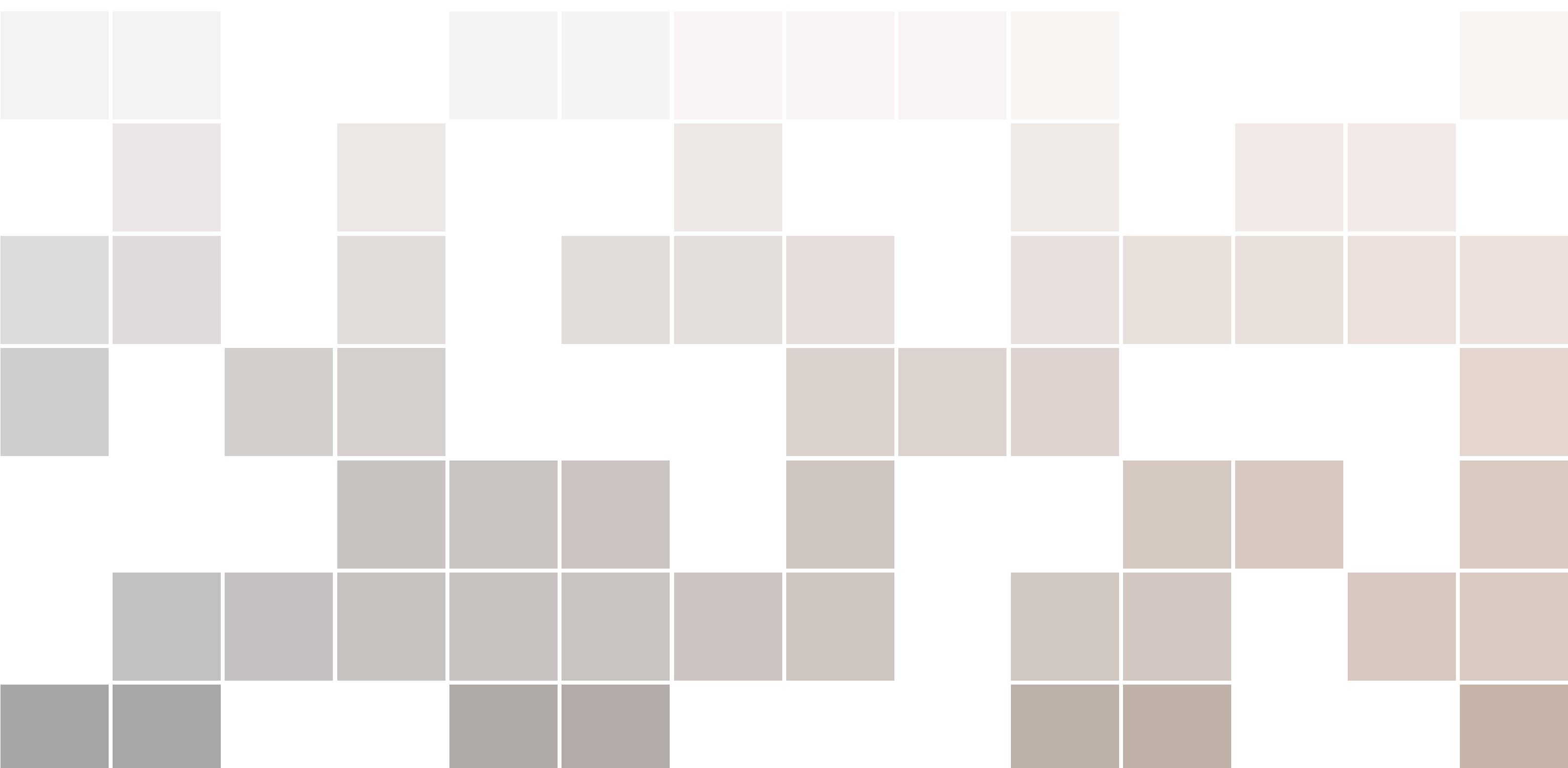
User Manual for LogicTerm 16

Next Generation Test & Debug Tool

Model: LT16M v2.008



LT16M User Manual
Document Version 1.3 | Revision A | March 2026

Copyright © 2025–2026 LogicTerm Inc. All rights reserved.

PUBLISHED BY LOGICTERM INC.
www.LogicTerm.com | logicterm16@gmail.com

LogicTerm Inc. designs and manufactures high-performance, affordable mixed-signal test and measurement instruments for embedded systems development. The LT16M is LogicTerm's flagship product, combining digital pattern generation, analog I/O, and programmable power delivery in a single compact instrument.

Trademarks. LogicTerm, LT16M, and LTStudio are trademarks or registered trademarks of LogicTerm Inc. All other product names, company names, and logos mentioned herein are the property of their respective owners.

Proprietary Notice. This document contains proprietary information of LogicTerm Inc. and is furnished for informational use only. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—without the prior written permission of LogicTerm Inc.

Disclaimer. The information in this document is provided "as is" and LogicTerm Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents. LogicTerm Inc. reserves the right to make changes to specifications, product descriptions, and circuit diagrams at any time without notice. LogicTerm Inc. assumes no liability for applications assistance or customer product design. The LT16M is not designed or intended for use in life-support equipment or applications where malfunction could cause serious injury or death.

Export Compliance. This product may be subject to applicable export control regulations. Users are responsible for compliance with all applicable laws and regulations.

First Edition, 2025. Revised March 2026.

Contents

I	Getting Started

1 Introduction . . . 9
1.1 The Concept 9
1.2 Use Cases and Capabilities 9
1.3 System Overview 10

2 Hardware Overview . . . 11
2.1 LT16M Box 11
2.2 Digital I/O Capabilities 12
2.3 User I/O 12
2.4 Analog Capabilities 12
2.5 Power Supply 13
2.6 Environmental and Operating Conditions 13
2.7 Safety Warnings and Precautions 13
2.8 Regulatory Compliance 14

3 Software Overview . . . 15
3.1 LTStudio IDE 15
3.2 LTStudio Control Menu 15
3.3 Supported File Types 16
3.4 Installation and USB Driver Setup 16

4 First-Time Setup . . . 18
4.1 Connecting the LT16M to a PC 18
4.2 Jumper Configuration 18
4.3 Verifying USB and Driver Installation 18
4.4 Using LTStudio: Running your first pattern! 18
4.5 Using LTStudio: Running your first service! 19
4.6 Using LTStudio: Running your first batch! 19

II Programming Architecture

5 Programming Model . . . 21
5.1 Dual-Level Execution 21
5.2 Object Flow Architecture 22

6 Programming Objects . . . 23
6.1 Overview 23
6.2 Formats 23
6.2.1 Output Format Ticks . . . 24
6.2.2 Input Format Ticks . . . 25
6.2.3 Formats Syntax . . . 25
6.3 Signals 26
6.3.1 Signals Syntax . . . 26
6.4 Patterns 28
6.4.1 Pattern Syntax . . . 29
6.4.2 Micro-Instructions . . . 29
6.4.3 Compiler Instructions . . . 41
6.4.4 Notes . . . 41
6.5 Services 41
6.5.1 Wrappers for services . . . 42
6.6 Custom Services 55
6.6.1 Creating a Service . . . 55
6.6.2 Logging and Exporting . . . 56
6.6.3 Control and Automation . . . 56
6.6.4 Service Invocation from Pattern . . . 56
6.6.5 Best Practices . . . 56
6.7 Batch Scripts 56
6.7.1 Wrappers for Batch Scripts . . . 56
6.7.2 Batch examples . . . 57

III Data Logging & Analysis

7 Database Architecture . . . 61
7.1 Overview 61

7.2 **Database Structure** 61
7.3 **View Tables** 61
7.4 **Exporting Data** 61
7.5 **Service and Batch Integration** 62
7.6 **Database Tables** 63
7.6.1 Records Table 63
7.6.2 Groups Table 63
7.6.3 GroupsInfo Table 63
7.6.4 IOFailsView Table 64
7.6.5 IOChangeView Table 64
7.6.6 IOCountersView Table 65
7.6.7 InfoView Table 65
7.6.8 AnalogDataView Table 65

IV Practical Examples

8 Quick Start Examples 67
8.1 **LED Blinking** 67
8.1.1 LED Blinking by changing the cycle 67
8.1.2 Slowdown LED Blinking using Repeats 68
8.1.3 Slowdown LED Blinking using Loops 68
8.1.4 Slowdown LED Blinking by mapping to `x[0]` 68
8.1.5 Slowdown LED Blinking by mapping to `x[15]` 69
8.1.6 Slowdown LED Blinking by using Keep and Toggle ticks 69
8.2 **Seven Segment Display** 69

9 Debugging Examples 71
9.1 **Detecting Stuck-at Faults** 71
9.2 **Open Circuit Detection** 72
9.3 **Short Detection Between Pins** 72
9.4 **Using IOCounters** 72
9.5 **Channel snoop** 72
9.6 **Oscilloscope** 72
9.7 **Logic Analyzer** 73

10 Production Examples 75
10.1 **Automated Test Sequence** 75
10.2 **Reusable Pattern with Parameters** 75
10.3 **Logging and Exporting Results** 75
10.4 **Pass/Fail Summary** 76
10.5 **PWM** 76
10.6 **SPI Master Controller** 76
10.7 **SPI Slave Device** 76
10.8 **I²C Master Controller** 76

10.9 Duty Cycle measurement 76
10.10 Frequency counter 77
10.11 Analog waveform generator 77
10.12 Melody generator 78
10.13 NPN Transistor Characterization 78
10.14 MOSFET Transistor Characterization 79
10.15 Flash programmer 79
10.16 EEPROM reader 79

V Reference

11 Troubleshooting and Optimization 82
11.1 Connection Issues 82
11.2 Pattern Execution Failures 82
11.3 Signal Integrity 82
11.4 Performance Optimization 83
11.5 Best Practices 83

12 Definitions 84
12.1 Object Types 84
12.2 Formats Syntax 84
12.3 Signals Syntax 84
12.4 Pattern Syntax 85
12.4.1 Cycle Syntax 85
12.4.2 ALU Syntax 85
12.4.3 Output Assignment Syntax 86
12.4.4 Control Syntax 86
12.4.5 IO Mask Syntax 86
12.4.6 Log Syntax 86
12.4.7 Drive-only Syntax 86
12.4.8 Branch Syntax 86
12.4.9 Compiler Instructions Syntax 86
12.5 Database Tables 87
12.5.1 Records Table 87
12.5.2 Groups Table 87
12.5.3 GroupsInfo Table 87
12.5.4 InfoView Table 87
12.5.5 AnalogDataView Table 88
12.5.6 IOCountersView Table 88
12.5.7 IOFailsView Table 88
12.5.8 IOChangeView Table 88

13 Descriptions and Definitions 89
13.1 Descriptions and Definitions 89
13.2 Glossary 89

Index 89

14 Version History 92
14.1 Version History 92

Getting Started

1 Introduction 9
1.1 The Concept
1.2 Use Cases and Capabilities
1.3 System Overview

2 Hardware Overview 11
2.1 LT16M Box
2.2 Digital I/O Capabilities
2.3 User I/O
2.4 Analog Capabilities
2.5 Power Supply
2.6 Environmental and Operating Conditions
2.7 Safety Warnings and Precautions
2.8 Regulatory Compliance

3 Software Overview 15
3.1 LTStudio IDE
3.2 LTStudio Control Menu
3.3 Supported File Types
3.4 Installation and USB Driver Setup

4 First-Time Setup 18
4.1 Connecting the LT16M to a PC
4.2 Jumper Configuration
4.3 Verifying USB and Driver Installation
4.4 Using LTStudio: Running your first pattern!
4.5 Using LTStudio: Running your first service!
4.6 Using LTStudio: Running your first batch!

1. Introduction

1.1 The Concept

Have you ever thought of a system that combines the deterministic performance of RTL languages with the flexibility of high-level scripting? The LT16M bridges this gap by offering a dual-level programming model: low-level digital pattern generation and high-level Python scripting for automation and analysis.

The LT16M, figure 1.1, is a comprehensive mixed-signal test and debug station designed for embedded systems engineers, educators, and hardware developers. It enables precise control over digital signals while allowing advanced data manipulation and visualization through Python.

Figure 1.1: LT16M box front panel

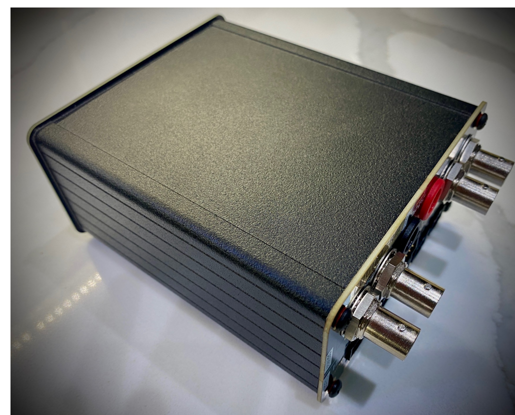


Figure 1.2: LT16M Box

1.2 Use Cases and Capabilities

The LT16M is ideal for:

- Debugging faulty circuits and integrated chips
- Prototyping digital and analog designs

- Mixed-signal waveform generation and capture
- Automated test sequencing and data visualization
- Educational labs and hands-on embedded systems training

1.3 System Overview

The LT16M system consists of:

- **LT16M Hardware Box** – Executes patterns, captures/generates analog signals, and manages power supplies
- **LTStudio IDE** – A Windows-based GUI for editing, compiling, and executing patterns and Python services
- **USB Interface** – Connects the PC to the LT16M for control and data exchange

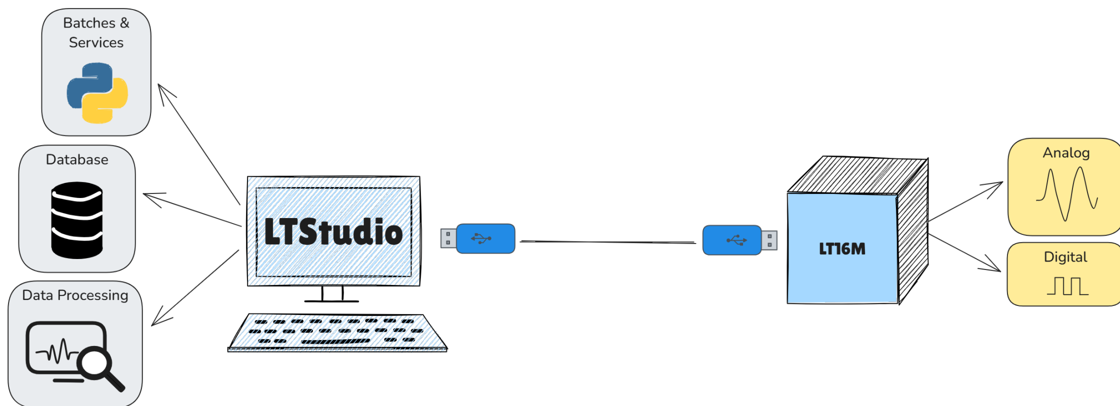


Figure 1.3: Top-level system architecture of LT16M

The LT16M integrates the capabilities of multiple instruments into one platform:

- Logic Analyzer
- Pattern Generator
- Oscilloscope
- Programmable Power Supply

This manual will guide you through setup, programming, data logging, and practical examples to help you make the most of LT16M.

2. Hardware Overview

2.1 LT16M Box

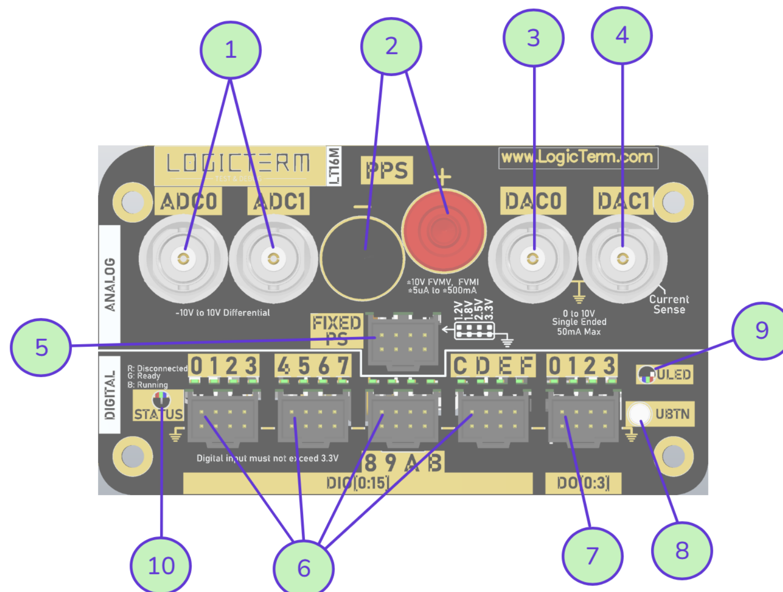


Figure 2.1: Front panel of LT16M

The LT16M hardware is a compact, mixed-signal test and debug station designed by LogicTerm. It integrates digital and analog I/O, power supplies, and user interface elements into a single enclosure.

