# SPIN2 Terminal Window User's Manual

**A Complete Guide to P2 Terminal Interfaces**

## Table of Contents

1. [Introduction](#introduction)
2. [Getting Started](#getting-started)
3. [Terminal Configuration](#terminal-configuration)
4. [Control Characters Reference](#control-characters-reference)
5. [Programming Patterns](#programming-patterns)
6. [Advanced Techniques](#advanced-techniques)
7. [Real-World Projects](#real-world-projects)
8. [Troubleshooting](#troubleshooting)
9. [Learning Exercises](#learning-exercises)
10. [Best Practices](#best-practices)

## Introduction

The SPIN2 Terminal Window system provides powerful text-based user interface capabilities built directly into the Parallax Propeller 2. Unlike external terminal emulators, P2's terminal system is integrated with the DEBUG framework, offering real-time display updates, multiple terminal instances, and seamless integration with your SPIN2 programs.

### What You'll Learn

By the end of this manual, you'll be able to: - Create interactive text-based user interfaces - Build real-time data displays and monitoring systems - Implement menu systems and user input handling - Debug programs with organized visual output - Design educational demonstrations and experiments

### Key Capabilities

**Display Features**: - **Flexible Resolution**: 1-300 columns × 1-200 rows - **Variable Font Sizes**: 6-200 point fonts for any viewing distance - **Color Schemes**: 4 simultaneous color combinations - **Real-Time Updates**: Live display changes during program execution

**Integration Benefits**: - **Native DEBUG Support**: Direct output from SPIN2 DEBUG statements - **Multi-Terminal**: Multiple independent terminal windows - **COG Independence**: Non-blocking operation across COGs - **Zero Dependencies**: No external software required

### When to Use Terminal Windows

**Ideal Applications**: - Interactive control panels and status displays - Educational programming demonstrations  
- Development debugging and program tracing - Data logging and sensor monitoring interfaces - Menu-driven configuration systems

**Not Ideal For**: - High-resolution graphics (use bitmap displays instead) - High-speed data visualization (consider scope displays) - Complex layouts (terminal is character-grid based)

## Getting Started

### Your First Terminal Window

Let's start with the simplest possible terminal example:

CON \_clkfreq = 10\_000\_000  
  
PUB main()  
 ' Create a basic terminal window  
 DEBUG(`TERM MyTerminal SIZE 40 20 TEXTSIZE 14)  
   
 ' Send some text to the terminal  
 DEBUG("`TERM.str(string("Hello, P2 Terminal!"))")  
   
 ' Keep the program running to see the output  
 repeat  
 waitms(1000)

**What This Does**: 1. **DEBUG(TERM MyTerminal ...)\*\*: Creates a new terminal window named "MyTerminal" 2. \*\*SIZE 40 20\*\*: Makes the terminal 40 characters wide by 20 lines tall 3. \*\*TEXTSIZE 14\*\*: Sets the font size to 14 points 4. \*\*DEBUG("TERM.str(...))**: Sends text to the terminal

### Understanding Terminal Names

Each terminal window needs a unique name. This allows you to: - Have multiple terminals open simultaneously - Send different content to different terminals - Organize output by function (status, debug, data, etc.)

PUB multiple\_terminals()  
 ' Create specialized terminals  
 DEBUG(`TERM Status SIZE 30 10 TEXTSIZE 12)  
 DEBUG(`TERM Debug SIZE 60 25 TEXTSIZE 10)  
 DEBUG(`TERM Data SIZE 20 15 TEXTSIZE 16)  
   
 ' Send content to specific terminals  
 DEBUG("`Status.str(string("System Ready"))")  
 DEBUG("`Debug.str(string("Starting main loop"))")  
 DEBUG("`Data.dec(42)")

### Basic Output Methods

SPIN2 provides several methods for sending data to terminals:

|  |  |  |
| --- | --- | --- |
| Method | Purpose | Example |
| .str() | Send string | DEBUG("TERM.str(string("Hello")))| |.dec()| Send decimal number |DEBUG("TERM.dec(123)") |
| .hex() | Send hexadecimal | DEBUG("TERM.hex($FF)")| |.bin()| Send binary |DEBUG("TERM.bin(%1010)") |
| .char() | Send single character | DEBUG("TERM.char(65)")` |

### Simple Interactive Example

Here's a more interesting example that shows real-time updates:

PUB counter\_demo() | count  
 DEBUG(`TERM Counter SIZE 25 8 TEXTSIZE 16)  
   
 count := 0  
 repeat  
 ' Clear the terminal and show current count  
 DEBUG("`Counter.char(0)") ' Clear screen  
 DEBUG("`Counter.str(string("Counter: "))")  
 DEBUG("`Counter.dec(count)")  
   
 count++  
 waitms(500) ' Update twice per second

**Learning Points**: - **char(0)**: Clears the terminal and homes the cursor - **Real-time updates**: The display changes as the program runs - **Readable formatting**: Combining text labels with dynamic values

## Terminal Configuration

### Size Configuration

Terminal size determines both the character capacity and the physical window size:

' Small terminal for status info  
DEBUG(`TERM Status SIZE 20 5 TEXTSIZE 18)  
  
