single-digit numbers, but it cannot handle multi-digit numbers directly. To display larger numbers, you must split the number into individual digits and convert each one to its corresponding ASCII code before sending them to the screen, ensuring that all digits are displayed correctly in sequence.

Syntax

```
add destination, {#}source
```

- **destination** = where the value will go
- **source** = where the value comes from
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - Omit # if it is a **variable or register**.

Program: `add.spin2`

```
'add.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov number1, #2
  mov number2, #2

  add number2, number1
```

```
add number2, #"0"

wypin number2, #TRANSMISSION_PIN

endprog
  jmp  #$

number1 res 1
number2 res 1
```

When this program runs, the terminal will display the character `"4"`. First, `mov number1, #2` stores the value 2 in `number1`, and `mov number2, #2` stores 2 in `number2`. Next, `add number2, number1` adds the value of `number1` to `number2`, so `number2` now holds 4. Then, `add number2, #"0"` converts the number 4 into its ASCII code by adding 48, giving the ASCII value for `"4"`. Finally, `wypin number2, #TRANSMISSION_PIN` sends this ASCII code to the terminal, resulting in the single character `"4"` appearing on the screen.

The SUB Instruction

The `SUB` instruction subtracts one value from another. The value from the source is **subtracted from the current value in the destination**, and the result is stored back in the destination register, updating it with the new value.

We could perform `sub A, #2.` This subtracts 2 from the value already stored in register A, and `sub B, A` subtracts the value in register A from register B. Just like with addition, numbers can be converted to ASCII characters for printing. The ASCII codes for the digits `'0'` through `'9'` range from 48 to 57, so to display a number between 0 and 9 as a character after performing a subtraction, you add 48 to the result. This converts the numeric value into its corresponding ASCII code, allowing it to appear correctly on the terminal or screen.

We can convert the result of a subtraction to displayable form. For example, if register A contains 6, performing sub A, `#2` subtracts 2, leaving A with the value 4. Adding 48 to A converts this numeric value to its corresponding ASCII code, 52, which represents the character '4'. Sending this value to a terminal will display '4'. This approach works well for single-digit results, but to display multi-digit numbers, you must split the number into individual digits and convert each one to its ASCII code before sending them to the terminal.

Syntax

`sub destination, {#}source`

- **destination** = where the value will go
- **source** = where the value comes from
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - `Omit # if it is a` **variable or register**.

Program: `sub.spin2`

```
'sub.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov number1, #5
  mov number2, #3

  sub number1, number2
  add number1, #"0"

  wypin number1, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1 res 1
  number2 res 1
```

When this program runs, the terminal will display the character `"2"`. First, `mov number1, #5` stores the value 5 in `number1`, and `mov number2, #3`

stores 3 in `number2`. Next, `sub number1, number2` subtracts `number2` from `number1`, leaving `number1` with the value `2`. Then, `add number1, #"0"` converts this number into its ASCII code by adding 48, giving the ASCII value for `"2"`. Finally, `wypin number1, #TRANSMISSION_PIN` sends the ASCII code to the terminal, resulting in the character `"2"` appearing on the screen.

The MUL Instruction

The `MUL` instruction multiplies two values. The value in the destination is multiplied by the source, and the **result is stored back in the destination register**, updating it with the new product. This allows you to perform precise multiplication directly on the values your program is working with.

The MUL instruction is used because it efficiently multiplies two numbers in a single operation. It works with both literal values and values stored in registers, making it versatile for many tasks. MUL is useful for arithmetic calculations, scaling values, or preparing data for output. This allows fast, single-cycle multiplication for integers, making your programs more efficient and precise.

Syntax

```
mul destination, {#}source
```

- **destination** = where the value will go
- **source** = where the value comes from
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - `Omit # if it is a` **variable or register**.

Program: mul.spin2

```
'mul.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
```

```
org 0
asmclk

wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
wypin  #1, #TRANSMISSION_PIN
dirh   #TRANSMISSION_PIN

mov number1, #2
mov number2, #3

mul number2, number1
add number2, #"0"
wypin number2, #TRANSMISSION_PIN

endprog
  jmp  #$

number1 res 1
number2 res 1
```

When this program runs, the terminal will display the character `"6"`. First, `mov number1, #2` stores the value `2` in `number1`, and `mov number2, #3` stores 3 in `number2`. Next, `mul number2, number1` multiplies `number2` by `number1`, so `number2` now holds 6. Then, `add number2, #"0"` converts this number into its ASCII code by adding 48, giving the ASCII value for `"6"`. Finally, `wypin number2, #TRANSMISSION_PIN` sends the ASCII code to the terminal, resulting in the character `"6"` appearing on the screen.

Exercise

In this exercise, you will practice manipulating variables using basic arithmetic instructions. First, increment a variable by 1 using the `ADD` instruction to increase its value. Next, decrease a variable and check whether it goes below zero, allowing you to handle underflow or trigger a conditional action. Finally, double a value using multiplication, which demonstrates how to scale a number efficiently in your program.

Chapter 4: Bitwise Operations

Bitwise operations include instructions that **manipulate individual bits** within a value. You could use a single byte to represent **eight different LEDs**, turning each one on or off individually by manipulating its corresponding bit. Bitwise operations are extremely efficient because they operate directly at the hardware level, allowing your program to make decisions and control devices **very quickly** without using extra memory or complex calculations. Typically, you'll use a bitmask during these operations. A **bitmask** is a pattern of bits used to select, set, or clear specific bits in a value.

The AND Instruction

The `AND` instruction is commonly used to check or isolate specific bits without changing the others. By combining a value with a **bitmask**—a pattern where 1s indicate the bits you care about and 0s elsewhere—you can test only the bits you want. Any bits that are 0 in the mask are cleared, leaving only the relevant bits. This makes `AND` extremely useful for reading hardware states, checking status flags, or handling multiple on/off signals packed into a single value.

If a register holds `00000101` and you apply a mask of `00000100` using `AND`, the result is `00000100`, showing that bit 2 was set while all other bits are cleared. `AND` sets each result bit to 1 only if both corresponding bits are 1, allowing you to isolate or clear specific bits without affecting the rest. This makes it an essential tool in embedded programming, hardware control, and efficient low-level operations.

Syntax

```
and destination, {#}source
```

- **destination** = where the value will go

- **source** = where the value comes from
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - `Omit # if it is a` **variable or register**.

Program: `and.spin2`

```
'and.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov number1, #13
  and number1, #5

  add number1, #"0"
  wypin number1, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1 res 1
```

The variable `number1` is first set to 13, which in binary is `00001101`. The AND instruction then performs a bitwise comparison with the value 5 (`00000101`), resulting in `00000101`, which is decimal 5. This value is then converted to an ASCII character by adding `"0"`, producing 53, which corresponds to the character `'5'`. The program sends this value to the terminal, so the expected output displayed on the screen is the single character 5. This demonstrates how AND can be used for bit masking to isolate specific bits in a value.

The `OR` Instruction

The OR instruction performs a **bitwise OR** between two values. Each bit in the result is set to `1` if **either or both** of the corresponding bits in the source or destination are `1`. The resulting value is stored back in the destination register. OR is commonly used to **set specific bits** without affecting the others, such as turning on particular LEDs, enabling flags, or configuring hardware states while leaving unrelated bits unchanged.

To turn on a specific bit, such as bit 2 for an LED, you can use a bitwise OR with a mask that has a 1 in that position (`00000100`). OR'ing the original value (`00000001`) with the mask results in `00000101`, keeping bit 0 ON while also turning on bit 2. This shows how OR can set specific bits safely without affecting the others, making it ideal for controlling hardware or managing multiple flags packed into a single byte or word.

Syntax

```
or destination, {#}source
```

- **destination** = where the value will go
- **source** = where the value comes from
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - `Omit # if it is a` **variable or register**.

Program: or.spin2

```
'or.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
```

```
dirh   #TRANSMISSION_PIN

mov number1, #4
or  number1, #3
add number1, #"0"
wypin number1, #TRANSMISSION_PIN

endprog
  jmp  #$

number1 res 1
```

The program first sets number1 to 4 (00000100 in binary). The OR instruction then performs a bitwise OR with 3 (00000011), resulting in 00000111, which is decimal 7. This value is then converted to an ASCII character by adding "0", giving 55, which corresponds to the character '7'. The program sends this value to the terminal, so the expected output displayed on the screen is the single character 7. This demonstrates how OR can be used to **turn on multiple bits at once** without affecting the other bits in the original value.

The XOR Instruction

The `XOR` instruction performs a bitwise exclusive OR between two values. Each result bit is set to 1 only if the corresponding bits in the source and destination are different; if they are the same, the result is 0. The resulting value is stored back in the destination register. `XOR` is commonly used to toggle specific bits, compare flags, or invert bits without affecting others.

It is especially useful for flipping bits in a register—changing 1 to 0 or 0 to 1—without altering the rest. By XOR'ing with a mask that has 1s in the positions you want to toggle, bits corresponding to 1s are inverted while bits corresponding to 0s remain unchanged. This makes `XOR` an efficient way to invert flags, switch LED states, or manipulate individual bits in a value safely.

Suppose a register holds the value `00000001`, where bit 0 represents a "Power" flag. To toggle bit 2 ("Status") without affecting bit 0, you can XOR the register with `00000100`. The result is `00000101`, so both the "Power" and "Status" bits are now on. Running the same XOR operation again will toggle bit 2 back off, demonstrating how XOR efficiently flips specific bits while leaving others unchanged.

Syntax

`xor destination, {#}source`

- **destination** = where the value will go
- **source** = where the value comes from
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - Omit # if it is a **variable or register**.

Program: `xor.spin2`

```
'xor.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov number1, #6
  xor number1, #3

  add number1, #"0"
  wypin number1, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1 res 1
```

We set number1 to 6, which in binary is 00000110. The XOR instruction then performs a bitwise exclusive OR with 3 (00000011), resulting in 00000101, which is decimal 5. This value is converted to an ASCII character by adding "0", producing 53, which corresponds to the character '5'. The program sends this value to the terminal, so the expected output displayed on the screen is the

single character 5. This example demonstrates how XOR can be used to toggle specific bits in a value while leaving the other bits unchanged.

The NOT Instruction

The `NOT` instruction performs a bitwise inversion of a value, flipping every bit in the destination register so that 1s become 0s and 0s become 1s. The result is stored back in the same register. This operation is useful for creating the binary opposite of a number, inverting flags, generating masks, or toggling all bits in a value at once without affecting other registers, making it a powerful tool in low-level and embedded programming.

If a register holds the binary value `00000001` (decimal 1), applying the NOT instruction flips every bit, resulting in `11111110`, which is decimal 254 in an 8-bit value. This demonstrates how NOT inverts all bits in a value, producing the binary opposite of the original number.

Syntax

```
not destination
```

- **destination** = where the value will go

Program: `not.spin2`

```
'not.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN
```

```
mov number1, #3
not number1

and number1, #%1111

add number1, #"0"
wypin number1, #TRANSMISSION_PIN

endprog
  jmp  #$

number1 res 1
```

We set `number1` to 3, which in binary is `00000011`. The `NOT` instruction inverts all bits, producing `11111100` in an 8-bit context. To focus on just the lowest 4 bits, the program then performs a bitwise AND with `1111`, resulting in `1100`, which is decimal 12. This value is then converted to an ASCII character by adding `"0"`, giving 60. ASCII 60 corresponds to the single character `'<'`, not the digits `"12"`.

Exercise

In this exercise, you will practice manipulating individual bits in a register. First, use AND with a mask to keep only bit 2 of number1, clearing all other bits. Next, use OR with a mask to turn on bit 0 without affecting the other bits. Then, use XOR with a mask to toggle bit 3, flipping it while leaving the rest unchanged. Finally, apply NOT to invert all bits of number1, creating the binary opposite of its current value.

Chapter 5: Shifting

Shifting instructions allow you to move the bits in a value **left or right**. Shifting left or right is useful for efficiently multiplying or dividing by powers of 2, manipulating individual flags, or working with packed binary data. By moving bits, you can quickly scale values, isolate or position specific bits, and perform low-level operations that are essential in embedded programming.

SHL — Shift Left

The `SHL` (Shift Left) instruction moves the bits in a value to the left, with each shift effectively multiplying the value by 2. It takes a destination register and a source—either another register or a literal—specifying how many positions to shift. New bits on the right are always set to 0, and the result is stored back in the destination register. `SHL` is commonly used for efficient multiplication by powers of 2, manipulating specific bits, or positioning data within a register, providing a fast, high-performance alternative to standard multiplication in low-level programming.

Syntax

```
shl destination, {#}source
```

- **destination** = where the value will go
- **source** = where the value comes from
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - `Omit # if it is a` **variable or register**.

Program: shl.spin2

```
'shl.spin2
CON
  _CLKFREQ          = 180_000_000
```

```
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov     number1, #2
  shl     number1, #2
  add     number1, #"0"
  wypin   number1, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1 res 1
```

Lets set `number1` to 2. The SHL instruction then shifts this value left by 2 positions, effectively multiplying it by 4, resulting in 8. Adding `"0"` converts this numeric value to its ASCII code, producing 56, which corresponds to the character `'8'`. Finally, the program sends this value to the terminal via the TX pin, so the expected output displayed on the screen is the single character `'8'`. This example demonstrates how shifting can be used for fast multiplication and combined with ASCII conversion for terminal output.

SHR — Shift Right

The `SHR` (Shift Right) instruction moves the bits of a value to the right, effectively dividing it by 2 for each position shifted. It takes a destination register and a source—either a register or a literal specifying how many positions to shift—and fills new bits on the left according to arithmetic shift rules. The result is stored back in the destination register. `SHR` is commonly used for fast division by powers of 2, extracting specific bits, or manipulating packed binary data in low-level programming.

Syntax

```
shr destination, {#}source
```

- **destination** = where the value will go
- **source** = where the value comes from
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - Omit # if it is a **variable or register**.

Program: shr.spin2

```
'shr.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov     number1, #16
  shr     number1, #1
  add     number1, #"0"
  wypin   number1, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1 res 1
```

Here, we set `number1` to 16. The SHR instruction then shifts this value right by 1 position, effectively dividing it by 2, resulting in 8. Adding `"0"` converts this numeric value to its ASCII code, producing 56, which corresponds to the character `'8'`. Finally, the program sends this value to the terminal via the TX pin, so the expected output displayed on the screen is the single character `'8'`. This example demonstrates how SHR can be used for fast division by powers of 2 and combined with ASCII conversion for terminal output.

SAL — Shift Arithmetic Left

The SAL (Shift Arithmetic Left) instruction shifts the bits of a value to the left, effectively multiplying it by 2 for each position shifted while preserving the sign of signed numbers. It takes a destination register and a source—either a register or an immediate value specifying how many positions to shift—and stores the result back in the destination register. SAL is useful for efficient multiplication of signed numbers, handling signed values correctly, and manipulating bits in low-level programming, making it ideal for real-time calculations, games, animations, or signal processing.

Syntax

```
sal destination, {#}source
```

- **destination** = where the value will go
- **source** = where the value comes from
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - Omit # if it is a **variable or register**.

Program: sal.spin2

```
'sal.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov     number1, #4
  sal     number1, #1
```

```
add     number1, #"0"
wypin   number1, #TRANSMISSION_PIN

endprog
  jmp  #$

number1 res 1
```

The variable `number1` is set to 4. The SAL instruction then shifts this value left by 1 position, effectively multiplying it by 2, resulting in 8. Adding `"0"` converts this numeric value to its ASCII code, producing 56, which corresponds to the character `'8'`. Finally, the program sends this value to the terminal via the TX pin, so the expected output displayed on the screen is the single character `'8'`. This demonstrates how SAL can be used for fast multiplication while preserving signed values and combining it with ASCII conversion for display.

SAR — Shift Arithmetic Right

The `SAR` (Shift Arithmetic Right) instruction shifts the bits of a value to the right, effectively dividing it by 2 for each position while preserving the sign of signed numbers through sign extension of the leftmost bit. It takes a destination register and a source—either a register or an immediate value specifying how many positions to shift—and stores the result back in the destination register. `SAR` is useful for fast, efficient division of signed numbers, bit manipulation, and low-level arithmetic operations in embedded programming, making it ideal for real-time applications like games, signal processing, or animations.

Syntax

```
sar destination, {#}source
```

- **destination** = where the value will go
- **source** = where the value comes from
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - `Omit # if it is a` **`variable or register`**.

Program: sar.spin2

```
'sar.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov     number1, #4
  sar     number1, #1
  add     number1, #"0"
  wypin   number1, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1 res 1
```

If the value 4 is shifted right by 1 using SAR, the result is 2. Adding `"0"` converts this numeric value to its ASCII code, producing the character `'2'`. Sending this value over the TX pin will display `'2'` on a connected terminal, demonstrating how SAR can be used for fast division by powers of 2 while preserving the sign for signed values and combining it with ASCII conversion for terminal output.

Exercise

In this exercise, you will practice using shift instructions to manipulate values. First, use `SHL` to shift the value in `number1` one position to the left, doubling its current value. Next, use `SHR` to shift `number1` two positions to the

right, effectively dividing it by four. Then, apply `SAL` to shift the signed value in `number1` three positions to the left, multiplying it by eight while preserving its sign. Finally, use `SAR` to shift the signed value in `number1` one position to the right, halving it while keeping the sign intact.

Chapter 6: Loops

Loops allow the CPU to repeatedly execute a block of code without rewriting instructions. Unlike high-level languages, loops are built using conditional tests and jump instructions, often paired with counters stored in registers. They are essential for tasks such as blinking LEDs, reading multiple inputs, or processing arrays in memory, where repetition at the hardware level would otherwise be inefficient and error-prone.

Loops are a fundamental tool in P2ASM, allowing the CPU to execute a section of code multiple times without repeating instructions. They provide precise control over timing and iteration by using counters, conditions, or flags, making them flexible for tasks such as controlling hardware, reading inputs, and processing data in memory. Loops are created by combining counters stored in registers with conditional or unconditional jump instructions that control program flow, keeping programs compact, readable, and less prone to errors.

By combining these instructions with registers as counters, you can create flexible loops that either repeat a section of code a precise number of times or continue running until a specific condition is met. This approach gives P2ASM programmers precise control over program flow, making loops essential for tasks ranging from hardware control to data processing.

JMP — Jump (Unconditional)

The `JMP` instruction causes an unconditional jump to a specified location in the program. Immediately after, execution continues from that target location. This makes `JMP` essential for tasks such as skipping code, repeating sections in loops, redirecting program flow, or handling errors.

Syntax

```
jmp #address
```

- **address** = the label, memory location, or constant to jump to.
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - Omit # if it is a **variable or register**.
  - **JMP** always unconditionally transfers execution to the target address.

Program: jmp.spin2

```
'jmp.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  jmp     #larger_value:

  smaller_value:
    mov     number1, #2

  larger_value:
    mov     number1, #4
    add     number1, #"0"
    wypin   number1, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1 res 1
```

A label smaller_value is defined but immediately jumps to larger_value using JMP #larger_value. This means the code under smaller_value is skipped, and execution continues at larger_value. At larger_value, the program sets number1 to 4, converts it to its ASCII equivalent by adding "0", and then transmits it using WYPIN on the TRANSMISSION_PIN. As a result, the value 4 is sent, and the instructions under smaller_value never execute.

DJNZ — Decrement and Jump if Not Zero

The DJNZ (Decrement and Jump if Not Zero) instruction decreases the value in a register by one and checks the result. If the value is not zero, execution jumps to the specified target, repeating the code; if it is zero, execution continues to the next instruction, ending the loop. By combining decrementing and conditional branching into a single operation, DJNZ enables compact, efficient count-controlled loops, making it ideal for timing loops, repetitive I/O, and buffer or memory processing in embedded systems where speed and code size matter.

Syntax

`djnz register, {#}target`

- **register** = the register or variable whose value will be decremented.
  - Always a register/variable, so do not use #.
- **target** = the address or label to jump to if the result is not zero..
  - `Use # for a literal numeric address.`
  - `Omit # if it is a label defined in your code.`

Program: djnz.spin2

```
'djnz.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
```

```
wypin  #1, #TRANSMISSION_PIN
dirh   #TRANSMISSION_PIN

mov     number1, #5

count_loop
  djnz    number1, #process_number
  jmp     #done

process_number
  mov     ascii_number, number1
  add     ascii_number, #"0"
  jmp     #count_loop

done
  wypin   ascii_number, #TRANSMISSION_PIN

endprog
  jmp  #$

number1 res 1
ascii_number    res 1
```

In the first step, `number1` is initialized to 5 and the program enters the `count_loop`. The `DJNZ` instruction decrements `number1` by 1 and checks if it is zero. If `number1` is not zero, execution jumps to `process_number`, where the current value of `number1` is copied to `ascii_number` and converted to its ASCII equivalent by adding `"0"`. The program then jumps back to `count_loop`. This process repeats until `number1` reaches zero. Finally, at the `done` label, the program sends the last stored value of `ascii_number` via `WYPIN` on the `TRANSMISSION_PIN`. As a result, only the final number before zero (1 in this case) is transmitted to the terminal, while all previous numbers are decremented but not printed.

TJNZ — Test and Jump if Not Zero

The TJNZ (Test and Jump if Not Zero) instruction checks the value of a register and conditionally alters program flow: if the register is not zero, execution jumps to a specified label; if it is zero, the program continues with the next instruction. Because TJNZ tests a register without modifying it, it is ideal for loops, counters, flags, and other conditional branches. By combining the test and jump in a single instruction, it simplifies code, eliminates the need for

separate compare operations, and makes status checks, loop control, and event polling more efficient and readable.

Syntax

`tjnz register, {#}target`

- **register** = the register or variable whose bit will be tested.
  - Always a register/variable, so do not use #.
- **target** = the address or label to jump to if the tested bit is not zero.
  - `Use # for a literal numeric address.`
  - `Omit # if it is a label defined in your code.`

Program: tjnz.spin2

```
'tjnz.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov     number1, #5

  count_loop
    tjnz    number1, #process_number
    jmp     #done

  process_number
    mov     ascii_number, number1
    add     ascii_number, #"0"
    sub     number1, #1
    jmp     #count_loop

  done
    wypin   ascii_number, #TRANSMISSION_PIN
```

```
endprog
  jmp  #$

number1 res 1
ascii_number res 1
```

We initialize `number1` to 5 as the counter. The program enters `count_loop`, where `TJNZ` tests if `number1` is non-zero. If it is non-zero, execution jumps to `process_number`; if it is zero, it jumps to `done` and ends the loop. In `process_number`, the current value of `number1` is copied into `ascii_number`, converted to its ASCII code by adding `"0"`, and then `number1` is decremented by 1. Execution then jumps back to `count_loop`. Because `WYPIN` is only at the `done` label, only the last value of `ascii_number` (1 in this case) is transmitted via the transmission pin, not the other numbers in the countdown.