- **16 bidirectional digital I/O (DIO)** pins with LED status indicators ⑥
- **4 digital drive-only (DO)** pins ⑦
- **2 high-speed ADCs** (±10V, 200 KS/s, 16-bit) ①
- **2 DACs** (0–10V, 1 MS/s, 16-bit) ③ ④
- **Programmable Power Supply (PPS)** with 7 current ranges and clamping ②

- **Fixed power rails:** 3.3V, 2.5V, 1.8V, 1.2V @ 200mA ⑤
- **User I/O:** RGB LED (ULED) ⑨ and push button (UBTN) ⑧
- **System status indicator** can be Red (Disconnected), Green (Ready), or Blue (Running) ⑩

2.2 Digital I/O Capabilities

- Fully programmable at 100 MS/s real-time sample rate for DIOs
- Fully programmable at 25 MHz update rate for DOs
- 1K-entry pattern buffer (16-bit wide)
- 4K sample capture buffer
- Selectable LVCMOS levels: 1.8V, 2.5V, 3.3V (via jumper, figure 2.2)
- 1mA source/sink per pin
- Integrated LEDs for all 20 digital pins

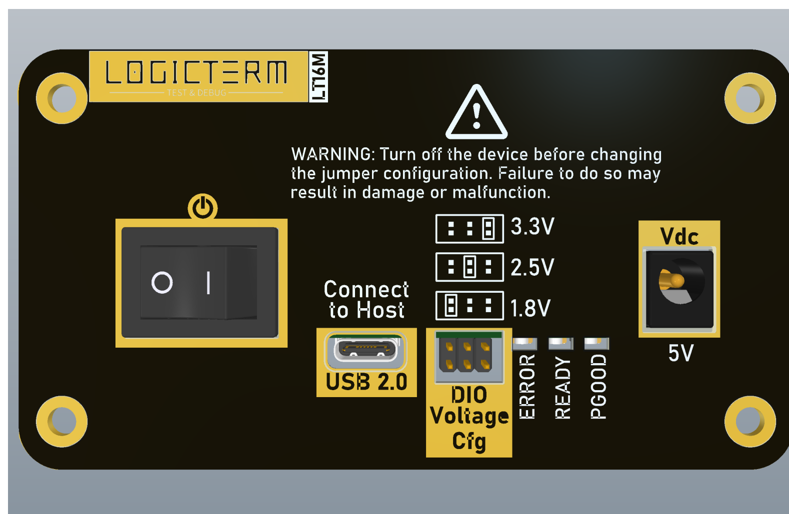


Figure 2.2: Back panel of LT16M showing jumper configuration

2.3 User I/O

- **UBTN:** User push button checked by pattern or service
- **ULED:** RGB LED controlled via pattern or service

2.4 Analog Capabilities

Analog-to-Digital Converters (ADC)

- Two differential channels
- ±10V input range
- 200 KS/s, 16-bit resolution
- 0.3 mV resolution, ±0.5% accuracy
- 128 dB CMRR, 94.2 dB typical SNR
- 1000 MΩ || 3 pF input impedance

Digital-to-Analog Converters (DAC)

- Two single-ended channels (DAC0 and DAC1)
- Output range: 0–10V

- Update rate: 1 MS/s
- Maximum output current: 50 mA per channel
- Resolution: 16-bit (0.153 mV per LSB)
- Output impedance: Few mΩ at low frequencies, approaching unity gain at high frequencies
- Slew rate: ????? V/μs
- Maximum load capacitance: ????? pF
- DAC1 includes current monitoring: 0.1 mA resolution (for ????? mA range)

2.5 Power Supply

- **Fixed Power Rails:** 3.3V, 2.5V, 1.8V, 1.2V @ 200mA maximum current each

Programmable Power Supply (PPS)

- **Voltage Ranges:** Three modes covering ±10V:
  - Range 1 (Bipolar): -6V to +6V
  - Range 2 (Positive): 0 to +10V
  - Range 3 (Negative): 0 to -10V
- **Current Ranges:** ±5 µA, ±25 µA, ±250 µA, ±2.5 mA, ±25 mA, ±250 mA, to +500 mA (source) / -250 mA (sink) – 7 ranges total
- **Settling Time:** 10 to 50 µs (range dependent)
- **Modes:** Force Voltage / Measure Voltage (FVMV), Force Voltage / Measure Current (FVMI)
- **Features:** Current clamping, programmable compliance voltage
- **Current Clamping:** Prevents overcurrent damage to device under test; user-programmable threshold
- **Load Characteristics:** Output impedance ????? Ω at low frequencies, unity gain at high frequencies

2.6 Environmental and Operating Conditions

Temperature

- Operating Temperature: -10°C to 50°C
- Storage Temperature: -20°C to 70°C

Humidity

- Operating: 10% to 90% relative humidity (non-condensing)
- Storage: 5% to 95% relative humidity (non-condensing)

Power Consumption

- Idle (no pattern running): ????? mA
- Active pattern (typical): ????? mA
- Maximum (all subsystems active): ????? W

2.7 Safety Warnings and Precautions

Warning 1 **Electrical Safety:**

- Do not apply voltages exceeding ±10 V to any analog input pins (ADC inputs)
- Do not exceed maximum current ratings: 1 mA per DIO pin, 50 mA per DAC output
- All digital pins include ESD protection diodes; use proper ESD protection when handling the device

- Do not touch internal components while the device is powered

Warning 2 Power Supply Operation:

- Use only the supplied power adapter with correct voltage and polarity
- Do not modify or override power supply specifications
- Do not apply external voltage to any power rails
- Do not load any fixed power rail beyond specified maximum of 200 mA per rail
- PPS supplies: Do not exceed maximum current limits for selected range

Warning 3 Jumper Configuration:

- Always power off the LT16M before changing jumper settings
- Allow 30 seconds after power-off before making changes
- Verify jumper position visually after configuration
- Reapply power and verify status LED (green = ready) before resuming operation
- Changing voltage levels while powered may cause permanent damage

Warning 4 Device Under Test (DUT) Protection:

- Use PPS current clamping to prevent overcurrent damage to unknown circuits
- Start with lowest current range when testing new devices
- Verify correct polarity before applying power to DUT
- Use current limiting techniques when testing suspected short circuits

2.8 Regulatory Compliance

Warning 5 Under construction!

The LT16M has been tested and certified for compliance with the following standards:

- FCC Part 15 Class B (Electromagnetic Interference): Certified
- CE Mark compliance (European Union): Certified
- RoHS 2.0 (Directive 2011/65/EU): Compliant
- WEEE Directive (2012/19/EU): Compliant

For detailed certification information and safety documentation, contact LogicTerm Inc. or visit the support website.

3. Software Overview

3.1 LTStudio IDE

LTStudio is a Windows-based integrated development environment (IDE) designed to control the LT16M hardware. It allows users to:

- Compile and load digital patterns
- Execute Python services and batch scripts
- Visualize and export logged data
- Manage projects and object libraries

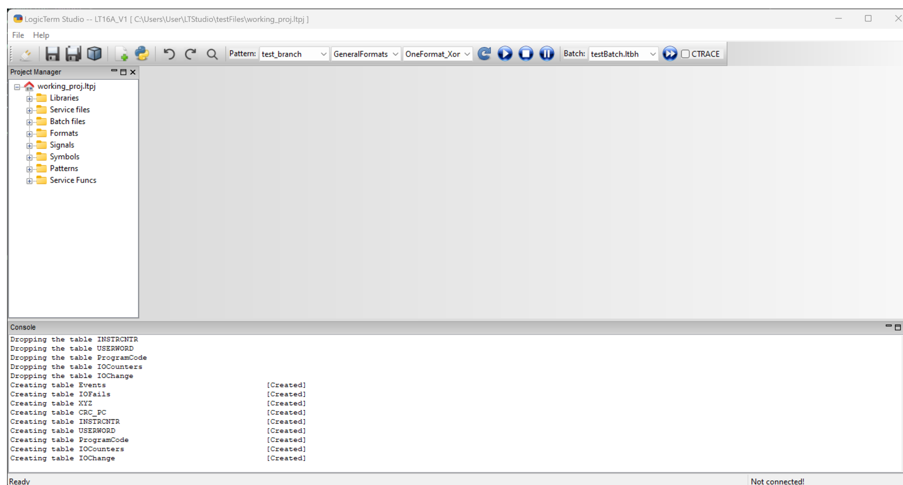


Figure 3.1: LTStudio startup screen

3.2 LTStudio Control Menu

1. Connect: connects LTStudio to the LT16M hardware box via a USB cable. LT16M Status LED changes color from Red to Green when connected.

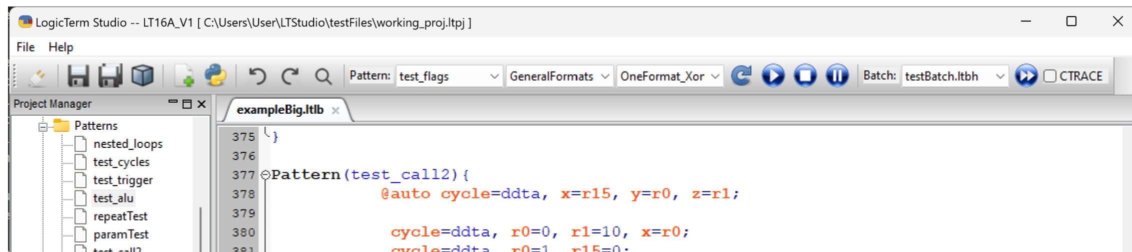


Figure 3.2: LTStudio control button bar

2. Save project: save all files into project and compile all.
3. Add object: add a new LT16M library or batch file to current project.
4. Add service: add a service file to current project.
5. Undo: undo last change
6. Redo: redo last change after it was undone
7. Find: search in current open tab
8. Pattern drop-down menu: choose one of the current compiled patterns.
9. Formats drop-down menu: choose one of the current compiled formats.
10. Signals drop-down menu: choose one of the current compiled signals.
11. Reset: reset digital pattern generator. It will bring the pattern generator to a known state.
12. Start pattern: start the current selected pattern/formats/signals.
13. Stop pattern: stop the running pattern.
14. Pause pattern: pause the running pattern.
15. Batch dropdown menu: choose one of the current batches.
16. Start batch: start the selected batch.

3.3 Supported File Types

LTStudio supports the following file extensions:

- `.ltlb` – Library files containing Formats, Signals, and Patterns
- `.ltpy` – Python service files for logging, plotting, and control
- `.ltbh` – Batch scripts for automated test sequences

3.4 Installation and USB Driver Setup

To connect LT16M to your PC, follow these steps:

1. Install LTStudio from `https://www.LogicTerm.com/download`
2. Install Zadig v2.9 or later from `https://zadig.akeo.ie`
3. Connect LT16M via USB-C to a USB 2.0 port
4. Power on the LT16M (Status LED turns red)
5. Open Zadig and select “List All Devices”
6. Choose `LT16D (Interface 1)` and install `libusb-win32`, as seen in Figure 3.3.

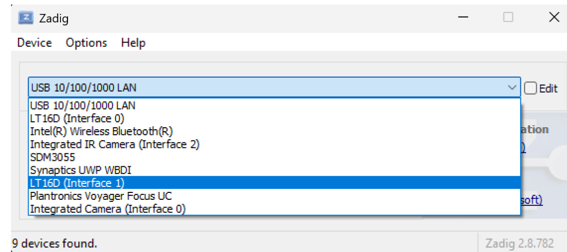


Figure 3.3: Zadig showing two interfaces for LT16M.

7. Turn off the LT16M device, and then, turn it back on.
8. In Device Manager, update `LT16D (Interface 0)` to use `USB Serial Converter A`, if it is different. For a successful setup, the device will appear in the device manager as Figure 3.4.
9. Launch LTStudio and press “Connect” — the Status LED should turn green

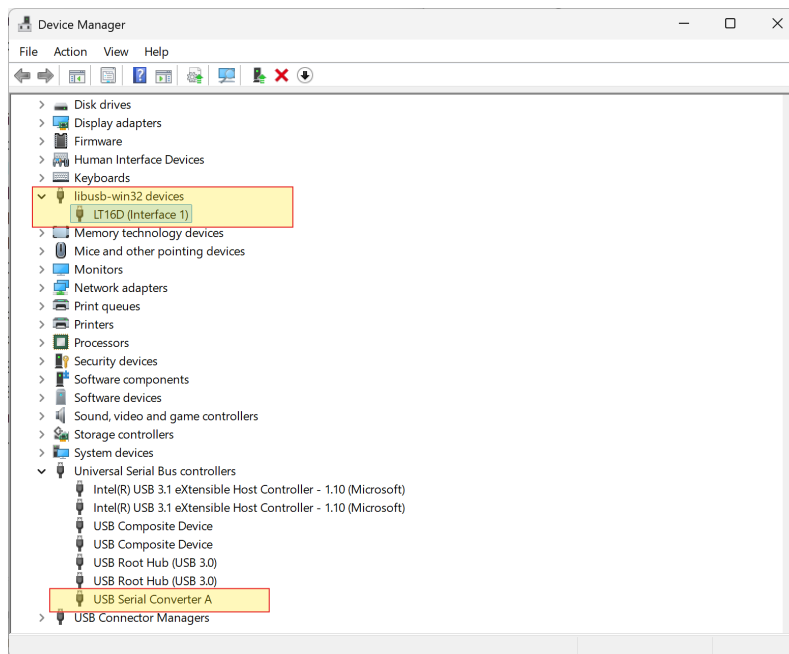


Figure 3.4: LT16M driver configuration in Device Manager

4. First-Time Setup

4.1 Connecting the LT16M to a PC

To set up the LT16M, first connect the supplied power adapter, then link the device to your PC using the included USB-C cable. Confirm that the unit is switched on, start LTStudio, and select “Connect.” The LT16M status LED displays red when idle and changes to green after LTStudio establishes a successful connection.

Warning 6 Use only the supplied power adapter. Using other power supplies may damage the device or impair its performance. ■

4.2 Jumper Configuration

The LT16M supports three digital I/O voltage levels: 1.8V, 2.5V, and 3.3V. These are selected using a jumper on the back panel, refer to Figure 2.2.

Warning 7 Always power off the LT16M before changing jumper settings. Changing voltage levels while the device is powered may cause permanent damage. ■

4.3 Verifying USB and Driver Installation

After connecting the LT16M:

1. Open Device Manager on your PC, as seen in Figure 3.4.
2. Under `libusb-win32 devices`, confirm that `LT16D (Interface 1)` is listed.
3. Under `Universal Serial Bus controllers`, confirm that `USB Serial Converter A` is installed for Interface 0.

4.4 Using LTStudio: Running your first pattern!

After installing the drivers, follow these steps to run your first pattern:

1. Open LTStudio from the Start Menu or desktop shortcut. Figure 3.2 illustrates the toolbar buttons available for different functions.
2. Power on LT16M and connect USB-C cable.
3. Click the **Connect** button in the toolbar.
4. Verify that the LT16M status LED turns green, confirming a successful connection.
5. Navigate to `File` → `New Project`, name the project `HelloWorld`, and select the desired directory.
6. Create a new library via `File` → `New Library`, naming it `HelloWorld_lib`.
7. Go to `Utilities` → `Templates` → `Patterns` → `Hello World Full Example - Pattern`, then copy the template text.
8. Paste the template text into `HelloWorld_lib`, save the current project, and note that the new pattern, formats, and signals now appear in their respective drop-down menus.
9. Select `helloWorldPattern`, `helloWorldFormats`, and `HelloWorldSignals`, then click `Start Pattern`.
10. Observe that the pattern maps the 16-bit counter to the 16 DIOs, causing the DIO LEDs to blink at varying rates.
11. Confirm that the LT16M status LED turns blue, indicating the system is running.
12. To stop the pattern, either press `Stop Pattern` or hold the `UBTN`.

4.5 Using LTStudio: Running your first service!

The simplest way to call a hardware function outside a pattern is to call it from the "Interactive Shell" in LTStudio. You can type `hw.setUSERLED(0xff, 0, 0)` and hit enter to execute. You should see the ULED in red. You may change the USERLED section 6.5.1. For creating a custom service and call it from a pattern, please follow these steps.

1. Make sure you are connected to the LT16M.
2. Create a new service file via `File` → `New Service`, naming it `HelloWorld_services`.
3. Go to `Utilities` → `Templates` → `Services` → `echo`, then copy the template text.
4. Paste the template text into `HelloWorld_services`, save the current project, and note that the new service now appears under services in project manager.
5. `Hello World Full Example - Service`
6. helloWorldPattern2 helloWorldFormats HelloWorldSignals helloWorldServices

4.6 Using LTStudio: Running your first batch!

You are now ready to explore patterns, signals, and services within LTStudio.

II Programming Architecture

5 Programming Model **21**

5.1 Dual-Level Execution

5.2 Object Flow Architecture

6 Programming Objects **23**

6.1 Overview

6.2 Formats

6.3 Signals

6.4 Patterns

6.5 Services

6.6 Custom Services

6.7 Batch Scripts

5. Programming Model

5.1 Dual-Level Execution

The LT16M system is built around a dual-level programming model that combines deterministic digital control with flexible high-level scripting:

- **Level 1: Pattern Execution on LT16M** Patterns are compiled and executed directly on the LT16M hardware. These patterns define precise digital signal behavior using a custom instruction set. They are ideal for timing-critical operations and deterministic control.
- **Level 2: Python Services on Host PC** Python scripts run on the host PC via LTStudio. These services can extract logged data, visualize results, control execution flow, and automate batch operations. They provide a high-level interface for analysis and orchestration.

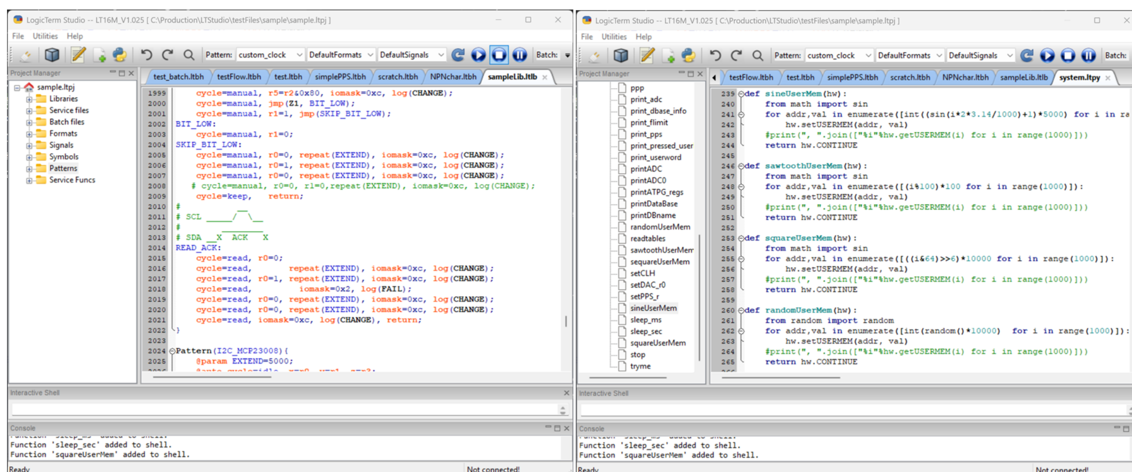


Figure 5.1: Dual-level programming model of LT16M. (Left) writing a pattern for digital signals and low-level conditioning. (Right) writing a service script for mixed signals interaction and high-level data manipulation.

5.2 Object Flow Architecture

The LT16M programming workflow is modular and object-oriented. Each test or debug session typically involves the following components:

- **Formats** – Define timing and signal structure across cycles
- **Signals** – Map physical DIO pins to logical pattern outputs
- **Patterns** – Generate digital waveforms and call services
- **Services** – Python functions for logging, plotting, and control with specific return
- **Batches** – Python scripts that automate execution of multiple objects

This architecture allows users to build reusable libraries, automate test sequences, and integrate digital and analog operations seamlessly.

6. Programming Objects

6.1 Overview

LT16M uses a modular object model to define, execute, and automate digital and analog tests. Each object type plays a specific role in the signal generation and data logging pipeline.

6.2 Formats

Figure 6.1: LTStudio Formats

Formats define the timing and structure of digital signals. Each format can contain up to 16 cycle configurations, and each cycle can specify output values, masks, and control flags.

- Up to 16 format definitions per pattern

- Each format definition has up to 16 cycles
- For a selected cycle, a format defines pin directions and four time ticks
- Ticks are time steps of 10 ns of a specific type. For example, an `oHHHH` format will drive a cycle of 40 ns of a high signal while an `oLHLH` will construct a clock-like waveform.

6.2.1 Output Format Ticks

Direction	Tick	Drive	Example	Example Notes
Output	`H`	High	`oHHHH`	Drives high for four ticks each is 10ns.
Output	`L`	Low	`oLLLL`	Drives low for 40 ns.
Output	`D`	Data	`oDDDD`	Drives based on mapped bit in Signals.
Output	`V`	inVerted data	`oVVVV`	Drives based on the inversion of the mapped bit.
Output	`K`	Keep prev. data	`oKKKK`	Keep the same data state as previous data.
Output	`T`	Toggle prev. data	`oTTTT`	Toggle the data state of previous data.
Output	`Z`	high-Zi	`oZZZZ`	Pin is tri-stated.