' Large terminal for detailed output  
DEBUG(`TERM Main SIZE 80 30 TEXTSIZE 12)  
  
' Wide terminal for data tables  
DEBUG(`TERM Data SIZE 120 15 TEXTSIZE 10)

**Size Guidelines**: - **20x5 to 40x10**: Status displays, simple counters - **40x20 to 60x25**: General purpose, menu systems - **80x30+**: Code debugging, detailed logs - **120x40+**: Data analysis, complex tables

### Font Size Selection

Font size affects both readability and how much content fits:

' Large fonts for presentations  
DEBUG(`TERM Demo SIZE 30 15 TEXTSIZE 24)  
  
' Medium fonts for general use   
DEBUG(`TERM Work SIZE 40 20 TEXTSIZE 14)  
  
' Small fonts for dense information  
DEBUG(`TERM Log SIZE 80 40 TEXTSIZE 8)

**Font Size Guidelines**: - **20-30 points**: Presentations, demonstrations, elderly users - **12-18 points**: General development work, normal viewing - **8-12 points**: Dense data, debugging output, small screens - **6-8 points**: Maximum information density, young eyes only

### Color Schemes

Each terminal supports 4 color combinations that you can switch between:

PUB color\_demo()  
 DEBUG(`TERM Colors SIZE 30 12 TEXTSIZE 14)  
   
 ' Configure color schemes (implementation varies by environment)  
 DEBUG(`Colors COLOR 0 $FFFFFF $000000) ' White on black  
 DEBUG(`Colors COLOR 1 $00FF00 $000000) ' Green on black   
 DEBUG(`Colors COLOR 2 $FFFF00 $0000FF) ' Yellow on blue  
 DEBUG(`Colors COLOR 3 $FF0000 $FFFF00) ' Red on yellow  
   
 ' Use different color schemes  
 DEBUG("`Colors.char(4)") ' Select scheme 0  
 DEBUG("`Colors.str(string("Normal text", 13, 10))")  
   
 DEBUG("`Colors.char(5)") ' Select scheme 1  
 DEBUG("`Colors.str(string("Success message", 13, 10))")  
   
 DEBUG("`Colors.char(6)") ' Select scheme 2  
 DEBUG("`Colors.str(string("Warning message", 13, 10))")  
   
 DEBUG("`Colors.char(7)") ' Select scheme 3  
 DEBUG("`Colors.str(string("Error message", 13, 10))")

**Color Strategy**: - **Scheme 0**: Normal text (default colors) - **Scheme 1**: Success/positive information (green) - **Scheme 2**: Warnings/cautions (yellow/orange) - **Scheme 3**: Errors/critical alerts (red)

## Control Characters Reference

### Essential Control Characters

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Character | Function | Usage |
| 0 | NUL | Clear & Home | DEBUG("TERM.char(0)")| | 1 | SOH | Home Only |DEBUG("TERM.char(1)") |
| 2 | STX | Set Column | DEBUG("TERM.char(2, 10)")| | 3 | ETX | Set Row |DEBUG("TERM.char(3, 5)") |
| 4 | EOT | Color #0 | DEBUG("TERM.char(4)")| | 5 | ENQ | Color #1 |DEBUG("TERM.char(5)") |
| 6 | ACK | Color #2 | DEBUG("TERM.char(6)")| | 7 | BEL | Color #3 |DEBUG("TERM.char(7)") |
| 8 | BS | Backspace | DEBUG("TERM.char(8)")| | 9 | TAB | Tab |DEBUG("TERM.char(9)") |
| 10 | LF | Line Feed | DEBUG("TERM.char(10)")| | 13 | CR | Carriage Return |DEBUG("TERM.char(13)") |

### Positioning and Movement

**Absolute Positioning**:

' Move to column 10, row 5  
DEBUG("`TERM.char(2, 10)") ' Set column to 10  
DEBUG("`TERM.char(3, 5)") ' Set row to 5  
DEBUG("`TERM.str(string("Text at (10,5)"))")

**Relative Movement**:

' Home cursor and build a simple display  
DEBUG("`TERM.char(1)") ' Home cursor  
DEBUG("`TERM.str(string("Line 1", 13, 10))") ' CR + LF  
DEBUG("`TERM.str(string("Line 2", 13, 10))")  
DEBUG("`TERM.char(8, 8, 8)") ' Backspace 3 times  
DEBUG("`TERM.str(string("\*\*\*"))") ' Overwrite last chars

### Screen Management

**Clearing Operations**:

PUB clear\_operations()  
 DEBUG(`TERM Demo SIZE 40 15 TEXTSIZE 14)  
   
 ' Fill screen with content  
 repeat 10  
 DEBUG("`Demo.str(string("Sample line of text", 13, 10))")  
   
 waitms(2000)  
   
 ' Clear everything and start fresh  
 DEBUG("`Demo.char(0)") ' Clear and home  
 DEBUG("`Demo.str(string("Screen cleared!"))")