TJZ — Test and Jump if Zero

The TJZ (Test and Jump if Zero) instruction checks whether a register contains zero. If it does, execution jumps to the specified label; otherwise, the program continues with the next instruction. Since TJZ tests a register without modifying it, it is useful for conditional branching, such as wait loops, monitoring flags, or detecting when a counter reaches zero. Commonly used in embedded control, status monitoring, and polling routines, TJZ complements TJNZ by providing the opposite condition check, giving programmers flexible control over program flow.

Syntax

`tjz register, {#}target`

- **register** = the register or variable whose bit will be tested.
    - Always a register/variable, so do not use #.
- **target** = the address or label to jump to if the tested bit is not zero.
    - `Use # for a literal numeric address.`

- Omit # if it is a label defined in your code.

Program: tjz.spin2

```
'tjz.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov     number1, #5

  process_number
    sub     number1, #1
    tjz     number1, #done
    mov     ascii_flag, number1
    add     ascii_flag, #"0"
    jmp     #process_number

  done
      wypin   ascii_flag, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1 res 1
  ascii_flag  res 1
```

We initialize `number1` to 5 as the counter. The program enters `process_number`, where `number1` is first decremented by 1. The `TJZ` instruction checks if `number1` is zero: if so, execution jumps to `done`; if not, it continues. The current value of `number1` is copied into `ascii_flag` and converted to its ASCII code by adding `"0"`. Execution then jumps back to `process_number`. Because the `WYPIN` instruction is only at the `done` label, only the last value of `ascii_flag` before the loop ends (1 in this case) is transmitted via the transmission pin, not the other numbers in the countdown.

Exercise

Practice using P2ASM jump instructions by creating loops and conditional branches with different behaviors. Start by using `JMP` to create a loop that sets `number1` to 1, jumps back to the loop label three times, and then sets `number1` to 5 and prints it as ASCII. Next, initialize a counter to 4 and use `DJNZ` to decrement it in a loop, printing the counter value as ASCII each iteration until it reaches zero. Then, set a counter to 3 and use `TJNZ` to repeat a block that increments `number1` by 2, printing its value as ASCII each time until the counter reaches zero. Finally, initialize a flag to 5 and use `TJZ` to wait in a loop while decrementing the flag, and once the loop finishes, set `number1` to 9 and print it as ASCII.

Chapter 7: Decision Making (Conditional Execution)

Decision-making instructions allow your program to react to different situations. Rather than executing every instruction sequentially, the program can check whether a register is zero, not zero, or has overflowed, compare two values, interpret numbers as signed or unsigned, and execute instructions only when specific conditions are met, without relying on jumps.

Decision-making relies on flags, which are special bits that indicate the result of an operation. The **Zero Flag (Z)** is set when a result equals zero, and the **Carry Flag (C)** is set when an operation produces a carry, a borrow, or certain unsigned comparisons. By using these flags, instructions can run only when needed, saving time and reducing the need for multiple jumps. This allows assembly code to implement conditional logic similar to if-else statements in higher-level languages, works with both signed and unsigned numbers, and keeps programs efficient and easier to read.

Decision Instructions

Comparison	CMP / Flags	IF_* Instruction(s)	Notes / Example
Equal (==)	`cmp a, b` `wz`	`if_z`	Executes if `a = b`
Not equal (!=)	`cmp a, b` `wz`	`if_nz`	Executes if `a ≠ b`
Less than (<)	`cmp a, b` `wc`	`if_c`	Executes if `a < b` (unsigned)
Greater than (>)	`cmp a, b` `wc`	`if_nc if_nz`	Executes if `a > b` (unsigned); combine to exclude equality
Less than or equal (<=)	`cmp a, b` `wc`	`if_be`	Executes if `a ≤ b` (unsigned); IF_BE = C=1 or Z=1
Greater than or equal (>=)	`cmp a, b` `wc`	`if_nc`	Executes if `a ≥ b` (unsigned)

CMP — Compare

The CMP (Compare) instruction evaluates the relationship between two values without modifying either operand. It is commonly used to make decisions based on equality or relative magnitude.

CMP performs an internal subtraction between a destination register and a source value, which may be another register or a constant. The result of this subtraction is not stored. Instead, CMP updates one or more condition flags that describe the relationship between the two values.

CMP can update condition flags using modifiers. The **wz** (write zero flag) modifier updates the Zero flag, setting **Z = 1** if the values are equal and **Z = 0** if they are not. The **wc** (write carry flag) modifier updates the Carry flag, which reflects the relative magnitude of the compared values, depending on whether the comparison is interpreted as signed or unsigned. The **wcz** (write carry and zero flags) modifier updates both flags simultaneously, allowing a single comparison to capture both equality and ordering information.

Because CMP does not change the compared values, it is well suited for decision-making, equality checks, range testing, and loop control. The flags it sets can be examined later to determine how program execution should proceed.

Syntax

```
cmp destination, {#}source wz
cmp destination, {#}source wc
```

- **destination** = the register or variable whose value will be compared
- **source** = the value to compare with
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - Omit # if it is a **variable or register**.
- **flags** = the value to compare with
  - wz = sets the WZ (zero) flag if destination - source = 0
  - wc sets the WC (carry) flag if destination < source (unsigned).

IF_Z — Execute If Zero

The IF_Z instruction executes the next instruction only if the Zero (Z) flag is set (Z = 1). It is used to control program flow based on whether a previous operation resulted in zero or whether two values are equal.

The Zero flag is usually set by a previous instruction, such as CMP or an arithmetic operation. When IF_Z runs, it checks this flag: if Z = 1, the next instruction is executed; if Z = 0, the next instruction is skipped. This allows inline conditional execution without needing a separate jump or branch instruction.

Syntax

`if_z instruction`

- **instruction** = the instruction to execute
  - The instruction can be any standard P2ASM instruction, including shifts, arithmetic, bitwise, etc.
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - Omit # if it is a **variable or register**.
  - Only runs the instruction if the zero flag (WZ) is set.

Program: if_z.spin2

```
'if_z.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
  wypin   #1, #TRANSMISSION_PIN
```

```
dirh    #TRANSMISSION_PIN

mov     number1, #2
mov     number2, #2

cmp     number1, number2 wz
if_z    mov     isEqual, #"1"

wypin   isEqual, #TRANSMISSION_PIN

endprog
  jmp  #$

number1 res 1
number2 res 1
isEqual res 1
```

Since `number1` and `number2` are both 2, `CMP` sets the Zero flag (Z) to 1. The `IF_Z` instruction then executes the next line, moving `"1"` into `isEqual`, which is sent to the transmission pin. This outputs 1. If the numbers were not equal, Z would be 0, `IF_Z` would not run, and nothing would be sent to the pin.

IF_NZ — Execute If Not Zero

The `IF_NZ` instruction runs the next instruction only if the Zero (Z) flag is clear (Z = 0). The Z flag is usually set by a previous instruction like `CMP` or an arithmetic operation. This lets your program **execute code only when two values are not equal**, making `IF_NZ` useful for decision-making and branching in cases where values differ.

`IF_NZ` is useful because it runs instructions only when a condition is not met, such as when two values are different. This avoids unnecessary jumps for simple checks, works together with `CMP` to create “if not equal” logic, and makes assembly programs easier to read by allowing inline conditional execution.

Syntax

```
if_nz instruction
```

- **`instruction`** = the instruction to execute
  - The instruction can be any standard P2ASM instruction, including shifts, arithmetic, bitwise, etc.

- Use # if it is a **literal constant** (number, hex, ASCII).
- Omit # if it is a **variable or register**.
- Only runs the instruction if the zero flag (WZ) is not set.

Program: if_nz.spin2

```
'if_nz.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
  wypin   #1, #TRANSMISSION_PIN
  dirh    #TRANSMISSION_PIN

  mov     number1, #2
  mov     number2, #3

  cmp     number1, number2 wz
  if_nz   mov     isNotEqual, #"0"

  wypin   isNotEqual, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1 res 1
  number2 res 1
  isNotEqual res 1
```

While number1 is 2 and number2 is 3, CMP sets the Zero flag (Z) to 0. The IF_NZ instruction then executes the next line, moving "0" into isNotEqual, which is sent to the transmission pin and outputs **0**. If the numbers had been equal, Z would be 1, IF_NZ would skip the instruction, and nothing would be sent to the pin.

CMP + IF_Z / IF_NZ — Conditional Execution Based on Comparison

By combining `CMP` with `IF_Z` and `IF_NZ`, your program can **conditionally execute instructions based on equality** without needing jumps or loops for simple checks.

Using `CMP` with `IF_Z` and `IF_NZ` makes decision-making in P2ASM simple and efficient. It avoids unnecessary jumps when you only need to run a single instruction conditionally, allows inline "if-else" style behavior, and is ideal for small equality checks, flags, and quick decision routines.

Syntax

```
cmp destination, {#}source wz
if_z    instruction
if_nz   instruction
```

Program: if_z_if_nz.spin2

```
' if_z_if_nz.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
  wypin   #1, #TRANSMISSION_PIN
  dirh    #TRANSMISSION_PIN

  mov     number1, #6
  mov     number2, #7

  cmp     number1, number2 wz
  if_z    mov     isEqual, #"1"
  if_nz   mov     isEqual, #"0"

  wypin   isEqual, #TRANSMISSION_PIN
```

```
endprog
  jmp  #$

number1 res 1
number2 res 1
isEqual res 1
```

For our program, `number1` is 6 and `number2` is 7, the values are not equal, so `CMP` sets the Zero flag (Z) to 0. The `IF_Z` instruction is skipped because Z = 0, while `IF_NZ` executes, moving `"0"` into `isEqual`, which is sent to the transmission pin and outputs 0. Only one of the conditional instructions runs depending on the Z flag, demonstrating inline conditional execution based on a comparison.

CMP + IF_C — Conditional Execution Based on Carry (Unsigned Comparison)

The **IF_C** instruction executes the next instruction only when the Carry flag (**C = 1**), making it useful for unsigned "less than" comparisons. When combined with **CMP wc**, it allows your program to conditionally execute instructions based on the relative size of two values without using jumps. This approach keeps the compared registers unchanged and provides an efficient way to implement decision-making, sorting routines, and loops where it is necessary to test whether one value is smaller than another.

Syntax

```
if_c instruction
```

- **instruction** = the instruction to execute
  - The instruction can be any standard P2ASM instruction, including shifts, arithmetic, bitwise, etc.
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - Omit # if it is a **variable or register**.

- Only runs the instruction if the carry flag (WC) is set.

Program: if_c.spin2

```
' if_c.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
  wypin   #1, #TRANSMISSION_PIN
  dirh    #TRANSMISSION_PIN

  mov     number1, #1
  mov     number2, #2

  cmp     number1, number2 wc
  if_c    mov     isLessThan, #"1"

  wypin   isLessThan, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1     res 1
  number2     res 1
  isLessThan  res 1
```

Here `number1` is 1 and `number2` is 2, `number1` is less than `number2`, so `CMP` sets the Carry flag (C) to 1. The `IF_C` instruction then executes, moving `"1"` into `isLessThan`, which is sent to the transmission pin and outputs 1. If `number1` were greater than or equal to `number2`, C would be 0, the `IF_C` instruction would not run, and nothing would be sent to the pin.

CMP + IF_NC — Conditional Execution Based on Not Carry (Unsigned Greater or Equal)

The `IF_NC` instruction then executes the next instruction only when C = 0, making it useful for unsigned "greater than or equal" comparisons. By combining `CMP wc` with `IF_NC`, your program can conditionally execute instructions based on relative size without using jumps.

Using `CMP` with `IF_NC` provides an efficient way to perform unsigned greater-than-or-equal comparisons in P2ASM. It allows instructions to execute conditionally without adding extra jumps, keeps the compared registers unchanged, and is especially useful in decision-making, sorting routines, and loops where you need to test whether one value is greater than or equal to another.

Syntax

```
if_nc instruction
```

- **instruction** = the instruction to execute
  - The instruction can be any standard P2ASM instruction, including shifts, arithmetic, bitwise, etc.
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - Omit # if it is a **variable or register**.
  - Only runs the instruction if the carry flag (WC) is not set.

Program: if_nc.spin2

```
' if_nc.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
```

```
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
  wypin   #1, #TRANSMISSION_PIN
  dirh    #TRANSMISSION_PIN

  mov     number1, #6
  mov     number2, #2

  cmp     number1, number2 wc
  if_nc   mov     isGreaterOrEqual, #"1"

  wypin   isGreaterOrEqual, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1            res 1
  number2            res 1
  isGreaterOrEqual   res 1
```

In this if_nc example, `number1` is 6 and `number2` is 2, `number1` is greater than or equal to `number2`, so `CMP` sets the Carry flag (C) to 0. The `IF_NC` instruction then executes, moving `"1"` into `isGreaterOrEqual`, which is sent to the transmission pin and outputs 1. If `number1` were less than `number2`, C would be 1, the `IF_NC` instruction would not run, and nothing would be sent to the pin.

CMP + IF_C / IF_NC — Unsigned Conditional Execution

The **IF_C** instruction executes the next instruction only when the Carry flag (**C = 1**), while **IF_NC** executes the next instruction only when **C = 0**. Together, these instructions enable unsigned comparisons with inline conditional execution, allowing your program to respond differently depending on whether a value is smaller or larger/equal, all without using jumps. By combining **CMP** with **IF_C** and **IF_NC**, conditional execution becomes simple and efficient: instructions can run only when needed, making this approach ideal for decision-making tasks such as sorting, checking thresholds, or controlling outputs. It also

keeps the original register values unchanged while using flags to guide program flow.

Syntax

```
cmp destination, {#}source wc
if_c  instruction
if_nc instruction
```

Program: if_c_if_nc.spin2

```
' if_c_if_nc.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
  wypin   #1, #TRANSMISSION_PIN
  dirh    #TRANSMISSION_PIN

  mov     number1, #4
  mov     number2, #8

  cmp     number1, number2 wc
  if_c    mov     isLessThan, #"1"
  if_nc   mov     isLessThan, #"0"

  wypin   isLessThan, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1     res 1
  number2     res 1
  isLessThan  res 1
```

The variable `number1` is 4 and `number2` is 8, 4 is less than 8, so `CMP` sets the Carry flag (C) to 1. The `IF_C` instruction executes, moving `"1"` into

`isLessThan`, which is sent to the transmission pin and outputs **1**. The `IF_NC` instruction is skipped because C ≠ 0. If `number1` were greater than or equal to `number2`, C would be 0, `IF_C` would skip, and `IF_NC` would execute instead, sending `"0"` to the pin.

CMP + IF_BE — Unsigned Below or Equal Conditional Execution

The **IF_BE** instruction executes the next instruction only when the first value is less than or equal to the second, which occurs when the Carry (**C = 1**) or Zero (**Z = 1**) flag is set. When combined with **CMP**, it enables efficient unsigned "≤" comparisons and inline conditional execution without extra jumps. This approach allows instructions to run conditionally for tasks such as threshold checks, sorting routines, or enforcing limits, while keeping the original registers unchanged and using the Carry and Zero flags to guide program flow.

Syntax

`if_be instruction`

- **instruction** = the instruction to execute
  - The instruction can be any standard P2ASM instruction, including shifts, arithmetic, bitwise, etc.
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - Omit # if it is a **variable or register**.
  - Executes the instruction **only if the first value is less than or equal to the second**.

Program: if_be.spin2

```
' if_be.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
```

```
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
  wypin   #1, #TRANSMISSION_PIN
  dirh    #TRANSMISSION_PIN

  mov     number1, #5
  mov     number2, #5

  cmp     number1, number2 wcz
  if_be   mov     isLessOrEqual, #"1"

  wypin   isLessOrEqual, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1       res 1
  number2       res 1
  isLessOrEqual res 1
```

If `number1` is 5 and `number2` is 5, `number1` is less than or equal to `number2`, so `CMP wcz` sets the Carry flag C = 0 and the Zero flag Z = 1. The `IF_BE` instruction executes, moving `"1"` into `isLessOrEqual`, which is sent to the transmission pin and outputs 1. If `number1` were greater than `number2`, C = 0 and Z = 0, `IF_BE` would skip, and nothing would be sent to the pin.

CMP + IF_NC / IF_NZ — Unsigned Greater than Conditional Execution

The IF_NC + IF_NZ combination executes the next instruction only when the first value is greater than the second, which occurs when the Carry flag (C = 0) and the Zero flag (Z = 0) are both clear. When combined with CMP, this enables efficient unsigned “>” comparisons and inline conditional execution without requiring extra jumps. This approach allows instructions to run

conditionally for tasks such as threshold checks, sorting routines, or enforcing limits, while keeping the original registers unchanged and using the Carry and Zero flags to guide program flow.

Syntax

```
cmp destination, {#}source wc
    if_nc  instruction
    if_nz instruction
```

Program: if_nc_if_nz.spin2

```
' if_nc_if_nz.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
  wypin   #1, #TRANSMISSION_PIN
  dirh    #TRANSMISSION_PIN

  mov     number1, #8
  mov     number2, #4

  cmp     number1, number2 wc
  if_nc if_nz  mov isGreater, #"1"   ' Executes if number1 > number2
  if_c         mov isGreater, #"0"   ' Executes if number1 <= number2

  wypin   isGreater, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1  res 1
  number2  res 1
  isGreater res 1
```

If `number1` is 8 and `number2` is 4, `number1` is greater than `number2`. The `CMP wc` instruction sets the Carry flag (C) = 0 (indicating `number1 ≥ number2`) and the Zero flag (Z) = 0 (indicating `number1 ≠ number2`). The combination `IF_NC IF_NZ` only executes when both conditions are true, so `"1"` is moved into `isGreater` and sent to the transmission pin, resulting in the output `1`. If `number1` were less than or equal to `number2`, either C = 1 or Z = 1, so `IF_NC IF_NZ` would skip, and `"0"` would be moved into `isGreater`, reflecting that the condition is false.

Chapter 8: Subroutines

Subroutines allow you to organize your program into smaller, logical sections. This allows for reuse code for tasks that repeat or require encapsulation. P2ASM supports procedures using `CALL`, `RET`, `JMP`, and local labels, giving you a structured way to manage program flow.

In P2ASM, `JMP` performs an unconditional jump to a label or address. `CALL` jumps to a location in code and saves the return address, allowing the program to return after execution. `RET` returns control to the instruction immediately following the last `CALL`. Local labels, prefixed with a dot (`.`), are visible only within the subroutine and are used for loops or conditional branches inside the subroutine.

Subroutines reduce code duplication, making programs easier to read, maintain, and debug. They enable structured programming even in assembly, allowing you to break complex tasks into manageable parts. Local labels prevent naming collisions between different procedures, keeping each section of code self-contained and organized.

JMP revisited Instruction

The JMP (Jump) instruction causes an unconditional transfer of control to a specified memory location, immediately setting the program counter to that address and skipping any instructions in between. Unlike CALL, it does not save

a return address. JMP is a fundamental tool in P2ASM for controlling program flow, enabling loops, branching, subroutines, error handling, or skipping sections of code. Its behavior is simple and predictable, as it always transfers execution to the target without checking flags or conditions.

Syntax

`jmp #address`

- **address** = the label, memory location, or constant to jump to.
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - Omit # if it is a **variable or register**.
  - **JMP** always unconditionally transfers execution to the target address.

Program: jmp.spin2

```
'jmp.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov     number1, #6

  jmp     #SetAndPrint

  mov     number1, #3   ' This will be skipped

  SetAndPrint
    mov     number1, #5
    add     number1, #"0"
    wypin   number1, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1 res 1
```

Execution begins with mov `number1, #6`, but the following instruction mov `number1, #3` is skipped because `JMP #SetAndPrint` immediately redirects execution to the `SetAndPrint` label. At `SetAndPrint`, `number1` is set to 5, converted to its ASCII code by adding `"0"`, and sent out via `WYPIN` on `TRANSMISSION_PIN`. The program then reaches `endprog` and terminates. This demonstrates how `JMP` can be used to skip instructions and jump directly to a specific section of code.

CALL Instruction

The `CALL` instruction in P2ASM jumps to a subroutine while saving the return address, allowing the program to resume execution immediately after the `CALL` once the subroutine finishes. Unlike `JMP`, which simply redirects execution, `CALL` enables structured programming by letting you reuse code without repeating it, improving organization and maintainability.

When executed, the processor first saves the address of the instruction following the `CALL`, then jumps to the specified subroutine label. The subroutine code runs normally, and a `RET` instruction returns execution to the saved address. This mechanism supports modular, reusable subroutines and structured programming patterns such as loops, conditional processing, and modular I/O handling, making programs easier to read, maintain, and debug.

Syntax

```
call #address
ret
```

- `address = the label, memory location, or constant to jump to.`
- `ret must be used` **`inside the subroutine`** `to return control.`
  - Use # if it is a **literal constant** (number, hex, ASCII).
  - `Omit # if it is a` **variable or register**.

- **JMP** always unconditionally transfers execution to the target address.

Program: call.spin2

```
'call.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov     number1, #0

  call    #SetAndPrint

  jmp     #endprog

  SetAndPrint
    mov     number1, #5
    add     number1, #"0"
    wypin   number1, #TRANSMISSION_PIN

    ret

  endprog
    jmp  #$

  number1 res 1
```

This program demonstrates the use of `call` to invoke a subroutine. Execution starts with mov `number1, #0`, then call `#SetAndPrint` jumps to the `SetAndPrint` subroutine while saving the return address. Inside the subroutine, `number1` is set to 5, converted to its ASCII code by adding `"0"`, and sent out via `wypin` on `TRANSMISSION_PIN`. After the subroutine finishes, `ret` returns execution to the instruction immediately following the `call`, which is jmp `#endprog`, terminating the program.

Global labels and Local labels

A label is a symbolic name representing a specific location in your code, allowing you to jump or call that location without remembering its numeric address. Labels improve code readability by giving descriptive names to program locations, making it easier to follow program flow and understand program structure. They also help organize code and maintain clarity in more complex programs.

Global labels define entry points for subroutines or main program sections. They can be targeted by instructions like `JMP` or `CALL`, allowing execution to jump to a subroutine or a specific part of the program. When a `CALL` is used, execution jumps to the subroutine while saving the return address, and `RET` returns control to the instruction immediately following the original `CALL`. Global labels provide a way to structure your program and reuse code efficiently.

Local labels, typically prefixed with a dot (.), are scoped only to the nearest preceding global label. They are commonly used for loops or internal branching within subroutines and can be safely reused in different subroutines, preventing naming conflicts. Local labels enable branching or looping within subroutines, supporting modular and organized code while keeping program flow clear and maintainable.