Table 6.1: List of different types of ticks for output (driving) Formats

Table 6.1 describes the types of output ticks. Figure 6.2 shows the format waveform for driving using `L`, `H`, and `Z` ticks. Note that the waveform will appear in the following cycle after an instruction is executed. Figure 6.3 exhibits the waveform for driving using `D`, `V`, `K`, and `T` ticks.

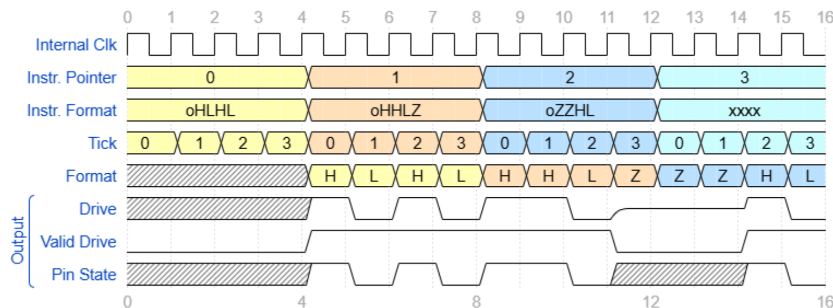


Figure 6.2: Format waveform for driving using `L`, `H` and `Z` ticks.

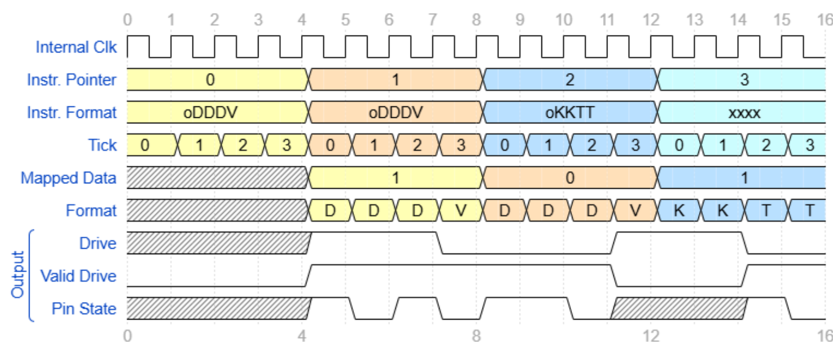


Figure 6.3: Format waveform for driving using `D`, `V`, `K`, and `T` ticks.

Hint 6.2.1 — The `T` and `K` ticks toggle or keep the previous state of the pin even when the pin was an input in the previous cycle..

6.2.2 Input Format Ticks

Direction	Tick	Read for	Example	Example Notes
Input	`H`	High	`iZZHZ`	Read comparator uses High to compare at 3rd tick.
Input	`L`	Low	`iLZZZ`	Read comparator uses Low to compare at 1st tick.
Input	`D`	Data	`iDDDD`	Read comparator uses mapped bit to compare at all ticks.
Input	`V`	inVerted data	`iVZZZ`	Read comparator uses inverted mapped bit.
Input	`M`	Memory read	`iMZZZ`	Read comparator uses data state of last read.
Input	`Z`	high-Zi	`iZZZZ`	Pin is tri-stated.

Table 6.2: List of different types of ticks for input (reading) Formats

For input cycles, ticks are used for the read comparator to compare the input data against. Table 6.2 describes the types of input ticks. Figure 6.4 shows the waveform for reading using `L`, `H`, `D`, and `V` ticks. The comparator will compare the pin state to the format output. Figure 6.5 shows the format output for reading using `M` and `Z` ticks.

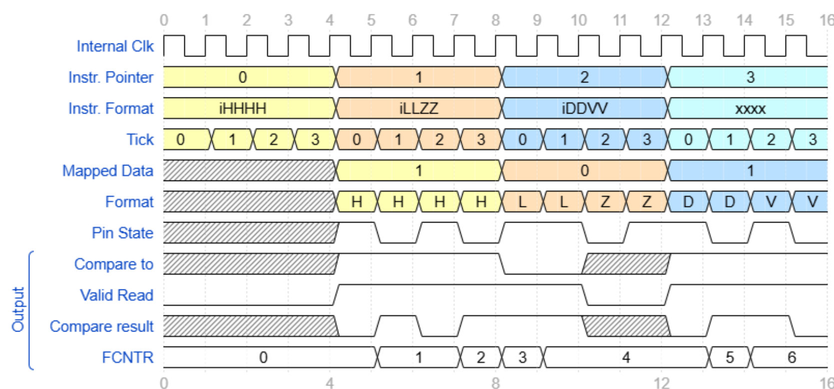


Figure 6.4: Format waveform for reading using `L`, `H`, `D`, and `V` ticks.

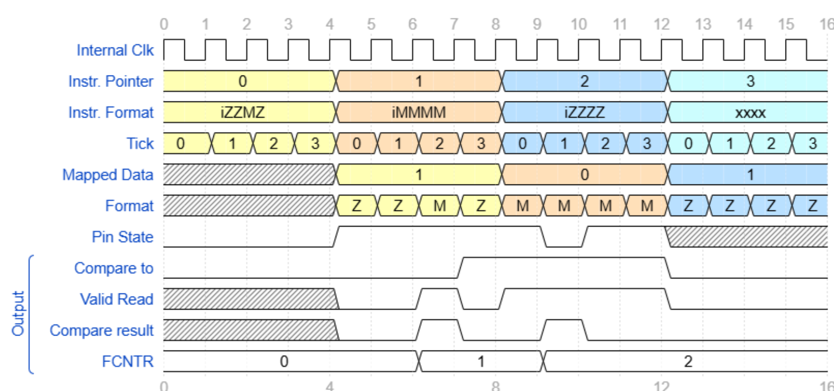


Figure 6.5: Format waveform for reading using `M` and `Z` ticks.

6.2.3 Formats Syntax

Please refer to section 12.2 for full syntax.

The first line in the format definition object is for the cycle naming header. The following code, in listing 6.1, defines the Formats object named `exampleFormat`. It has two formats with two

cycles.

```
Formats(exampleFormat){
        # CYCLE        0       1
    # FORMAT
      cycle_sel  = [ led_on, led_off];  # Header
      LED_FRMT   = [  oHHHH,  oLLLL ];  # All outputs format
      BTN_FRMT   = [  iHHHH,  iLLLL ];  # All inputs format
    }

Formats(exampleFormat2){
    cycle_sel  = [low,    data, inv  ];
    READ_FRMT  = [iZZLZ, iZZZD, iZZZV]; # Inputs. Z is tri-state.
                                        # D and V rely on the pin selected data.
    WRITE_FRMT = [oLLLL, oDDDD, oVVVV]; # Ouputs. if data is 0, D is 0 & V is 1.
                                        # if data is 1, D is 1 & V is 0.
    }
```

Listing 6.1: Example Formats

If pins use `LED_FRMT`, they will drive high when the cycle is `led_on` and low when the cycle is `led_off`. On the other hand, pins that use the format `BTN_FRMT` expect high on read when the selected cycle is `led_on`, and expect low when the cycle is `led_off`.

Valid LT16M variable names:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A−z, 0−9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- A variable name cannot be any of the Python keywords.

Warning 8 Number of defined `cycle_sel` must match the number of defined formats in each of the format selections, as shown in listing 6.2,

```
Formats(badExampleFormat){
        # CYCLE        0       1       3
    # FORMAT
      cycle_sel  = [ led_on, led_off, extra];  # Header
      LED_FRMT   = [  oHHHH,  oLLLL,  oZZZZ];  # defined for 3 cycles
      BTN_FRMT   = [  iHHHH,  iLLLL ];         # defined for 2 cycles!
    }
```

Listing 6.2: Syntax error!

6.3 Signals

Signal objects map logical outputs from a pattern to physical DIO pins on the LT16M.

- Define pin-to-bit mapping
- Support labeling and grouping
- Used in conjunction with formats to drive hardware

6.3.1 Signals Syntax

Please refer to section 12.3 for full syntax.

The LT16M features 16 digital I/O pins operable at 100Mbps. The signal object allows for unique labeling, mapping to physical pin and data sources, and formatting. These labels are commonly employed in VCD files. Rather than using generic signal names like `pin`[0], `pin`[1], `pin`[2], and `pin`[3], more descriptive names such as `CS_N`, `SCLK`, `MOSI`, and `MISO` can be applied.

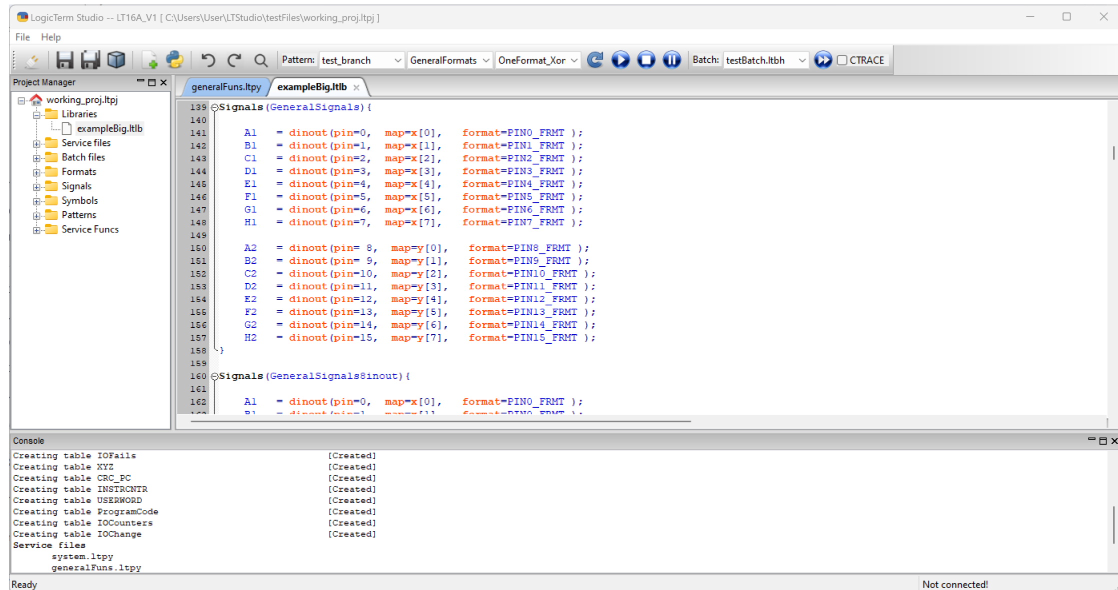


Figure 6.6: LTStudio Signals

The data source for a pin can be fixed values (0 or 1), one bit of the output registers (`x[0], x[1]... x[15]` or `y[0], y[1]... y[15]` or `z[0], z[1]... z[15]`)

> **Hint 6.3.1** *Signals*, `dio`, `pin`, `map`, and `format` are reserved keywords.

For `format1` and `sig1` signals in Listing 6.3, three signals—`SCLK`, `BTN`, and `BTN_N`—are defined. `SCLK` is assigned to `DIO[0]`, always sourcing data from 0. Under `CLK_FRMT`, `SCLK` operates at 25MHz, cycling high for 2 ticks, then low for 2 ticks. `BTN` is associated with `DIO[1]`, with data constantly sourced from 0. `BTN_FRMT` dictates that `BTN` should be low on `fstCy` and high on `scdCy`. Conversely, `BTN_N`, sourcing data from 1, expects the opposite state to `BTN`.

```
1  Formats(format1){
2          # CYCLE          0        1
3      # FORMAT
4        cycle_sel  = [   fstCy , scdCy];    # Header
5        BTN_FRMT   = [  iDDDD ,  iVVVV ];   # All inputs format
6        CLK_FRMT   = [  oHHLL ,  oHHLL ];   # Clock
7      }
8
9  Signals(sig1){
10     SCLK  = dio(pin=0, map=0, format=CLK_FRMT);
11     BTN   = dio(pin=1, map=0, format=BTN_FRMT);
12     BTN_N= dio(pin=2, map=1, format=BTN_FRMT);
13 }
```

Listing 6.3: Example Signals 1

For the `spi_frmt` formats and `spi_sigs` signals illustrated in Listing 6.4, the illustration presents a possible setup for an SPI application. The `CS_` is assigned to `DIO[0]` with a constant 0 as the data source using the `CS_` format. `CS_` format is consistently configured to `H`, `L`, or `Z`, none of which utilize the data source. Thus, altering the `CS_` data source to 1 remains without impact. Similarly, `SCLK`, `MISO`, and `RST_N` follow the same configuration as `CS_`. Conversely, `MOSI` is connected to `x[7]`, mapping it to the seventh bit of a loaded byte, which enables MSB loading upon left-shifting the data.

```
1  Formats(spi_frmt){
2
3        # CYCLE     0       1       2       3       4
```

```
4     # FORMAT
5       cycle_sel  = [idle , start , clkh , clkl , rst];
6       CS_        = [oHHHH, oLLLL, oLLLL, oLLLL, oZZZZ];
7       SCLK       = [oLLLL, oLLLL, oHHHH, oLLLL, oZZZZ];
8       MOSI       = [oLLLL, oDDDD, oDDDD, oDDDD, oZZZZ];
9       MISO       = [iHHHH, iHHHH, iHHHH, iHHHH, iHHHH];
10      RST_N      = [oHHHH, oHHHH, oHHHH, oHHHH, oLLLL];
11  }
12
13  Signals(spi_sigs){
14     # To be used with GeneralFormats
15      CS_    = dio(pin=0,  map=0,    format=CS_   );
16      SCLK   = dio(pin=1,  map=0,    format=SCLK  );
17      MOSI   = dio(pin=2,  map=x[7], format=MOSI  );
18      MISO   = dio(pin=3,  map=0,    format=MISO  );
19      RST_N  = dio(pin=4,  map=0,    format=RST_N );
20  }
```

Listing 6.4: Example Signals 2

Hint 6.3.2 Selected format in a Signals object must be one of the formats definitions in the current Formats.

6.4 Patterns

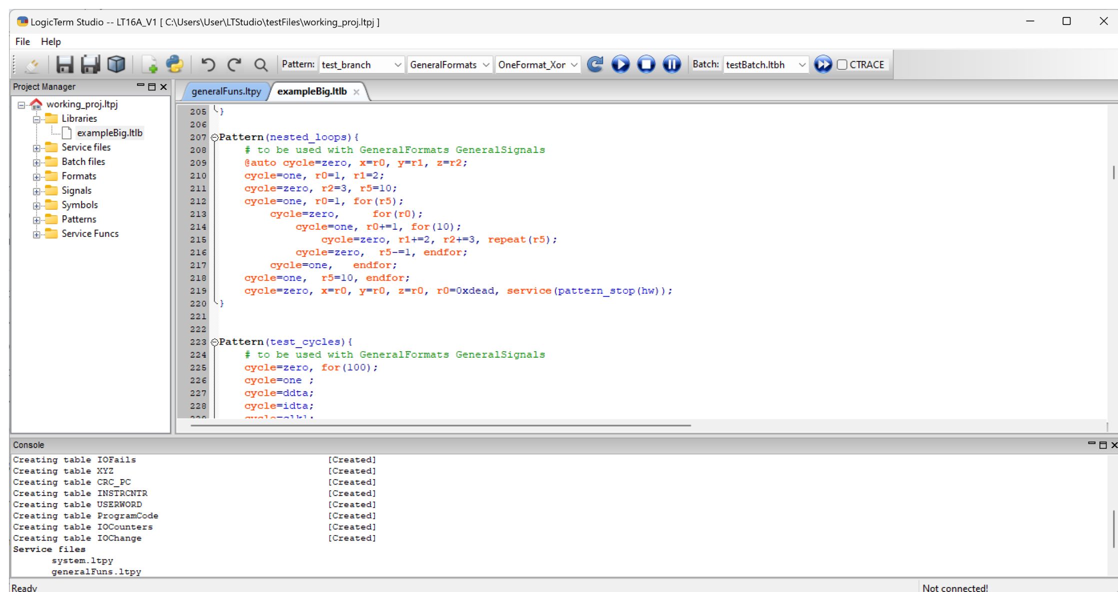


Figure 6.7: LTStudio Pattern

The `Pattern` object defines a programmable sequence of instructions executed by the LT16M's digital core. Each instruction contain micro-instructions that define signal behavior, branching, logging, and service calls. Instructions span 4 ticks (40 ns total) and can control digital I/Os, perform arithmetic, manage memory, and invoke host services.

- 16 General-Purpose Registers (GPRs)
- 2 Timers (24-bit)
- 1K user memory
- 4K logging buffer
- 3 output registers: `x`, `y`, `z`

6.4.1 Pattern Syntax

```
Pattern Object:
===============
   pattern               := Pattern(<patternName>){[compilerInstruction][Instruction]...}
   compilerInstruction   := @using <symbolName>;
                            @auto cycle_uInstr<,uInstr>;
                            @param <parameterName>=<ParameterValue>, ...;
   Instruction           := [(Label)cycle_uInstr[<,ALU1_uInstr><,ALU2_uInstr><,XYZ_uInstr>
                                                 <,iomask_uInstr><,cntrl_uInstr><,log_uInstr>
                                                 <,do_uInstr><,branch_uInstr>];
   validName             := [a-zA-Z_][a-zA-Z0-9_]*
   rx                    := r0, r1 ... r15
   imm_16                := 16-bit integer number
   Label                 := <validName>:
```

Please refer to section 12.4 for full syntax.

6.4.2 Micro-Instructions

Each instruction begins with a `cycle` assignment. The rest are optional and unordered. Table 6.3 list all micro-instructions.

Micro-instruction	Optional	Description	**Example**
Cycle Assignment	No	Select cycle from predefined formats	`cycle=rst`
ALU1	Yes	Execute arithmetic/logical operations	`r1=r2|r3`
ALU2	Yes	Execute arithmetic/logical operations	`r2=r5*r10`
X Assignment	Yes	Assign x to one GPR	`x=r1`
Y Assignment	Yes	Assign x to one GPR	`y=r0`
Z Assignment	Yes	Assign x to one GPR	`z=r15`
Control	Yes	Timer, Flag, Counter controls	`clr(T0)`
IO Strobe	Yes	IO mask configuration for reading	`iomask=0xffff`
Data log	Yes	Log `ADC`, `FAIL`, `INFO`, etc.	`log(FAIL)`
Data Output	Yes	Configure Drive-Only pins	`do=0xf`
Branch	Yes	Execute branch operations	`jmp`

Table 6.3: Micro-instruction for pattern object

Cycle Assignment

Cycle assignment for assigning the current instruction to one of the cycles defined in Signals object.

ALU Operations

ALU assignments for performing up to two arithmetic and logical operations. Operations done by ALU1 are executed in parallel to those done by ALU2. Load/Store operations are also done by ALU1/ALU2. The compiler assigns the first ALU operation in an instruction to ALU1 before it assigns to ALU2.

Please refer to subsection 12.4.2 for full syntax.

ALU operations (register with register)

The supported ALU operations, where both operands are GPRs, are listed in table 6.4.

ALU operations (register with immediate)

The supported ALU operations, where the second operand is immediate, are listed in table 6.5.