**Home vs Clear+Home**: - **Home only** (char(1)): Cursor to top-left, content remains - **Clear & Home** (char(0)): Erase all content AND move cursor to top-left

## Programming Patterns

### Pattern 1: Status Display

A common pattern for showing system status:

PUB status\_display() | temperature, pressure, status  
 DEBUG(`TERM Status SIZE 35 12 TEXTSIZE 14)  
   
 repeat  
 ' Read sensor values (simulated)  
 temperature := get\_temperature()  
 pressure := get\_pressure()  
 status := get\_system\_status()  
   
 ' Update display  
 DEBUG("`Status.char(0)") ' Clear  
 DEBUG("`Status.char(4)") ' Normal color  
 DEBUG("`Status.str(string("=== SYSTEM STATUS ===", 13, 10))")  
   
 ' Temperature with color coding  
 DEBUG("`Status.str(string("Temperature: "))")  
 if temperature > 80  
 DEBUG("`Status.char(7)") ' Error color  
 elseif temperature > 70  
 DEBUG("`Status.char(6)") ' Warning color  
 else  
 DEBUG("`Status.char(5)") ' Success color  
 DEBUG("`Status.dec(temperature)")  
 DEBUG("`Status.str(string("°F", 13, 10))")  
   
 ' Pressure display  
 DEBUG("`Status.char(4)") ' Back to normal  
 DEBUG("`Status.str(string("Pressure: "))")  
 DEBUG("`Status.dec(pressure)")  
 DEBUG("`Status.str(string(" PSI", 13, 10))")  
   
 ' System status  
 DEBUG("`Status.str(string("Status: "))")  
 if status == STATUS\_OK  
 DEBUG("`Status.char(5)") ' Success color  
 DEBUG("`Status.str(string("NORMAL"))")  
 else  
 DEBUG("`Status.char(7)") ' Error color  
 DEBUG("`Status.str(string("FAULT"))")  
   
 waitms(1000) ' Update every second

### Pattern 2: Menu System

Interactive menu with user selection:

PUB menu\_system() | choice  
 DEBUG(`TERM Menu SIZE 40 15 TEXTSIZE 16)  
   
 repeat  
 show\_menu()  
 choice := get\_user\_choice()  
 handle\_choice(choice)  
  
PRI show\_menu()  
 DEBUG("`Menu.char(0)") ' Clear screen  
 DEBUG("`Menu.char(4)") ' Normal color  
 DEBUG("`Menu.str(string("╔══════════════════════════════════╗", 13, 10))")  
 DEBUG("`Menu.str(string("║ MAIN MENU ║", 13, 10))")  
 DEBUG("`Menu.str(string("╠══════════════════════════════════╣", 13, 10))")  
 DEBUG("`Menu.str(string("║ 1. Start System ║", 13, 10))")  
 DEBUG("`Menu.str(string("║ 2. Configure Settings ║", 13, 10))")  
 DEBUG("`Menu.str(string("║ 3. View Status ║", 13, 10))")  
 DEBUG("`Menu.str(string("║ 4. Run Diagnostics ║", 13, 10))")  
 DEBUG("`Menu.str(string("║ 5. Exit ║", 13, 10))")  
 DEBUG("`Menu.str(string("╚══════════════════════════════════╝", 13, 10))")  
 DEBUG("`Menu.str(string("Select option (1-5): "))")  
  
PRI handle\_choice(choice)  
 case choice  
 1: start\_system()  
 2: configure\_settings()  
 3: view\_status()  
 4: run\_diagnostics()  
 5: shutdown\_system()  
 other:   
 DEBUG("`Menu.char(7)") ' Error color  
 DEBUG("`Menu.str(string("Invalid choice!", 13, 10))")  
 waitms(1500)

### Pattern 3: Data Logging

Continuous data logging with scrolling display:

PUB data\_logger() | timestamp, value1, value2, line\_count  
 DEBUG(`TERM Logger SIZE 60 25 TEXTSIZE 10)  
   
 ' Header  
 DEBUG("`Logger.str(string("TIME VALUE1 VALUE2 STATUS", 13, 10))")  
 DEBUG("`Logger.str(string("--------- -------- -------- --------", 13, 10))")  
   
 line\_count := 2  
 repeat  
 ' Get data  
 timestamp := get\_timestamp()  
 value1 := read\_sensor1()  
 value2 := read\_sensor2()  
   
 ' Check if we need to scroll  
 if line\_count >= 24  
 ' Scroll by clearing and rewriting header  
 DEBUG("`Logger.char(0)")  
 DEBUG("`Logger.str(string("TIME VALUE1 VALUE2 STATUS", 13, 10))")  
 DEBUG("`Logger.str(string("--------- -------- -------- --------", 13, 10))")  
 line\_count := 2  
   
 ' Log data line  
 DEBUG("`Logger.dec(timestamp)")  
 DEBUG("`Logger.char(9, 9)") ' Two tabs for alignment  
 DEBUG("`Logger.dec(value1)")  
 DEBUG("`Logger.char(9, 9)") ' Two tabs  
 DEBUG("`Logger.dec(value2)")  
 DEBUG("`Logger.char(9)") ' One tab  
   
 ' Status based on values  
 if value1 > 100 or value2 > 100  
 DEBUG("`Logger.char(7)") ' Error color  
 DEBUG("`Logger.str(string("HIGH"))")  
 else  
 DEBUG("`Logger.char(5)") ' Success color  
 DEBUG("`Logger.str(string("OK"))")  
   