Syntax

```
global_label = Program-wide label.

.local_label = Section-only label (belongs to the nearest
global label above it).
```

Program: label.spin2

```
'label.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)
  DELAY             = 180_000_000 / 6

DAT
  org 0
  asmclk
```

```
  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  call #PrintEven
  call #PrintOdd
  jmp #endprog

  PrintEven
    mov number1, ##0
    .loop
      mov ascii_out, number1
      add ascii_out, #"0"
      wypin ascii_out, #TRANSMISSION_PIN
      call #flush
      add number1, ##2
      cmp number1, ##10 wc
      if_ge jmp #.endLoop
      jmp #.loop
    .endLoop
    ret

  PrintOdd
    mov number1, ##1
    .loop
      mov ascii_out, number1
      add ascii_out, #"0"
      wypin ascii_out, #TRANSMISSION_PIN
      call #flush
      add number1, ##2
      cmp number1, ##11 wcz
      if_nc jmp #.endLoop
      jmp #.loop
    .endLoop
    ret

flush
    mov count, ##DELAY
.delay
    sub count, #1 wz
    if_nz jmp #.delay
    ret

  endprog
    jmp  #$

  number1   res 1
  ascii_out res 1
  count     res 1
```

We print, in this program, even and odd numbers sequentially as ASCII characters to the `TRANSMISSION_PIN`, with a short delay between each output. First, the

`PrintEven` subroutine runs, initializing `number1` to 0 and incrementing by 2 each loop iteration until it reaches 10, producing the sequence 0, 2, 4, 6, 8. Next, the `PrintOdd` subroutine runs, starting `number1` at 1 and incrementing by 2 until it reaches 11, producing 1, 3, 5, 7, 9. Each value is converted to ASCII by adding `"0"` before sending it via `WYPIN`. The `flush` subroutine introduces a brief delay between transmissions. After both subroutines finish, the program ends, producing the full output sequence: 0 2 4 6 8 1 3 5 7 9.

Exercise

First, use `JMP` to skip a line that sets `number1` to 2 and instead set `number1` to 5, then print it. Next, create a subroutine that sets `number1` to 7 and prints it, using `CALL` and `RET` to return to the main program. Within a subroutine, practice a local label loop that runs three times, incrementing `number1` by 1 and printing each value. Finally, create multiple subroutines: one that prints even numbers from 0 to 4 and another that prints odd numbers from 1 to 5 sequentially, calling each subroutine from the main program to reinforce structured, modular coding.

Chapter 9: Timing & Waits

Timing control in P2ASM is essential for synchronizing tasks, generating delays, and coordinating I/O operations. Programs can pause execution for a specific number of clock cycles or wait until an event occurs. Common techniques include using instructions for very short delays, creating subtraction loops with a counter register for longer or variable delays, and employing hardware timers or counters when precise or extended timing is required. A simple delay can be implemented by loading a register with a count value and decrementing it until it reaches zero, with the `CMP` instruction and `wcz` or `wc` flags used to test when the counter has finished.

Delays are typically measured in clock cycles, which can be converted to seconds or microseconds using the system clock frequency (`_CLKFREQ`). Timing control ensures proper synchronization for serial or I/O communication, provides consistent spacing between repeated signals such as LED blinking, and helps prevent race conditions in multitasking or sensor-reading loops. It is also essential for generating precise pulse widths, timeouts, or other time-sensitive control signals. By accurately managing delays, programs can maintain reliable operation and predictable behavior in hardware interactions.

WAITX instruction

The `WAITX` instruction pauses the executing COG for a precise number of system clock cycles. `WAITX` provides exact timing without manually decrementing counters, making it simpler and more reliable. It is ideal for tasks that require precise timing, such as serial communication, pulse generation, sensor reading etc.

`WAITX` accepts a delay value specifying the number of system clock cycles to pause execution. Once the delay completes, the COG immediately resumes execution. The delay value can range from 0 to 4,294,967,295, allowing both very short and very long pauses. Short delays from 0 to 511 cycles can be

specified using the `#value` form, while longer delays require the `##value` form, which uses an extra instruction word to encode a full 32-bit number.

With `_CLKFREQ = 180,000,000`, each system clock cycle lasts approximately 5.56 nanoseconds. Using the full 32-bit range of `WAITX`, the maximum delay of 4,294,967,295 cycles corresponds to roughly 23.86 seconds, demonstrating the instruction's ability to provide both very short and very long precise delays.

Syntax

```
Waitx {# or ##}value
```

- `#value = waits 0–511 clock cycles`
- `##value = waits 0–4,294,967,295 clock cycles`

Program: waitx.spin2

```
'waitx.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)
  DELAY             = 180_000_000

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  waitx ##DELAY

  mov character, #"A"
  wypin character, #TRANSMISSION_PIN

  endprog
    jmp  #$

  character res 1
```

This program demonstrates a precise delay using `waitx`. Execution begins with waitx `##DELAY`, which halts the COG for the specified number of system clock cycles (in this case, corresponding to a 1-second delay). After the pause, mov `character, #"A"` loads the ASCII code for the letter **A** into the variable `character`, and `wypin` transmits it via the `TRANSMISSION_PIN`. The program then ends with `endprog`.

Instruction: NOP (No Operation)

`NOP` stands for "No Operation." It executes without performing any functional task but still consumes one system clock cycle, allowing execution to move to the next instruction. Each NOP cycle is deterministic, with a duration of 1 / `_CLKFREQ` (for example, at 180 MHz, one NOP takes approximately 5.56 nanoseconds). Multiple NOPs can be executed in sequence to create slightly longer delays, though using many NOPs for extended timing becomes impractical—longer delays are better handled with `WAITX` or loop counters.

The NOP instruction is useful for introducing precise, single-cycle timing gaps between instructions and synchronizing events in timing-sensitive hardware tasks, such as bit-banging or pulse generation. Its simplicity and predictability make it ideal for minimal delays without adding overhead beyond the one clock cycle it consumes.

Syntax

`nop`

- `Single-cycle delay`

Program: nop.spin2

```
'nop.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)
```

```
DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  nop

  mov character, #"A"
  wypin character, #TRANSMISSION_PIN

  endprog
    jmp  #$

  character res 1
```

The program starts with a single-cycle `NOP`, which performs no operation but consumes one system clock cycle. This creates an imperceptibly short delay before the next instruction. Immediately after, `MOV character, #"A"` loads the ASCII code for **A** into the `character` variable, and `WYPIN` transmits it through the `TRANSMISSION_PIN`. The program then ends with `endprog`. On a terminal, the letter **A** appears instantly, but the `NOP` demonstrates how a one-cycle delay can be inserted for precise timing control in a sequence of instructions.

GETCT/WAITCT1/ADDCT1 instruction

The `GETCT` instruction reads the current value of a counter/timer into a register, allowing your program to track the timer's progress without blocking execution. It is commonly used in combination with `ADDCT1` and `WAITCT1` to implement precise, non-blocking or scheduled timing operations.

Syntax

```
getct target_register, CT1
```

- target_register = register that receives the counter value
- CT1 = counter/timer 1
- Copies the current counter value into the target register.
- Does not block or pause execution.
- The counter continues running.
- Commonly used with WAITX or time comparisons.
- The value is a 32-bit system counter.

The ADDCT1 instruction adds a specified value to counter/timer 1 (CT1), allowing you to schedule future events relative to the current counter value. It is typically used with GETCT and WAITCT1 to create precise delays or periodic timing without manually recalculating absolute times.

Syntax

```
addct1 value
```

- value = the number of clock cycles to add to counter/timer 1 (CT1).
- Useful for scheduling events in the future based on the current counter.

The WAITCT1 instruction in Propeller 2 assembly pauses program execution until counter/timer 1 (CT1) reaches a specified value. As a blocking instruction, it stops the program until the counter event occurs, providing precise timing for periodic events and hardware-controlled delays. WAITCT1 is efficient because the program is automatically blocked without manually polling flags or using software loops, and it simplifies code compared to using POLLCT1, making programs easier to read, maintain, and integrate.

Syntax

waitct1

- The instruction waits until the value in the CT1 compare register is reached.
- It automatically blocks the COG until the timeout, no loop or flag polling is needed.

Program: waitct1.spin2

```
'waitct1.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov     counter, #0
  mov     delay, ##_CLKFREQ
  mov     time_duration, #5

  sleep
    getct   timeout
    .loop
      addct1  timeout, delay
      waitct1
      add     counter, #1
      cmp     counter, time_duration wz
      if_nz jmp     #.loop
    .end

  wypin   #"#", #TRANSMISSION_PIN

  endprog
    jmp  #$

  timeout       res 1
  counter       res 1
  delay         res 1
```

```
  time_duration res 1
```

We initialize a seconds counter to 0, set a delay to `_CLKFREQ` (one second worth of clock cycles), and set `time_duration` to 5. The program reads the current CT value into `timeout`, then repeatedly schedules the next timeout by adding `delay` to `timeout` and executes `waitct1` to block the COG until that scheduled moment. After each wait, the program increments `counter` and repeats until `counter` equals `time_duration`. When the loop finishes (after five precise one-second waits, ~5 seconds total), the program sends a single # on `TRANSMISSION_PIN`. The waits are hardware-timed and block the COG, ensuring exact one-second increments.

POLLCT1 instruction (Non-blocking Timer Poll)

The `POLLCT1` instruction checks whether Counter Timer 1 (CT1) has reached its target value without stopping program execution. Unlike `WAITCT1`, which blocks until the timer expires, `POLLCT1` runs immediately and updates the Carry (C) flag to indicate whether the timer has passed its target. This non-blocking behavior allows the program to make conditional decisions based on timer status, enabling responsive and flexible control.

`POLLCT1` is useful for non-blocking delays, letting a program perform other tasks while monitoring a timer. It provides fine-grained control over timing loops and is especially valuable for multitasking, responsive event handling, or implementing timeouts without halting program flow.

Syntax

```
pollct1
```

- Typically used after scheduling a target with ADDCT1.
- Often combined with conditional instructions like IF_C or IF_NC to loop until the timer expires.

Program: pollct1.spin2

```
'pollct1.spin2
```

```
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov counter, #0
  mov delay, ##_CLKFREQ
  mov time_duration, #5

  sleep
    getct timeout
   .loop
     addct1 timeout, delay
   .wait
     pollct1
     if_c jmp #.wait
     add counter, #1
     cmp counter, time_duration wz
     if_nz jmp #.loop
   .end

   wypin #"#", #TRANSMISSION_PIN

  endprog
    jmp  #$

  timeout       res 1
  counter       res 1
  delay         res 1
  time_duration res 1
```

We measure a five-second interval using non-blocking timer checks. The program begins by initializing a counter to `0` and a one-second delay value, then reads the current CT value into `timeout`. In the main loop, it schedules a future timeout using `ADDCT1`, based on the current CT value. The inner `.wait` loop repeatedly polls the timer with `POLLCT1` until the scheduled moment is reached, **without blocking the COG**, and sets the C flag when the timeout occurs. After each timeout, the counter increments. This process repeats until the counter reaches five, corresponding to five precise one-second intervals. When the loop

finishes, the program outputs a # character on the transmission pin, signaling that the full timed interval has completed.

WAITX vs NOP

Both **NOP** and **WAITX** can be used to introduce delays, but they differ significantly in efficiency and practicality. A **NOP** instruction consumes exactly one clock cycle while performing no useful work, making it suitable only for extremely short, single-cycle timing adjustments. In contrast, **WAITX** can stall for an arbitrary number of cycles, allowing multi-cycle delays to be handled cleanly and efficiently without wasting instruction space. Choosing the appropriate instruction for the situation is key to writing fast, memory-efficient, and maintainable code.

`NOP` is useful for very small timing adjustments, such as single-cycle tweaks or padding within tightly tuned instruction sequences. For longer or more precise delays, `WAITX` is the better choice, providing cycle-accurate pauses without bloating the program with repeated instructions. Long sequences of NOPs are impractical for meaningful delays, as even millisecond-scale pauses can require hundreds of thousands of instructions, wasting time, space, and stressing the COG's LUT cache and hub execution. They also make code harder to read and obscure intent. In contrast, `WAITX` allows flexible, variable-length, and precisely controlled delays, keeping code efficient, readable, and easy to maintain.

Program: delay.spin2

```
'delay.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk
```

```
wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
wypin  #1, #TRANSMISSION_PIN
dirh   #TRANSMISSION_PIN

nop
nop
nop

mov character, #"A"
wypin character, #TRANSMISSION_PIN

waitx ##_CLKFREQ

mov character, #"B"
wypin character, #TRANSMISSION_PIN

endprog
  jmp  #$

character res 1
```

Let’s show two delay methods and their visible effect: the three consecutive NOP instructions introduce a tiny, three-cycle pause before transmitting 'A', which is effectively instantaneous to a human observer (~50 ns at 180 MHz). The program then places the ASCII character "A" into character and outputs it on the transmission pin. Next, waitx ##_CLKFREQ performs a multi-cycle stall equal to one second (≈_CLKFREQ cycles at 180 MHz), starting from the current COG clock, after which the program stores "B" into character and transmits it, producing "B" exactly one second after the WAITX instruction begins. The three NOPs simply provide a minimal timing offset, and the line character res 1 reserves the single-byte storage used for both transmissions.

Exercise

Practice timing and delays in P2ASM with this exercise. Start by inserting three consecutive NOP instructions, then set a character to "X" and print it. Next, use WAITX #50 for a short pause before setting a character to "Y" and printing it, followed by a longer delay using WAITX ##_CLKFREQ*2 to pause for two seconds, then set a character to "Z" and print it. For a more advanced exercise, create a 3-second loop with GETCT, ADDCT1, and WAITCT1 that increments a counter each second before printing "#". Finally, use POLLCT1 in a non-blocking loop to check CT1 every second for four seconds while

incrementing a counter, then print "@", reinforcing both blocking and non-blocking timing techniques.

Chapter 10: Hardware Functions

The Propeller 2 microcontroller features a built-in hardware math engine called **CORDIC** (COordinate Rotation DIgital Computer), which performs complex arithmetic operations far faster than software routines. Using a **shift-and-add algorithm**, CORDIC efficiently computes trigonometric, vector, and other arithmetic functions, reducing code size by replacing long instruction sequences with a single hardware operation. Its deterministic timing makes it ideal for real-time applications, and it supports high-precision fractional math without requiring floating-point hardware, enabling efficient, compact, and predictable calculations on the Propeller 2.

QMUL — Unsigned Multiplication Using CORDIC

The `QMUL` instruction leverages the Propeller 2's CORDIC hardware to perform **unsigned multiplication** directly in hardware using a **shift-and-add approach**. The hardware shifts one operand and adds it conditionally to an accumulating sum, efficiently computing the product in just a few cycles. This method executes far faster than equivalent software routines, conserves clock cycles and program memory, and provides precise, deterministic results for unsigned values. `QMUL` also integrates seamlessly with other CORDIC operations, such as division or square root, enabling flexible and efficient arithmetic workflows on the Propeller 2.

To use QMUL, first load the two unsigned values into normal registers, for example: `mov number1, ##3` and `mov number2, ##2`. Then execute `qmul source1, source2` to instruct the CORDIC hardware to perform the multiplication; note that the source registers themselves are not modified. Finally, retrieve the computed product from the CORDIC output register QX using `getqx destination`, which stores the result in a general-purpose

Syntax

qmul source1, source2

- source1 = register or literal value
- source2 = register or literal value

getqx destination

- destination = This is the **register, variable, or memory location** where the value from the queue will be stored.
- Use getqx destination to retrieve the result.

Program: qmul.spin2

```
'qmul.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov     number1, #3
  mov     number2, #2

  qmul    number1, number2
  getqx   number1

  add     number1, #"0"
  wypin   number1, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1 res 1
```

```
number2 res 1
```

Here we multiply 3 by 2 using the Propeller 2's qmul instruction. First, the numbers are loaded into number1 and number2. Executing qmul number1, number2 multiplies the two 32-bit signed values, storing the high 32 bits of the 64-bit product in the qx register, which is then retrieved into number1 using getqx. In this specific case, the numbers are small, so qx contains the exact product. The program converts the numeric result 6 to its ASCII equivalent by adding "0"—note that this approach works only for single-digit results—and transmits it via the transmission pin, producing the character "6" as output. The variables number1 and number2 each reserve a single byte of storage for this operation. This calculation and transmission occur immediately and non-blocking, unlike waitx or waitct1 delays.

QDIV — Unsigned Division Using CORDIC

The `QDIV` instruction uses the Propeller 2's CORDIC hardware to perform unsigned integer division efficiently. It computes source1 `÷ source2` far faster than a software routine, eliminating the need for loops or long instruction sequences. Division is achieved using a shift-and-subtract approach, where the numerator is shifted and compared to the denominator in successive steps, with the quotient built incrementally. This hardware-based method ensures precise, deterministic results while saving program memory and clock cycles.

To use `QDIV`, load the numerator and denominator into registers (for example, `mov number1, ##6` for the numerator and `mov number2, ##3` for the denominator), then execute `qdiv source1, source2`. The CORDIC engine performs the division, and the quotient is retrieved using `getqx dest`; the remainder can be fetched with `getqy dest` if needed. `QDIV` can also be combined with `QFRAC` for precise fractional results, providing flexibility for advanced arithmetic operations on the Propeller 2 while keeping code simple and efficient.

Syntax

```
qdiv source1, source2
getqx destination
```

getqy destination

- source1 = register or literal value, used as the numerator
- source2 = register or literal value, used as the denominator
- destination = This is the register, variable, or memory location where the value from the queue will be stored.
- Use getqx destination to retrieve the quotient.
- Use getqy destination to retrieve the remainder (optional).

Program: qdiv.spin2

```
'qdiv.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov     number1, #6
  mov     number2, #3

  qdiv    number1, number2
  getqx   number1

  add     number1, #"0"
  wypin   number1, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1 res 1
  number2 res 1
```

The `qdiv` instruction used here divides 6 by 3. The numerator and denominator are first loaded into `number1` and `number2`. Executing `qdiv number1, number2` directs the CORDIC hardware to perform the division, producing a 32-bit quotient stored in the qx register, which is then retrieved into `number1` using `getqx`. In this case, the numbers divide evenly, so the quotient is exact. The program converts the numeric result `2` to its ASCII equivalent by adding `"0"`—note that this approach works correctly only for single-digit results—and transmits it via the transmission pin, producing the character `"2"` as output. The variables `number1` and `number2` each reserve a single byte of storage for this operation. This calculation and transmission occur immediately and non-blocking, unlike waitx or waitct1 delays.

Exercise

With this exercise, start by using `QMUL` to multiply 4 × 5 and print the result. Next, perform unsigned division with `QDIV` by dividing 8 ÷ 2 and printing the quotient.

Chapter 11: Hub Memory Data Access

Hub RAM is the Propeller 2's main shared memory, used to store larger programs and data accessible by all cogs. P2ASM provides instructions for reading and writing 8-bit, 16-bit, or 32-bit values, allowing you to manipulate data safely and efficiently. These instructions ensure precise memory access without accidentally modifying nearby values, making them especially useful for working with buffers, arrays, or tables.

Hub memory access also has instructions that are used to handle hub RAM safely and precisely. They prevent errors that can happen if the wrong data size is read or written, which could corrupt nearby values. These instructions are important when working with structured programs that use arrays, buffers, or other shared data.

Variables in P2ASM

Variables are placeholders in memory that store data your program can read or change. Each variable has three important parts: a **name**, which is the label you use in your code to access it; a **type**, which defines the size and format of the data (bit, byte, word, or long); and a **value or initialization**, which is an optional starting value stored in the variable.

Memory alignment is how data is arranged in memory for efficient access. Single bytes can be stored anywhere, but words (2 bytes) should start at even addresses, and longs (4 bytes) must start at addresses divisible by 4. Following these rules helps the CPU read and write data correctly and efficiently. When accessing a specific location, using `##@label` ensures the address is handled correctly as a full 32-bit value.

Example

```
character byte "A", 0, 0, 0  ' 4 bytes: ASCII "A" with 3 nulls
                             ‘ to align to 4-byte boundary
```

```
number1   word 0, 0          ' 2 words = 4 bytes (may require
                             ‘ padding to align)
number2   long 0             ' 4-byte aligned long
```

Program: variables.spin2

```
'variables.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  wypin character, #TRANSMISSION_PIN
  waitx #511

  mov number1, #1
  add number1, #"0"
  wypin number1, #TRANSMISSION_PIN
  waitx #511

  mov number2, #2
  add number2, #"0"
  wypin number2, #TRANSMISSION_PIN

  endprog
    jmp  #$

  character byte "A", 0, 0, 0
  number1   word 0, 0
  number2   long 0
```

We send three characters—`"A"`, `"1"`, and `"2"`—out through the transmission pin using the `wypin` instruction. First, the program transmits the `character` variable, which holds `"A"` (not `"0"`). It then loads the numbers 1 and 2 into `number1` (a 2-byte word) and `number2` (a 4-byte long), converts them into their ASCII characters by adding `"0"`, and sends them as well. Short

delays (`waitx #511`) separate each transmission; these delays are very short and produce small but noticeable spacing. The variable definitions at the end use byte (`character`), word (`number1`), and long (`number2`) types, demonstrating different sizes, ensuring proper alignment, and allowing reliable memory access.

RDBYTE instruction

`RDBYTE` reads a single byte (8 bits) from hub RAM and copies it into a register so the CPU can manipulate it. Since the CPU cannot access hub memory directly, this instruction is essential for working with data stored outside the cog's local memory. `RDBYTE` allows programs to safely inspect, modify, or use values such as characters in a string, entries in a byte buffer, or small numerical variables. By transferring a single byte at a time into a register, `RDBYTE` ensures precise memory access without affecting adjacent data, making it a fundamental tool for handling hub-stored data in P2ASM programs.

Syntax

```
rdbyte destination, address
```

- **destination** — The cog register where the byte will be stored.
- **address** — The hub memory address of the byte to read.
- Use @label to get the address of a label, and ##@label for the full 32-bit address.

Program: rdbyte.spin2

```
'rdbyte.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
```

```
org 0
asmclk

wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
wypin  #1, #TRANSMISSION_PIN
dirh   #TRANSMISSION_PIN

rdbyte data_byte, ##@character
wypin data_byte, #TRANSMISSION_PIN

endprog
  jmp  #$

character byte "A", 0, 0, 0
data_byte      res 1
```

We read a single byte from the `character` variable in hub RAM into the COG register `data_byte` using `RDBYTE`. The value stored is the character `"A"`. The program then immediately transmits that byte via the transmission pin using `wypin`. The variable `character` is defined as a 4-byte sequence (`"A", 0, 0, 0`), but only the first byte is read and sent. `data_byte` is a one-byte COG register space reserved to hold the read value. This transfer from hub RAM to the COG register occurs non-blocking, and the transmission happens immediately according to the UART timing. As a result, the character `"A"` is transmitted.