ALU Op.	Description	Result (r3)
r3=r0+r1	Addition	8
r3=r1-r0	Subtraction	6
r3=r0|r1	Logical OR	7
r3=r0&r1	Logical AND	1
r3=r0^r1	Logical XOR	6
r3=r1>r0	Set if greater than: r1>r0?0xffff:0	0xffff
r3=r1<r0	Set if less than: r1<r0?0xffff:0	0
r3=r2*r2	Lower 16-bit of Multiplication	0x0001
r3=r2**r2	Higher 16-bit of Multiplication	0xfffe
r3=r1>>>r1	Rotate right	0x8003
r3=r1<<<r0	Rotate left	0x000e

Table 6.4: ALU operations: Register with register assuming r0=1, r1=7, r2=0xffff

ALU Op.	Description	Result (r3)
r3=r0+7	Addition	8
r3=r1-1	Subtraction	6
r3=r0|7	Logical OR	7
r3=r0&7	Logical AND	1
r3=r0^7	Logical XOR	6
r3=r1>>1	Shift right	3
r3=r1<<1	Shift left	0x000e
r3=r2*0xffff	Lower 16-bit of Multiplication	0x0001
r3=r2**0xffff	Higher 16-bit of Multiplication	0xfffe
r3=r1>>>7	Rotate right	0x8003
r3=r1<<<1	Rotate left	0x000e

Table 6.5: ALU operations: Register with immediate assuming r0=1, r1=7, r2=0xffff

Register Assignment

Register assignment can be to an immediate, GPR, mathematical expression or io. An assignment's right-hand side may consist of a mathematical expression, evaluated using Python syntax. Supported operations include {+, -, *, //, |, &, ^, >>, <<}. The expression's result is always truncated to an integer and limited to 16 bits. Using io, users can read the current DIO states.

```
1 Pattern(myPat){
2     cycle=idle, r1=0;                                     # r1=0x0000
3     cycle=idle, r1=0xffff, r2=0b0000_1111_0000_1111;    # r1=0xffff r2=0x0f0f
4     cycle=idle, r1=r2,     r2=r1;                        # r1=0x0f0f r2=0xffff (swap!)
5     cycle=idle, r1=100/2,  r2=((100+1)/2);               # r1=50     r2=50
6     cycle=idle, r1=-1,     r2=65536;                     # r1=0xffff r2=0
7     cycle=idle, r3=io;                                    # r3 = the state of the DIOs
8     ...
```

Listing 6.5: Register assignment

Auxiliary register assignment

```
1 Pattern(myPat){
2     cycle=idle, SEED=0x8765_4321;        # 32-bit assignment
3     cycle=idle, FLIMIT=10;               # 24-bit assignment
```

```
4     cycle=idle, T0=0xffffff;            # 24-bit assignment
5     cycle=idle, T1=0xffffff-10;         # 24-bit assignment
6     ...
```

Listing 6.6: Auxiliary register assignment

Hint 6.4.1 Auxiliary register assignment consume both ALUs, ALU1 and ALU2. In other words, Instructions with auxiliary register assignment cannot have any other ALU micro-instructions.

Random assignment (Using CRC-CCITT (XModem) as Linear-feedback shift register)

The lower 16-bit of SEED will be used for the ALU1 LFSR and the higher ones for ALU2 LFSR. CRC-CCITT (XModem) is used.

```
1 Pattern(myPat){
2     cycle=idle, SEED=0x8765_4321;
3     cycle=idle, r0=0, r1=1;
4     cycle=idle, r0=rand(r0), r1=rand(r1);
5     ...
```

Listing 6.7: Linear-feedback shift register

```
1  def rand(reg_val, SEED=0x0000):
2      """ Compute CRC-CCITT (XModem) for a 16-bit integer with a custom seed.
3      Polynomial: 0x1021
4      """
5      crc = SEED
6      for shift in (8, 0):  # Process high byte first, then low byte
7          crc ^= ((reg_val >> shift) & 0xFF) << 8
8          for _ in range(8):
9              crc = ((crc << 1) ^ 0x1021) if (crc & 0x8000) else (crc << 1)
10             crc &= 0xFFFF
11     return crc
```

Listing 6.8: Linear-feedback shift register python function

```
rand with default (SEED=0x0000): rand(0)=0000, rand(1)=1021, rand(2)=2042, rand(3)=3063, ...
rand with SEED=0x0001: rand(0)=1021, rand(1)=0000, rand(2)=3063, rand(3)=2042, ...
rand with SEED=0x1234: rand(0)=13c6, rand(1)=03e7, rand(2)=3384, rand(3)=23a5, ...
rand with SEED=0xffff: rand(0)=1d0f, rand(1)=0d2e, rand(2)=3d4d, rand(3)=2d6c, ...
```

Memory Access

The ATPG system includes a user memory accessible for both reading and writing by the services and ATPG itself. This memory holds 1024 entries, each being 16 bits in size. Entries can be accessed by either the pattern generator or the host PC through a service or batch process, but only one memory access is allowed per instruction. This feature facilitates data exchange between the pattern generator and services; for instance, the DAC can retrieve a sine wave dataset preloaded by a service from this memory.

```
1  Pattern(myPat){
2      cycle=idle, r0=0, r1=10;
3      cycle=idle, r2=5, r3=2;       # r0=0, r1=10, r2=5, r3=2
4                                    #MEM: 0:xxxx  1:xxxx  2:xxxx  3:xxxx
5
6      cycle=idle, mem[2]=r1;       # Writes 10 to memory location 2
7      cycle=idle, mem[r0]=r2;      # Writes 5 to memory location 0
8      cycle=idle, r0=mem[r3];      # Reads memory location 2 into r0
9      cycle=idle, r1=mem[0];       # Reads memory location 0 into r1
10                                  # r0=10, r1=5, r2=5, r3=2
11                                  #MEM: 0:5  1:xxxx  2:10  3:xxxx
```

Listing 6.9: Memory assignmnet

DAC Control

The system includes two single-ended Digital-to-Analog Converters (DACs) that operate within a 0 to 10V range and support a rate of 1 MS/s, with a current capacity of less than 50mA. The assigned values for the DACs are in millivolts, setting the assignment range between 0 and 10,000. Unlike DAC0, DAC1 features current monitoring and offers a 16-bit resolution. The DAC assignments should be spaced at least 15 cycles.

```
1  Pattern(myPat){
2      cycle=idle, r0=0,     r1=1000;
3      cycle=idle, r2=5000, r3=10_000;
4      cycle=idle, dac0=r0;              # DAC0= 0.000V, DAC1=OLD_VAL
5      cycle=idle, repeat(15);           # DAC delay
6      cycle=idle, dac0=r1;              # DAC0= 1.000V, DAC1=OLD_VAL
7      cycle=idle, repeat(15);           # DAC delay
8      cycle=idle, dac0=r2;              # DAC0= 5.000V, DAC1=OLD_VAL
9      cycle=idle, repeat(15);           # DAC delay
10     cycle=idle, dac0=r3;              # DAC0=10.000V, DAC1=OLD_VAL
11     cycle=idle, dac1=r0;              # DAC0=10.000V, DAC1= 0.000V
12     cycle=idle, repeat(15);           # DAC delay
13     cycle=idle, dac1=r1;              # DAC0=10.000V, DAC1= 1.000V
14     cycle=idle, repeat(15);           # DAC delay
15     cycle=idle, dac1=r2;              # DAC0=10.000V, DAC1= 5.000V
16     cycle=idle, repeat(15);           # DAC delay
17     cycle=idle, dac0=r0, dac1=r0;     # DAC0= 0.000V, DAC1= 0.000V
18     ...
```

Listing 6.10: DAC assignment

Output Assignment

Each instruction generates three outputs: x, y, and z. These outputs can be directed to any of the general-purpose registers (r0, r1, ..., r15). The designated registers serve as data sources for pin assignments specified within Signals objects, as detailed in ??. Assignment is not sticky and it is defaulted to r0 when it is not assigned in an instruction.

```
1  Pattern(myPat){
2                                             #   X      Y      Z
3      cycle=idle, r0=0x0000, r1=0x1111; # 0x0000 0x0000 0x0000
4      cycle=idle, r2=0x2222, r3=0x3333; # 0x0000 0x0000 0x0000
5      cycle=idle, x=r0, y=r1, z=r2;     # 0x0000 0x1111 0x2222
6      cycle=idle;                       # 0x0000 0x0000 0x0000
7      cycle=idle, x=r1, y=r1, z=r1;     # 0x1111 0x1111 0x1111
8      cycle=idle, x=r2, y=r2;           # 0x2222 0x2222 0x0000
9      cycle=idle, x=r3, r3=0xffff;      # 0xffff 0x0000 0x0000
10
11     ...
```

Listing 6.11: X/Y/Z Output Assignments

Control Flags

Control assignment for clearing timers, persistent fail flag and/or fail counter. There can be more than one clear micro-instruction.

- clr(T0) sets timer 0 (T0) to 0
- clr(T1) sets timer 1 (T1) to 0
- clr(PF) clears the persistent fail flag
- clr(FCNTR) clears the io fail counters

```
1  Pattern(myPat){                        #              FCNTR
2                                         #  T0   T1   [0] [1]  PF
3      cycle=idle, T0=10;                 #  10    0    0   0    0
4      cycle=idle, T1=20;                 #   9   20    0   0    0
5      cycle=idle, clr(T1);               #   8    0    0   0    0
```

```
6     cycle=idle, clr(T0);                        #  0    0    0   0   0
7     cycle=low, iomask=0x3;                      #  0    0    1   1   1
8     cycle=low, iomask=0x3;                      #  0    0    2   2   1
9     cycle=low, iomask=0x3;                      #  0    0    3   3   1
10    cycle=low, clr(FCNTR);                      #  0    0    0   0   1
11    cycle=low, clr(PF);                         #  0    0    0   0   0
12    cycle=low, T0=9, iomask=0x1;                #  9    0    1   0   1
13    cycle=low, T1=5, iomask=0x2;                #  8    5    1   1   1
14    cycle=low;                                  #  7    4    1   1   1
15    cycle=low, clr(T0), clr(T1), clr(PF), clr(FCNTR); #  0    0    0   0   0
16    ...
```

Listing 6.12: Control Micro-instruction

IO Mask

IO mask provides the mask for IOs that are being read in the current cycle. IO strobe is enabled by the selected format. Logging the mismatched IO to the database is controlled by the log micro-instruction.

Warning 9 `iomask` will be disabled for pins with an output cycle.

```
1  Formats(myFrmt){
2        # CYCLE      0      1        2      3        4
3    # FORMAT
4      cycle_sel = [zero , one   ,   hiz  , low   , high  ];
5      PIN0_FRMT = [oLLLL, oHHHH,   iZZZZ, iLLLL, iHHHH ];
6  }
7  Pattern(myPat){
8                # Assuming that DIO[0] is tied to GND
9                # Assuming that DIO[1] is tied to VCC
10                                # FAIL[0] FAIL[1]   FCNTR[0] FCNTR[1]
11     cycle=low;                 #   0       0          0        0
12     cycle=low, iomask=0b00;    #   0       0          0        0
13     cycle=low, iomask=0b01;    #   0       0          0        0
14     cycle=low, iomask=0b10;    #   0       1          0        1
15     cycle=low, iomask=0b11;    #   0       1          0        2
16     cycle=high, iomask=0b00;   #   0       0          0        2
17     cycle=high, iomask=0b01;   #   1       0          1        2
18     cycle=high, iomask=0b10;   #   0       0          1        2
19     cycle=high, iomask=0b11;   #   1       0          2        2
20
21     cycle=zero, iomask=0b11;   #   0       0          2        2  # cycle is output!
22     cycle=one,  iomask=0b11;   #   0       0          2        2  # no read
23     ...
```

Listing 6.13: IO Mask Assignment

Logging

This micro-instruction enables the logging of different types of data to the database. Data can be

- read mis-match (aka fail) – FAIL
- instruction information – INFO
- IOs fail counters (Lower/Upper) – FCNTRL/FCNTRH
- IO changes – CHANGE
- ADC reads – ADC

Logging is only allowed for one type of data `(FAIL|CHANGE|INFO|FCNTRL|FCNTRH|ADC)`.

Warning 10 The pattern generator has a buffer of 4K entries. The size of `FAIL/CHANGE` packets is 1; meaning you can log 4K packets at highest speed. The size of `INFO/ADC` packets is 2;

meaning you can log 2K packets at highest speed. The size of FCNTRL/FCNTRH packets is 4; meaning you can log 1K packets at highest speed. ■

FAIL

Each instruction can record read mismatches, termed as FAIL. A FAIL packet is generated when any enabled IOs, as determined by the iomask, experiences a mismatch. This packet is recorded in the IOFails table which contains the following fields:

- X: X Output register (16-bit)
- Y: Y Output register (16-bit)
- Z: Lower byte of Z Output register (8-bit)
- Tick: Tick bit mask indicating the fail occurrence (4-bit)
- IO: IO bit mask for failed IOs (16-bit)

```
1  Formats(myFrmt){
2        # CYCLE     0      1        2      3       4
3    # FORMAT
4      cycle_sel = [zero , one  ,   hiz  , low  , high , chng ];
5      PIN0_FRMT = [oLLLL, oHHHH,   iZZZZ, iLLLL, iHHHH, iZMZZ];
6  }
7  Pattern(myPat){
8               # Assuming that DIO[0] is tied to GND
9               # Assuming that DIO[1] is tied to VCC
10                                                     # FAIL[0] FAIL[1]
11     cycle=low,  r0=0,                log(FAIL);  #   0       0
12     cycle=low,  r0+=1, iomask=0b00, log(FAIL);  #   0       0
13     cycle=low,  r0+=1, iomask=0b01, log(FAIL);  #   0       0
14     cycle=low,  r0+=1, iomask=0b10, log(FAIL);  #   0       1   <- A fail is logged
15     cycle=low,  r0+=1, iomask=0b11, log(FAIL);  #   0       1   <- A fail is logged
16     cycle=high, r0+=1, iomask=0b00, log(FAIL);  #   0       0
17     cycle=high, r0+=1, iomask=0b01, log(FAIL);  #   1       0   <- A fail is logged
18     cycle=high, r0+=1, iomask=0b10, log(FAIL);  #   0       0
19     cycle=high, r0+=1, iomask=0b11, log(FAIL);  #   1       0   <- A fail is logged
20 # OUTPUT
21 #TABLE[IOFails] (count=0):
22 #   -----------------------------------------------------------------
23 #      id     |     X     |     Y     |     Z     |   Tick    |    IO
24 #   -----------------------------------------------------------------
25 #
26 #
27 #     ...
```

Listing 6.14: Log Enable – FAIL

CHANGE

Each instruction is capable of recording a read change, referred to as CHANGE. A CHANGE packet is generated whenever any enabled IOs, identified by the iomask, undergo a change since the last read. Initially, the system sets the last read to 0 at the beginning of a pattern.

This packet is documented in the IOChange table, which includes the following fields:

- InstrCntr: A 44-bit instruction counter that begins at 0 at the pattern's start.
- IO: A 16-bit IO bit mask for changed IOs

Warning 11 A memory read (M) tick must occur for CHANGE packets to be logged successfully. ■

```
1  Formats(myFrmt){
2        # CYCLE     0      1        2      3       4
3    # FORMAT
4      cycle_sel = [zero , one  ,   hiz  , low  , high , chng ];
5      PIN0_FRMT = [oLLLL, oHHHH,   iZZZZ, iLLLL, iHHHH, iZMZZ];
6  }
7  Pattern(myPat){
```

```
8               # Assuming that DIO[0] is tied to GND
9               # Assuming that DIO[1] is tied to VCC
10                                                 # FAIL[0] FAIL[1]
11      cycle=chng, r0=0,                log(CHANGE); #   0      0
12      cycle=chng, r0+=1, iomask=0b00, log(CHANGE); #   0      0
13      cycle=chng, r0+=1, iomask=0b01, log(CHANGE); #   0      0
14      cycle=chng, r0+=1, iomask=0b10, log(CHANGE); #   0      1   <- A fail is logged
15      cycle=chng, r0+=1, iomask=0b11, log(CHANGE); #   0      1   <- A fail is logged
16      cycle=chng, r0+=1, iomask=0b00, log(CHANGE); #   0      0
17      cycle=chng, r0+=1, iomask=0b01, log(CHANGE); #   1      0   <- A fail is logged
18      cycle=chng, r0+=1, iomask=0b10, log(CHANGE); #   0      0
19      cycle=chng, r0+=1, iomask=0b11, log(CHANGE); #   1      0   <- A fail is logged
20  # OUTPUT
21  % TABLE[IOChange] (count=1):
22  %   -------------------------------
23  %       id    |InstrCntr |    IO
24  %   -------------------------------
25  %       3      |     1      |     1
26
27      ...
```

Listing 6.15: Log Enable – CHANGE

INFO

Each instruction is capable of recording some of the instruction data, referred to as `INFO`. An `INFO` packet is generated whenever it is enabled in the instruction.

This packet is documented in the INFO table, which includes the following fields:

- X: X Output register (16-bit)
- Y: Y Output register (16-bit)
- Z: Z Output register (16-bit)
- curPC: current program counter (10-bit)
- nextPC: next program counter (10-bit)
- InstrCntr: A 44-bit instruction counter that begins at 0 at the pattern's start.

Warning 12 Do not call `log(INFO)` for more than 2K instructions before calling a service.

```
1  Formats(myFrmt){
2        # CYCLE     0      1        2      3       4
3    # FORMAT
4      cycle_sel  = [zero , one   ,  hiz  , low  , high , chng ];
5      PINO_FRMT  = [oLLLL, oHHHH,   iZZZZ, iLLLL, iHHHH, iZMZZ];
6  }
7  Pattern(myPat){
8               # Assuming that DIO[0] is tied to GND
9               # Assuming that DIO[1] is tied to VCC
10                                                 # FAIL[0] FAIL[1]
11      cycle=low,  r0=0,                log(INFO); #   0      0
12      cycle=low,  r0+=1, iomask=0b00, log(INFO); #   0      0
13      cycle=low,  r0+=1, iomask=0b01, log(INFO); #   0      0
14      cycle=low,  r0+=1, iomask=0b10, log(INFO); #   0      1   <- A fail is logged
15      cycle=low,  r0+=1, iomask=0b11, log(INFO); #   0      1   <- A fail is logged
16      cycle=high, r0+=1, iomask=0b00, log(INFO); #   0      0
17      cycle=high, r0+=1, iomask=0b01, log(INFO); #   1      0   <- A fail is logged
18      cycle=high, r0+=1, iomask=0b10, log(INFO); #   0      0
19      cycle=high, r0+=1, iomask=0b11, log(INFO); #   1      0   <- A fail is logged
20  # OUTPUT
21  TABLE[Info] (count=0):
22    ----------------------------------------------------------------------------
23       id    |     X     |     Y     |     Z     |  curPC    |  nextPC   |InstrCntr
24    ----------------------------------------------------------------------------
25
26      ...
```

Listing 6.16: Log Enable – INFO

FCNTRL/FCNTRH

Each instruction is capable of reporting the fail counter per each of the IOs, referred to as FCNTRL/FCNTRH. FCNTRL packet reports the fail counters for IO[0] to IO[7], while FCNTRH packet reports the fail counters for IO[8] to IO[15].