 DEBUG("`Logger.char(4, 13, 10)") ' Back to normal, newline  
 line\_count++  
   
 waitms(500) ' Log every half second

### Pattern 4: Progress Indicators

Visual progress displays for long operations:

PUB progress\_demo() | i, percent, bar\_length  
 DEBUG(`TERM Progress SIZE 50 8 TEXTSIZE 14)  
   
 DEBUG("`Progress.str(string("Processing Data...", 13, 10, 10))")  
   
 repeat i from 0 to 100  
 percent := i  
 bar\_length := percent / 2 ' 50-character bar  
   
 ' Position cursor for progress bar  
 DEBUG("`Progress.char(2, 0)") ' Column 0  
 DEBUG("`Progress.char(3, 3)") ' Row 3  
   
 ' Draw progress bar  
 DEBUG("`Progress.str(string("["))")  
 repeat 50  
 if i-- > 0  
 DEBUG("`Progress.char(6)") ' Warning color for filled  
 DEBUG("`Progress.str(string("█"))")  
 else  
 DEBUG("`Progress.char(4)") ' Normal color for empty  
 DEBUG("`Progress.str(string("░"))")  
   
 DEBUG("`Progress.char(4)") ' Back to normal  
 DEBUG("`Progress.str(string("] "))")  
 DEBUG("`Progress.dec(percent)")  
 DEBUG("`Progress.str(string("%"))")  
   
 waitms(50) ' Animation speed  
   
 DEBUG("`Progress.char(13, 10, 10)")  
 DEBUG("`Progress.char(5)") ' Success color  
 DEBUG("`Progress.str(string("Processing Complete!"))")

## Advanced Techniques

### Multi-Terminal Coordination

Using multiple terminals for different purposes:

PUB multi\_terminal\_system()  
 ' Specialized terminals for different functions  
 DEBUG(`TERM Status SIZE 30 8 TEXTSIZE 16)  
 DEBUG(`TERM Debug SIZE 80 20 TEXTSIZE 10)  
 DEBUG(`TERM Data SIZE 40 25 TEXTSIZE 12)  
 DEBUG(`TERM Control SIZE 25 6 TEXTSIZE 14)  
   
 ' Launch monitoring in separate COG  
 cognew(monitor\_cog(), @monitor\_stack)  
   
 ' Main program with coordinated displays  
 main\_program\_loop()  
  
PRI monitor\_cog() | temp, voltage  
 repeat  
 temp := read\_temperature()  
 voltage := read\_voltage()  
   
 ' Update status terminal  
 DEBUG("`Status.char(1)") ' Home  
 DEBUG("`Status.str(string("Temp: "))")   
 DEBUG("`Status.dec(temp)")  
 DEBUG("`Status.str(string("°F Voltage: "))")   
 DEBUG("`Status.dec(voltage)")  
 DEBUG("`Status.str(string("V"))")  
   
 ' Log to data terminal  
 DEBUG("`Data.dec(get\_timestamp())")  
 DEBUG("`Data.str(string(", "))")  
 DEBUG("`Data.dec(temp)")  
 DEBUG("`Data.str(string(", "))")  
 DEBUG("`Data.dec(voltage)")  
 DEBUG("`Data.str(string("\n"))")  
   
 waitms(1000)

### Dynamic Terminal Management

Creating and destroying terminals based on program state:

VAR byte terminal\_active[10] ' Track active terminals  
  
PUB dynamic\_terminals() | mode, terminal\_id  
 repeat  
 mode := get\_operating\_mode()  
   
 case mode  
 MODE\_SETUP:  
 ensure\_terminal("Setup", 40, 15, 14)  
 close\_terminal("Runtime")  
   
 MODE\_RUNTIME:  
 ensure\_terminal("Runtime", 60, 25, 10)  
 close\_terminal("Setup")  
   
 MODE\_DEBUG:  
 ensure\_terminal("Debug", 80, 30, 8)  
 ensure\_terminal("Variables", 30, 20, 10)  
  
PRI ensure\_terminal(name, cols, rows, textsize)  
 ' Only create if not already active  
 if not terminal\_exists(name)  
 DEBUG(`TERM {{name}} SIZE {{cols}} {{rows}} TEXTSIZE {{textsize}})  
 mark\_terminal\_active(name)

### Custom Formatting Functions

Building reusable formatting functions:

PUB format\_table\_row(terminal, col1, col2, col3, col4)  
 ' Standardized table row formatting  
 DEBUG("`{{terminal}}.str(string("|")))")  
 format\_field(terminal, col1, 12) ' 12-char wide field  
 DEBUG("`{{terminal}}.str(string("|")))")  
 format\_field(terminal, col2, 8) ' 8-char wide field  
 DEBUG("`{{terminal}}.str(string("|")))")  
 format\_field(terminal, col3, 10) ' 10-char wide field  
 DEBUG("`{{terminal}}.str(string("|")))")  
 format\_field(terminal, col4, 15) ' 15-char wide field  
 DEBUG("`{{terminal}}.str(string("|\n")))")  
  