WRBYTE instruction

`WRBYTE` writes a single byte (8 bits) from a register into hub RAM, allowing the CPU to update shared memory safely. Since the CPU cannot modify hub memory directly, `WRBYTE` is used to store processed values back into hub RAM after calculations or modifications have been made in a register. It is essential for updating byte-sized data such as characters, flags, or entries in buffers and arrays.

`WRBYTE` works hand-in-hand with `RDBYTE`: first, a byte is read from hub memory into a register, then processed, and finally written back using `WRBYTE`. This makes it the fundamental instruction for managing byte-level data in shared memory. By enabling precise, controlled updates, `WRBYTE` ensures that

individual bytes can be modified without affecting neighboring memory, which is crucial when working with strings, buffers, or other shared data structures in P2ASM programs.

Syntax

```
wrbyte source, address
```

- **source** — register holding the byte to write.
- **address** — Hub memory address where the byte will be stored.
- Use @label for labels and ##@label for full 32-bit addresses.

Program: wrbyte.spin2

```
'wrbyte.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov data_byte, #"B"
  wrbyte data_byte, ##@character

  rdbyte data_byte, ##@character
  wypin  data_byte, #TRANSMISSION_PIN

  endprog
    jmp  #$

  character byte "A", 0, 0, 0
  data_byte      res 1
```

The character "B" is written into the hub-memory variable character using wrbyte. First, the program loads "B" into the COG register data_byte (not “temp”), then writes that byte to the address of character, replacing the original "A". Next, the program reads the updated value back from hub RAM into data_byte using rdbyte and immediately transmits it via the transmission pin using wypin. The variable character is defined as a 4-byte sequence ("A", 0, 0, 0), but only the first byte is written and read. data_byte is a one-byte COG register space reserved for this operation. This hub-to-COG transfer and transmission occur non-blocking, and as a result, the transmitted character is "B".

RDBYTE looping

RDBYTE reads a single byte (8 bits) from hub RAM into a register, and although it operates on one byte at a time, it can be used within a loop to access multiple bytes in sequence. This makes it ideal for processing strings, buffers, arrays, or other collections of byte-sized data stored in hub memory. By combining RDBYTE with a pointer and a loop, your program can efficiently step through each memory address, reading bytes one after another, often stopping when a terminator, such as 0, is encountered.

When paired with WRBYTE, this approach allows not only reading but also updating or transmitting data efficiently, making sequential hub memory access both flexible and powerful. Hub memory is organized one byte at a time, but using a pointer or label like ##@label ensures that each address is treated correctly. This combination of RDBYTE, pointers, and loops provides a robust method for handling strings, buffers, and other sequential byte-sized structures in Propeller 2 programs.

Syntax

```
rdbyte destination, address
```

- **destination =** register where the byte will be stored.
- **Address =** Hub memory address of the byte to read.
- Use @label or ##@label for full 32-bit addresses.

Program: multi_rdbyte.spin2

```
'multi_rdbyte.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)
  DELAY             = 180_000_000

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov char_ptr, ##@character

  read_loop
    rdbyte current_char, char_ptr
    cmp    current_char, #0 wz
    if_z   jmp #done
    wypin  current_char, #TRANSMISSION_PIN
    waitx ##DELAY
    add    char_ptr, #1
    jmp    #read_loop
    done

  endprog
    jmp  #$

  character     byte "A", "B", "C", "D", 0
  char_count    res 1
  current_char  res 1
  char_ptr      res 1
```

This program reads and transmits a string of characters stored in hub memory. First, `char_ptr` is set to point to the start of the `character` array, which contains `"A"`, `"B"`, `"C"`, `"D"`, followed by a `0` terminator. Inside the loop, `rdbyte` reads one byte at a time from hub RAM into the COG register `current_char`. If the byte is `0`, the loop ends. Otherwise, the program immediately transmits the byte via the transmission pin using `wypin`, waits briefly (`waitx ##DELAY`), and then increments `char_ptr` to point to the next byte. This process repeats until the `0` terminator is reached, transmitting the

entire string in order. All hub-to-COG transfers and transmissions occur non-blocking. The `character` array is stored as consecutive bytes in hub RAM, while `char_ptr` and `current_char` are single-byte COG registers reserved for the pointer and read value, respectively.

RDBYTE for Strings

A string in Propeller 2 assembly is a sequence of characters stored in memory, with each character occupying 1 byte. Strings are typically null-terminated, ending with a 0, so the program can determine where the string ends. `RDBYTE` can be used to read each character from hub RAM into a register, and by combining it with a pointer and a loop, your program can process the string one character at a time. The loop continues reading bytes until it encounters the null terminator, ensuring that only valid characters are processed. This method makes `RDBYTE` an essential tool for handling strings efficiently, allowing programs to inspect, modify, or output characters sequentially while working safely with hub memory.

Syntax

```
rdbyte destination, address
```

- **destination =** register to store the byte.
- **address =** Hub memory address of the byte.
- Use ##@label for full 32-bit address.

Program: rdbyte_strings.spin2

```
'rdbyte_strings.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)
  DELAY             = 180_000_000 / 100
```

```
DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov char_ptr, ##@character

  read_loop
    rdbyte current_char, char_ptr
    cmp    current_char, #0 wz
    if_z   jmp #done
    wypin  current_char, #TRANSMISSION_PIN
    waitx  ##DELAY
    add    char_ptr, #1
    jmp    #read_loop
    done

  endprog
    jmp  #$

character      byte "Hello assembly!", 0
char_count     res 1
current_char   res 1
char_ptr       res 1
```

The program begins by loading the hub address of the string `character` into the pointer `char_ptr`. It then reads one byte at a time from hub RAM into the COG register `current_char` using `RDBYTE`. Each byte is checked against the null terminator (`0`); if it matches, the loop ends. Otherwise, the byte is immediately transmitted via the transmission pin using `wypin`, and the program waits briefly (`waitx ##DELAY`) before incrementing `char_ptr` to point to the next byte. This process repeats, reading and transmitting each character in order until the null terminator is reached. The `character` array is stored consecutively in hub RAM, while `char_ptr` and `current_char` are single-byte COG registers reserved for the pointer and read value, respectively. As a result, the string `"Hello assembly!"` is transmitted character by character. All hub-to-COG transfers and transmissions occur non-blocking, with precise timing dictated by the `waitx` delay.

RDWORD instruction

`RDWORD` reads a 16-bit word (2 bytes) from hub RAM into a register, allowing the CPU to work with the value safely since it cannot access hub memory directly. When using `RDWORD`, the hub memory address must be **word-aligned**, pointing to the correct 2-byte boundary. You can reference the address with `@label` and use ## to ensure full 32-bit encoding.

Once the 16-bit word is loaded into a register, it can be processed, transmitted, or written back to hub memory using `WRWORD`. `RDWORD` is especially useful for reading numerical data stored in hub RAM, such as word-sized variables in buffers, arrays, or other data structures. By providing safe, precise access to 16-bit values, `RDWORD` simplifies handling larger or structured data in Propeller 2 programs.

Syntax

```
rdword destination, address
```

- **destination** = Cog register where the 16-bit word will be stored.
- **Address** = Hub memory address of the word.
- Use `##@label` to reference a label's hub address.

Program: rdword.spin2

```
'rdword.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN
```

```
rdword number_value, ##@number1
add number_value, #"0"
wypin  number_value, #TRANSMISSION_PIN

endprog
  jmp  #$

number1       word 6, 0
number_value  res 1
```

Our program reads a 16-bit word from the hub-memory variable `number1` into the COG register `number_value` using `rdword`. In this case, `number1` is defined as a 16-bit word containing the value `6`. The program then converts the numeric value to its ASCII equivalent by adding `"0"` and immediately transmits it via the transmission pin using `wypin`. `number_value` is a one-byte COG register reserved to hold the read value. The `rdword` instruction transfers the 16-bit word from hub RAM to the COG register non-blocking, and the transmission occurs immediately. As a result, the character `"6"` is output.

WRWORD instruction

`WRWORD` writes a 16-bit word (2 bytes) from a register into hub RAM, allowing the CPU to safely update memory since it cannot modify hub RAM directly. When using `WRWORD`, the hub memory address must be **word-aligned**, meaning it should be divisible by 2. Use `@label` to reference the hub address and ## to ensure full 32-bit encoding.

`WRWORD` is commonly paired with `RDWORD` to read, process, and write numerical data, buffers, or other word-sized values in hub memory. By enabling safe, precise updates of 16-bit variables and structured data, `WRWORD` simplifies working with larger programs and shared memory in Propeller 2 assembly.

Syntax

```
wrword source, address
```

- **source** = register containing the 16-bit value to store.
- **address =** Hub memory address where the word will be written.

- Use `##@label` for full 32-bit hub addresses.

Program: wrword.spin2

```
'wrword.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov     number_value, ##7
  wrword  number_value, ##@value

  rdword  number_value, ##@value
  add number_value, #"0"
  wypin   number_value, #TRANSMISSION_PIN

  endprog
    jmp  #$

  value         word 0
  number_value  res 1
```

We are writing a 16-bit number into the hub-memory variable `value` using `wrword`. First, the number `7` is loaded into the COG register `number_value` and then written to the address of `value` in hub RAM. The program then reads the updated 16-bit word back from hub RAM into `number_value` using `rdword`, converts it to its ASCII equivalent by adding `"0"`, and immediately transmits it via the transmission pin using `wypin`. The variable `value` is defined as a 16-bit word in hub RAM, and `number_value` is a one-byte COG register reserved to hold the value. The hub-to-COG transfers and transmission occur non-blocking. As a result, the character `"7"` is output.

RDLONG instruction

`RDLONG` reads a 32-bit long value (4 bytes) from hub RAM into a register, allowing the CPU to safely process data it cannot access directly in hub memory. The hub address must be **long-aligned**, meaning divisible by 4, and can be referenced using `@label` with ## for full 32-bit encoding. `RDLONG` is part of the basic hub memory transfer instructions, alongside `RDBYTE` and `RDWORD`.

This instruction is ideal for accessing 32-bit numerical data stored in buffers, arrays, tables, or other long-sized variables. Once transferred into a register, the program can safely manipulate, process, or transmit the value without directly touching hub RAM. `RDLONG` is often paired with `WRLONG` to enable efficient reading and writing of 32-bit values, making it a key tool for handling larger or structured data in Propeller 2 programs.

Syntax

```
rdlong destination, address
```

- **destination =** register to store the 32-bit value.
- **address =** Hub memory address of the long.
- Use ##@label for full 32-bit hub addresses.

Program: rdlong.spin2

```
'rdlong.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  rdlong  number_value, ##@number1
  add number_value, #"0"
```

```
wypin   number_value, #TRANSMISSION_PIN

endprog
  jmp  #$

number1       long 3
number_value  res 1
```

Our code reads a 32-bit value from the hub-memory variable number1 into the COG register number_value using rdlong. In this case, number1 is defined as a 32-bit long containing the value 3. The program then converts the numeric value to its ASCII equivalent by adding "0" and immediately transmits it via the transmission pin using wypin. The number_value register is a one-byte COG register reserved to hold the read value. The transfer from hub RAM to the COG register is non-blocking, and the transmission occurs immediately. As a result, the character "3" is output.

WRLONG instruction

`WRLONG` writes a 32-bit long value (4 bytes) from a register into hub RAM, allowing the CPU to safely store processed data since it cannot modify hub memory directly. It is part of the basic hub memory transfer instructions, along with `WRBYTE` and `WRWORD`, and is commonly used to update long-sized variables, buffers, or numerical data.

When using `WRLONG`, the hub memory address must be **long-aligned**, meaning divisible by 4. Use `@label` to reference the location and ## to ensure the assembler encodes the full 32-bit address. Following these rules guarantees that 32-bit values are stored correctly and can be read back safely with `RDLONG`. `WRLONG` is ideal for saving large numerical variables, updating buffers, storing counters, configuration values, lookup tables, or any other 32-bit data that needs to persist in shared memory for later use.

Syntax

```
wrlong source, address
```

- **source =** register containing the 32-bit value to store.
- **address =** Hub memory address to write the value.

- Use ##@label for full 32-bit addresses.

Program: wrlong.spin2

```
'wrlong.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov     number_value, #5
  wrlong  number_value, ##@number1

  rdlong  number_value, ##@number1
  add number_value, #"0"
  wypin   number_value, #TRANSMISSION_PIN

  endprog
    jmp  #$

  number1       long 0
  number_value  res 1
```

A 32-bit value is assigned into the hub-memory variable `number1` using `wrlong`. First, the number `5` is loaded into the COG register `number_value` and then written to the address of `number1` in hub RAM. The program then reads the 32-bit value back from hub RAM into `number_value` using `rdlong`, converts it to its ASCII equivalent by adding `"0"`, and immediately transmits it via the transmission pin using `wypin`. The variable `number1` is defined as a 32-bit long in hub RAM, and `number_value` is a one-byte COG register reserved to hold the value. Both the hub-to-COG transfer and transmission occur non-blocking. As a result, the character `"5"` is transmitted.

Exercise

Practice reading and writing data in hub RAM with this exercise. Start by writing the value `0x55` into the third byte of a buffer using `WRBYTE`. Next, load a 16-bit word from hub memory, add 5 to it, and store it back in the same location with `RDWORD` and `WRWORD`. Then, load a 32-bit long value from hub RAM and transmit it through a smart pin, reinforcing the use of `RDLONG`. Finally, copy a 32-bit value from one long variable to another in hub memory using `RDLONG` and `WRLONG`, giving practice with safe and efficient handling of byte-, word-, and long-sized data in Propeller 2 assembly.

Chapter 12 — How Numbers Work in P2 Assembly

Numbers are stored in binary inside registers or hub memory. Before doing math or printing values, it's important to understand how these numbers are represented. The Propeller 2 works with both unsigned and signed integers, and it can store values in different sizes such as bits, bytes, words, and longs. Arithmetic instructions use these sizes and signed types to determine how the numbers are processed.

Binary numbers work by using each bit to represent a power of two. For example, the 8-bit value 01010101 adds up to 85 in decimal. When dealing with signed values, the Propeller 2 uses two's-complement format to represent negative numbers. This means you invert all the bits of a number and then add 1. Using this method, the value –1 in an 8-bit signed format becomes 11111111.

The Propeller 2 works entirely in binary internally, even though we often write numbers in decimal for convenience. Decimal values are only human-friendly representations, while the actual operations inside the chip manipulate raw binary bits. When you want to display a number—such as sending it over UART or showing it on an LCD—the value must be converted from its binary form into ASCII characters so it can be printed correctly.

The ADD instruction performs basic addition between values stored in registers. It takes two numbers, adds them together, and places the result in the destination register. If the total becomes larger than what the selected data size can hold, an overflow can occur, meaning the result wraps around instead of fitting in the expected range.

The SUB instruction subtracts one value from another and stores the result in the destination register. When working with unsigned values, a negative result cannot be represented, so it wraps around to a large positive number. However, when using signed arithmetic, the same operation correctly produces a negative value using two's-complement format.

The DIV instruction performs integer division between two registers, producing a whole-number result. Because this is integer math, any remainder is

simply discarded, so the result is always truncated. Division is especially useful when converting numbers into individual decimal digits for printing, since repeatedly dividing and taking remainders helps break a value down into separate characters.

The QDIV instruction uses the Propeller 2's CORDIC hardware to perform division much faster than the standard DIV instruction. When the operation completes, the quotient is placed in the QX register and the remainder in the QY register. Because of its speed, QDIV is the preferred method for converting numbers into digits when printing values inside tight loops.

Different data sizes must be correctly aligned in hub memory to ensure they can be read and written properly. Bytes can be stored at any address, but words must be placed at even addresses, and longs must start at addresses divisible by four. If data is not aligned correctly, the Propeller 2 may read or write the wrong values, leading to unexpected behavior.

Printing Numbers Using SUB

Before printing a number, you first store it in a register. Then, you convert the binary value into decimal digits using instructions like SUB, DIV, or QDIV. Finally, each digit is sent as an ASCII character through a SmartPin to display it on a UART, LCD, or other output.

You can print numbers without using division instructions. One simple method is **repeated subtraction**, which counts how many times powers of ten fit into a number. This is useful for small numbers or when avoiding heavier instructions like DIV or QDIV.

Program: print_number_sub.spin2

```
'print_number_sub.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
```

```
wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
wypin  #1, #TRANSMISSION_PIN
dirh   #TRANSMISSION_PIN

mov     number_store, #245
wrlong  number_store, ##@number1
rdlong  number_store, ##@number1
mov     value, number_store

mov     digit, #0
hundreds_loop
  cmp     value, #100   wc
  if_nc  sub     value, #100
  if_nc  add     digit, #1
  if_nc  jmp     #hundreds_loop

add     digit, #"0"
wypin   digit, #TRANSMISSION_PIN
waitx   ##1000

mov     digit, #0
tens_loop
  cmp     value, #10    wc
  if_nc  sub     value, #10
  if_nc  add     digit, #1
  if_nc  jmp     #tens_loop

add     digit, #"0"
wypin   digit, #TRANSMISSION_PIN
waitx   ##1000

mov     digit, value
add     digit, #"0"
wypin   digit, #TRANSMISSION_PIN

endprog
  jmp  #$

number1        long 0
number_store   res 1
value          res 1
digit          res 1
```

This code prints the number 245 digit by digit through the TRANSMISSION_PIN. First, the number 245 is stored in the COG register number_store and written into the hub-memory variable number1 using wrlong. It is then read back into number_store with rdlong, and copied into value for digit extraction. The program computes the hundreds digit by repeatedly subtracting 100 from value and incrementing digit until value is less than 100, then converts digit to its ASCII equivalent by adding "0" and transmits it using wypin, followed by a brief waitx delay. The tens digit is extracted similarly using repeated subtraction of 10, converted to ASCII, and transmitted with another short delay. Finally, the remaining ones digit is converted to ASCII and

transmitted. All digits are sent as ASCII characters, with each step non-blocking except for the short waitx pauses. As a result, the digits "2", "4", and "5" are transmitted sequentially, accurately representing the original number 245.

Printing Numbers Using QDIV

Here is a faster method to extract decimal digits using the QDIV instruction, which performs integer division in hardware. Unlike repeated subtraction, QDIV gives both the quotient and the remainder in a single operation. This makes it much more efficient for breaking a number into digits, especially when printing values in loops or working with large numbers.

Program: print_number_qdiv.spin2

```
'print_number_qdiv.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
```

```
wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
wypin  #1, #TRANSMISSION_PIN
dirh   #TRANSMISSION_PIN

mov number_store, #245
wrlong  number_store, ##@number1
rdlong  number_store, ##@number1
mov     value, number_store

qdiv    value, #100
getqx   digit
add     digit, #"0"
wypin   digit, #TRANSMISSION_PIN

getqy   value
waitx   ##1000

qdiv    value, #10
getqx   digit
add     digit, #"0"
wypin   digit, #TRANSMISSION_PIN

getqy   value
waitx   ##1000

mov     digit, value
add     digit, #"0"
wypin   digit, #TRANSMISSION_PIN

endprog
  jmp  #$

number1        long 0
number_store   res 1
value          res 1
digit          res 1
```

We will print the number 245 digit by digit through the TRANSMISSION_PIN using the qdiv instruction. First, the number 245 is stored in the COG register number_store and written into the hub-memory variable number1 using wrlong. It is then read back into number_store with rdlong and copied into value for digit extraction. The hundreds digit is obtained by dividing value by 100 with qdiv; the quotient is retrieved from qx into digit, converted to its ASCII equivalent by adding "0", and transmitted via wypin. The remainder after the division is retrieved from qy into value for the next step. The tens digit is similarly extracted by dividing value by 10 using qdiv, converting the quotient to ASCII, and transmitting it. Finally, the remaining ones digit in value is converted to ASCII and transmitted. Short waitx delays are used between digits to ensure sequential output. All hub-to-COG transfers and transmissions occur non-blocking. As a result, the digits "2", "4", and "5" are transmitted

sequentially, accurately representing the original number stored in number_store.

Exercises

Practice working with numbers, arithmetic, and memory in Propeller 2 assembly with this exercise. Begin by converting given decimal numbers into 8-bit binary to understand their bit patterns. Next, use `ADD` and `SUB` on both signed and unsigned numbers to observe sums, differences, and overflow behavior. Then, pick a number in a register and use shifts or masks to isolate specific bits, recording their values in binary to see how individual bits can be manipulated. Finally, store bytes, words, and longs at different hub memory addresses and read them back to explore what happens when data is misaligned, reinforcing safe and precise memory access techniques.

Chapter 13 — Cog Control

A **cog** in the Propeller 2 is like a small, separate processor inside the chip that can run its own program at the same time as the others. This lets the Propeller 2 handle many tasks in parallel, such as reading sensors, updating a display, or generating sound—all without slowing each other down. Learning how to control cogs is important because it allows you to start and stop tasks, organize which cog does what, and safely share data through the chip's main memory (hub RAM).

The Propeller 2 chip has **8 cogs (0 through 7)**, and each cog has its own small workspace with registers and a stack, plus the ability to access the chip's main memory, called **hub RAM**. Because each cog can run instructions on its own, the chip can perform many tasks at the same time—like handling input/output, keeping track of time, or communicating with other devices—without one task slowing down the others.

Key features of cog control include the ability to start and stop cogs on the fly and give them access to specific parts of hub memory. You can manage cogs using instructions like COGINIT, COGSTOP, COGID, and COGNEW. Each cog keeps track of its own program with its instruction pointer (program counter) and stack, so it can run independently without interfering with other cogs.

Syntax

```
coginit cogID, ##@routine

cogid  destination_register

cogstop cogID
```

- **cogID =** The ID of the cog (0–7) or #newcog to use the next free cog.
- **routine =** Hub memory address of the routine for the cog to execute.
- **destination_register =** A cog register to store the cog's own ID.

In the Propeller 2, each **cog** is an independent processor core that can run a program concurrently with other cogs. Using `COGINIT`, you can start a cog running a specific routine stored in hub memory, enabling parallel task execution. `COGSTOP` allows you to halt a cog when its work is finished or no longer needed, giving you control over system resources. Meanwhile, `COGID` lets a cog identify itself by returning its ID, which is useful for coordinating tasks across multiple cogs. Together, these instructions provide basic but essential control over cog operation, allowing programs to manage and monitor concurrent execution efficiently.