This packet is documented in the IOCounters table, which includes the following fields:

- IO: The IO number (4-bit)
- Counter: fail counter – number of mismatches (24-bit)

> **Warning 13** Do not call log(FCNTRL) or log(FCNTRH) for more than 1K instructions before calling a service.

```
1  Formats(myFrmt){
2        # CYCLE      0       1         2       3        4
3    # FORMAT
4      cycle_sel  = [zero  , one   ,    hiz   , low   , high  , chng ];
5      PINO_FRMT  = [oLLLL , oHHHH ,   iZZZZ , iLLLL , iHHHH , iZMZZ];
6  }
7  Pattern(myPat){
8                # Assuming that DIO[0] is tied to GND
9                # Assuming that DIO[1] is tied to VCC
10                                                 # FAIL[0] FAIL[1]
11     cycle=low,   r0=0,                   ;  #    0      0
12     cycle=low,   r0+=1, iomask=0b00;  #    0      0
13     cycle=low,   r0+=1, iomask=0b01;  #    0      0
14     cycle=low,   r0+=1, iomask=0b10;  #    0      1  <- A fail is logged
15     cycle=low,   r0+=1, iomask=0b11;  #    0      1  <- A fail is logged
16     cycle=high, r0+=1, iomask=0b00;  #    0      0
17     cycle=high, r0+=1, iomask=0b01;  #    1      0  <- A fail is logged
18     cycle=high, r0+=1, iomask=0b10;  #    0      0
19     cycle=high, r0+=1, iomask=0b11, log(FCNTRL);  #   1      0  <- A fail is logged
20     cycle=high, r0+=1, iomask=0b11, log(FCNTRH);  #   1      0  <- A fail is logged
21
22 # OUTPUT
23 TABLE[IOCounters] (count=8):
24   --------------------------------
25       id     |     IO     | Counter
26   --------------------------------
27       4       |      0       |      1
28       5       |      1       |      0
29       6       |      2       |      0
30       7       |      3       |      0
31       8       |      4       |      0
32       9       |      5       |      0
33       10      |      6       |      0
34       11      |      7       |      0
35
36     ...
```

Listing 6.17: Log Enable – FCNTRL/FCNTRH

ADC

Each instruction is capable of reporting the ADC readings, referred to as ADC. ADC packet includes four ADC readings, ADC0 voltage, ADC1 voltage, DAC1 current and PPS current or voltage.

This packet is documented in the AnalogData table, which includes the following fields:

- Type: adc0, adc1, dac1mi, ppsmi, ppsmv
- Value: float number for current or voltage
- Unit: Value unit – V or mA
- curPC: current program counter (10-bit)
- InstrCntr: A 44-bit instruction counter that begins at 0 at the pattern's start.

Warning 14 Do not call `log(ADC)` for more than 2K instructions before calling a service.

```
1  Formats(myFrmt){
2        # CYCLE      0       1         2       3        4
3    # FORMAT
4      cycle_sel  = [zero  , one   ,   hiz   , low   , high  , chng ];
5      PINO_FRMT  = [oLLLL , oHHHH ,   iZZZZ , iLLLL , iHHHH , iZMZZ];
6  }
7  Pattern(myPat){
8               # Assuming that DIO[0] is tied to GND
9               # Assuming that DIO[1] is tied to VCC
10                                                 # FAIL[0] FAIL[1]
11     cycle=low ,  r0=0,              ;  #   0      0
12     cycle=low ,  r0+=1, log(ADC);   #   0      0
13     cycle=low ,  r0+=1, log(ADC);   #   0      0
14     cycle=low ,  r0+=1, log(ADC);   #   0      1   <- A fail is logged
15     cycle=low ,  r0+=1, log(ADC);   #   0      1   <- A fail is logged
16     cycle=high , r0+=1, log(ADC);   #   0      0
17     cycle=high , r0+=1, log(ADC);   #   1      0   <- A fail is logged
18     cycle=high , r0+=1, log(ADC);   #   0      0
19     cycle=high , r0+=1, log(ADC);   #   1      0   <- A fail is logged
20     cycle=high , r0+=1, log(ADC);   #   1      0   <- A fail is logged
21
22 # OUTPUT
23 #TABLE[AnalogData] (count=0):
24 #------------------------------------------------------------------
25 #     id     |    Type    |  Value     |   Unit     |  curPC    |InstrCntr
26 #------------------------------------------------------------------
27 #     165          adc1       12.6         V          12       120342
28 #     167          adc0       12e-3        V          12       212423
29 #     186          dac1mi     0.3          mA         12       212515
30 #     187          ppsmi      1.232        mA         12       212513
31 #     189          ppsmv      5.0          V          12       212514
32 #     ...
```

Listing 6.18: Log Enable – ADC

Drive-Only Assignment

This micro-instruction sets the four drive-only pins to a certain value. The value of these pins are set in the current cycle.

```
Data Drive-Only Micro-Instruction:
===================================
    imm_4b       := 4 bit integer
    do_uInstr    := do=imm_4b
```

```
1  # Connect DIO[3:0] to DO[3:0]
2  Formats(do_frmt){
3    # FORMAT
4      cycle_sel  = [ low  ];
5      PINO_FRMT  = [ iZZLZ ];
6  }
7
8  Pattern(test_do){
9                                                 #        (last)          (first)
10     cycle=low , r0=0;                          # Tick:     3      2    1     0
11     cycle=low , iomask=0xf , log(FAIL), r0+=1, do=1;  # do=xxxx xxxx xxxx 0000
12     cycle=low , iomask=0xf , log(FAIL), r0+=1, do=2;  # do=xxxx xxxx xxxx xxxx
13     cycle=low , iomask=0xf , log(FAIL), r0+=1, do=3;  # do=xxxx xxxx xxxx xxxx
14     cycle=low , iomask=0xf , log(FAIL), r0+=1;        # do=0000 xxxx xxxx xxxx
15     cycle=low , iomask=0xf , log(FAIL), r0+=1, do=4;  # do=xxxx 0000 0000 0000
16     cycle=low , iomask=0xf , log(FAIL), r0+=1, do=5;  # do=xxxx xxxx xxxx xxxx
17     cycle=low , iomask=0xf , log(FAIL), r0+=1, do=6;  # do=xxxx xxxx xxxx xxxx
18     cycle=low , iomask=0xf , log(FAIL), r0+=1, do=7;  # do=xxxx xxxx xxxx xxxx
19     cycle=low , iomask=0xf , log(FAIL), r0+=1, do=8;  # do=xxxx xxxx xxxx xxxx
20     cycle=low , iomask=0xf , log(FAIL), r0+=1, do=9;  # do=xxxx xxxx xxxx xxxx
21     cycle=low , iomask=0xf , log(FAIL), r0+=1, do=10; # do=xxxx xxxx xxxx xxxx
```

```
22      cycle=low, iomask=0xf, log(FAIL), r0+=1, do=11; # do=xxxx xxxx xxxx xxxx
23      cycle=low, iomask=0xf, log(FAIL), r0+=1, do=12; # do=xxxx xxxx xxxx xxxx
24      cycle=low, iomask=0xf, log(FAIL), r0+=1, do=13; # do=xxxx xxxx xxxx xxxx
25      cycle=low, iomask=0xf, log(FAIL), r0+=1, do=14; # do=xxxx xxxx xxxx xxxx
26      cycle=low, iomask=0xf, log(FAIL), r0+=1, do=15; # do=xxxx xxxx xxxx xxxx
27      cycle=low, service(pattern_stop(hw));
28  }
29
30      ...
```

Listing 6.19: Data Drive-Only Assignments

Branching

This micro-instruction is responsible for the flow control of the pattern execution. Branches can be jumps, repeats, loops, subroutine calls and service calls. An instruction can have no more than one branch.

Please refer to subsection 12.4.8 for full syntax.

Jump

Jumps are categorized into two types. Unconditional jumps directly alter the program counter to a designated label without conditions. Conversely, conditional jumps depend on certain flag states to determine execution. The flags available include fail (F), persistent fail (PF), fail limit exceeded (FLE), timers (T0, T1), user flag (UF), and zero flag (Z1).

Flag	Variation	Description
F	F NF	Fail flag for last read
PF	PF NPF	Persistent fail: was there any fail?
FLE	FLE NFLE	Fail limit exceeded: Fail counter for any IO is greater that FLIMIT
T0	T0 NT0	Timer 0 is expired
T1	T1 NT1	Timer 1 is expired
UF	UF NUF	User flag: a flag representing the UBTN (1 when pressed).
Z1	Z1 NZ1	Zero flag from ALU1; Note: No zero flag for ALU2.

Table 6.6: Flags used for conditional branching.

```
1   Pattern(test_branch){
2       # to be used with GeneralFormats GeneralSignals
3       @auto cycle=ddta, x=r0, y=r0, z=r0;
4       LABEL_0 :   cycle=l1st, r0=0xf, jmp(0, FAIL_SUB);  # same as jmp(FAIL_SUB)
5                   cycle=l1st, T0=10, clr(T0);
6       LABEL_1 :   cycle=l1st, r0=1,  jmp(1, LABEL_3);
7       LABEL_2 :   cycle=l1st, r0=2;   # Never gonna be reached!
8       LABEL_3 :   cycle=l1st, r0+=1, jmp(NT0, LABEL_3);
9       cycle=l1st, service(echo(hw,"Time is done!"));
10      LABEL_4  : cycle=l1st,  r0=4, jmp(1,  LABEL_5);
11      LABEL_5  : cycle=l1st,  r0=5, jmp(1,  LABEL_7);
12      LABEL_6  : cycle=l1st,  r0=6, jmp(1,  LABEL_3);
13      LABEL_7  : cycle=l1st,  r0=7, jmp(1,  LABEL_15);
14      LABEL_8  : cycle=l1st,  r0=8, jmp(1,  LABEL_3);
15      LABEL_9  : cycle=l1st,  r0=9, jmp(1,  LABEL_3);
16      LABEL_10 : cycle=l1st, r0=10, jmp(1,  LABEL_3);
17      LABEL_11 : cycle=l1st, r0=11, jmp(1,  LABEL_3);
18      LABEL_12 : cycle=l1st, r0=12, jmp(1,  LABEL_3);
19      LABEL_13 : cycle=l1st, r0=13, jmp(1,  LABEL_3);
20      LABEL_14 : cycle=l1st, r0=14, jmp(1,  LABEL_3);
21      LABEL_15 : cycle=l1st, r0=15, jmp(1,  HAHA);
22
23      PASS: cycle=l1st, r0=0xFFFF;
24            cycle=l1st, jmp(LABEL_4);
```

```
    FAIL_SUB: cycle=l1st, r0=0xDEAD;

    HAHA:  cycle=l1st, service(echo(hw,"Here"));

    cycle=zero, service(pattern_stop(hw));
}
```

Listing 6.20: Branching using conditional and unconditional jumps

Call

Calls are categorized into two types. Unconditional calls modify the program counter unconditionally, whereas conditional calls rely on flag conditions such as fail (F), persistent fail (PF), fail limit exceeded (FLE), timers (T0, T1), user flag (UF), and zero flag (Z1) to determine execution. Each call subroutine mandates a `return` statement, and users can nest up to 16 calls.

```
Pattern(test_call2){
          @auto cycle=ddta, x=r15, y=r0, z=r1;

           cycle=ddta, r0=0, r1=10, x=r0;
           cycle=ddta, r0=1, r15=0;
           cycle=ddta, r0=2, do=0xf, call(1, CALL_TRUE); # Always jump do=0xf,
           cycle=ddta, r0=3;
           cycle=zero, r0=4, call(0, CALL_ERROR) ;  # Never jump

           cycle=zero, r2=r1-10, r0=5; # Z1 FLAG is set
           cycle=zero, r0=6, do=0xf, call(Z1, CALL_TRUE) ;

           cycle=zero, r0=7, r2=r1+r0;
           cycle=zero, r0=8, do=0xf, call(NZ1, CALL_TRUE) ;

           cycle=zero, r0=9, iomask=0xffff; # Will not fail
           cycle=zero, r0=10, do=0xf, call(NF, CALL_TRUE);

           cycle=zero, r0=11;
           cycle=zero, r0=12, do=0xf, call(NPF, CALL_TRUE);

           cycle=high, r0=13, iomask=0xffff; # Will fail
           cycle=zero, r0=14;
           cycle=zero, r0=15, do=0xf, call(F, CALL_TRUE);

           cycle=zero, r0=16;
           cycle=zero, r0=17, do=0xf, call(PF, CALL_TRUE); # Already has a fail

           cycle=zero, r0=18;
           cycle=ddta, FLIMIT=10;
           cycle=zero, r0=19, do=0xf, call(NFLE, CALL_TRUE); # Already has a fail

           cycle=high, r0=20, iomask=0xffff, repeat(3); # Assuming FLIMIT_REG=10
           cycle=zero, r0=21, do=0xf, call(FLE, CALL_TRUE);
           cycle=ddta, T0=10;
           cycle=zero, r0=22, clr(T0);
           cycle=zero, r0=23, do=0xf, call(NT0, CALL_TRUE); # Assuming T0=10

           cycle=zero, r0=24, repeat(8);
           cycle=zero, r0=25, do=0xf, call(T0, CALL_TRUE);
           cycle=ddta, T1=84;
           cycle=zero, r0=26;
           cycle=zero, r0=27, clr(T1);
           cycle=zero, r0=28, do=0xf, call(NT1, CALL_TRUE); # Assuming T1=20

           cycle=zero, r0=29, repeat(80);
           cycle=zero, r0=30, do=0xf, call(T1, CALL_TRUE);

           cycle=zero, r0=31;
           cycle=zero, r0=32, do=0xf,call(NUF, CALL_TRUE);

           cycle=zero, r0=33;
           cycle=zero, r0=34, call(UF, CALL_ERROR);
```

```
54
55              cycle=ddta,do=5,    service(pattern_stop(hw));
56
57 CALL_TRUE:   cycle=ddta, r15=r15<<1;
58              cycle=ddta, r15=r15|1, return;
59
60 CALL_ERROR:  cycle=ddta, r0=0xffff, r15=0;
61              cycle=ddta, r0=0xffff, return;
62
63 }
```

Listing 6.21: Branching using conditional and unconditional calls

Loop

Hardware mechanisms facilitate the repetition of code blocks via loops, initiated with `for` and terminated with `endfor`. Repetition counts can be 16-bit immediate or sourced from a GPR. Users can nest up to 16 for loops.

```
1  Pattern(nested_loops){
2      # to be used with GeneralFormats GeneralSignals
3      @auto cycle=zero, x=r0, y=r1, z=r2;
4      cycle=one, r0=1, r1=2;
5      cycle=zero, r2=3, r5=10;
6      cycle=one, r0=1, for(r5);
7          cycle=zero,      for(r0);
8              cycle=one, r0+=1, for(10);
9                  cycle=zero, r1+=2, r2+=3, repeat(r5);
10             cycle=zero,  r5-=1, endfor;
11         cycle=one,   endfor;
12     cycle=one,  r5=10, endfor;
13     cycle=zero, x=r0, y=r0, z=r0, r0=0xdead, service(pattern_stop(hw));
14 }
```

Listing 6.22: Branching using `for` loops

Repeat

Repeat branching allows for the execution of an instruction multiple times, with repetition counts either 16-bit immediate or obtained from a GPR.

```
1  Pattern(repeatTest){
2      # to be used with GeneralFormats GeneralSignals
3      @auto cycle=zero, x=r0, y=r1, z=r2;
4
5      cycle=zero, r1=0, r2=0xff;
6      cycle=zero, r0=3, r2=0xaa;
7
8      cycle=zero, r0+=1, r1+=2, for(10);
9      cycle=one,  r0+=1, r1+=2, repeat (2);
10     cycle=clk1, r0+=1, r1+=2, repeat (2);
11     cycle=zero, r0+=1, r1+=2, repeat (r0);
12     cycle=clk2, r0+=1, r1+=2, repeat (4);
13     cycle=one,  r0+=1, r1+=2, endfor;
14   END: cycle=zero, service(pattern_stop(hw));
15 }
```

Listing 6.23: Branching using repeats

Service

Service branching enables the pattern generator to invoke a Python function while maintaining the current instruction until the service returns, either as `CONTINUE` to proceed, or `STOP` to halt the pattern.

```
1  Pattern(service_call){
2      cycle=low, r1+=1, service( echo(hw, 'Hello World') ); # Prints and CONTINUE
3      cycle=low, r0=0x1111;
```

```
4       cycle=low, r0=0x2222;
5       cycle=low, service(pattern_stop(hw));                    # STOP
6   }
```

Listing 6.24: Stalling the pattern and calling a service on host

6.4.3 Compiler Instructions

```
compilerInstruction  := @using <symbolName>;
                        @auto cycle_uInstr<,uInstr>;
                        @param <parameterName>=<ParameterValue>, ...;
```

Patterns might incorporate compiler instructions, which are preparatory and not executed. The `@auto` and `@using` instructions aid in code clarity, while the `@param` instruction facilitates pattern parameterization.

```
1   Pattern(param_test){
2       @param TIMES = 1, P1=1, P2=1, P3=1, P4=1;
3       @auto cycle=zero, x=r0, y=r1, z=r2, do=0xf;
4
5       cycle=zero, r0=0xba11, for(TIMES);
6           cycle=zero, repeat(P1);
7           cycle=zero, repeat(P2);
8           cycle=zero, repeat(P3);
9           cycle=zero, repeat(P4);
10          cycle=zero, repeat(P1);
11          cycle=zero, repeat(P2);
12          cycle=zero, repeat(P3);
13          cycle=zero, repeat(P4);
14      cycle=zero, endfor;
15      cycle=zero,  service(pattern_stop(hw));
16  }
```

Listing 6.25: Compiler instructions are ment to improve pattern readability and parameterization.

6.4.4 Notes

- Only one branch per instruction is allowed.
- Auxiliary register assignments (e.g., `T0=...`) consume both ALUs.
- Logging buffer is 4K entries; use `service()` to offload.
- `x`, `y`, `z` default to `r0` if not assigned.