PRI format\_field(terminal, value, width) | i, len, spaces  
 ' Right-align value in field of specified width  
 len := strsize(value)  
 spaces := width - len  
   
 repeat i from 0 to spaces-1  
 DEBUG("`{{terminal}}.char(32)") ' Space character  
   
 DEBUG("`{{terminal}}.str({{value}})")

### Real-Time Animation

Creating animated displays for visual feedback:

PUB spinner\_animation(terminal) | frame  
 frame := 0  
 repeat 20 ' 20 animation frames  
 DEBUG("`{{terminal}}.char(8)") ' Backspace  
 case frame // 4  
 0: DEBUG("`{{terminal}}.char(124)") ' |  
 1: DEBUG("`{{terminal}}.char(47)") ' /  
 2: DEBUG("`{{terminal}}.char(45)") ' -  
 3: DEBUG("`{{terminal}}.char(92)") ' \  
   
 frame++  
 waitms(100)  
  
PUB progress\_dots(terminal, count) | i  
 repeat i from 0 to count-1  
 DEBUG("`{{terminal}}.char(46)") ' Period  
 waitms(200)  
   
 repeat i from 0 to count-1  
 DEBUG("`{{terminal}}.char(8)") ' Backspace to erase  
 waitms(50)

## Real-World Projects

### Project 1: Environmental Monitoring Station

CON   
 \_clkfreq = 10\_000\_000  
 TEMP\_SENSOR\_PIN = 16  
 HUMIDITY\_SENSOR\_PIN = 17  
 LIGHT\_SENSOR\_PIN = 18  
  
PUB environmental\_monitor() | temp, humidity, light, hour, minute  
 ' Setup display terminals  
 setup\_displays()  
   
 repeat  
 ' Read all sensors  
 temp := read\_temperature(TEMP\_SENSOR\_PIN)  
 humidity := read\_humidity(HUMIDITY\_SENSOR\_PIN)  
 light := read\_light\_level(LIGHT\_SENSOR\_PIN)  
 get\_time(@hour, @minute)  
   
 ' Update main display  
 update\_main\_display(temp, humidity, light, hour, minute)  
   
 ' Log data  
 log\_data(temp, humidity, light)  
   
 ' Check for alerts  
 check\_alerts(temp, humidity)  
   
 waitms(5000) ' Update every 5 seconds  
  
PRI setup\_displays()  
 ' Main status display  
 DEBUG(`TERM Status SIZE 45 12 TEXTSIZE 16)  
 DEBUG("`Status.char(0)")  
 DEBUG("`Status.str(string("Environmental Monitoring Station\n"))")  
 DEBUG("`Status.str(string("================================\n\n"))")  
   
 ' Data logging display  
 DEBUG(`TERM Log SIZE 70 20 TEXTSIZE 10)  
 DEBUG("`Log.str(string("TIME TEMP HUMID LIGHT STATUS\n"))")  
 DEBUG("`Log.str(string("---- ---- ----- ----- ------\n"))")  
   
 ' Alert display  
 DEBUG(`TERM Alerts SIZE 35 8 TEXTSIZE 14)  
 DEBUG("`Alerts.char(0)")  
 DEBUG("`Alerts.str(string("SYSTEM ALERTS\n"))")  
 DEBUG("`Alerts.str(string("=============\n"))")  
  
PRI update\_main\_display(temp, humidity, light, hour, minute)  
 ' Position at data area (skip header)  
 DEBUG("`Status.char(2, 0)") ' Column 0  
 DEBUG("`Status.char(3, 4)") ' Row 4  
   
 ' Temperature with color coding  
 DEBUG("`Status.str(string("Temperature: "))")  
 if temp > 80  
 DEBUG("`Status.char(7)") ' Red for hot  
 elseif temp < 60  
 DEBUG("`Status.char(6)") ' Blue for cold  
 else  
 DEBUG("`Status.char(5)") ' Green for normal  
 DEBUG("`Status.dec(temp)")  
 DEBUG("`Status.char(4)") ' Back to normal  
 DEBUG("`Status.str(string("°F\n"))")  
   
 ' Humidity display  
 DEBUG("`Status.str(string("Humidity: "))")  
 if humidity > 70  
 DEBUG("`Status.char(6)") ' Warning for high humidity  
 else  
 DEBUG("`Status.char(4)") ' Normal  
 DEBUG("`Status.dec(humidity)")  
 DEBUG("`Status.str(string("%\n"))")  
   
 ' Light level  
 DEBUG("`Status.char(4)")  
 DEBUG("`Status.str(string("Light Level: "))")  
 DEBUG("`Status.dec(light)")  
 DEBUG("`Status.str(string(" lux\n\n"))")  
   
 ' Current time  
 DEBUG("`Status.str(string("Last Update: "))")  
 if hour < 10  
 DEBUG("`Status.char(48)") ' Leading zero  
 DEBUG("`Status.dec(hour)")  
 DEBUG("`Status.char(58)") ' Colon  
 if minute < 10  
 DEBUG("`Status.char(48)") ' Leading zero  
 DEBUG("`Status.dec(minute)")  
  