Program: cog_control.spin2

```
'cog_control.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)
  DELAY             = 180_000_000 / 2

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  start
    call #print_cogid
    coginit #newcog, ##@print_cogid
    waitx ##DELAY
    cogstop #1

  jmp #endprog

  print_cogid
    cogid cog_number
    add cog_number, #"0"
    wypin cog_number, #TRANSMISSION_PIN
    waitx ##DELAY
    ret

  endprog
    jmp  #$

  cog_number res 1
```

The main cog executes the print_cogid routine first and outputs its id (0) to the transmission pin using wypin. Next, a second cog is started with coginit to run the same routine independently, which outputs its own id (1). Both cogs operate concurrently and independently, each using its own cog register space, without interfering with each other. Short waitx ##DELAY pauses ensure the outputs are spaced clearly. After the delay, the main cog stops the second cog using cogstop #1. The serial output will display the characters "0" followed by "1", demonstrating that each cog correctly identifies itself, and illustrating parallel execution, cog initialization, and basic cog control. The variable cog_number is a one-byte cog register used to store each cog's id for ASCII conversion and transmission.

The Propeller 2 allows you to run programs on **multiple cogs simultaneously**, enabling true parallel processing. By starting two or more cogs with `COGINIT`, each cog can execute its own routine independently, such as handling input on one cog while generating output on another. Using `COGID`, each cog can identify itself and perform tasks based on its ID, which helps coordinate shared resources or avoid conflicts. When a cog completes its task, `COGSTOP` can be used to halt it, freeing the core for other operations. Running multiple cogs efficiently allows programs to perform concurrent tasks, improve responsiveness, and fully utilize the processing power of the Propeller 2.

Program: cog_addition.spin2

```
'cog_addition.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)
  DELAY             = 180_000_000 / 2

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN
```

```
main
  coginit #1, ##@addition

  waitx   ##DELAY

  rdword  calculation_result, ##@calculation_result_hub
  add     calculation_result, #"0"
  wypin   calculation_result, #TRANSMISSION_PIN

  jmp #endprog

addition
  mov     calculation_result, #4
  add     calculation_result, #2
  wrword  calculation_result, ##@calculation_result_hub
  cogstop #1
  ret

endprog
  jmp  #$

calculation_result      res 1
calculation_result_hub  word 0
```

The main cog starts a second cog using coginit #1 to execute the addition routine independently. The second cog calculates the sum of 4 + 2, stores the 16-bit result in hub memory (calculation_result_hub) using wrword, and then stops itself with cogstop #1. After waiting long enough with waitx ##DELAY for the second cog to finish, the main cog reads the value from hub memory into the cog register calculation_result using rdword, converts it to its ASCII equivalent by adding "0", and immediately transmits it via the transmission pin using wypin. The variables calculation_result_hub (hub RAM) and calculation_result (cog register) are used to store the calculation result and facilitate hub-to-cog transfer. As a result, the serial output displays the character "6", demonstrating that the addition was performed in a separate cog and the main cog successfully retrieved and transmitted the result.

Using multiple cogs in the Propeller 2 allows a program to divide work across independent cores, improving efficiency and responsiveness. For example, one cog can handle real-time input, such as reading a sensor or keyboard, while another cog simultaneously updates a display or controls LEDs. By starting each cog with COGINIT and using COGID to identify which cog is executing, routines can be tailored to specific tasks without interfering with each other. Once a cog

finishes its assigned work, COGSTOP can safely halt it, freeing resources and preventing unnecessary CPU usage. This approach demonstrates how multi-cog programming can simplify complex tasks and enable parallel, synchronized operations within a single Propeller 2 system.

Program: cog_multicog.spin2

```
'cog_multicog.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)
  DELAY             = 180_000_000 / 2

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov     working_value, #0
  wrlong  working_value, ##@shared_counter

  start
    coginit #1, ##@incrementer
    coginit #2, ##@reporter

  jmp     #endprog


  incrementer
    .loop
      rdlong  working_value, ##@shared_counter
      cmp     working_value, #10 wz
      if_z    jmp #stop_cog

      add     working_value, #1
      wrlong  working_value, ##@shared_counter
      waitx   ##DELAY
    jmp     #.loop


  reporter
    .loop
      rdlong  working_value, ##@shared_counter
      cmp     working_value, #10 wz
      if_z    jmp #stop_cog
```

```
    add     working_value, #"0"
    wypin   working_value, #TRANSMISSION_PIN
    waitx   ##DELAY
  jmp     #.loop

stop_cog
  cogid   working_value
  cogstop working_value

endprog
  jmp  #$

shared_counter   long 0
working_value    res 1
```

Cog 1 (incrementer) repeatedly reads the 32-bit hub-memory variable shared_counter into the cog register working_value, increments it by 1, and writes it back to hub memory using rdlong and wrlong. At the same time, Cog 2 (reporter) reads the same counter from hub memory, converts it to its ASCII equivalent by adding "0", and immediately transmits it via the transmission pin using wypin. Both cog monitor the counter and automatically stop themselves with cogstop once the value reaches 10. Short waitx ##DELAY pauses ensure that increments and transmissions occur at visible intervals. The shared_counter variable in hub RAM allows inter-cog communication, while working_value is a one-byte cog register used for temporary storage. As a result, the serial output displays the numbers "0" through "9" sequentially, demonstrating parallel execution, safe sharing of hub memory between cogs, and proper self-stopping to prevent unnecessary execution.

A cog in the Propeller 2 can be dedicated to flipping or toggling memory values in hub RAM, allowing a program to perform continuous or repetitive tasks independently. By starting a cog with `COGINIT`, you can run a routine that repeatedly reads a value from hub memory, modifies it (for example, inverting bits or toggling a flag), and writes it back. Using `COGID`, the cog can identify itself to coordinate memory access if multiple cogs are involved, preventing conflicts. Once the flipping task is no longer needed, `COGSTOP` can halt the cog safely. This method leverages a separate core to handle ongoing memory operations without blocking the main program, demonstrating efficient use of parallel processing for dynamic data manipulation.

Program: cog_memoryflip.spin2

```
'cog_memoryflip.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)
  DELAY             = 180_000_000 / 2

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  main
    wrbyte #"+", #shared_char

    coginit #1, ##@swap_char
    waitx ##DELAY

    rdbyte working_value, ##shared_char
    wypin working_value, ##TRANSMISSION_PIN

    jmp #endprog

  swap_char
    rdbyte working_value, ##shared_char
    cmp working_value, #"-" wz
    if_z mov working_value, #"+"
    if_nz mov working_value, #"-"
    wrbyte working_value, ##shared_char
    cogstop #1

  endprog
    jmp  #$

  working_value  res 1
  shared_char    byte 0
```

The main cog first writes the character "+" into the shared hub-memory variable shared_char using wrbyte. A second cog is then started with coginit to execute the swap_char routine independently. This routine reads the value from shared_char into the cog register working_value using rdbyte, checks whether it is "-"; if so, it changes it to "+", otherwise it changes it to "-", and writes the updated character back to hub memory using wrbyte. After a short delay (waitx ##DELAY), the main cog reads the modified value from shared_char into working_value and immediately transmits it via the transmission pin using

wypin. Both hub-to-cog transfers and transmissions occur non-blocking. The variable shared_char resides in hub RAM to allow inter-cog communication, while working_value is a one-byte cog register used for temporary storage. As a result, the serial output displays the character "-", demonstrating that a separate cog can independently modify a shared hub-memory variable and that the main cog can successfully retrieve and use the updated value.

Chapter 14: String Manipulation

A string is simply a sequence of characters stored in memory, typically ending with a null byte (0). The Propeller 2 Assembly language provides no built-in string functions, so all operations must be performed manually using byte-level instructions such as **RDBYTE** and **WRBYTE.** Strings are commonly used for serial communication (UART text output), constructing messages, and comparing or modifying character sequences.

Print a String

Printing a string in P2 assembly involves reading characters sequentially from hub RAM until a null terminator (0) is found. Because the Propeller 2 has no built-in string functions, each character must be fetched manually using `RDBYTE` and then sent to an output device such as a UART SmartPin. The process continues one byte at a time until the null terminator is reached, signaling the end of the string. This method is fundamental for displaying messages or sending text to serial consoles.

String Length

Finding the length of a string requires iterating through the bytes stored in memory until the null terminator is encountered. A counter is incremented for each character processed, resulting in the total number of visible characters in the string. Since the P2 does not track string sizes automatically, this manual traversal ensures precision and allows later routines—such as formatting, validation, or buffer management—to rely on the correct character count.

Program: string_length.spin2

```
'string_length.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  mov stringPointer, ##@mystring
  call #get_string_length

  add string_length, #"0"
  wypin string_length, #TRANSMISSION_PIN

  jmp     #endprog

  get_string_length
    mov     string_length, #0
    .loop
      rdbyte  currentChar, stringPointer
      cmp     currentChar, #0 wz
      if_z    ret
      add     string_length, #1
      add     stringPointer, #1
    jmp #.loop

  endprog
    jmp  #$

  string_length    res 1
  currentChar      res 1
  stringPointer    res 1

  mystring         byte "Hello!",0
```

We demonstrate how to calculate the length of a null-terminated string in Propeller 2 assembly. This program demonstrates how to calculate the length of a null-terminated string in Propeller 2 assembly. The COG register stringPointer is initialized to the hub-memory address of the string mystring. The subroutine get_string_length repeatedly reads each byte from hub RAM into the COG register currentChar using RDBYTE, checks for the null terminator (0), and

increments the counter string_length for each character processed. The loop continues until the null terminator is encountered. After the subroutine returns, string_length contains the number of characters in the string (excluding the null terminator). The program then converts string_length to its ASCII equivalent by adding "0" and transmits it via the transmission pin using wypin. The variables stringPointer and currentChar are one-byte COG registers used for pointer tracking and temporary storage, while string_length is a COG register counter. The string mystring resides in hub RAM. This method is fundamental in P2 programming, as strings do not carry inherent length information and must be manually processed.

String Number Length

Sometimes you may want to **limit the maximum length** of a string when counting characters. This prevents errors if the string is longer than expected. In this example, we count up to a specific number of characters, stopping either at the **null terminator** or the **limit**.

Program: string_number_length.spin2

```
'string_number_length.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  main
```

```
  mov stringPointer, ##@mystring
  call #get_string_length

  add string_length, #"0"
  wypin string_length, #TRANSMISSION_PIN

  jmp     #endprog

get_string_length
  mov string_length, #0
  mov string_limit, #9
  .loop
      rdbyte  currentChar, stringPointer
      cmp     currentChar, #0 wz
      if_z    ret
      cmp     string_limit, string_length wz
      if_z    ret
      add     string_length, #1
      add     stringPointer, #1
  jmp     #.loop

endprog
  jmp  #$

string_length    res 1
string_limit     res 1
currentChar      res 1
stringPointer    res 1

mystring         byte "Hello!",0
```

We calculate the length of a null-terminated string in Propeller 2 assembly. Our program calculates the length of a null-terminated string in Propeller 2 assembly with an enforced maximum limit. The COG register stringPointer is initialized to the hub-memory address of the string mystring. The subroutine get_string_length repeatedly reads each byte from hub RAM into the COG register currentChar using rdbyte, checks for either the null terminator (0) or that the counter string_length has reached the maximum string_limit, and increments string_length for each character processed. The loop exits when either condition is met. After returning from the subroutine, string_length contains the number of characters in the string (up to the limit). The program then converts string_length to its ASCII equivalent by adding "0" and transmits it via the transmission pin using wypin. The variables stringPointer and currentChar are one-byte COG registers used for pointer tracking and temporary storage, while string_length and string_limit are COG registers for counting and limiting. The string mystring resides in hub RAM. In this example, the string "Hello!" produces a length of 6, transmitted as the ASCII character "6". This routine demonstrates safe and fundamental string handling in P2 assembly,

where strings do not carry inherent length information and must be manually processed.

String compare

String comparison is a byte-by-byte process where each character from the first string is matched with the corresponding character from the second string. The comparison continues until a mismatch is found or both strings reach the null terminator at the same position. If all characters match up to the terminator, the strings are considered equal. This method is the basis for implementing searching, sorting, or command-parsing routines in low-level programs.

Program: string_compare.spin2

```
'string_compare.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  main
    mov     stringPointer1, ##@mystring1
    mov     stringPointer2, ##@mystring2
    call    #string_compare

    add     string_result, #"0"
    wypin   string_result, #TRANSMISSION_PIN

    jmp     #endprog

  string_compare
    .loop
      rdbyte  currentChar1, stringPointer1
      rdbyte  currentChar2, stringPointer2
```

```
    cmp     currentChar1, currentChar2 wz
    if_nz   jmp     #.different
    cmp     currentChar1, #0 wz
    if_z    jmp     #.same
    add     stringPointer1, #1
    add     stringPointer2, #1
  jmp     #.loop

  .same
    mov     string_result, #0
ret

  .different
    mov     string_result, #1
ret

endprog  endprog
  jmp  #$

currentChar1    res 1
currentChar2    res 1
stringPointer1  res 1
stringPointer2  res 1
string_result   res 1

mystring1       byte "Hello!",0
mystring2       byte "Hello!",0
```

This compares two null-terminated strings in Propeller 2 assembly. The COG registers stringPointer1 and stringPointer2 are initialized to the hub-memory addresses of mystring1 and mystring2. The subroutine string_compare reads each byte from the strings into the one-byte COG registers currentChar1 and currentChar2 using RDBYTE, and compares them. If a mismatch is found at any position, the subroutine immediately sets string_result to 1, indicating the strings are different, and returns. If both strings reach their null terminators simultaneously without a mismatch, string_result is set to 0, indicating the strings are identical. After returning from the subroutine, string_result is converted to its ASCII equivalent by adding "0" and transmitted via the transmission pin using wypin. The variables currentChar1 and currentChar2 serve as temporary COG registers for byte comparison, while string_result holds the comparison outcome. The strings mystring1 and mystring2 reside in hub RAM. This routine provides a reliable, byte-by-byte comparison of any pair of null-terminated strings and outputs "0" for identical strings or "1" for different strings.

String Number Compare

Sometimes you only want to compare the first N characters of two strings. This is useful when you want a prefix match, when the strings may have different lengths, or when you want to limit processing to a specific number of characters. In such a comparison, you check each character up to the specified length or stop sooner if a null terminator is reached. If all compared characters match, the strings are considered equal within that range; if a mismatch is found, they are considered different. This approach allows efficient checking of string prefixes without needing to process the entire string.

Program: string_number_compare.spin2

```
'string_number_compare.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  main
    mov stringPointer1, ##@mystring1
    mov stringPointer2, ##@mystring2
    mov compareLength, #6
    call #string_n_compare

    add stringResult, #"0"
    wypin stringResult, #TRANSMISSION_PIN

    jmp     #endprog

  string_n_compare
    .loop
      cmp     compareLength, #0 wz
      if_z    jmp     #.same

      rdbyte  currentChar1, stringPointer1
      rdbyte  currentChar2, stringPointer2
      cmp     currentChar1, currentChar2 wz
```

```
    if_nz   jmp     #.different
    cmp     currentChar1, #0 wz
    if_z    jmp     #.same

    add     stringPointer1, #1
    add     stringPointer2, #1
    sub     compareLength, #1
    jmp     #.loop

  .same
    mov     stringResult, #0
ret

  .different
    mov     stringResult, #1
ret

endprog
  jmp  #$

currentChar1     res 1
currentChar2     res 1
stringPointer1   res 1
stringPointer2   res 1
stringResult     res 1
compareLength    res 1

mystring1        byte "Hello!",0
mystring2        byte "Hello?",0
```

We are going to compare up to a specified number of characters from two null-terminated strings in Propeller 2 assembly. The COG registers stringPointer1 and stringPointer2 are initialized to the hub-memory addresses of mystring1 and mystring2, and compareLength is set to the maximum number of characters to compare. The subroutine string_n_compare reads one byte at a time from hub RAM into the COG registers currentChar1 and currentChar2 using rdbyte. Each pair of bytes is compared: if a mismatch is found, stringResult is immediately set to 1 and the subroutine returns. The loop also exits if a null terminator (0) is encountered in either string or if compareLength reaches zero, in which case stringResult is set to 0, indicating the strings match up to the specified length. After returning, stringResult is converted to its ASCII equivalent by adding "0" and transmitted via the transmission pin using wypin. The temporary COG registers currentChar1 and currentChar2 hold each byte for comparison, while stringResult stores the final outcome. The strings reside in hub RAM. In this example, mystring1 is "Hello!" and mystring2 is "Hello?"; the program compares the first six characters and outputs "0" if they match within that limit, or "1" if a difference is found. This routine is useful for prefix

matching, safely limiting string comparisons, and handling strings of differing lengths.

String copy

Copying a string is done by reading each byte from a source address and writing it to a destination address, repeating the process until the null terminator is copied as well. The terminator ensures the destination string is properly closed and safe for later operations. This operation is essential when building dynamic messages, managing multiple buffers, or preparing data for output, because the P2 provides no direct memory-copy instructions for strings.

Program: string_copy.spin2

```
'string_copy.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  main
    mov     sourcePointer, #@mystring1
    mov     destinationPointer, #@mystring2
    call    #string_copy

    mov     destinationPointer, #@mystring2
      .loop_transmit
        rdbyte  currentChar, destinationPointer
        cmp     currentChar, #0 wz
        if_z    jmp #endprog
        wypin   currentChar, #TRANSMISSION_PIN
```

```
      waitx ##DELAY
      add     destinationPointer, #1
    jmp #.loop_transmit

    jmp #endprog

string_copy
  .loop
    rdbyte  currentChar, sourcePointer
    wrbyte  currentChar, destinationPointer
    add     sourcePointer, #1
    add     destinationPointer, #1
    cmp     currentChar, #0 wz
    if_z    ret
    jmp #.loop

endprog
  jmp  #$

currentChar         res 1
sourcePointer       res 1
destinationPointer  res 1

mystring1           byte "Hello!",0
mystring2           byte 0,0,0,0,0,0,0       ' enough space for copy
```

Our program demonstrates how to copy a null-terminated string from one memory locThis program copies a null-terminated string from one hub memory location to another in Propeller 2 assembly and then transmits it via the transmission pin. The COG registers sourcePointer and destinationPointer are initialized to the addresses of mystring1 and mystring2, respectively. The subroutine string_copy reads one byte at a time from the source using rdbyte, writes it to the destination using wrbyte, and increments both pointers. The loop terminates when a null terminator (0) is encountered, ensuring the full string is copied. After returning, the main program resets destinationPointer to the start of mystring2 and iterates through the copied string: each byte is read into the temporary COG register currentChar, transmitted via wypin, delayed briefly with waitx, and the pointer incremented. This guarantees the copied string is output character by character in order. The variables sourcePointer, destinationPointer, and currentChar clearly indicate their roles, and the space allocated for mystring2 ensures it can fully accommodate the copied string. The original string remains unchanged, while the copy is safely stored in hub memory and transmitted sequentially.

String number copy

Copying a string of characters from one location to another is common. The process involves taking each character from a source string and placing it into a destination, one by one, until the end of the string is reached. If the string is shorter than the space available, the remaining locations can be filled with zeros. This method is useful for creating exact copies of messages, preparing data for output, or organizing strings in memory, giving a beginner a clear understanding of how string data can be moved and managed in a program.

Program: string_number_copy.spin2

```
'string_number_copy.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)
  DELAY             = 180_000_000

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  main
    mov sourcePointer, #@mystring1
    mov destinationPointer, #@mystring2
    mov maxCopyBytes, #2
    call #string_number_copy

    mov     destinationPointer, #@mystring2
    .loop_transmit
      rdbyte  currentChar, destinationPointer
      cmp     currentChar, #0 wz
      if_z    jmp #endprog
      wypin   currentChar, #TRANSMISSION_PIN
      waitx ##DELAY
      add     destinationPointer, #1
      jmp #.loop_transmit

      jmp #endprog

  string_number_copy
    .loop
```

```
    cmp     maxCopyBytes, #0 wz
    if_z    ret

    rdbyte  currentChar, sourcePointer
    cmp     currentChar, #0 wz
    if_z    jmp #.fill_zero

    wrbyte  currentChar, destinationPointer
    add     sourcePointer, #1
    add     destinationPointer, #1
    sub     maxCopyBytes, #1
    jmp     #.loop

  .fill_zero
    mov     currentChar, #0
  .fill_loop
    wrbyte  currentChar, destinationPointer
    add     destinationPointer, #1
    sub     maxCopyBytes, #1
    cmp     maxCopyBytes, #0 wz
    if_z    ret
    jmp #.fill_loop

endprog
  jmp  #$

currentChar        res 1
sourcePointer      res 1
destinationPointer res 1
maxCopyBytes       res 1

mystring1          byte "Hello!",0
mystring2          byte 0,0,0,0,0,0,0
```

When this program runs, it copies a limited number of characters from one null-terminated string to another in Propeller 2 assembly. The main program initializes sourcePointer and destinationPointer to the addresses of mystring1 and mystring2, respectively, and sets maxCopyBytes to 2, limiting the copy to two characters. The subroutine string_number_copy reads one byte at a time from the source using rdbyte, writes it to the destination using wrbyte, and increments both pointers while decrementing maxCopyBytes. If the source string ends before the limit is reached, the routine fills the remaining space in the destination with zeros to ensure proper null-termination. After returning, the main program resets destinationPointer to the start of mystring2 and transmits each byte sequentially through wypin, with a full-second delay (waitx ##DELAY) between transmissions. The loop stops when a null terminator (0) is encountered. This ensures only the intended number of characters is copied and transmitted—in this example, the characters "H" and "e"—demonstrating safe, length-limited string copying in hub memory. The variables currentChar,

sourcePointer, destinationPointer, and maxCopyBytes clearly indicate their roles in controlling the copy process and temporary storage.

String concatenate

String concatenation is performed by manually copying two null-terminated strings into a single buffer. The process begins by copying the first string byte-by-byte into the destination until its null terminator is reached. Then, instead of stopping, the routine continues by reading each character of the second string and writing it immediately after the first string's final byte. After the second string's null terminator is copied, the combined buffer contains a seamless concatenation of both strings. This method is essential for building messages, merging input fragments, constructing command lines, or creating dynamic text output in environments where no built-in string library exists.