6.5 Services

Services are Python functions that run on the host PC. They can be triggered by patterns to perform tasks such as:

- Extracting logged data
- Plotting results
- Controlling external instruments
- Managing test flow

The first argument for a service must be *hw* and it **must** return, `hw.CONTINUE` to continue running the calling pattern or `hw.STOP` to stop the pattern. If a system call or any python function does not return `hw.CONTINUE` or `hw.STOP`, the the user must create a service that handles those returns.

```
1   def echo(hw, msg):
2       """ print the string <s> """
3       print(msg)
4       return hw.CONTINUE
5
6   def sleep_sec(hw, num_secs=1):
```

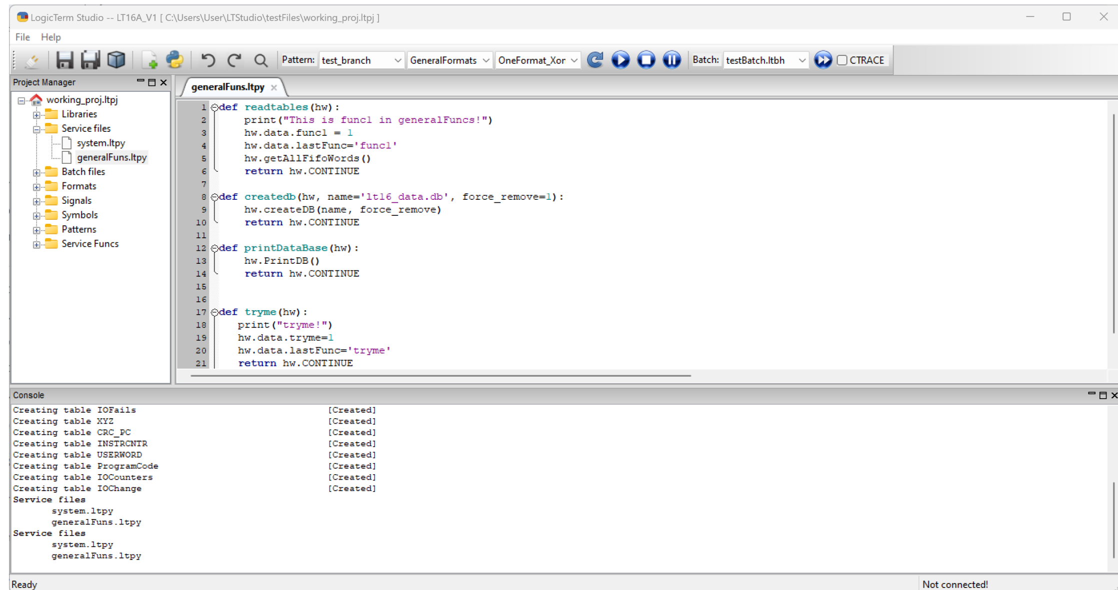


Figure 6.8: LTStudio Services

```
7      """ Sleep for <num_secs> seconds -- default=1 """
8      from time import sleep
9      sleep(num_secs)
10     return hw.CONTINUE
11
12 def stop(hw):
13     """  stop the pattern  """
14     return hw.STOP
```

Listing 6.26: Service example

How it is invoked:

```
1 Pattern(myPat){
2     cycle=idle;
3     cycle=idle, service(echo(hw,"Hello World!"));
4     cycle=idle, service(sleep_sec(hw,5));
5     cycle=idle, service(echo(hw,"Bye!"));
6     cycle=idle, service(stop(hw));
```

Listing 6.27: Invoking services. Note that none of the services return something other than `hw.CONTINUE` or `hw.STOP`

6.5.1 Wrappers for services

Accessing GPRs and pattern registers

- `hw.printGPRs` prints the value of all 16 GRPs.
- `hw.getGPR` returns the value for a specific GPR.
- `hw.getPC` returns the current program counter that is the service is being called from.
- `hw.getInstrCounter` returns the current instruction counter.

```
1 def printATPG_regs(hw):
2     hw.printGPRs()
3     print("GPR[0] =%.04x"% hw.getGPR(0))
4     print("Current program counter is at", hw.getPC())
5     print("Current instruction counter is at", hw.getInstrCounter())
6     return hw.CONTINUE
```

Listing 6.28: Create a service for the system call wrappers with returned data.

```
Pattern(service_pat_call){
    cycle=low, r0=0x0000,   r1=0x1111;
    cycle=low, r2=0x2222,   r3=0x3333;
    cycle=low, r4=0x4444,   r5=0x5555;
    cycle=low, r6=0x6666,   r7=0x7777;
    cycle=low, r8=0x8888,   r9=0x9999;
    cycle=low, r10=0xaaaa, r11=0xbbbb;
    cycle=low, r12=0xcccc, r13=0xdddd;
    cycle=low, r14=0xeeee, r15=0xffff;
    cycle=low, service( hw.printGPRs() );
    cycle=low, r0+=1, service( hw.printGPRs() ); # Note that r0 cannot be predicted
    cycle=low, r0+=1, service( printATPG_regs(hw) );
    cycle=low, service(stop(hw));
}
```

Listing 6.29: Accessing GPRs services

```
Console
Running [3] service_pat_call using the format=DefaultFormats and signal=DefaultSignals and parameters= {}
==============================================================================================
Format/Signal/Pattern were successfully compiled
Executing service >> hw.printGPRs()
r0=0000 r1=1111 r2=2222 r3=3333 r4=4444 r5=5555 r6=6666 r7=7777 r8=8888 r9=9999 r10=aaaa r11=bbbb r12=cccc r13=dddd r14=eeee r15=ffff
Executing service >> hw.printGPRs()
r0=2ca3 r1=1111 r2=2222 r3=3333 r4=4444 r5=5555 r6=6666 r7=7777 r8=8888 r9=9999 r10=aaaa r11=bbbb r12=cccc r13=dddd r14=eeee r15=ffff
Executing service >> printATPG_regs(hw)
r0=ca40 r1=1111 r2=2222 r3=3333 r4=4444 r5=5555 r6=6666 r7=7777 r8=8888 r9=9999 r10=aaaa r11=bbbb r12=cccc r13=dddd r14=eeee r15=ffff
GPR[0] =79bb
Current program counter is at 10  while the next is at 10
Current instruction counter is at 13030529
Executing service >> stop(hw)
Pattern Done:  CRC= 93394e51   InstrCounter= 18669089 curPC= 11, nextPC= 11

The pattern has been successfully stopped
```

Figure 6.9: Output of `service_pat_call` pattern for printing GPRs, PC, nextPC and InstrCounter.

Accessing USERLED

- `hw.setUSERLED` sets the User RGB LED.
- `hw.getUSERLED` gets the current status of the User RGB LED.

```
def invert_userled(hw):
    rgb_val=hw.getUSERLED()
    r = (rgb_val>>16)&0xff
    g = (rgb_val>>8)&0xff
    b = (rgb_val>>0)&0xff
    hw.setUSERLED( r^0xff, g^0xff, b^0xff )
    return hw.CONTINUE
```

Listing 6.30: Create a service for the system call wrappers (`hw.getUSERLED`) with returned data.

```
Pattern(service_userled){
    cycle=low;
    cycle=low, service(hw.setUSERLED(0x00, 0x00, 0x00));
    cycle=low, service(hw.setUSERLED(0xff, 0x00, 0x00));
    cycle=low, service(hw.setUSERLED(0x00, 0xff, 0x00));
    cycle=low, service(hw.setUSERLED(0x00, 0x00, 0xff));
    cycle=low, service(hw.setUSERLED(0xff, 0xff, 0xff));
    cycle=low;
    cycle=low, service(invert_userled(hw));
    cycle=low, service(stop(hw));
}
```

Listing 6.31: Accessing USERLED

Accessing USERWORD

USERWORD is a hardware register that carries its value between patterns.

- `hw.setUSERWORD` sets the value of the 64 bit User Word.
- `hw.getUSERWORD` gets the value of the 64 bit User Word.

```
Console
Running [5] service_userled using the format=DefaultFormats and signal=DefaultSignals and parameters= {}
=====================================================================================================
Format/Signal/Pattern were successfully compiled
Executing service >> hw.setUSERLED(0x00,0x00,0x00)
Executing service >> hw.setUSERLED(0xff,0x00,0x00)
Executing service >> hw.setUSERLED(0x00,0xff,0x00)
Executing service >> hw.setUSERLED(0x00,0x00,0xff)
Executing service >> hw.setUSERLED(0xff,0xff,0xff)
Executing service >> invert_userled(hw)
Executing service >> stop(hw)
Pattern Done:  CRC= 552d22c8   InstrCounter= 7685869 curPC= 8, nextPC= 8
The pattern has been successfully stopped
```

Figure 6.10: The output of `service_userled` pattern

```
1 def print_userword(hw):
2     print("USERWORD=%.016x"%(hw.getUSERWORD()))
3     return hw.CONTINUE
```

Listing 6.32: Create a service for (`hw.getUSERWORD`)

```
1 Pattern(service_userword){
2     cycle=low;
3     cycle=low, service(print_userword(hw));
4     cycle=low, service(hw.setUSERWORD(0xfedcba9876543210));
5     cycle=low, service(print_userword(hw));
6     cycle=low, service(stop(hw));
7 }
```

Listing 6.33: Accessing USERWORD

```
Console
Running [6] service_userword using the format=DefaultFormats and signal=DefaultSignals and parameters= {}
=====================================================================================================
Format/Signal/Pattern were successfully compiled
Executing service >> print_userword(hw)
USERWORD=0000000000000000
Executing service >> hw.setUSERWORD(0xfedcba9876543210)
Executing service >> print_userword(hw)
USERWORD=fedcba9876543210
Executing service >> stop(hw)
Pattern Done:  CRC= 6904bb59   InstrCounter= 5746088 curPC= 4, nextPC= 4
The pattern has been successfully stopped
```

Figure 6.11: Output for `service_userword` pattern

Accessing USERBUTTON

- `hw.waitUSERBUTTON(msg)` will bring a GUI window waiting for user to press UBTN to proceed. Default message is "Press UBTN to proceed".
- `hw.getUSERBUTTON` returns the value of the shift register for button status, active high. 32 bit register that samples the user LED at 1msec rate.

```
1 def print_pressed_userbutton(hw):
2     print("Before pressing: %.08x"%hw.getUSERBUTTON())
3     print("Waiting for user to press:")
4     ubtn=hw.getUSERBUTTON()
5     while ubtn==0:
6         ubtn=hw.getUSERBUTTON()
7     print("After pressing:%.08x"%ubtn)
8     return hw.CONTINUE
```

Listing 6.34: Create a service for (`hw.getUSERBUTTON`)

```
1 Pattern(service_userbtn){
2     cycle=low;
3     cycle=low, r0+=1, service(hw.waitUSERBUTTON("Please press user button ..."));
4     cycle=low, service(print_pressed_userbutton(hw));
5     cycle=low, service(stop(hw));
6 }
```

Listing 6.35: Accessing USERBUTTON

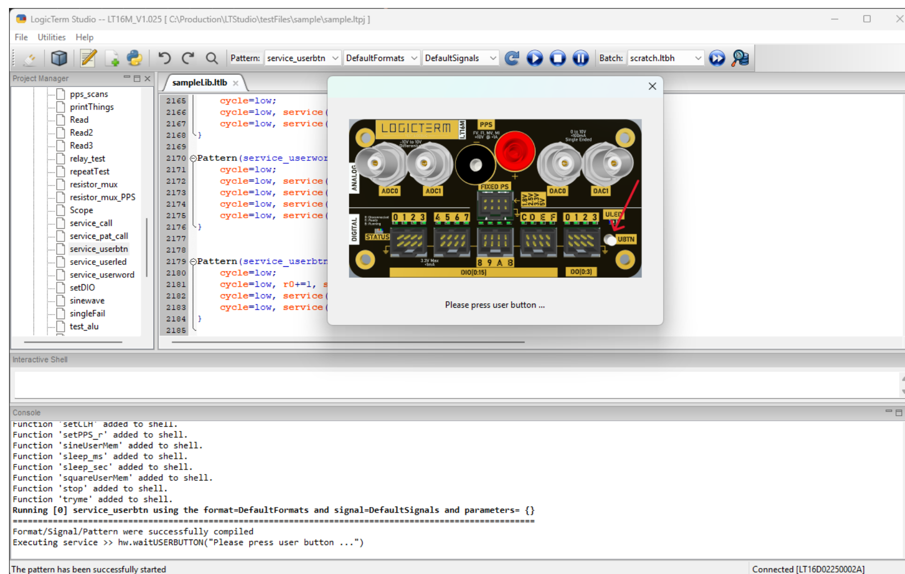


Figure 6.12: `hw.waitUSERBUTTON(msg)` creates a pop-up window.

Manipulating the database

- `hw.PrintDB`
- `hw.getDbName`
- `hw.runQuery`

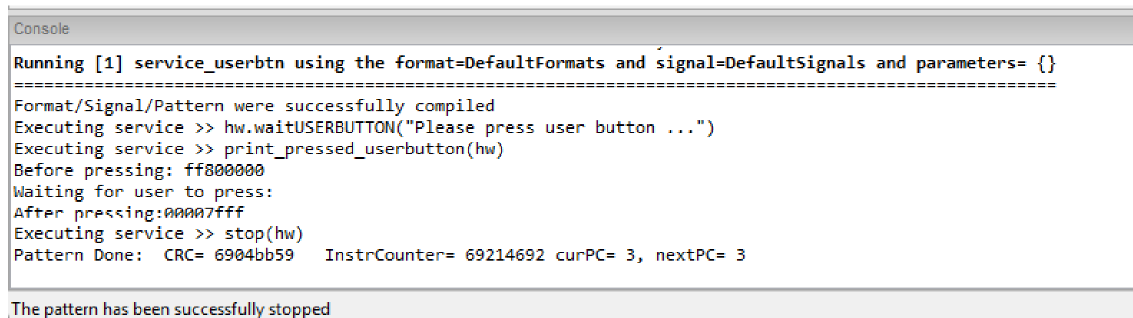


Figure 6.13: Output of `service_userbutton` pattern. Note that user button is partially pressed.

```
def print_dbase_info(hw):
    print("Current database is %s"%hw.getDbName())
    print("Full print of the database:")
    hw.PrintDB()
    print("Running a query and printing results:")
    data=hw.runQuery("select IO, Counter from IOCounters")
    for io,cntr in data:
        print(f"{io}: {cntr}")
    return hw.CONTINUE
```

Listing 6.36: Create a service for manipulating the database.

```
Pattern(service_dbase){
    cycle=low;
   # cycle=low, service(hw.createDB("heho.sqlite", 1));
    cycle=low, service(hw.PrintDB());
    cycle=low, log(FCNTRL);
    cycle=low, service(print_dbase_info(hw));
    cycle=low, service(pattern_stop(hw));
}
```

Listing 6.37: Manipulating the database

```
Pattern Output:
===============
Executing service >> print_dbase_info(hw)
Current database is :memory:
Full print of the database:
TABLE[Records] (count=9):
  ---------------------
      id     |    Type
  ---------------------
      1       |   Groups
      2       | IOCounters
...
      9       | IOCounters

TABLE[Groups] (count=1):
  --------------------------------------------
      id     |    Name    |  Level    |Recorded_at
  --------------------------------------------
      1       | [0] service_dbase |     1       | 2025-11-08 14:50:48.541187

TABLE[GroupsInfo] (count=1):
  ------------------------------------------------------
      id     | Formats   | Siganls   |  Params   |PinLabels
  ------------------------------------------------------
      1       | DefaultFormats | DefaultSignals |    | A1,B1,C1,D1,E1,F1,G1,..

TABLE[IOFails] (count=0):
  -----------------------------------------------------------------
      id     |    X      |    Y      |    Z      |   Tick    |    IO
  -----------------------------------------------------------------

TABLE[IOChange] (count=0):
  --------------------------------
      id     |InstrCntr |     IO
  --------------------------------

TABLE[IOCounters] (count=16):
  --------------------------------
      id     |     IO     | Counter
  --------------------------------
      2       |      0       |      0
      3       |      1       |      0
...
      7       |      5       |      0
```

```
TABLE[Info] (count=0):
   ---------------------------------------------------------------------------
       id     |     X      |     Y      |     Z      |  curPC    |  nextPC   |InstrCntr
   ---------------------------------------------------------------------------

TABLE[AnalogData] (count=0):
   ----------------------------------------------------------------
       id     |    Type    |   Value    |   Unit     |  curPC    |InstrCntr
   ----------------------------------------------------------------

Running a query and printing results:
0: 0
1: 0
...
7: 0

Executing service >> pattern_stop(hw)
Pattern Done:  CRC= fbac7c3a   InstrCounter= 4288893 curPC= 5, nextPC= 5
```

Setting Data Groups

- `hw.setGroupName` provides the means to create data group within a pattern.

```
1  Pattern(service_grouping){
2      @auto cycle=low, x=r0, y=r1, z=r2;
3      cycle=low, service(hw.setGroupName("FastData"));
4      cycle=low, r0+=1, log(INFO), repeat(1000);
5      cycle=low, service(hw.setGroupName("SlowData"));
6      cycle=low, r0+=1, log(INFO), for(1000);
7      cycle=low, repeat(1000);
8      cycle=low, endfor;
9      cycle=low, service(pattern_stop(hw));
10 }
```

Listing 6.38: Data Grouping

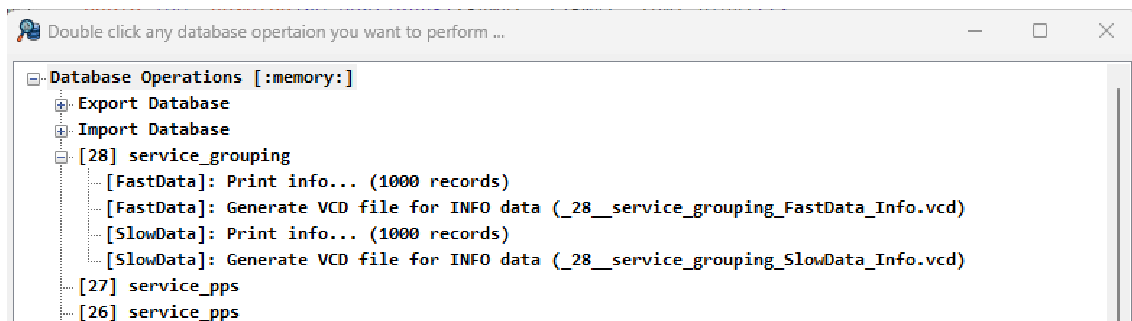


Figure 6.14: Database operation window is showing the subgroups of data within the `service_grouping` pattern.

Accessing User Memory

- `hw.setUSERMEM(address, value)` writes the 16bit value to the mem[address].
- `hw.getUSERMEM(address)` returns the 16bit value stored in mem[address].

```
1  def sineUserMem(hw):
2      from math import sin
3      for addr,val in enumerate([int((sin(i*2*3.14/1000)+1)*5000) for i in range
       (1000)]):
4          hw.setUSERMEM(addr, val)
5      #print(", ".join(["%i"%hw.getUSERMEM(i) for i in range(1000)]))
6      return hw.CONTINUE
7
8  def sawtoothUserMem(hw):
9      for addr,val in enumerate([(i%100)*100 for i in range(1000)]):
10         hw.setUSERMEM(addr, val)
```

```
11      #print(", ".join(["%i"%hw.getUSERMEM(i) for i in range(1000)]))
12      return hw.CONTINUE
13
14  def squareUserMem(hw):
15      for addr,val in enumerate([((i&64)>>6)*10000 for i in range(1000)]):
16          hw.setUSERMEM(addr, val)
17      #print(", ".join(["%i"%hw.getUSERMEM(i) for i in range(1000)]))
18      return hw.CONTINUE
19
20  def randomUserMem(hw):
21      from random import random
22      for addr,val in enumerate([int(random()*10000)  for i in range(1000)]):
23          hw.setUSERMEM(addr, val)
24      #print(", ".join(["%i"%hw.getUSERMEM(i) for i in range(1000)]))
25      return hw.CONTINUE
```

Listing 6.39: Create services for loading the USERMEM with 1000 sample of a sinewave, sawtooth, squarewave, and random data.