PRI log\_data(temp, humidity, light) | timestamp  
 timestamp := get\_timestamp()  
   
 DEBUG("`Log.dec(timestamp)")  
 DEBUG("`Log.str(string(" "))")  
 DEBUG("`Log.dec(temp)")  
 DEBUG("`Log.str(string(" "))")  
 DEBUG("`Log.dec(humidity)")  
 DEBUG("`Log.str(string(" "))")  
 DEBUG("`Log.dec(light)")  
 DEBUG("`Log.str(string(" "))")  
   
 ' Status indicator  
 if temp > 80 or humidity > 70  
 DEBUG("`Log.char(7)") ' Red  
 DEBUG("`Log.str(string("ALERT"))")  
 else  
 DEBUG("`Log.char(5)") ' Green  
 DEBUG("`Log.str(string("OK"))")  
   
 DEBUG("`Log.char(4, 10)") ' Normal color, newline  
  
PRI check\_alerts(temp, humidity)  
 if temp > 85  
 show\_alert("HIGH TEMPERATURE ALERT!")  
 elseif temp < 50  
 show\_alert("LOW TEMPERATURE ALERT!")  
   
 if humidity > 80  
 show\_alert("HIGH HUMIDITY ALERT!")  
 else  
 clear\_alerts()  
  
PRI show\_alert(message)  
 DEBUG("`Alerts.char(2, 0)") ' Column 0  
 DEBUG("`Alerts.char(3, 3)") ' Row 3  
 DEBUG("`Alerts.char(7)") ' Error color  
 DEBUG("`Alerts.str({{message}})")  
 DEBUG("`Alerts.char(4)") ' Back to normal  
  
PRI clear\_alerts()  
 DEBUG("`Alerts.char(2, 0)") ' Column 0  
 DEBUG("`Alerts.char(3, 3)") ' Row 3  
 DEBUG("`Alerts.str(string("All systems normal "))")

### Project 2: Interactive Calculator

PUB calculator() | num1, num2, operator, result, input\_state  
 DEBUG(`TERM Calc SIZE 40 15 TEXTSIZE 16)  
   
 input\_state := INPUT\_NUM1  
   
 repeat  
 show\_calculator\_display(num1, operator, num2, result, input\_state)  
   
 ' Get user input (implementation depends on input method)  
 process\_input(@num1, @operator, @num2, @result, @input\_state)  
   
 waitms(100)  
  
PRI show\_calculator\_display(num1, operator, num2, result, state)  
 DEBUG("`Calc.char(0)") ' Clear screen  
   
 ' Title  
 DEBUG("`Calc.str(string("┌────────────────────────────────────┐\n"))")  
 DEBUG("`Calc.str(string("│ P2 CALCULATOR │\n"))")  
 DEBUG("`Calc.str(string("├────────────────────────────────────┤\n"))")  
   
 ' Display area  
 DEBUG("`Calc.str(string("│ "))")  
   
 ' Show current expression  
 if state > INPUT\_NUM1  
 DEBUG("`Calc.dec(num1)")  
 if state > INPUT\_OPERATOR  
 DEBUG("`Calc.char(32, {{operator}}, 32)") ' Space, operator, space  
 if state > INPUT\_NUM2  
 DEBUG("`Calc.dec(num2)")  
 if state == SHOW\_RESULT  
 DEBUG("`Calc.str(string(" = "))")  
 DEBUG("`Calc.dec(result)")  
   
 ' Pad and close display line  
 repeat (35 - get\_display\_length(num1, operator, num2, result, state))  
 DEBUG("`Calc.char(32)") ' Space padding  
 DEBUG("`Calc.str(string(" │\n"))")  
   
 ' Instructions based on state  
 DEBUG("`Calc.str(string("├────────────────────────────────────┤\n"))")  
 case state  
 INPUT\_NUM1:  
 DEBUG("`Calc.str(string("│ Enter first number: │\n"))")  
 INPUT\_OPERATOR:  
 DEBUG("`Calc.str(string("│ Enter operator (+, -, \*, /): │\n"))")  
 INPUT\_NUM2:  
 DEBUG("`Calc.str(string("│ Enter second number: │\n"))")  
 SHOW\_RESULT:  
 DEBUG("`Calc.str(string("│ Press any key to continue... │\n"))")  
   
 DEBUG("`Calc.str(string("└────────────────────────────────────┘"))")

## Troubleshooting

### Common Issues and Solutions

**Problem: Terminal window doesn't appear** - **Check**: Terminal name conflicts - **Solution**: Use unique names for each terminal - **Example**: Change TERM Main to TERM Main\_{{timestamp}}

**Problem: Text appears garbled or overlapped** - **Check**: Control character usage - **Solution**: Always clear screen before major updates - **Example**: Use DEBUG("TERM.char(0)")` before rewriting content

**Problem: Colors don't work as expected** - **Check**: Color scheme configuration - **Solution**: Explicitly set color schemes with COLOR commands - **Reset**: Use DEBUG("TERM.char(4)")` to return to default colors

**Problem: Terminal content scrolls unexpectedly** - **Check**: Terminal size vs. content length - **Solution**: Either increase terminal size or manage content length - **Monitor**: Track line count and clear when approaching limit