Program: string_concatenate.spin2

```
'string_concatenate.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)
  DELAY             = 180_000_000

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  main
    mov sourcePointer1, #@mystring2
    mov destinationPointer, #@mystring3
    call #string_copy

    mov sourcePointer2, #@mystring1
    mov destinationPointer, #@mystring3
    call #string_concat

    mov transmitPointer, #@mystring3
```

```
  .loop_transmit
    rdbyte  currentChar, transmitPointer
    cmp     currentChar, #0 wz
    if_z    jmp #endprog
    wypin   currentChar, #TRANSMISSION_PIN
    waitx ##DELAY
    add     transmitPointer, #1
    jmp #.loop_transmit

    jmp #endprog

string_copy
  .loop_copy
    rdbyte  currentChar, sourcePointer1
    wrbyte  currentChar, destinationPointer
    add     sourcePointer1, #1
    add     destinationPointer, #1
    cmp     currentChar, #0 wz
    if_z    ret
    jmp #.loop_copy

string_concat
  .loop_find_end
    rdbyte  currentChar, destinationPointer
    cmp     currentChar, #0 wz
    if_z    jmp #.append_start
    add     destinationPointer, #1
    jmp #.loop_find_end

  .append_start
    .loop_append
      rdbyte  currentChar, sourcePointer2
      wrbyte  currentChar, destinationPointer
      add     sourcePointer2, #1
      add     destinationPointer, #1
      cmp     currentChar, #0 wz
      if_z    ret
      jmp #.loop_append

endprog
  jmp  #$

currentChar         res 1
sourcePointer1      res 1
sourcePointer2      res 1
destinationPointer  res 1
transmitPointer     res 1

mystring1           byte "World!",0
mystring2           byte "Hello,",0,0,0,0,0,0
mystring3           byte 256
```

Let’s demonstrate how to combine two strings into a single buffer and transmit the result character by character. First, the string "Hello," stored in

mystring2 is copied into the buffer mystring3 using the string_copy routine, which reads each byte from the source with rdbyte and writes it to the destination with wrbyte, stopping at the null terminator. Next, the string "World!" from mystring1 is appended to the end of mystring3 using the string_concat routine, which first locates the null terminator in the destination buffer and then sequentially writes each byte of the second string, including its null terminator, to the buffer. After concatenation, the main program sets transmitPointer to the start of mystring3 and transmits each character one by one through the TRANSMISSION_PIN using wypin, with a full-second delay (waitx ##DELAY) between characters, stopping at the null terminator. The final result, "Hello,World!", appears sequentially on the serial output. Temporary registers currentChar, sourcePointer1, sourcePointer2, destinationPointer, and transmitPointer clearly indicate the roles of each value in reading, writing, appending, and transmitting the string. This routine illustrates safe string copy, concatenation, and sequential transmission in hub memory while preserving proper null-termination.

String number concatenate

String concatenation can be done by first copying a source string into a destination buffer and then appending a specific number of characters from another string. A null terminator is added at the end to maintain a valid string. This approach is useful for working with fixed-size buffers, ensuring that memory is not overwritten while combining strings.

Program: string_number_concatenate.spin2

```
'string_number_concatenate.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)
  DELAY             = 180_000_000

DAT
  org 0
```

```
asmclk

wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
wypin  #1, #TRANSMISSION_PIN
dirh   #TRANSMISSION_PIN

main
  mov sourcePointer2, #@mystring2
  mov destinationPointer, #@mystring3
  call #string_copy

  mov sourcePointer1, #@mystring1
  mov destinationPointer, #@mystring3
  mov maxAppendCount, #2
  call #string_n_concat

  mov transmitPointer, #@mystring3
  .loop_transmit
    rdbyte  currentChar, transmitPointer
    cmp     currentChar, #0 wz
    if_z    jmp #endprog
    wypin   currentChar, #TRANSMISSION_PIN
    waitx ##DELAY
    add     transmitPointer, #1
    jmp #.loop_transmit

  jmp #endprog

string_copy
  .loop_copy
    rdbyte currentChar, sourcePointer2
    wrbyte currentChar, destinationPointer
    add sourcePointer2, #1
    add destinationPointer, #1
    cmp currentChar, #0 wz
    if_z ret
    jmp #.loop_copy

string_n_concat
  .loop_find_end
    rdbyte currentChar, destinationPointer
    cmp currentChar, #0 wz
    if_z jmp #.append_start
    add destinationPointer, #1
    jmp #.loop_find_end

  .append_start
    .loop_append
      cmp maxAppendCount, #0 wz
      if_z jmp #.null_terminate

      rdbyte currentChar, sourcePointer1
      cmp currentChar, #0 wz
      if_z jmp #.null_terminate
      wrbyte currentChar, destinationPointer
      add sourcePointer1, #1
```

```
      add destinationPointer, #1
      sub maxAppendCount, #1
      jmp #.loop_append

  .null_terminate
    mov     currentChar, #0
    wrbyte  currentChar, destinationPointer
ret

endprog
  jmp  #$

currentChar         res 1
sourcePointer1      res 1
sourcePointer2      res 1
destinationPointer  res 1
transmitPointer     res 1
maxAppendCount      res 1

mystring1           byte "World!",0
mystring2           byte "Hello,",0,0,0,0,0,0
mystring3           byte 256
```

Here we demonstrate how to copy a string and append a limited number of characters from another string in Propeller 2 assembly, then transmit the final result through a smart pin. First, the string "Hello," in mystring2 is copied into the buffer mystring3 using the string_copy routine. This routine reads each byte from the source with RDBYTE and writes it to the destination with WRBYTE, stopping at the null terminator. Next, the string_n_concat routine appends up to maxAppendCount characters from mystring1 ("World!") to the end of mystring3. It first finds the null terminator of the destination buffer, then sequentially writes each byte from the source until either the maximum number of characters is appended or the source string's null terminator is reached. After appending, the routine ensures the resulting string is properly null-terminated.

Finally, the main program sets transmitPointer to the start of mystring3 and transmits each character one by one over the TRANSMISSION_PIN using wypin, with a full-second delay (waitx ##DELAY) between characters, stopping at the null terminator. Temporary registers—currentChar, sourcePointer1, sourcePointer2, destinationPointer, transmitPointer, and maxAppendCount—clearly indicate the role of each value in reading, writing, appending, and transmitting the string. The output will display the characters sequentially: H, e, l, l, o, ,, W, o, representing the concatenated string "Hello,Wo". This example

illustrates safe string copying, controlled-length concatenation, proper null-termination, and sequential character transmission in hub memory.

String to decimal

Converting a numeric string to an integer involves reading ASCII characters from a null-terminated string and translating them into their numeric values. Each character is processed by subtracting the ASCII code for `'0'` (0x30), which yields the corresponding decimal digit. For single-digit strings, this operation is equivalent to the standard `atoi` function in high-level languages. The resulting integer can then be used in arithmetic operations, comparisons, or transmitted over serial interfaces. Extending this approach with loops allows multi-digit numbers to be fully converted into usable integers.

Program: string_to_decimal.spin2

```
'string_to_decimal.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)
  DELAY             = 180_000_000

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  main
    mov stringPointer, ##@testStr
    call #strToDec

    add numericValue, #"0"
    wypin   numericValue, #TRANSMISSION_PIN
    waitx   ##DELAY

    jmp     #endprog
```

```
strToDec
  rdbyte  currentChar, stringPointer
  sub     currentChar, #"0"
  mov     numericValue, currentChar
  ret

endprog
  jmp  #$

testStr byte "3", 0

numericValue      res 1
currentChar       res 1
stringPointer     res 1
```

Here we are converting a single-digit string into its numeric value. The string "3" is stored in testStr as a null-terminated string. The subroutine strToDec reads the first byte of the string using RDBYTE into the temporary register currentChar. Since ASCII digits start at 0x30, the routine subtracts "0" from the character to obtain the numeric integer value and stores it in numericValue. This effectively converts the ASCII character '3' into the integer 3.

In the main program, numericValue is then converted back to its ASCII character by adding "0" and transmitted through TRANSMISSION_PIN using wypin. A brief delay (waitx ##DELAY) ensures proper spacing on the serial line.

The temporary registers currentChar, stringPointer, and numericValue clearly indicate their roles: reading the input character, pointing to the source string, and storing the numeric conversion result. The output will transmit the character '3', demonstrating that the program correctly converts a string digit into a number and then transmits it.

This approach is useful for processing numeric input in string form while keeping the value ready for display or further arithmetic operations. It can be extended to handle multi-digit strings by iterating through the string and accumulating the total numeric value.

Decimal to character

Converting a single-digit integer into its ASCII representation involves adding the ASCII code for `'0'` (0x30) to the numeric value. The routine `DecToStr` takes an integer between 0 and 9 and stores the resulting ASCII character in a buffer or variable. This character can then be transmitted over

serial, displayed, or combined with other strings. This process is essentially the inverse of the `strToDec` routine and is useful for generating text output from numeric calculations in low-level P2 program

Program: decimal_to_character.spin2

```
'decimal_to_character.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)
  DELAY             = 180_000_000

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  main
    mov number, #4
    call #DecToStr

    add asciiChar, #"0"
    wypin asciiChar, #TRANSMISSION_PIN
    waitx ##DELAY

    jmp #endprog

  DecToStr
    mov asciiChar, number
    ret

  endprog
    jmp  #$

  asciiChar   res 1
  number      res 1
```

The result of this program is that the single-digit integer stored in number (in this case 4) is converted to its corresponding ASCII character and transmitted over the TRANSMISSION_PIN. The integer value 4 is stored in the register number, and the TRANSMISSION_PIN is configured for asynchronous output using wrpin and wxpin, with dirh setting the pin as output. The

subroutine DecToStr moves the integer from number into the variable asciiChar, and the main program adds the ASCII value of "0" to convert the integer into its ASCII representation. For example, 4 + "0" results in 0x34, which corresponds to the character '4'. This character is then sent out through the transmission pin using wypin, with a short waitx delay to ensure proper spacing for serial reception. The program's variables include number, which holds the original integer, and asciiChar, which holds the converted character ready for output. When run, the program transmits the character '4' on the serial line, demonstrating how a numeric value can be converted to a character for display or further serial communication. The approach works for any single-digit integer and the program enters an infinite loop after transmission, allowing repeated inspection if needed. This routine is a fundamental building block for displaying numeric values as characters and can be extended to handle multi-digit numbers for serial output.

Decimal to string

Converting a multi-digit decimal string into an integer requires processing each character sequentially. Each ASCII digit is read from the null-terminated string and converted to its numeric value by subtracting `'0'` (0x30). The integer is accumulated using the formula `result = result × 10 + digit`, effectively building the number as each new digit is added. Once the full string has been processed, the resulting integer can be used for calculations, comparisons, or output. For display or serial transmission, the integer can be printed one digit at a time, using a routine such as `DecToStr` to convert each digit back into an ASCII character.

Program: decimal_to_string.spin2

```
'decimal_to_string.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)
  DELAY             = 180_000_000

DAT
  org 0
```

```
asmclk

wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
wypin  #1, #TRANSMISSION_PIN
dirh   #TRANSMISSION_PIN

main
  call #stringToInteger
  call #printInteger
  jmp #endprog

stringToInteger
  mov stringPointer, ##@testString
  mov convertedInteger, #0
  .loop
    rdbyte currentCharacter, stringPointer
    cmp currentCharacter, #0 wz
    if_z ret
    sub currentCharacter, #"0"
    qmul convertedInteger, #10
    getqx convertedInteger
    add convertedInteger, currentCharacter
    add stringPointer, #1
    jmp #.loop

printInteger
  mov divisor, ##1_000_000_000
  mov leadingZeroFlag, #0
  mov tempInteger, convertedInteger
  .loop
    mov     currentDigit, #0
    qdiv    tempInteger, divisor
    getqx   currentDigit
    getqy   tempInteger
    cmp     currentDigit, #0 wz
    if_z cmp leadingZeroFlag, #0 wz
    if_z jmp #.skipPrint
    if_nz mov leadingZeroFlag, #1
    add     currentDigit, #"0"
    wypin   currentDigit, #TRANSMISSION_PIN
    waitx   ##DELAY
  .skipPrint
    qdiv    divisor, #10
    getqx   divisor
    cmp     divisor, #0 wz
    if_nz jmp #.loop
    cmp     leadingZeroFlag, #0 wz
    if_z jmp #.loop
  ret

endprog
  jmp  #$

testString byte "451", 0

currentDigit res 1
```

```
divisor res 1
tempInteger res 1
leadingZeroFlag res 1

convertedInteger res 1
currentCharacter res 1
stringPointer res 1
```

This code takes a string of ASCII digits and converts it into an actual integer, then prints that integer as a sequence of ASCII characters. The stringToInteger routine begins by initializing a pointer to the null-terminated input string testString and a register convertedInteger to zero. It reads each character one by one, subtracts the ASCII value of "0" to obtain its numeric value, multiplies the accumulated integer by 10 using qmul, and adds the new digit. This process continues until the null terminator is encountered, resulting in the full numeric value stored in convertedInteger. The printInteger routine then extracts each decimal digit by dividing the integer by decreasing powers of ten with qdiv, retrieves both the quotient and remainder using getqx and getqy, converts the digit to its ASCII representation by adding "0", and transmits it sequentially through TRANSMISSION_PIN with short waitx delays. A leading-zero flag ensures that only significant digits are printed, skipping unnecessary zeros. For the example string "451", the program transmits the characters '4', '5', and '1' in order, accurately representing the numeric value of the original string. This approach demonstrates a complete round-trip conversion between string and numeric forms in P2 assembly, suitable for processing numeric input and producing formatted output.

Decimal to string revisited

Converting a multi-digit integer into a null-terminated ASCII string involves first breaking the number into individual digits. The routine `DecToStr` stores the integer in a variable such as `result` and converts each digit into its ASCII equivalent, temporarily writing them in reverse order to a buffer. After all digits are processed, the string is reversed in-place to produce the correct sequence. The resulting null-terminated string can then be transmitted over a SmartPin

using `printStr` or used elsewhere in the program. This routine functions like a classic `itoa`, handling integers up to 32 bits and enabling formatted numeric output in low-level P2 programs.

Program: decimal_to_string2.spin2

```
'decimal_to_string2.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)
  DELAY             = 180_000_000

DAT
  org 0
  asmclk

  wrpin  ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin  ##BAUD_MODE, #TRANSMISSION_PIN
  wypin  #1, #TRANSMISSION_PIN
  dirh   #TRANSMISSION_PIN

  main
    mov result, ##994421
    mov ptra, ##@testStr

    call #decToStr
    mov ptra, ##@testStr
    call #printStr

    jmp #endprog

  decToStr
    cmp result, #0 wz
    if_z jmp #.zero_case

    mov string_start_pointer, ptra
    mov quotient, result

    .digit_loop
      qdiv quotient, #10
      getqx quotient
      getqy current_char
      add current_char, #"0"
      wrbyte  current_char, ptra
      add ptra, #1
      cmp quotient, #0 wz
      if_nz jmp #.digit_loop
      wrbyte  #0, ptra
      mov string_left_index, string_start_pointer
      sub ptra, #1
```

```
  .reverse_loop
    cmp string_left_index, ptra wc
    if_nc ret
    rdbyte current_char, string_left_index
    rdbyte string_right_index, ptra
    wrbyte string_right_index, string_left_index
    wrbyte current_char, ptra
    add string_left_index, #1
    sub ptra, #1
    jmp #.reverse_loop
  .zero_case
    mov ptra, string_start_pointer
    mov current_char, #"0"
    wrbyte current_char, ptra
    add ptra, #1
    wrbyte  #0, ptra
  ret

printStr
  .loop
    rdbyte  current_char, ptra
    cmp     current_char, #0 wz
    if_z    ret
    call    #tx_char
    add     ptra, #1
    jmp #.loop

tx_char
  wypin   current_char, #TRANSMISSION_PIN
  waitx   ##DELAY
  ret

endprog
  jmp  #$

testStr byte "", 0
digit res 1
divisor res 1
tmp res 1
number res 1
leading_zero  res 1
result  res 1
current_char res 1
string_start_pointer res 1
quotient res 1
string_left_index res 1
string_right_index res 1
```

This `decimal to string` program converts a decimal integer into a string and transmits it character by character over a serial output pin. The integer to be converted is stored in the variable `result`, and the resulting string is written into the buffer `testStr`. The main program initializes the integer and the string pointer, then calls the `decToStr` subroutine to perform the

conversion and `printStr` to transmit each character sequentially through the `TRANSMISSION_PIN`. In `decToStr`, the integer is repeatedly divided by 10 using `qdiv` to extract each digit starting from the least significant digit. Each digit is converted to its ASCII representation and stored in the string buffer. Because the digits are generated in reverse order, the subroutine subsequently reverses the string in place to obtain the correct sequence. A special case handles the number zero by directly writing the character `"0"` to the buffer. The `printStr` subroutine reads each character from the null-terminated string and calls `tx_char` to transmit it with a short `waitx` delay between characters, ensuring proper serial output timing. As a result, an integer like `994421` is converted to the string `"994421"` and transmitted sequentially as individual ASCII characters, producing an accurate textual representation of the original number.

Chapter 15: SmartPin Setup Instructions (for UART)

The Propeller 2's SmartPin system simplifies handling advanced I/O tasks by offloading them from the COG processors to dedicated hardware. This allows features like UART communication, PWM signal generation, and counting to operate independently and efficiently. By leveraging SmartPins, developers can perform precise, high-speed I/O operations without using extensive processor cycles. In this chapter, we explore how to use SmartPins for asynchronous UART transmission, enabling reliable serial data transfer at a configurable baud rate while keeping the COGs free for other tasks.

On the Propeller 2, any I/O pin can be turned into a SmartPin by configuring its registers with the WRPIN, WXPIN, and WYPIN instructions. When a SmartPin is set to asynchronous transmit mode (P_ASYNC_TX), it automatically manages all aspects of UART timing and framing. This means you only need to write the data bytes to the pin, and the SmartPin takes care of sending the start bit, data bits, and stop bit—simplifying serial communication.

To receive asynchronous UART data on the Propeller 2, the SmartPin system provides dedicated instructions that simplify handling incoming signals. **TESTP** allows the program to efficiently poll the reception pin, detecting when a new byte is available without halting the COG. Once data arrives, **RDPIN** reads the byte from the SmartPin's FIFO buffer, and bit shifting and masking isolate the meaningful 8-bit value. **AKPIN** then acknowledges that the byte has been processed, clearing the reception flag so the SmartPin can handle the next incoming byte. Using these instructions in combination, developers can implement continuous, high-speed data reception while relying on the SmartPin hardware to manage all UART timing, framing, and serialization, leaving the COG free for other operations.

Syntax

```
testp destination, {#}source
```

- **destination** = where the value will go

- **source** = where the value comes from
    - Use # if it is a **literal constant** (number, hex, ASCII).
    - `Omit # if it is a` **`variable or register`**`.`
    - `TESTP` **`tests one or more bits in a source value`** `and sets the` **`carry flag (C)`** `based on the result.`
    - It **does not modify the source data**, only updates the carry flag.
    - Commonly used for **polling bits in registers, variables, or SmartPins**, such as detecting when a UART SmartPin has a new byte or checking specific status bits.

`rdpin destination, {#}source`

- **destination** = where the value will go
- **source** = where the value comes from
    - Use # if it is a **literal constant** (number, hex, ASCII).
    - `Omit # if it is a` **`variable or register`**
    - `RDPIN` **`reads the current value`** `from a SmartPin's` **`input or FIFO buffer`** `into the specified` **`destination`** `(cog register).`
    - The read value may require **bit shifting or masking** to isolate the desired bits, such as extracting an **8-bit UART byte** from the SmartPin data.
    - Commonly used in **UART programs** to retrieve **received bytes** from the RX SmartPin.
    - **Does not modify the SmartPin**, it only copies the value into the destination.

`akpin destination, {#}source`

- **destination** = where the value will go
- **source** = where the value comes from
    - Use # if it is a **literal constant** (number, hex, ASCII).

- Omit # if it is a **variable or register**
- AKPIN signals the SmartPin that the previously read data has been processed.
- This clears the reception flag or read status, allowing the SmartPin to handle the next incoming value.
- Commonly used in UART programs after RDPIN to acknowledge received bytes without blocking the cog.
- Does not read or modify the data itself, it only informs the SmartPin that the previous data is done.

Basic UART Echo with SmartPins

We demonstrate a basic UART echo program using the Propeller 2's SmartPin system. The program receives serial data on one pin and immediately sends it back on another, providing a simple way to verify both receive (RX) and transmit (TX) functionality.

Program: echo.spin2

```
'echo.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  RECEPTION_PIN     = 63
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin   ##(P_ASYNC_RX | P_HIGH_15K), #RECEPTION_PIN
  wxpin   ##BAUD_MODE, #RECEPTION_PIN
  dirh    #RECEPTION_PIN

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
```

```
wypin   #1, #TRANSMISSION_PIN
dirh    #TRANSMISSION_PIN

main
.echo_loop
  testp   #RECEPTION_PIN wc
  if_nc  jmp     #.echo_loop
  rdpin   reception_byte, #RECEPTION_PIN
  shr     reception_byte, #24
  and     reception_byte, #255
  akpin   #RECEPTION_PIN
  wypin   reception_byte, #TRANSMISSION_PIN
  jmp     #.echo_loop

endprog
  jmp  #$

reception_byte    res 1
```

The echo program sets up the Propeller 2 to perform a continuous UART echo using SmartPin hardware for both reception and transmission. It configures one SmartPin for asynchronous RX input and another SmartPin for asynchronous TX output, setting the baud rate and timing parameters appropriately. In the main loop, the program continuously monitors the RX pin for incoming data using testp, and when a character is detected, it reads the byte from the SmartPin FIFO, shifts and masks it to isolate the 8-bit value, acknowledges the reception with akpin, and immediately sends the same byte out through the TX pin using wypin. This means any character sent to the RX pin is instantly echoed back; for example, sending "A" results in "A" being transmitted back, and a sequence like "1, 2, 3" is returned in the same order. This behavior confirms that the SmartPins are correctly handling UART framing and serialization, the baud rate is correctly configured, and the echo loop operates continuously without requiring the COG to manage low-level bit timing. As a result, the system provides a reliable and real-time serial interface, suitable for terminal I/O, debugging output, or interactive command-based applications.

Single Digit Input and Check (0–9)

Building on the previous UART echo program, this version adds input validation. This example introduces key programming concepts such as

character testing, conditional branching, and string output using SmartPins, demonstrating how the Propeller 2 can make decisions and communicate results over a serial interface.