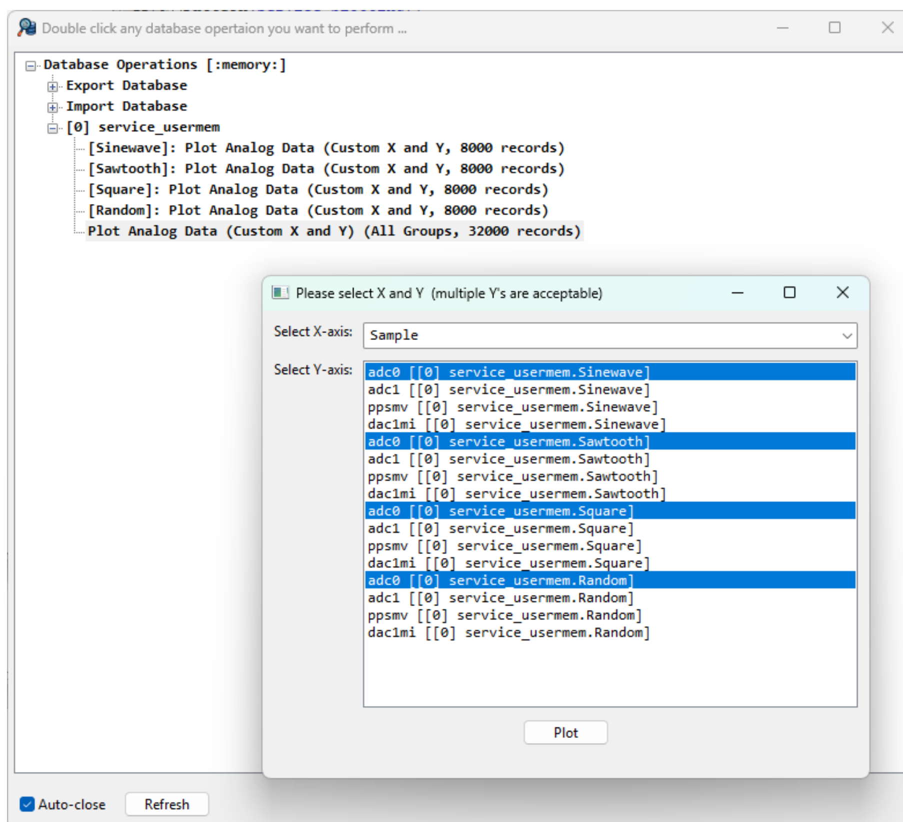


Figure 6.15: Example of how to plot data using database operations.

```
1   Pattern(service_usermem){
2       @param SAMPLES=2000;
3       @auto cycle=zero, x=r0, y=r1, z=r4;
4       cycle=zero, r3=998;
5
6       cycle=zero, service(hw.setGroupName("Sinewave"));
7       cycle=zero, service(sineUserMem(hw));
8       cycle=zero, r0=0, call(SETDAC0);
9
10      cycle=zero, service(hw.setGroupName("Sawtooth"));
11      cycle=zero, service(sawtoothUserMem(hw));
12      cycle=zero, r0=0, call(SETDAC0);
13
14      cycle=zero, service(hw.setGroupName("Square"));
```

```
15      cycle=zero, service(squareUserMem(hw));
16      cycle=zero, r0=0, call(SETDAC0);
17
18      cycle=zero, service(hw.setGroupName("Random"));
19      cycle=zero, service(randomUserMem(hw));
20      cycle=zero, r0=0, call(SETDAC0);
21
22      # NOTE: you must wait 15 cycles before you can load the next DAC value
23      cycle=zero, repeat(15);
24      cycle=zero, dac0=r0;
25
26      cycle=zero, service(pattern_stop(hw));
27
28  SETDAC0: cycle=zero, r1=mem[r0], for (SAMPLES);
29           cycle=zero, r4=r0>r3, dac0=r1,  log(ADC);
30           cycle=zero, repeat(1000);
31           cycle=zero, r0=r0|r4;
32           cycle=zero, r0+=1, endfor;
33           cycle=zero, return;
34  }
```

Listing 6.40: A pattern for driving sinewave, sawtooth, squarewave, and random data on DAC0 and read it back on the ADC0. Note that you must connect the loopback cable between DAC0 and ADC0.

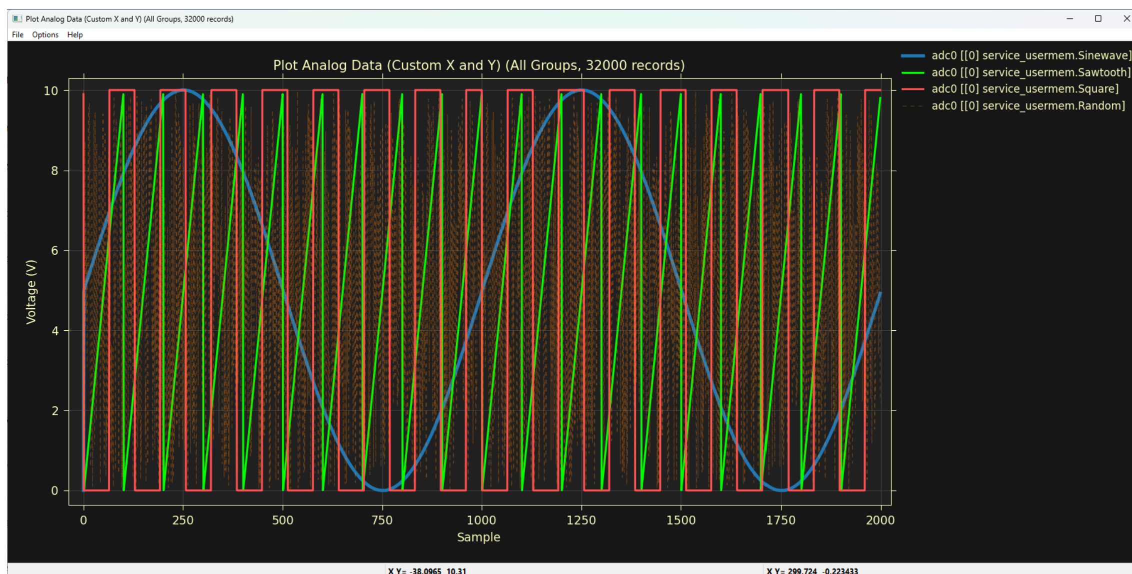


Figure 6.16: Creating a plot using the "Plot Analog Data" option inside the database operations utility in LTSudio. Note that "Sample" was used as the X axis and Y data were the ADC0 data.

Accessing Fail Limit (FLIMIT)

- `hw.setFLIMIT(value)` sets the fail limit for the fail limit exceed flag (FLE).
- `hw.getFLIMIT` returns the value of the FLIMIT.

```
1  def print_flimit(hw):
2      print("FLIMIT=%.06x"%hw.getFLIMIT()) # Note: that FLIMIT is 24bit
3      return hw.CONTINUE
```

Listing 6.41: Create a service for accessing Fail Limit (FLIMIT).

```
1  Pattern(service_flimit){
2      cycle=low;
3      cycle=low, FLIMIT=0x87654321;          # Set from the pattern
4      cycle=low, service(print_flimit(hw));
```

```
5       cycle=low,  service(hw.setFLIMIT(100)); # Set by a service
6       cycle=low,  service(print_flimit(hw));
7       cycle=low,  service(pattern_stop(hw));
8   }
```

Listing 6.42: Accessing Fail Limit (FLIMIT)

```
Console
Running [2] service_flimit using the format=DefaultFormats and signal=DefaultSignals and parameters= {}
==========================================================================================================
Format/Signal/Pattern were successfully compiled
Executing service >> print_flimit(hw)
FLIMIT=654321
Executing service >> hw.setFLIMIT(100)
Executing service >> print_flimit(hw)
FLIMIT=000064
Executing service >> pattern_stop(hw)
Pattern Done:  CRC= 552d22c8   InstrCounter= 6160709 curPC= 5, nextPC= 5

The pattern has been successfully stopped
```

Figure 6.17: Output of `service_flimit` pattern

Accessing DACs

- `hw.setDAC0(voltage)` sets the voltage for DAC0.
- `hw.setDAC1(voltage)` sets the voltage for DAC1.
- `hw.getDAC1MI` returns the current for DAC1 in mA.

```
1   def measure_current(hw, load_res):
2       cur=hw.getDAC1MI()
3       print("Current is %imA. Voltage=%.2fV"%(cur, cur*load_res/1000))
4       return hw.CONTINUE
```

Listing 6.43: Create a service for accessing DACs.

```
1   Pattern(service_dacs){
2       cycle=low;
3       cycle=low,  service(hw.setDAC0(0));
4       cycle=low,  service(hw.setDAC0(2.5));
5       cycle=low,  service(hw.setDAC0(7));
6       cycle=low,  service(hw.setDAC0(10));
7
8       cycle=low,  service(hw.setDAC1(0));
9       cycle=low,  service(measure_current(hw, 220));
10      cycle=low,  service(hw.setDAC1(1));
11      cycle=low,  service(measure_current(hw, 220));
12      cycle=low,  service(hw.setDAC1(5));
13      cycle=low,  service(measure_current(hw, 220));
14      cycle=low,  service(hw.setDAC1(10));
15      cycle=low,  service(measure_current(hw, 220));
16      # Reset back DACs to 0V
17      cycle=low,  service(hw.setDAC0(0));
18      cycle=low,  service(hw.setDAC1(0));
19      cycle=low,  service(pattern_stop(hw));
20  }
```

Listing 6.44: Accessing DACs

Accessing ADCs

- `hw.getADC0`/`hw.getADC1` returns the ADC read.

```
1   def print_adc(hw):
2       print("ADC0=%.03f  ADC1=%.03f"%(hw.getADC0(),hw.getADC1()))
3       return hw.CONTINUE
```

Listing 6.45: Create a service for accessing ADC.

```
Console
Running [12] service_dacs using the format=DefaultFormats and signal=DefaultSignals and parameters= {}
==========================================================================================================
Format/Signal/Pattern were successfully compiled
Executing service >> hw.setDAC0(0)
Executing service >> hw.setDAC0(2.5)
Executing service >> hw.setDAC0(7)
Executing service >> hw.setDAC0(10)
Executing service >> hw.setDAC1(0)
Executing service >> measure_current(hw,220)
Current is 0mA. Voltage=0.00V
Executing service >> hw.setDAC1(1)
Executing service >> measure_current(hw,220)
Current is 0mA. Voltage=0.00V
Executing service >> hw.setDAC1(5)
Executing service >> measure_current(hw,220)
Current is 0mA. Voltage=0.01V
Executing service >> hw.setDAC1(10)
Executing service >> measure_current(hw,220)
Current is 0mA. Voltage=0.01V
Executing service >> hw.setDAC0(0)
Executing service >> hw.setDAC1(0)
Executing service >> pattern_stop(hw)
Pattern Done:  CRC= 6904bb59   InstrCounter= 16821698 curPC= 15, nextPC= 15

The pattern has been successfully stopped
```

Figure 6.18: Output for `service_dacs` with no load

```
1  Pattern(service_adcs){
2      cycle=low;
3      # Loopback cables between the ADCs and the DACs
4      cycle=low, service(hw.setDAC0(0));
5      cycle=low, service(print_adc(hw));
6
7      cycle=low, service(hw.setDAC0(2.5));
8      cycle=low, service(print_adc(hw));
9
10     cycle=low, service(hw.setDAC1(5));
11     cycle=low, service(hw.setDAC0(7));
12     cycle=low, service(print_adc(hw));
13
14     cycle=low, service(hw.setDAC0(10));
15     cycle=low, service(print_adc(hw));
16
17     # Reset back DACs to 0V
18     cycle=low, service(hw.setDAC0(0));
19     cycle=low, service(hw.setDAC1(0));
20     cycle=low, service(pattern_stop(hw));
21 }
```

Listing 6.46: Accessing ADCs

Accessing PPS

`hw.setPPSV` `hw.configPPS(opmode, cur_range, volt_range`:

- opmode is "FVMV" or "FVMI". This will affect the result of `hw.getPPS`.
- cur_range is "5uA", "25uA", "250uA", "2.5mA", "25mA", "250mA", or "500mA" for maximum source or sink current of $\pm$ 5μA, $\pm$ 25μA, $\pm$ 250μA, $\pm$ 2.5mA, $\pm$ 25mA, $\pm$ 250mA, or $+500$mA$/-250$mA, respectively.
- volt_range is "BIP_6V", "UNI_P10V", or "UNI_N10V" for ranges -6V to 6V, 0 to 10V or 0 to -10V.

`hw.getPPS` returns (measurement, type). type is V or I. The unit is voltage or mA.

```
1 def print_pps(hw):
2     (meas, v_or_i)=hw.getPPS()
```

```
Console
Running [20] service_dacs using the format=DefaultFormats and signal=DefaultSignals and parameters= {}
==========================================================================================
Format/Signal/Pattern were successfully compiled
Executing service >> hw.setDAC0(0)
Executing service >> hw.setDAC0(2.5)
Executing service >> hw.setDAC0(7)
Executing service >> hw.setDAC0(10)
Executing service >> hw.setDAC1(0)
Executing service >> measure_current(hw,220)
Current is 0mA. Voltage=0.00V
Executing service >> hw.setDAC1(1)
Executing service >> measure_current(hw,220)
Current is 4mA. Voltage=0.97V
Executing service >> hw.setDAC1(5)
Executing service >> measure_current(hw,220)
Current is 22mA. Voltage=4.86V
Executing service >> hw.setDAC1(10)
Executing service >> measure_current(hw,220)
Current is 44mA. Voltage=9.72V
Executing service >> hw.setDAC0(0)
Executing service >> hw.setDAC1(0)
Executing service >> pattern_stop(hw)
Pattern Done:  CRC= 6904bb59   InstrCounter= 30036224 curPC= 15, nextPC= 15

The pattern has been successfully stopped
```

Figure 6.19: Output for `service_dacs` with 220 Ω load

```
3     unit = "V" if v_or_i=="V" else "mA"
4     print("PPS_M%s=%.03f%s"%(v_or_i, meas, unit))
5     return hw.CONTINUE
```

Listing 6.47: Create a service for accessing PPS.

```
1  Pattern(service_pps){
2      cycle=low;
3      cycle=low, service(hw.configPPS('FVMV', '25mA', 'UNI_P10V'));
4      cycle=low, service(hw.setPPSV(0.123));
5      cycle=low, service(print_pps(hw));
6      cycle=low, service(hw.setPPSV(5.555));
7      cycle=low, service(print_pps(hw));
8      cycle=low, service(hw.setPPSV(9.876));
9      cycle=low, service(print_pps(hw));
10     cycle=low, service(hw.configPPS('FVMI', '25mA', 'UNI_P10V'));
11     cycle=low, service(print_pps(hw));
12     cycle=low, service(pattern_stop(hw));
13 }
```

Listing 6.48: Accessing PPS

Plotting wrappers

- `hw.plot(data, title, win, ylabel1)`: data is a list of lists with the first list is the data headers. title is to be viewed in the top of the plot. win is the window number to be used – keep changing it to create multiple windows. ylabel1 additional Y label.
- `hw.plotall(what_li, win)` automatically plots the curves listed in what_li which can be any combination of "adc0", 'adc1", "ppsmv", "ppsmi" and "dac1mi".

```
1  def create_plot(hw):
2      data=[["X", "FastData", "SlowData"]]
3      x_data=range(1000)
4      f_data=hw.runQuery("select InstrCntr from InfoView where GroupName LIKE '%%
       FastData%%' limit 1000")
5      f_data = [row[0] for row in f_data]
6      f_data = [entry-min(f_data) for entry in f_data]
7      s_data=hw.runQuery("select InstrCntr from InfoView where GroupName LIKE '%%
       SlowData%%' limit 1000")
```

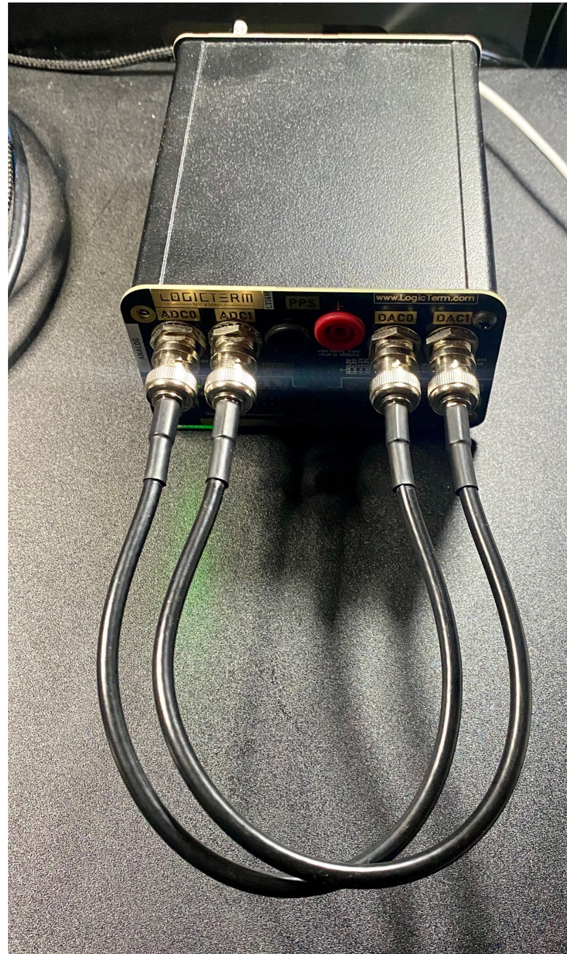


Figure 6.20: Loopback cables between the ADCs and the DACs

```
8        s_data = [row[0] for row in s_data]
9        s_data = [entry-min(s_data) for entry in s_data]
10       data.extend(zip(x_data, f_data,s_data))
11       hw.plot(data, 'Fast vs Slow', 1, 'Instruction Counter')
12       return hw.CONTINUE
```

Listing 6.49: Creating a service to plot data extracted from the database.

```
1  Pattern(service_plotting){
2      cycle=low;
3      cycle=low, service(create_plot(hw));
4      cycle=low, dac0=r0, log(ADC),  for(1000);
5      cycle=low, repeat(100);
6      cycle=low, r0+=20,             endfor;
7      cycle=low, service(hw.plotall(['adc0', 'adc1', 'dac1mi']));
8      cycle=low, service(pattern_stop(hw));
9  }
```

Listing 6.50: Accessing PPS

```
Console
Running [21] service_adcs using the format=DefaultFormats and signal=DefaultSignals and parameters= {}
==========================================================================================
Format/Signal/Pattern were successfully compiled
Executing service >> hw.setDAC0(0)
Executing service >> print_adc(hw)
ADC0=-0.002  ADC1=-0.003
Executing service >> hw.setDAC0(2.5)
Executing service >> print_adc(hw)
ADC0=2.498  ADC1=-0.004
Executing service >> hw.setDAC1(5)
Executing service >> hw.setDAC0(7)
Executing service >> print_adc(hw)
ADC0=6.999  ADC1=4.996
Executing service >> hw.setDAC0(10)
Executing service >> print_adc(hw)
ADC0=10.003  ADC1=4.998
Executing service >> hw.setDAC0(0)
Executing service >> hw.setDAC1(0)
Executing service >> pattern_stop(hw)
Pattern Done:  CRC= 6904bb59   InstrCounter= 15134140 curPC= 12, nextPC= 12

The pattern has been successfully stopped
```