**Problem: Poor performance with multiple terminals** - **Check**: Update frequency and terminal count - **Solution**: Reduce update frequency or use fewer terminals - **Optimize**: Update only changed content, not entire display

### Debugging Techniques

**Trace Terminal Operations**:

PUB debug\_terminal\_ops()  
 ' Create debug terminal for tracing operations  
 DEBUG(`TERM Trace SIZE 60 10 TEXTSIZE 10)  
   
 ' Trace each operation  
 DEBUG("`Trace.str(string("Creating main terminal...\n"))")  
 DEBUG(`TERM Main SIZE 40 20 TEXTSIZE 14)  
   
 DEBUG("`Trace.str(string("Sending test message...\n"))")  
 DEBUG("`Main.str(string("Hello, Terminal!"))")  
   
 DEBUG("`Trace.str(string("Operations complete.\n"))")

**Content Validation**:

PRI validate\_terminal\_content(terminal\_name, expected\_lines)  
 ' Use a validation terminal to check content  
 DEBUG(`TERM Validator SIZE 50 5 TEXTSIZE 12)  
   
 if get\_terminal\_line\_count(terminal\_name) == expected\_lines  
 DEBUG("`Validator.char(5)") ' Green  
 DEBUG("`Validator.str(string("✓ Content valid"))")  
 else  
 DEBUG("`Validator.char(7)") ' Red   
 DEBUG("`Validator.str(string("✗ Content mismatch"))")

## Learning Exercises

### Exercise 1: Build a Digital Clock

**Objective**: Create a real-time digital clock display

**Requirements**: - Display current time in HH:MM:SS format - Update every second - Use different colors for hours, minutes, and seconds - Add AM/PM indicator

**Starter Code**:

PUB digital\_clock() | hours, minutes, seconds, ampm  
 DEBUG(`TERM Clock SIZE 25 8 TEXTSIZE 20)  
   
 repeat  
 ' Get current time (implement these functions)  
 hours := get\_hours()  
 minutes := get\_minutes()   
 seconds := get\_seconds()  
 ampm := get\_ampm()  
   
 ' Your code here:  
 ' 1. Clear the display  
 ' 2. Format and display the time  
 ' 3. Use colors for different components  
 ' 4. Add the AM/PM indicator  
   
 waitms(1000)

**Learning Goals**: - Practice terminal clearing and positioning - Learn color control techniques - Understand real-time display updates - Work with time formatting

### Exercise 2: Create a Text-Based Game

**Objective**: Build a simple guessing game

**Game Rules**: - Computer picks random number 1-100 - Player has 7 guesses - Display "Too High", "Too Low", or "Correct!" - Show guess count and remaining guesses - Use colors for feedback

**Template**:

PUB guessing\_game() | secret, guess, attempts, max\_attempts  
 max\_attempts := 7  
 secret := random\_number(1, 100)  
 attempts := 0  
   
 ' Setup game display  
 ' Your code here  
   
 repeat while attempts < max\_attempts  
 ' Show game state  
 ' Get player guess (implement input method)  
 ' Check guess and provide feedback  
 ' Update attempt counter  
   
 if guess == secret  
 ' Handle win condition  
 quit  
   
 ' Handle loss condition if loop exits

### Exercise 3: System Monitor Dashboard

**Objective**: Create a multi-panel system monitoring display

**Requirements**: - CPU usage display (simulated) - Memory usage with progress bar - Network activity indicator - System uptime counter - Alert panel for warnings

**Advanced Features**: - Animated progress bars - Color-coded status indicators - Historical data graphs (ASCII art) - Configurable update intervals

### Exercise 4: Interactive Menu Builder

**Objective**: Build a reusable menu system framework

**Features to Implement**: - Dynamic menu creation from arrays - Nested submenu support - Keyboard navigation simulation - Visual selection highlighting - Menu item enable/disable states

**Design Pattern**:

VAR  
 byte menu\_items[10][32] ' Menu item strings  
 byte menu\_count ' Number of items  
 byte selected\_item ' Currently selected  
 byte menu\_enabled[10] ' Enable/disable states  
  
PUB menu\_framework()  
 ' Initialize menu system  
 setup\_menu()  
   
 repeat  
 display\_menu()  
 handle\_selection()  
 waitms(100)

### Exercise 5: Data Visualization Challenge

**Objective**: Create ASCII art charts and graphs

**Chart Types to Implement**: - Horizontal bar charts - Vertical bar charts  
- Line graphs using ASCII characters - Pie charts with text representation - Histogram displays

**Sample Data Visualization**:

Sales by Quarter  
================  
Q1 ████████████████████ 75%  
Q2 ██████████████████████████ 95%  
Q3 ████████████ 45%  
Q4 ██████████████████████████████ 100%

## Best Practices

### Performance Optimization

**1. Minimize Full Screen Updates**

' Bad: Clearing entire screen frequently  
repeat  
 DEBUG("`TERM.char(0)") ' Full clear every time  
 update\_entire\_display()  
 waitms(100)  
  
' Good: Update only changed areas  
repeat  
 update\_changed\_areas\_only()  
 waitms(100)