Program: get_digit.spin2

```
'get_digit.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  RECEPTION_PIN     = 63
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

dat
  org 0
  asmclk

  wrpin   ##(P_ASYNC_RX | P_HIGH_15K), #RECEPTION_PIN
  wxpin   ##BAUD_MODE, #RECEPTION_PIN
  dirh    #RECEPTION_PIN

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
  wypin   #1, #TRANSMISSION_PIN
  dirh    #TRANSMISSION_PIN

  main
    .get_input
      testp   #RECEPTION_PIN wc
      if_nc  jmp     #.get_input

      rdpin   input_character, #RECEPTION_PIN
      shr     input_character, #24
      and     input_character, #$FF
      akpin   #RECEPTION_PIN

      wypin   input_character, #TRANSMISSION_PIN
      call    #.flush_transmission
      cmp     input_character, #"0" wc
      if_c   jmp     #.not_number
      cmp     input_character, #"9" wz,wc
      if_a   jmp     #.not_number

      mov     string_pointer, ##@msg_number
      call    #.print_string
      jmp     #.done

    .not_number
      mov     string_pointer, ##@msg_notnum
      call    #.print_string
    .done
```

```
  jmp     #.get_input

.print_string
.nextch
  rdbyte  current_byte, string_pointer
  cmp     current_byte, #0 wz
  if_z   ret
  wypin   current_byte, #TRANSMISSION_PIN
  call    #.flush_transmission
  add     string_pointer, #1
  jmp     #.nextch

.flush_transmission
  rdpin   transmission_busy, #TRANSMISSION_PIN wc
  if_c   jmp     #.flush_transmission
  ret

endprog
  jmp  #$

input_character    res 1
current_byte       res 1
transmission_busy  res 1
string_pointer     res 1

msg_number     byte 13,10,"It is a number from 0 to 9",13,10,0
msg_notnum     byte 13,10,"Not a number!",13,10,0
```

The program get digit enables the Propeller 2 to read characters from a terminal via UART, identify whether they are numeric digits, and provide appropriate feedback over the same serial interface. In the main loop, the program waits for a character to arrive on the RX pin, reads it from the SmartPin FIFO, isolates the 8-bit value, and immediately echoes it back for confirmation. It then checks whether the character falls between '0' and '9'. If so, it prints the message “It is a number from 0 to 9”; otherwise, it prints “Not a number!”. The message printing routine reads each character from the message string and transmits it sequentially, using a flush subroutine to ensure that the transmission pin is ready before sending the next character. This cycle repeats indefinitely, demonstrating that the Propeller 2 can reliably capture user input, perform ASCII-based conditional logic, and output messages over UART in real time.

Multi-character input

Let's extend the previous serial input examples to handle multi-character input and simple string comparison. This example introduces concepts such as string buffering, looped character reading, and character-by-character comparison, which are essential foundations for building command interpreters and text-based interfaces.

Program: input_match.spin2

```
'input_match.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  RECEPTION_PIN     = 63
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

dat
  org       0
  asmclk

  wrpin   ##(P_ASYNC_RX | P_HIGH_15K), #RECEPTION_PIN
  wxpin   ##BAUD_MODE, #RECEPTION_PIN
  dirh    #RECEPTION_PIN

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
  wypin   #1, #TRANSMISSION_PIN
  dirh    #TRANSMISSION_PIN

  main
    mov buffer_pointer, #input_buffer_array

    .read_char
      testp   #RECEPTION_PIN wc
      if_nc  jmp #.read_char

      rdpin input_character, #RECEPTION_PIN
      shr input_character, #24
      and input_character, #$FF
      akpin #RECEPTION_PIN

      wypin input_character, #TRANSMISSION_PIN
      call #.flush_tx

      wrbyte input_character, buffer_pointer
      add buffer_pointer, #1

      cmp input_character, ##$0D wz
      if_nz  jmp #.read_char

      wrbyte #0, buffer_pointer
      mov command_pointer, #@cmd_hello
```

```
     mov buffer_pointer, #input_buffer_array
     call #.check_match

     jmp #main

.check_match
  rdbyte input_character, buffer_pointer
  rdbyte compare_character, command_pointer
  cmp compare_character, #0 wz
  if_z  jmp #.print_world
  cmp input_character, compare_character wz
  if_nz  ret
  add buffer_pointer, #1
  add command_pointer, #1
  jmp #.check_match

 .print_world
   mov message_pointer, #@msg_world
   call #.print_string
   ret

 .print_string
 .nextch
   rdbyte compare_character, message_pointer
   cmp compare_character, #0 wz
   if_z ret
   wypin compare_character, #TRANSMISSION_PIN
   call #.flush_tx
   add message_pointer, #1
   jmp #.nextch

 .flush_tx
   rdpin compare_character, #TRANSMISSION_PIN wc
   if_c jmp #.flush_tx
   ret

endprog
  jmp  #$

input_character       res 1
compare_character     res 1
buffer_pointer        res 1
command_pointer       res 1
message_pointer       res 1
input_buffer_array    res 10

cmd_hello   byte "hello",0
msg_world   byte "p2asm",13,10,0
```

The program reads a string typed from the terminal, ending when Enter is pressed. If the entered string exactly matches `"hello"`, it responds by printing `"world"` on the terminal. Any other input is ignored, and the program waits for the next string. This demonstrates that the Propeller 2 can handle multi-character input, store it in a buffer, perform character-by-character comparison

with a target string, and output messages via UART using SmartPins. Users can type different strings to verify that only the exact match triggers the response.

Line Editing with CRLF

Continuing to build a user input system, this version adds support for typing text on a terminal, correcting mistakes with the Backspace key, completing a line by pressing Enter (CR), and sending a carriage return and line feed (CRLF) sequence for clean terminal output. These enhancements are an important step toward creating interactive serial interfaces, such as command prompts or user input menus, allowing users to enter, edit, and submit text efficiently.

Program: line_edit.spin2

```
'line_edit.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  RECEPTION_PIN     = 63
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin   ##(P_ASYNC_RX | P_HIGH_15K), #RECEPTION_PIN
  wxpin   ##BAUD_MODE, #RECEPTION_PIN
  dirh    #RECEPTION_PIN

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
  wypin   #1, #TRANSMISSION_PIN
  dirh    #TRANSMISSION_PIN

  main
    mov     buffer_pointer, #input_buffer_array

    .read_char
      testp   #RECEPTION_PIN wc
      if_nc   jmp     #.read_char

      rdpin   input_character, #RECEPTION_PIN
      akpin   #RECEPTION_PIN
```

```
    shr     input_character, #24
    and     input_character, #$FF

    cmp     input_character, ##$0D wz
    if_z    jmp     #.line_done

    cmp     input_character, ##$08 wz
    if_z    jmp     #.handle_backspace

    cmp     input_character, ##$7F wz
    if_z    jmp     #.handle_backspace

    wypin   input_character, #TRANSMISSION_PIN
    call    #flush_tx

    wrbyte  input_character, buffer_pointer
    add     buffer_pointer, #1

    jmp     #.read_char

  .handle_backspace
    cmp     buffer_pointer, #input_buffer_array wz
    if_z    jmp     #.read_char

    sub     buffer_pointer, #1

    wypin   ##$08, #TRANSMISSION_PIN
    call    #flush_tx
    wypin   ##32, #TRANSMISSION_PIN
    call    #flush_tx
    wypin   ##$08, #TRANSMISSION_PIN
    call    #flush_tx

    jmp     #.read_char

  .line_done
    wrbyte  #0, buffer_pointer

    wypin   ##$0D, #TRANSMISSION_PIN
    call    #flush_tx
    wypin   ##$0A, #TRANSMISSION_PIN
    call    #flush_tx

    jmp     #main

flush_tx
  rdpin   transmission_busy, #TRANSMISSION_PIN wc
  if_c    jmp     #flush_tx
  ret

endprog
  jmp  #$

input_character       res 1
transmission_busy     res 1
buffer_pointer        res 1
input_buffer_array    res 32
```

Users can type text into the terminal, with each character immediately echoed back. If a mistake is made, pressing Backspace removes the last character both visually on the terminal and from the internal input buffer. Pressing Enter (CR) completes the line, sending a carriage return and line feed (CRLF) to move to a new line cleanly. The typed line is null-terminated in memory, ready for further processing. This demonstrates real-time character handling, line editing, and proper terminal formatting, forming the foundation for interactive command prompts, menus, or other text-based serial interfaces.

Read Line

This read line example demonstrates a complete UART input routine for the Propeller 2, capable of receiving text from a terminal, echoing typed characters, handling Backspace editing, ending input when Enter (CR) is pressed, and outputting a proper Carriage Return + Line Feed (CRLF) to format the text correctly on the terminal. By combining SmartPin setup, character echoing, and simple text editing, this chapter establishes the foundation for building more advanced interactive serial programs.

Program: read_line.spin2

```
'read_line.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  RECEPTION_PIN     = 63
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin   ##(P_ASYNC_RX | P_HIGH_15K), #RECEPTION_PIN
  wxpin   ##BAUD_MODE, #RECEPTION_PIN
  dirh    #RECEPTION_PIN

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
  wypin   #1, #TRANSMISSION_PIN
  dirh    #TRANSMISSION_PIN
```

```
main
  mov input_buffer_arg, #input_buffer_array
  mov max_buffer_length, #32
  call #read_line
  jmp #main

read_line
mov buffer_pointer, input_buffer_arg
mov character_count, #0
.read_char
  testp   #RECEPTION_PIN wc
  if_nc   jmp #.read_char

  rdpin   input_character, #RECEPTION_PIN
  akpin   #RECEPTION_PIN

  shr     input_character, #24
  and     input_character, #$FF

  cmp     input_character, ##$0D wz
  if_z    jmp #.line_done

  cmp     input_character, ##$08 wz
  if_z    jmp #.handle_backspace

  cmp     input_character, ##$7F wz
  if_z    jmp #.handle_backspace

  cmp     character_count, max_buffer_length wz
  if_z    jmp #.read_char

  wypin   input_character, #TRANSMISSION_PIN
  call    #flush_tx

  wrbyte  input_character, buffer_pointer
  add     buffer_pointer, #1
  add     character_count, #1

  jmp     #.read_char

.handle_backspace
  cmp     character_count, #0 wz
  if_z    jmp #.read_char

  sub     buffer_pointer, #1
  sub     character_count, #1

  wypin   ##$08, #TRANSMISSION_PIN
  call    #flush_tx
  wypin   ##32, #TRANSMISSION_PIN
  call    #flush_tx
  wypin   ##$08, #TRANSMISSION_PIN
  call    #flush_tx

  jmp     #.read_char
```

```
.line_done
  wrbyte  #0, buffer_pointer

  wypin   ##$0D, #TRANSMISSION_PIN
  call    #flush_tx
  wypin   ##$0A, #TRANSMISSION_PIN
  call    #flush_tx
ret

flush_tx
  rdpin   transmission_busy, #TRANSMISSION_PIN wc
  if_c    jmp #flush_tx
  ret

endprog
  jmp  #$

input_character      res 1
transmission_busy    res 1
buffer_pointer       res 1
character_count      res 1
input_buffer_arg     res 1
max_buffer_length    res 1
input_buffer_array   res 32
```

Users can type text into the terminal, with each character immediately echoed back. If a mistake is made, pressing Backspace removes the last character both from the display and the internal input buffer. Pressing Enter (CR) completes the line, automatically sending a Carriage Return + Line Feed (CRLF) to move to a new line on the terminal. The typed line is stored in memory as a null-terminated string, ready for further processing. This demonstrates real-time character handling, line editing, and proper terminal formatting, forming the foundation for interactive command prompts, menus, or other text-based serial interfaces on the Propeller 2.

Send string

Here we learn how to use the Propeller 2 SmartPins for full UART communication, including reading input strings from a terminal, comparing text, and sending replies. This example demonstrates reading user input using a subtroutine `read_line`, echoing typed characters, sending strings back via the subroutine `send_string`, and comparing the input to a predefined command.

Program: send_string.spin2

```
'send_string.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  RECEPTION_PIN     = 63
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin   ##(P_ASYNC_RX | P_HIGH_15K), #RECEPTION_PIN
  wxpin   ##BAUD_MODE, #RECEPTION_PIN
  dirh    #RECEPTION_PIN

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
  wypin   #1, #TRANSMISSION_PIN
  dirh    #TRANSMISSION_PIN

  main
    mov input_buffer_arg, #input_buffer
    mov max_buffer_length, #32
    call #read_line

    mov compare_pointer, #@cmd_hello
    mov buffer_pointer, #input_buffer

  check_loop
    rdbyte input_character, buffer_pointer
    rdbyte compare_character, compare_pointer
    cmp input_character, compare_character wz
    if_nz jmp #skip_response
    cmp input_character, #0 wz
    if_z  jmp #send_world
    add buffer_pointer, #1
    add compare_pointer, #1
    jmp #check_loop

  send_world
    mov input_buffer_arg, #@msg_world
    call #send_string
    jmp #main

  skip_response
    jmp #main

  read_line
    mov buffer_pointer, input_buffer_arg
    mov character_count, #0

    read_char
      testp #RECEPTION_PIN wc
```

```
    if_nc jmp #read_char
    rdpin input_character, #RECEPTION_PIN
    shr input_character, #24
    and input_character, #$FF
    akpin #RECEPTION_PIN

    cmp input_character, ##$0D wz
    if_z   jmp #line_done
    cmp input_character, ##$08 wz
    if_z   jmp #handle_backspace

    wypin input_character, #TRANSMISSION_PIN
    call #flush_transmission
    cmp character_count, max_buffer_length wz
    if_z jmp #read_char
    wrbyte input_character, buffer_pointer
    add buffer_pointer, #1
    add character_count, #1
    jmp #read_char

handle_backspace
  cmp character_count, #0 wz
  if_z   jmp #read_char
  sub buffer_pointer, #1
  sub character_count, #1
  wypin ##$08, #TRANSMISSION_PIN
  wypin ##32, #TRANSMISSION_PIN
  wypin ##$08, #TRANSMISSION_PIN
  call #flush_transmission
  jmp #read_char

line_done
  wrbyte #0, buffer_pointer
  wypin ##$0D, #TRANSMISSION_PIN
  wypin ##$0A, #TRANSMISSION_PIN
  call #flush_transmission
  ret

send_string
  mov string_pointer, input_buffer_arg
  loop_send
    rdbyte output_character, string_pointer
    cmp output_character, #0 wz
    if_z jmp #done_send
    wypin output_character, #TRANSMISSION_PIN
    call #flush_transmission
    add string_pointer, #1
    jmp #loop_send

done_send
ret

flush_transmission
  rdpin transmission_busy, #TRANSMISSION_PIN wc
  if_c jmp #flush_transmission
  ret
```

```
  endprog
    jmp  #$

  input_character       res 1
  compare_character     res 1
  output_character      res 1
  buffer_pointer        res 1
  character_count       res 1
  input_buffer_arg      res 1
  max_buffer_length     res 1
  string_pointer        res 1
  compare_pointer       res 1
  transmission_busy     res 1
  input_buffer          res 32

  cmd_hello byte "hello", 0
  msg_world byte "p2 assembly!", 0
```

As user types, each character is immediately echoed back. Mistakes can be corrected using Backspace, which removes the character both visually and from the input buffer. Pressing Enter (CR) completes the input line, sending a proper Carriage Return + Line Feed (CRLF) to the terminal. The program then compares the typed line to the predefined command string "hello". If the input matches, the system responds by sending "world" back to the terminal; otherwise, it ignores the input and waits for a new line. All communication occurs at 2,000,000 baud, demonstrating real-time input handling, text comparison, and string output over UART using the Propeller 2 SmartPin system.

Tokenized Echo Command

We extend the UART functionality by introducing string tokenization, allowing the Propeller 2 to parse an input line into individual words or parameters. Users can type simple commands. By recognizing the first token as a command, the system can respond appropriately — in this case, sending the second token back to the terminal. This approach lays the groundwork for more advanced command-line interfaces and interactive serial control on the Propeller 2.

Program: token_echo.spin2

```
'token_echo.spin2
CON
```

```
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  RECEPTION_PIN     = 63
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

DAT
  org 0
  asmclk

  wrpin   ##(P_ASYNC_RX | P_HIGH_15K), #RECEPTION_PIN
  wxpin   ##BAUD_MODE, #RECEPTION_PIN
  dirh    #RECEPTION_PIN

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
  wypin   #1, #TRANSMISSION_PIN
  dirh    #TRANSMISSION_PIN

  main
    mov input_buffer_arg, #input_buffer
    mov max_length, #32
    call #read_line

    mov buffer_pointer, #input_buffer
    call #parse_two_tokens

    mov compare_ptr, #@cmd_echo
    mov string_pointer, token1_ptr

  .check_token1
    rdbyte input_character, string_pointer
    rdbyte compare_character, compare_ptr
    cmp input_character, compare_character wz
    if_nz jmp #main
    cmp input_character, #0 wz
    if_z jmp #.send_token2
    add string_pointer, #1
    add compare_ptr, #1
    jmp #.check_token1

  .send_token2
    mov input_buffer_arg, token2_ptr
    call #send_string
    jmp #main

  parse_two_tokens
    mov token1_ptr, buffer_pointer
    mov token2_ptr, #0

  .loop1
    rdbyte input_character, buffer_pointer
    cmp input_character, #0 wz
    if_z jmp #.done_tokens
    cmp input_character, ##32 wz
    if_z jmp #.found_space1
```

```
  add buffer_pointer, #1
  jmp #.loop1

.found_space1
  wrbyte #0, buffer_pointer
  add buffer_pointer, #1
  mov token2_ptr, buffer_pointer

.skip_spaces
  rdbyte input_character, buffer_pointer
  cmp input_character, ##32 wz
  if_z add buffer_pointer, #1
  if_nz jmp #.done_tokens
  jmp #.skip_spaces

.done_tokens
ret

read_line
  mov buffer_pointer, input_buffer_arg
  mov character_count, #0

.read_char
  testp #RECEPTION_PIN wc
  if_nc jmp #.read_char

  rdpin input_character, #RECEPTION_PIN
  akpin #RECEPTION_PIN

  shr input_character, #24
  and input_character, #$FF

  cmp input_character, ##$0D wz
  if_z jmp #.line_done

  cmp input_character, ##$08 wz
  if_z jmp #.handle_backspace

  cmp input_character, ##$7F wz
  if_z jmp #.handle_backspace

  cmp character_count, max_length wz
  if_z jmp #.read_char

  wypin input_character, #TRANSMISSION_PIN
  call #flush_tx

  wrbyte input_character, buffer_pointer
  add buffer_pointer, #1
  add character_count, #1
  jmp #.read_char

.handle_backspace
  cmp character_count, #0 wz
  if_z jmp #.read_char

  sub buffer_pointer, #1
```

```
  sub character_count, #1

  wypin ##$08, #TRANSMISSION_PIN
  call #flush_tx
  wypin ##32, #TRANSMISSION_PIN
  call #flush_tx
  wypin ##$08, #TRANSMISSION_PIN
  call #flush_tx

  jmp #.read_char

.line_done
  wrbyte #0, buffer_pointer
  wypin ##$0D, #TRANSMISSION_PIN
  call #flush_tx
  wypin ##$0A, #TRANSMISSION_PIN
  call #flush_tx
ret

send_string
  mov string_pointer, input_buffer_arg
.loop_send
  rdbyte input_character, string_pointer
  cmp input_character, #0 wz
  if_z jmp #.done_send
  wypin input_character, #TRANSMISSION_PIN
  call #flush_tx
  add string_pointer, #1
  jmp #.loop_send
.done_send
ret

flush_tx
  rdpin transmission_busy, #TRANSMISSION_PIN wc
  if_c jmp #flush_tx
  ret

endprog
  jmp  #$

input_character     res 1
compare_character   res 1
transmission_busy   res 1
buffer_pointer      res 1
character_count     res 1
input_buffer_arg    res 1
max_length          res 1
string_pointer      res 1
token1_ptr          res 1
token2_ptr          res 1
input_buffer        res 32
compare_ptr         res 1

cmd_echo byte "echo", 0
```

This code demonstrates a basic Propeller 2 command-line interface over UART. It reads a line of text from the terminal, echoing each character as it is typed and supporting backspace for corrections. When Enter is pressed, the input is split into two tokens. The first token is compared to the predefined command `"echo"`, and if it matches, the second token is sent back to the terminal. Lines that do not match are ignored, and the program loops to wait for new input. This example showcases interactive text input, simple string parsing, command recognition, and UART output — forming the foundation for flexible serial command interfaces.

Chapter 16: Introduction to SPI (Serial Peripheral Interface)

SPI, or Serial Peripheral Interface, is one of the most widely used communication protocols in microcontrollers. It provides a fast, full-duplex connection between a master device—typically the microcontroller—and one or more slave devices, such as EEPROMs, sensors, displays, flash chips, or DACs. Unlike UART, which transmits data asynchronously, SPI operates synchronously, meaning that data transmission is coordinated by a shared clock signal. This synchronization enables higher speeds and precise timing control. On the Propeller 2, SmartPins can manage SPI in hardware, making communication both efficient and flexible.

SPI communication relies on four main signals. SCLK is the clock line generated by the master, which synchronizes data transfer. MOSI (Master Out, Slave In) carries data sent from the master to the slave, while MISO (Master In, Slave Out) carries data from the slave back to the master. CS (Chip Select) is used by the master to enable communication with a specific slave device. Only one slave's SS line is active (low) at a time; when a slave's CS line is high, it ignores all SPI bus activity. Together, these four lines enable fast, full-duplex, and precisely timed data exchange between master and slave devices.

In SPI communication, the clock signal is like a steady beat or metronome that the master device creates to keep everyone in sync. Every "tick" of this clock tells the devices exactly when to send or read a bit of data. The timing of the data transfer is controlled by two settings: CPOL (Clock Polarity), which decides whether the clock starts high or low, and CPHA (Clock Phase), which decides whether data is read at the beginning or end of each tick. Together, these create four SPI modes: Mode 0 (clock idle low, data read on rising edge), Mode 1 (clock idle low, data read on falling edge), Mode 2 (clock idle high, data read on falling edge), and Mode 3 (clock idle high, data read on rising edge). Both master and slave must use the same mode to communicate correctly.

In SPI, data transfer begins when the master selects a slave by pulling its CS line low. The master then sends data bits to the slave through MOSI while at the same time reading bits from the slave through MISO. Each tick of the clock signal shifts one bit in both directions. After the transmission is complete, the

master sets the CS line high to deselect the slave. SPI is full-duplex, meaning it can send and receive data at the same time. Even if the master only wants to send data, it still receives dummy bits from the slave during the process.

Learning SPI is valuable because it provides fast, versatile, and precise communication between the Propeller 2 and a wide range of peripherals. SPI supports full-duplex transfers at high speeds—often tens of megahertz—making it ideal for time-sensitive tasks like streaming graphics to a display or reading high-speed sensor data. Its simple hardware design relies on basic logic-level pins without complex addressing, allowing devices to be controlled directly with shifts and toggles or by using the Propeller 2's SmartPins for hardware-timed communication. SPI is also widely supported across industry-standard devices, including flash memory, EEPROMs, ADCs, DACs, displays, real-time clocks, and various sensors, giving your programs universal connectivity. Mastering SPI not only enables efficient control of these devices but also lays the foundation for more advanced protocols such as QSPI, SD card SPI mode, or HyperBus memory interfaces.

Example

Here's the *general flow* of SPI operations in assembly (pseudocode style):

```
set CS low                  ' select device
repeat N bits
  output bit on MOSI
toggle clock
input bit from MISO
set CS high                 ' deselect device
```

In practice, this can be implemented using **SmartPins** for hardware-driven SPI timing.

Reading the JEDEC ID from SPI Flash

Flash memory is a **non-volatile storage medium**, meaning it retains data even when power is removed. It is widely used in embedded systems for boot memory, firmware updates, data logging, and configuration storage. On the Propeller 2, SPI Flash typically holds the program image for booting, but user code can also access it by reading and writing bytes through SPI commands. A common first step in working with SPI Flash is reading the **JEDEC ID**, a standardized three-byte identifier that provides the manufacturer ID, memory type, and capacity code. This not only confirms that the SPI bus is functioning but also ensures successful communication with the Flash device.

Most SPI Flash chips support the **Read JEDEC ID** command (`0x9F`). When the Propeller 2 sends this command, the Flash responds with three bytes: the first byte is the Manufacturer ID (for example, `0xEF` for Winbond or `0xC2` for Macronix), the second byte is the Memory Type ID, and the third byte is the Capacity Code, which indicates the chip's size (for example, `0x18 = 128 Mbit = 16 MB`). Reading the JEDEC ID is valuable for hardware verification, identifying the exact Flash device for bootloader purposes, and laying the foundation for memory operations such as reading, writing, or erasing sectors. This method works with nearly all SPI Flash chips, making it a reliable first step in SPI Flash programming.

Program: spi_display_jedec.spin2

```
'spi_display_jedec.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  RECEPTION_PIN     = 63
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

  SPI_CS   = 61
  SPI_CLK  = 60
  SPI_MOSI = 59
  SPI_MISO = 58

  SPI_CLK_MODE = P_TRANSITION + P_OE
  SPI_MOSI_MODE = P_SYNC_TX + P_OE | ((SPI_CLK - SPI_MOSI) & %111) << 24
  SPI_MISO_MODE = P_SYNC_RX | ((SPI_CLK - SPI_MISO) & %111) << 24
  SPI_CLK_DIV = (_CLKFREQ / 100_000_000) #> 4

  SPI_8BIT_CONTROL = $27
```

```
DAT
  org 0
  asmclk

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
  wypin   #1, #TRANSMISSION_PIN
  dirh    #TRANSMISSION_PIN

  wrpin   ##SPI_CLK_MODE, #SPI_CLK
  wrpin   ##SPI_MOSI_MODE, #SPI_MOSI
  wrpin   ##SPI_MISO_MODE, #SPI_MISO
  wxpin   ##SPI_CLK_DIV, #SPI_CLK
  drvl    #SPI_CLK

  wxpin   ##SPI_8BIT_CONTROL, #SPI_MOSI
  wxpin   ##SPI_8BIT_CONTROL, #SPI_MISO

  call    #print_jedec

  jmp     #endprog


  print_jedec
    drvl    #SPI_CS

    mov spi_command, #$9F
    call #spi_transmit

    call #spi_receive
    mov     pr0, spi_reception
    call    #print_hex_byte
    mov     uart_transmission, #" "
    call    #uart_transmit_char

    call #spi_receive
    mov     pr0, spi_reception
    call    #print_hex_byte
    mov     uart_transmission, #" "
    call    #uart_transmit_char

    call #spi_receive
    mov     pr0, spi_reception
    call    #print_hex_byte
    mov     uart_transmission, #" "
    call    #uart_transmit_char

    drvh    #SPI_CS

    mov     uart_transmission, #13
    call    #uart_transmit_char
    mov     uart_transmission, #10
    call    #uart_transmit_char
  ret

  spi_transmit
    shl spi_command,#24
```

```
  rev spi_command

  wypin spi_command,#SPI_MOSI
  drvl #SPI_MOSI

  wypin #16,#SPI_CLK

  .waitRdy
    testp #SPI_CLK wz
    if_nz jmp #.waitRdy
  ret

spi_receive
  drvl    #SPI_MISO
  fltl    #SPI_MOSI
 wypin   #16,#SPI_CLK
.waitDone
  testp   #SPI_MISO wz
  if_nz   jmp #.waitDone
  rdpin   spi_reception,#SPI_MISO
  rev     spi_reception
  zerox   spi_reception,#7
  ret

print_hex_byte
  mov hexadecimal_copy, pr0
  shr pr0, #4
  and pr0, #$F
  call #print_nibble
  mov pr0, hexadecimal_copy
  and pr0, #$F
  call #print_nibble
  ret

print_nibble
  cmp pr0, #10 wc
  if_b add pr0, #"0"
  if_nc add pr0, #"A"-10
  mov uart_transmission, pr0
  call #uart_transmit_char
  ret

uart_transmit_char
  wypin   uart_transmission, #TRANSMISSION_PIN
  .txwait
  rdpin   pr2, #TRANSMISSION_PIN wc
  if_c    jmp #.txwait
  ret

endprog
  jmp  #$

spi_command         res 1
spi_reception       res 1
uart_transmission   res 1
hexadecimal_copy    res 1
```

The program spi_display_jedec.spin2 initializes the Propeller 2 for both UART and SPI communication to read the JEDEC ID from an SPI flash device and display it over a serial terminal. The CON section defines system parameters, including the system clock _CLKFREQ, UART baud rate BAUD_RATE, and pin assignments for UART (TRANSMISSION_PIN, RECEPTION_PIN) and SPI (SPI_CS, SPI_CLK, SPI_MOSI, SPI_MISO). It also configures smartpin modes for asynchronous UART and synchronous SPI, sets the SPI clock divider (SPI_CLK_DIV) to control the SPI speed, and defines 8-bit SPI transfers. The DAT section contains the main program, which initializes the UART and SPI smartpins, sets the SPI pins low or high as needed, and ensures proper pin directions.

The print_jedec routine selects the SPI flash by pulling CS low, sends the JEDEC ID command (0x9F) via spi_transmit, and receives three bytes using spi_receive. Each received byte is reversed to account for MSB-first SPI convention, masked to extract valid bits, and printed as hexadecimal using print_hex_byte and print_nibble. Between bytes, spaces are transmitted over UART for readability. After all three bytes are sent, CS is set high with drvh, and newline characters are transmitted to finish the output. The spi_transmit and spi_receive routines handle full-duplex SPI communication at the hardware level using smartpins, including clock pulses and bit reversal. The print_hex_byte and print_nibble routines convert numeric SPI data into ASCII hexadecimal characters for terminal display, and uart_transmit_char ensures each character is fully transmitted before continuing.

As a result, when executed with a typical SPI flash such as a Winbond 16 MB device, the program outputs the JEDEC ID in hexadecimal format (for example, EF 40 18), where the first byte identifies the manufacturer, the second indicates memory type, and the third specifies capacity. This demonstrates how the Propeller 2 can read SPI flash identification data and display it over UART using smartpin hardware for efficient asynchronous and synchronous serial communication.

Reading from SPI Flash

The next step is to focus on **reading a single byte from an SPI Flash chip** at a specific address. This operation introduces the basic SPI communication commands needed to access flash memory and serves as the foundation for more advanced tasks, such as reading larger memory blocks, erasing sectors, or writing new data. By mastering byte-level reads, you gain the skills necessary to interact with flash memory for firmware updates, configuration storage, or other embedded system applications. Understanding this process is essential for safely and efficiently accessing and managing non-volatile memory.

To read a byte from an SPI Flash chip, the microcontroller first **pulls the chip select (CS) line low** to select the device. It then sends the `0x03 READ` command over SPI, followed by the 24-bit address of the memory location to access. The flash chip responds by sending the requested byte back on the MISO line. After the byte is received, the CS line is set high to deselect the chip, and the data can be printed over UART to verify the operation. This sequence establishes the core procedure for all SPI Flash interactions, forming the basis for reading larger blocks, updating memory, or performing other flash operations.

Program: spi_read_byte.spin2

```
'spi_read_byte.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  RECEPTION_PIN     = 63
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

  SPI_CS   = 61
  SPI_CLK  = 60
  SPI_MOSI = 59
  SPI_MISO = 58

  SPI_CLK_MODE   = P_TRANSITION + P_OE
  SPI_MOSI_MODE  = P_SYNC_TX + P_OE | ((SPI_CLK - SPI_MOSI) & %111) << 24
  SPI_MISO_MODE  = P_SYNC_RX | ((SPI_CLK - SPI_MISO) & %111) << 24
  SPI_CLK_DIV    = (_CLKFREQ / 100_000_000) #> 4
  SPI_8BIT_CONTROL  = $27

DAT
  org 0
  asmclk

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
  wypin   #1, #TRANSMISSION_PIN
```

```
dirh    #TRANSMISSION_PIN

wrpin   ##SPI_CLK_MODE, #SPI_CLK
wrpin   ##SPI_MOSI_MODE, #SPI_MOSI
wrpin   ##SPI_MISO_MODE, #SPI_MISO
wxpin   ##SPI_CLK_DIV, #SPI_CLK
drvl    #SPI_CLK

wxpin   ##SPI_8BIT_CONTROL, #SPI_MOSI
wxpin   ##SPI_8BIT_CONTROL, #SPI_MISO

mov flash_address, ##0
mov page_size, ##255

call #read_flash_256
jmp  #endprog

read_flash_256
  drvl #SPI_CS
  mov spi_command, #$03
  call #spi_transmit

  mov spi_command, flash_address
  shr spi_command, #16
  and spi_command, #$FF
  call #spi_transmit

  mov spi_command, flash_address
  shr spi_command, #8
  and spi_command, #$FF
  call #spi_transmit

  mov spi_command, flash_address
  and spi_command, #$FF
  call #spi_transmit

  mov byte_count, ##512


.read_loop
  call #spi_receive
  mov uart_transmission, spi_reception
  call #uart_transmit_char
  djnz byte_count, #.read_loop

  drvh #SPI_CS
  ret

spi_transmit
  shl spi_command,#24
  rev spi_command
  wypin spi_command,#SPI_MOSI
  drvl #SPI_MOSI
  wypin #16,#SPI_CLK
  waitx ##1
  .waitRdy
    testp   #SPI_CLK wz
```

```
    if_nz   jmp #.waitRdy
  ret

spi_receive
  drvl #SPI_MISO
  fltl #SPI_MOSI
  wypin #16,#SPI_CLK
  waitx ##1
  .waitDone
    testp #SPI_CLK wz
    if_nz  jmp #.waitDone
    rdpin spi_reception,#SPI_MISO
    rev spi_reception
    zerox spi_reception,#7
  ret

uart_transmit_char
  wypin uart_transmission, #TRANSMISSION_PIN
  .txwait
    rdpin pr2, #TRANSMISSION_PIN wc
    if_c  jmp #.txwait
  ret

endprog
  jmp  #$

spi_command        res 1
flash_address      res 1
byte_count         res 1
spi_reception      res 1
uart_transmission  res 1
page_size          res 1
```

The program begins by initializing `flash_address` to 0x000000, pointing to the first byte of flash memory, and `page_size` to 255, which sets up the block size for reading. The main routine calls `read_flash_256` to handle the SPI read, then jumps to `endprog` when complete. This structure separates setup from the actual memory reading, making the code modular and reusable for different addresses or page sizes.

In `read_flash_256`, CS is pulled low to select the flash chip, and the 0x03 READ command is sent to initiate reading. The 24-bit address is transmitted in three bytes (MSB first) to indicate where to start reading. A loop then reads 512 bytes sequentially via `spi_receive`, storing each byte in `spi_reception` and sending it over UART with `uart_transmit_char`. Finally, CS is pulled high to end the transaction.

The `spi_transmit` subroutine sends a byte over SPI by shifting and reversing it, writing it to MOSI, and pulsing the clock until the transfer

completes. Conversely, `spi_receive` reads a byte from MISO, reversing it to ensure correct interpretation. `uart_transmit_char` sends a byte over UART, waiting for the previous transmission to finish, ensuring clean output.

When run on a Propeller 2 connected to SPI Flash, the program reads 512 consecutive bytes starting from address 0x000000 and prints them over UART. If the flash contains ASCII text, readable characters appear; otherwise, the output may look like binary data, confirming that SPI communication and memory reading work correctly.

Writing to SPI Flash

Flash memory often includes **hardware protection features** to prevent accidental modification of stored data. Most SPI Flash chips provide a **Status Register** (accessed via the `RDSR` command) that indicates whether writing is allowed and which memory sectors are protected. Key fields include the **WEL (Write Enable Latch)** bit, which must be set using the `0x06 Write Enable` command before any write or erase operation; the **BPx bits**, which define block-level protection for specific sectors; and the **SRWD (Status Register Write Protect)** bit, which locks the status register itself. Before performing any write or erase operations, it is essential to check and configure these bits to ensure the flash memory is in a writable state.

Enabling writes to SPI Flash is straightforward but critical for safe memory management. First, the **CS line** is pulled low to select the chip, then the **Write Enable command (0x06)** is sent via SPI to set the WEL bit. After the command, CS is pulled high, and the Status Register can optionally be read to verify that WEL is set. As long as WEL remains cleared, the chip will reject any write or erase commands, protecting firmware, configuration, and critical data from accidental modification. This controlled approach allows precise updates to specific blocks or sectors, ensuring safe firmware updates, bootloader execution, and configuration storage without risking corruption of the device.

Writing data to SPI Flash is a multi-step process because the chip cannot overwrite memory directly without preparation. First, you must enable writing by sending the Write Enable (0x06) command, which sets the WEL (Write Enable Latch) bit in the Status Register. Next, if necessary, you erase a sector (typically 4 KB) using the Sector Erase command (0x20), which resets all bytes

in that sector to 0xFF. Then, the Byte or Page Write command (0x02) is sent, including the target address and the data byte(s) to be written. After issuing a write, you must wait for the operation to complete by polling the Status Register until the BUSY bit clears. Optionally, you can verify the write by reading the byte(s) back to ensure the correct value was stored. This structured process ensures data integrity and prevents accidental overwrites.

Many SPI Flash chips organize memory into pages, usually 256 bytes in size. Writing an entire page at once is much more efficient than writing single bytes individually. Single-byte writes are slower and can cause more wear on the memory over time, whereas page writes allow you to send up to 256 consecutive bytes in a single SPI transaction, improving speed and reducing wear. This method is ideal for bulk data updates, such as firmware uploads or storing large configuration tables. Flash page writes **wrap around at page boundary** (256 bytes). Ensure you don't cross pages unintentionally.

Program: spi_write_byte.spin2

```
'spi_write_byte.spin2
CON
  _CLKFREQ          = 180_000_000
  BAUD_RATE         = 2_000_000
  TRANSMISSION_PIN  = 62
  RECEPTION_PIN     = 63
  BIT_PERIOD        = (_CLKFREQ / BAUD_RATE)
  BAUD_MODE         = (BIT_PERIOD << 16) | (8 - 1)

  SPI_CS   = 61
  SPI_CLK  = 60
  SPI_MOSI = 59
  SPI_MISO = 58

  SPI_CLK_MODE   = P_TRANSITION + P_OE
  SPI_MOSI_MODE  = P_SYNC_TX + P_OE | ((SPI_CLK - SPI_MOSI) & %111) << 24
  SPI_MISO_MODE  = P_SYNC_RX | ((SPI_CLK - SPI_MISO) & %111) << 24
  SPI_CLK_DIV    = (_CLKFREQ / 100_000_000) #> 4
  SPI_8BIT_CONTROL  = $27

DAT
  org 0
  asmclk

  wrpin   ##(P_ASYNC_TX | P_OE), #TRANSMISSION_PIN
  wxpin   ##BAUD_MODE, #TRANSMISSION_PIN
  wypin   #1, #TRANSMISSION_PIN
  dirh    #TRANSMISSION_PIN

  wrpin   ##SPI_CLK_MODE, #SPI_CLK
  wrpin   ##SPI_MOSI_MODE, #SPI_MOSI
```

```
wrpin   ##SPI_MISO_MODE, #SPI_MISO
wxpin   ##SPI_CLK_DIV, #SPI_CLK
drvl    #SPI_CLK

wxpin   ##SPI_8BIT_CONTROL, #SPI_MOSI
wxpin   ##SPI_8BIT_CONTROL, #SPI_MISO

mov flash_address, ##0
mov page_size, ##255

call #spi_write_enable
call #spi_erase_sector
call #spi_write_enable

mov flash_address, #$0
mov spi_reception, #"a"
call #spi_byte_write

jmp  #endprog

spi_write_enable
  drvl #SPI_CS
  mov spi_command, #$06
  call #spi_transmit
  drvh #SPI_CS
  .wait_wel
    call #spi_read_status_register
    testb status_register, #2 wz
    if_z jmp #.wait_wel
  ret

spi_read_status_register
  drvl #SPI_CS
  mov spi_command, #$05
  call #spi_transmit
  call #spi_receive
  mov status_register, spi_reception
  drvh #SPI_CS
  ret

spi_erase_sector
  drvl #SPI_CS
  mov spi_command, #$20
  call #spi_transmit
  call #spi_send_address24
  drvh #SPI_CS
  .wait_busy
    call #spi_read_status_register
    testb status_register, #0 wz
    if_z jmp #.wait_busy
  ret

spi_send_address24
  mov spi_command, flash_address
  shr spi_command, #16
  and spi_command, #$FF
  call #spi_transmit
```

```
  mov spi_command, flash_address
  shr spi_command, #8
  and spi_command, #$FF
  call #spi_transmit

  mov spi_command, flash_address
  and spi_command, #$FF
  call #spi_transmit
  ret

spi_byte_write
  drvl #SPI_CS
  mov spi_command, #$02
  call #spi_transmit
  call #spi_send_address24
  mov byte_count, #256
  .page_loop
    mov spi_command, spi_reception
    call #spi_transmit
    djnz byte_count, #.page_loop
    drvh #SPI_CS
  ret

spi_transmit
  shl spi_command,#24
  rev spi_command
  wypin spi_command,#SPI_MOSI
  drvl #SPI_MOSI
  wypin #16,#SPI_CLK
  waitx ##1
  .waitRdy
    testp   #SPI_CLK wz
    if_nz   jmp #.waitRdy
  ret

spi_receive
  drvl #SPI_MISO
  fltl #SPI_MOSI
  wypin #16,#SPI_CLK
  waitx ##1
  .waitDone
    testp #SPI_CLK wz
    if_nz  jmp #.waitDone
    rdpin spi_reception,#SPI_MISO
    rev spi_reception
    zerox spi_reception,#7
  ret

uart_transmit_char
  wypin uart_transmission, #TRANSMISSION_PIN
  .txwait
    rdpin pr2, #TRANSMISSION_PIN wc
    if_c  jmp #.txwait
  ret

endprog
```

```
    jmp  #$

  spi_command        res 1
  flash_address      res 1
  byte_count         res 1
  spi_reception      res 1
  uart_transmission  res 1
  page_size          res 1
  status_register    res 1
```

The program starts by initializing `flash_address` to 0x000000, indicating the first memory location in the SPI Flash, and `page_size` to 255, defining the number of bytes in a flash page. This setup ensures the program knows where to begin writing and how many bytes each write operation will handle.

The main routine writes data to the SPI Flash in a safe sequence. It first calls `spi_write_enable` to allow write operations, then erases the sector containing the target address using `spi_erase_sector`. After enabling writing again, it writes 256 bytes of the character `"e"` to address 0x00 using `spi_byte_write`. This ensures the flash memory is cleared and ready for new data, preventing accidental overwrites.

The `spi_write_enable` subroutine sends the 0x06 Write Enable command to set the WEL (Write Enable Latch) in the status register. It repeatedly checks the WEL bit until it confirms the flash is ready to accept write commands. The `spi_read_status_register` routine sends the 0x05 command to retrieve the status register, which holds important flags like WEL and WIP (Write In Progress), indicating whether the chip is ready or busy.

`spi_erase_sector` pulls CS low, sends the 0x20 Sector Erase command, transmits the 24-bit sector address via `spi_send_address24`, and then pulls CS high to start the internal erase. It loops, polling the status register until the WIP bit clears, ensuring the sector is fully erased before writing. The `spi_send_address24` routine splits the 24-bit address into three bytes (MSB first) and sends them over SPI, required for any read, write, or erase operation.

`spi_byte_write` programs a full page (256 bytes) starting at `flash_address`. It sends the Page Program command (0x02), the 24-bit address, and then loops to transmit all 256 bytes using the value in `spi_reception`. CS is pulled high at the end to complete the transaction, allowing efficient bulk writing rather than sending single bytes one at a time.

The `spi_transmit` routine shifts and reverses a byte, writes it to the MOSI smartpin, triggers 16 clock pulses on SPI_CLK, and waits for completion,

ensuring correct SPI timing. `spi_receive` reads a byte from MISO, floats MOSI, triggers clock pulses, and stores the reversed, aligned result in `spi_reception`. `uart_transmit_char` sends a byte over UART, waiting for the previous transmission to finish to ensure reliable output.

When executed on a Propeller 2, this program erases the first sector of SPI Flash, enables writing, and writes a full page of `"e"` at address `0x00`. The data is stored permanently until overwritten, and if connected to a UART terminal, it can display transmitted data for verification. This demonstrates safe, efficient SPI Flash programming.

Chapter 17 – Practice Programs

These are programs you should implement on your own to practice and reinforce what you've learned about the P2ASM.

- **ASCII Table Printer** – Display all 128 ASCII characters.
- **Echo** – Read characters from UART and echo them back.
- **Sleep** – Pause program execution for a set duration.
- **Uptime** – Show the time since the program started.
- **Basic Calculator** – Perform simple arithmetic operations from UART input.
- **List COGs 0-7** – Display the status of all Propeller 2 cores.
- **Editor** – Simple text editor using UART.

Looking Ahead: From Learning Examples to Production-Ready Code

Throughout this text, you've explored fundamental concepts in Propeller 2 programming — from UART communication and string parsing to bitwise operations and shifting. The programs presented are designed for clarity, learning, and experimentation. They intentionally simplify hardware interactions, use fixed-size buffers, and rely on busy loops or direct manipulation to make the examples easy to follow.

In a production environment, however, these programs would require enhancements to ensure safety, efficiency, and maintainability. For instance, error handling would need to be robust: input values should be validated, buffers protected against overflow, and hardware responses monitored for unexpected states. Blocking loops, such as those waiting on UART RX or TX, would be replaced with interrupt-driven routines or hardware FIFOs, allowing the CPU to perform other tasks concurrently.

Buffer management would be more sophisticated, using circular buffers or dynamic allocation where appropriate, and all memory accesses would include rigorous bounds checking. Command parsing would scale to support multiple

commands, optional parameters, and case-insensitive comparison, with clear feedback for invalid input. Similarly, bitwise operations and shifting routines would be modularized, using descriptive constants for masks, flags, and bit positions to enhance readability and maintainability.

Finally, production code emphasizes modular design and separation of concerns. Low-level drivers handle hardware-specific tasks, utility routines manage data manipulation, and the main program coordinates the overall flow. This structure makes code easier to expand, debug, and maintain over time, whether controlling LEDs, reading sensors, or managing complex communication protocols.

In short, the programs you've written form the foundation of embedded programming skills. With careful attention to robustness, efficiency, and modularity, these foundations can be built into production-ready systems capable of reliable, real-world operation.