**2. Batch Terminal Operations**

' Bad: Many separate DEBUG calls  
DEBUG("`TERM.str(string("Temperature: "))")  
DEBUG("`TERM.dec(temp)")  
DEBUG("`TERM.str(string("°F"))")  
  
' Good: Combine into fewer calls  
DEBUG("`TERM.str(string("Temperature: "))")   
DEBUG("`TERM.dec(temp)")  
DEBUG("`TERM.str(string("°F"))")

**3. Use Appropriate Update Frequencies** - **Status displays**: 1-2 seconds - **Data logging**: 5-10 seconds  
- **Real-time monitoring**: 100-500ms - **User interfaces**: 50-100ms response time

### Code Organization

**1. Separate Display Logic**

' Good: Separate data from display  
PUB main\_program()  
 repeat  
 collect\_data()  
 process\_data()  
 update\_displays() ' Separate function  
 waitms(1000)  
  
PRI update\_displays()  
 update\_status\_display()  
 update\_data\_display()  
 update\_alert\_display()

**2. Use Consistent Naming**

' Terminal naming convention  
DEBUG(`TERM Status\_Main SIZE 40 20 TEXTSIZE 14)  
DEBUG(`TERM Debug\_Trace SIZE 80 25 TEXTSIZE 10)  
DEBUG(`TERM Data\_Log SIZE 60 30 TEXTSIZE 12)

**3. Create Reusable Functions**

PRI draw\_box(terminal, width, height, title)  
 ' Reusable box drawing function  
 ' Implementation here  
  
PRI format\_number(terminal, value, width, decimal\_places)  
 ' Standardized number formatting  
 ' Implementation here

### Error Handling

**1. Validate Terminal Parameters**

PRI create\_safe\_terminal(name, cols, rows, textsize)  
 ' Validate parameters before creating terminal  
 if cols < 1 or cols > 300  
 cols := 40 ' Default  
 if rows < 1 or rows > 200   
 rows := 20 ' Default  
 if textsize < 6 or textsize > 200  
 textsize := 14 ' Default  
   
 DEBUG(`TERM {{name}} SIZE {{cols}} {{rows}} TEXTSIZE {{textsize}})

**2. Handle Data Range Issues**

PRI safe\_display\_value(terminal, value, min\_val, max\_val)  
 ' Clamp values to safe display range  
 if value < min\_val  
 value := min\_val  
 elseif value > max\_val  
 value := max\_val  
   
 DEBUG("`{{terminal}}.dec({{value}})")

### Accessibility Considerations

**1. Use Sufficient Color Contrast** - Avoid red/green combinations (colorblind users) - Ensure text remains readable in all color schemes - Provide non-color status indicators (symbols, text)

**2. Font Size Guidelines** - Minimum 12-point for general use - 16-point or larger for presentations - Consider viewing distance in font selection

**3. Clear Visual Hierarchy** - Use consistent formatting for similar content - Group related information visually - Provide clear navigation cues

### Documentation Standards

**1. Comment Terminal Purposes**

' Main status display - shows current system state  
DEBUG(`TERM Status SIZE 40 15 TEXTSIZE 14)  
  
' Debug trace - detailed program execution info  
DEBUG(`TERM Debug SIZE 80 25 TEXTSIZE 10)

**2. Document Update Frequencies**

' Update status every 2 seconds (user-visible changes)  
repeat  
 update\_status\_display()  
 waitms(2000)

**3. Explain Complex Formatting**

PRI format\_data\_table()  
 ' Creates aligned table with fixed-width columns:  
 ' Column 1: 10 chars (timestamp)  
 ' Column 2: 8 chars (sensor value)  
 ' Column 3: 6 chars (status)

## Conclusion

The SPIN2 Terminal Window system provides a powerful foundation for creating professional text-based interfaces on the Parallax Propeller 2. From simple status displays to complex interactive systems, terminals offer the flexibility and performance needed for both educational and production applications.

### Key Takeaways

**Start Simple**: Begin with basic text output and gradually add features like colors, positioning, and multiple terminals.

**Plan Your Layout**: Design your display layout before coding. Consider information hierarchy and user workflow.

**Optimize Performance**: Update only what changes, use appropriate refresh rates, and batch operations when possible.

**Think Reusable**: Create functions and patterns that can be reused across projects.

**Test Thoroughly**: Verify your displays work correctly across different terminal sizes and with various data ranges.

### Next Steps

1. **Practice the Exercises**: Work through the learning exercises to build familiarity
2. **Build a Real Project**: Apply these concepts to solve an actual problem
3. **Explore Integration**: Combine terminals with other P2 features like Smart Pins and COGs
4. **Share and Learn**: Connect with the P2 community to share techniques and learn new approaches

The terminal system is just one part of the P2's comprehensive development environment. As you become comfortable with terminals, explore how they integrate with debugging tools, scope displays, and other P2 capabilities to create complete, professional applications.

**Document Information**: - **Version**: 1.0 - **Last Updated**: 2025-08-15 - **Source**: SPIN2 Terminal Window functionality from P2 Documentation - **Target Audience**: P2 developers from beginner to advanced - **Scope**: Complete guide to terminal window usage and best practices