Figure 6.21: Output for service_adcs with loopback cables between the ADCs and the DACs

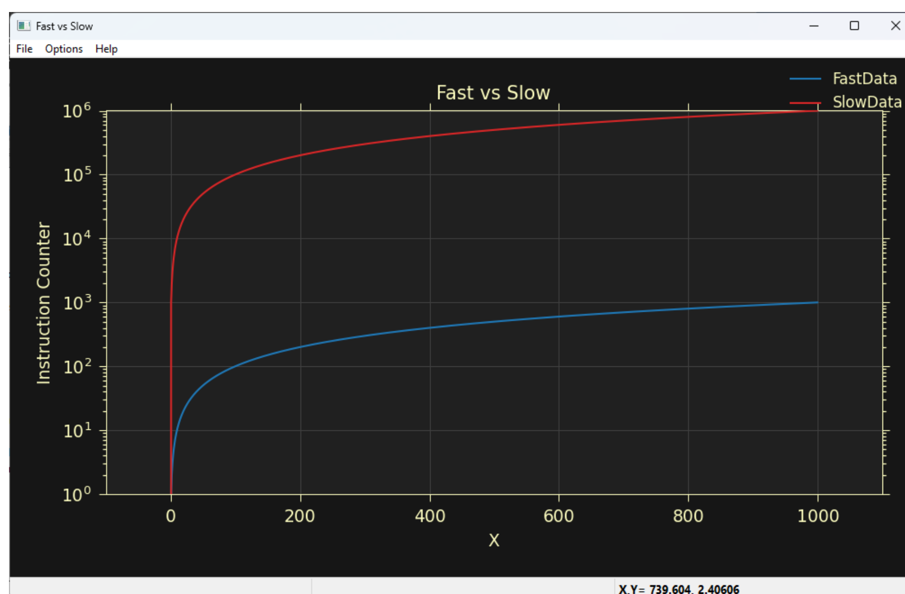


Figure 6.23: The output plot of `hw.plot`

```
Console
Running [27] service_pps using the format=DefaultFormats and signal=DefaultSignals and parameters= {}
==========================================================================================
Format/Signal/Pattern were successfully compiled
Executing service >> hw.configPPS('FVMV','25mA','UNI_P10V')
Executing service >> hw.setPPSV(0.123)
Executing service >> print_pps(hw)
PPS_MV=0.123V
Executing service >> hw.setPPSV(5.555)
Executing service >> print_pps(hw)
PPS_MV=5.555V
Executing service >> hw.setPPSV(9.876)
Executing service >> print_pps(hw)
PPS_MV=9.875V
Executing service >> hw.configPPS('FVMI','25mA','UNI_P10V')
Executing service >> print_pps(hw)
PPS_MI=0.007mA
Executing service >> pattern_stop(hw)
Pattern Done:  CRC= 6904bb59   InstrCounter= 12241585 curPC= 10, nextPC= 10

The pattern has been successfully stopped
```

Figure 6.22: Output for service_pps

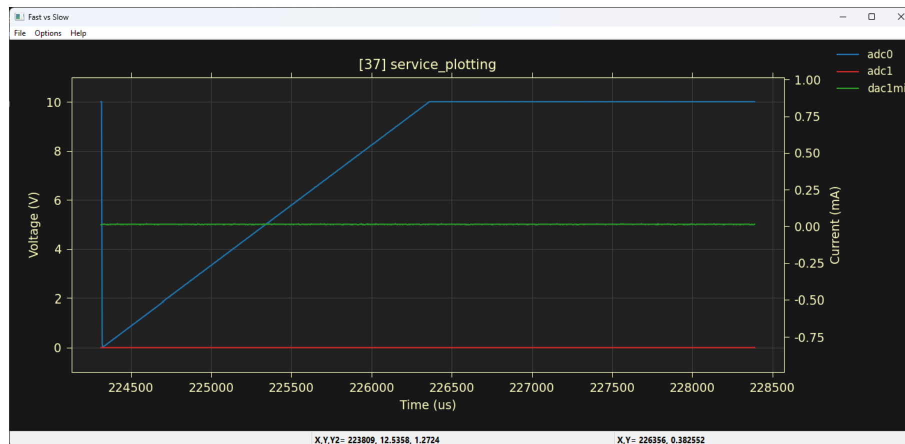


Figure 6.24: The output plot of `hw.plotall`

Other functions

- `hw.sleep` allows create time pauses using Python sleep function. Note that this is a software function and may not be precise.

6.6 Custom Services

Services are Python functions that run on the host PC and interact with the LT16M database, patterns, and hardware. They can be triggered from patterns or batch scripts to perform logging, plotting, control, and analysis.

6.6.1 Creating a Service

Each service is defined in a '.ltpy' file and must include a function with a unique name. Example:

```
1 def plot_waveform():
2     data = db.query("SELECT * FROM AnalogData WHERE Type = 'adc1'")
3     plot(data)
```

Key Features

- Access to SQLite database via `hw.runQuery()`
- Built-in plotting and export functions
- Can be called from patterns using `service(service_name)`

6.6.2 Logging and Exporting

Services can log results and export data for post-processing:

```
1 def log_and_export():
2     db.save("session.db")
3     db.export("session.xlsx")
```

6.6.3 Control and Automation

Services can control test flow and external instruments:

```
1 def control_supply():
2     set_voltage("PPS", 3.3)
3     wait(0.5)
4     measure_current("PPS")
```

6.6.4 Service Invocation from Pattern

Patterns can trigger services using the ‘CALL‘ instruction:

```
1 PATTERN RunTest
2   FORMAT F0 { DIO[0:3] = 0b1100; }
3   F0; LOG;
4   CALL plot_waveform;
```

6.6.5 Best Practices

- Keep services modular and reusable
- Validate inputs and handle exceptions
- Use consistent naming and documentation

6.7 Batch Scripts

Batch scripts are Python scripts that automate the execution of multiple patterns and services. They are ideal for production testing, regression runs, and complex workflows.

- Load and execute multiple objects
- Control timing and sequencing
- Perform post-processing and export

6.7.1 Wrappers for Batch Scripts

- `hw.pat_run(pattern_name, formats, signals, **params)` starts the <pattern_name> pattern using <formats> and <signals> with the provided parameters. This call is a blocking call and the batch calling it will be stalled until the pattern is stopped.
- `hw.pat_reset` resets a ATPG/LT16 hardware.

```
1 Pattern(custom_clock){
2     @param PERIOD_LOW=10, PERIOD_HIGH=10;
3     cycle=zero, r0=0, service(echo(hw, "Push user button to stop the pattern.") );
4
5 CYCLE:
6     cycle=ddta,  repeat(PERIOD_LOW);
7     cycle=idta,  repeat(PERIOD_HIGH-1);
8     cycle=idta,  jmp(NUF, CYCLE); # Press button to stop pattern
```

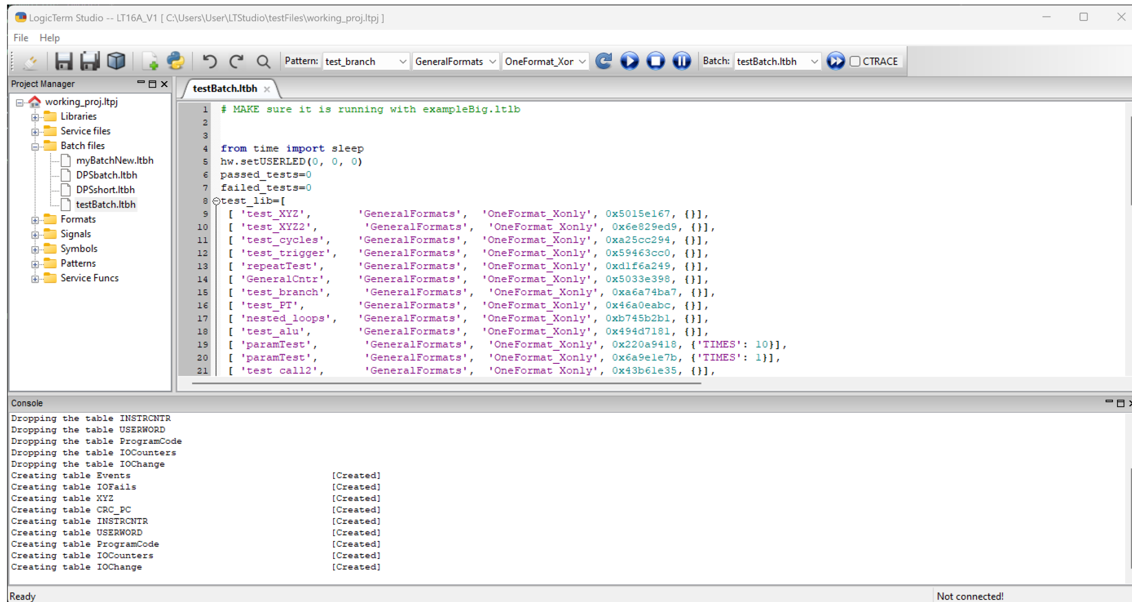


Figure 6.25: LTStudio Batch

```
9      cycle=low, service(pattern_stop(hw));
10 }
```

Listing 6.51: Creating a pattern with parameters

```
1  print("Calling custom_clock with default parameters (10, 10)") # Seeing a freq=625
       KHz Period=1.6us
2  hw.pat_run( "custom_clock", "DefaultFormats", "DefaultSignals")
3
4  print("Calling custom_clock with different parameters")
5  params={'PERIOD_LOW':1000, 'PERIOD_HIGH':500}
6  hw.pat_run( "custom_clock", "DefaultFormats", "DefaultSignals", **params) # 8.33K
       Period=120us
7
8  print("Calling custom_clock with different parameters")
9  print("Calling custom_clock with different parameters")
10 params={'PERIOD_LOW':60000, 'PERIOD_HIGH':60000}
11 hw.pat_run( "custom_clock", "DefaultFormats", "DefaultSignals", **params) # 104.17
       Hz 9.6ms
12
13 # Blinking the USER RGB LED
14 hw.setUSERLED(0xff,0,0)
15 hw.sleep(1)
16
17 hw.setUSERLED(0,0xff,0)
18 hw.sleep(1)
19
20 hw.setUSERLED(0,0,0xff)
21 hw.sleep(1)
22
23 print("pat_reset resets the hardware")
24 hw.pat_reset()
```

Listing 6.52: A batch file that starts/reset patterns

6.7.2 Batch examples

```
1 #######  Drawing NPN Characteristic Curve ####
2 #   +     -------------------------------
3 #   |\
```

```
#  | \
#  |  \    -- Ic-->       VCE
#  |PPS\___/\/\/\/\________o____
#  |   /                        +-- ADC0
#  |  /                         |
#  | /     +                    |
#  |/      |\                   + C
#  +       | \                 /
#          |  \   --Ib-->    |/
#          DAC0\__/\/\/\/\_B_|
#          |   /  Rb=10K     |
#          |  /              |\
#          | /                 \
#          |/                   + E
#          +                    |
#                               |
#                             ======
#                              ====
#                               ==
#################################################
setVBE=hw.setDAC0 # 0 -->0A, 1V -->100uA, 2V-->200uA
setVCE=hw.setPPSV
getVCE=hw.getADC0
def getIC(): return hw.getPPS()[0]

PPS_SAMPLES=1
VCE_SAMPLES=100
VBE_SAMPLES=10
VBE_MAX=10
# X Axes and Curve Names ["VCE", "IB=20uA", "IB=40uA", ..., "IB=200uA" ]
data=[["VCE"]+["IB=%iuA"%(vb/0.01/5) for vb in range(VBE_SAMPLES+1)]]
hw.configPPS(opmode='FVMI', cur_range='500mA', volt_range='UNI_P10V')

for vce in range(VCE_SAMPLES+1):
    print("Setting VCE to", vce/VCE_SAMPLES*10)
    setVCE(vce/VCE_SAMPLES*10)
    hw.sleep(0.01)
    record=[getVCE()]
    # Using VBE 0 to 2V for 0uA to 200uA (Rb=10K)
    for vbe in range(VBE_SAMPLES+1):
        setVBE(vbe/5)
        record.append(sum([getIC() for i in range(PPS_SAMPLES) ])/PPS_SAMPLES)
    data.append(record)

hw.plot(data,  title='VCE vs IC NPN Transistor', win=1, ylabel1='IC (mA)')

# Put it back to safe voltages
hw.setPPSV(0)
setVCE(0)
setVBE(0)
```

Listing 6.53: A batch file

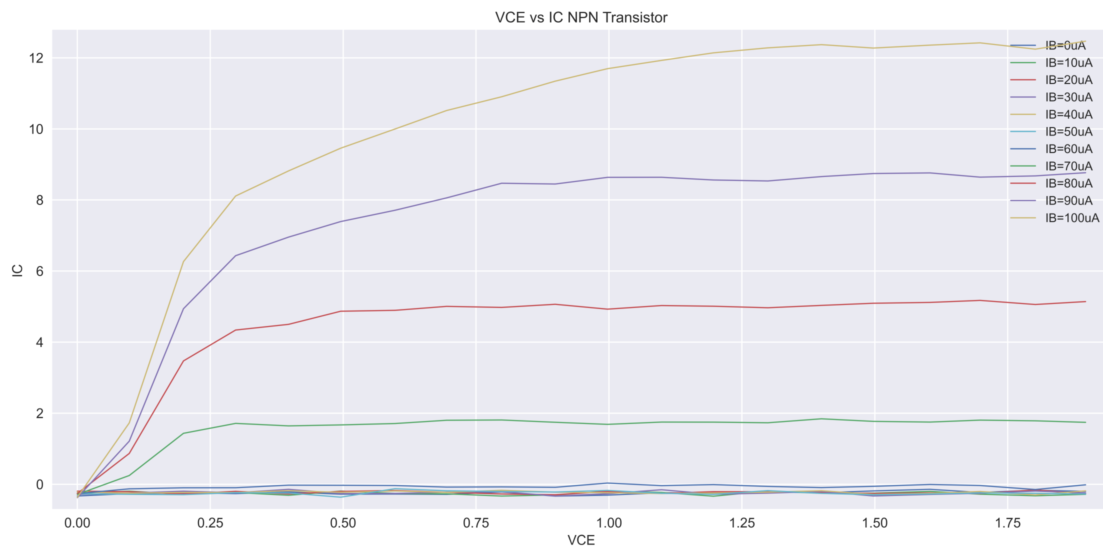


Figure 6.26: LTStudio Batch for drawing the NPN transistor characterization curve.

III

7 Database Architecture 61
7.1 Overview
7.2 Database Structure
7.3 View Tables
7.4 Exporting Data
7.5 Service and Batch Integration
7.6 Database Tables

7. Database Architecture

7.1 Overview

LT16M uses an in-memory SQLite database to log digital and analog data during pattern execution. This architecture enables fast access, structured grouping, and seamless export to external formats.

7.2 Database Structure

The database is composed of multiple tables that capture different aspects of execution:

- **Records** – Raw digital and analog samples
- **Groups** – Logical grouping of records by test phase or pattern
- **GroupsInfo** – Metadata for each group (name, timestamp, pattern ID)
- **Info** – Instruction info
- **IOChange** – Changes to pins
- **AnalogData** – Captured ADC samples with timestamps
- **IOCounters** – Per-pin counters for toggles, transitions, and activity
- **IOFails** – Pin-level failure logs

7.3 View Tables

LTStudio provides simplified views for quick access to grouped data, as seen in figure 7.2.

7.4 Exporting Data

Users can export logged data from LTStudio in multiple formats:

- **Excel (.xlsx)** – Tabular export for analysis and reporting
- **CSV (.csv)** – Lightweight format for scripting and automation
- **SQLite (.db)** – Full database export for advanced querying

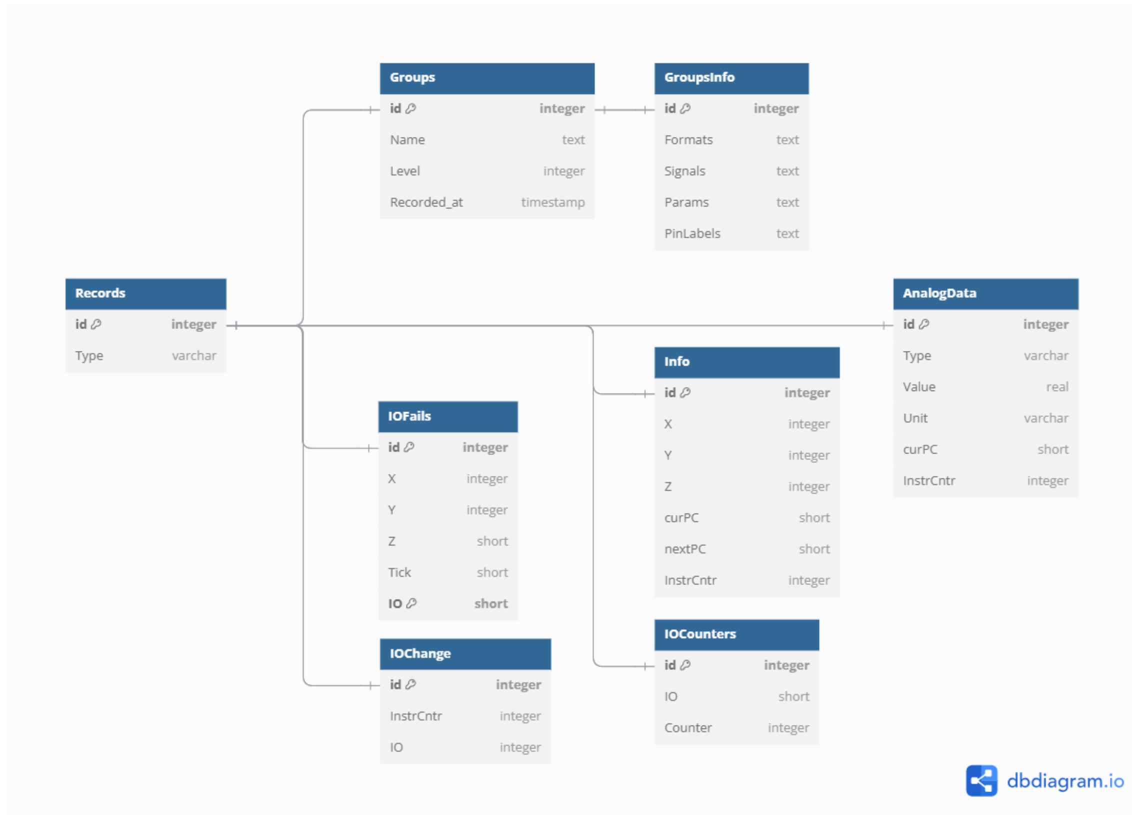


Figure 7.1: LT16M database structure

7.5 Service and Batch Integration

Python services and batch scripts can query the database during or after pattern execution to:

- Extract analog waveforms
- Count digital transitions
- Detect failures and anomalies
- Generate plots and reports

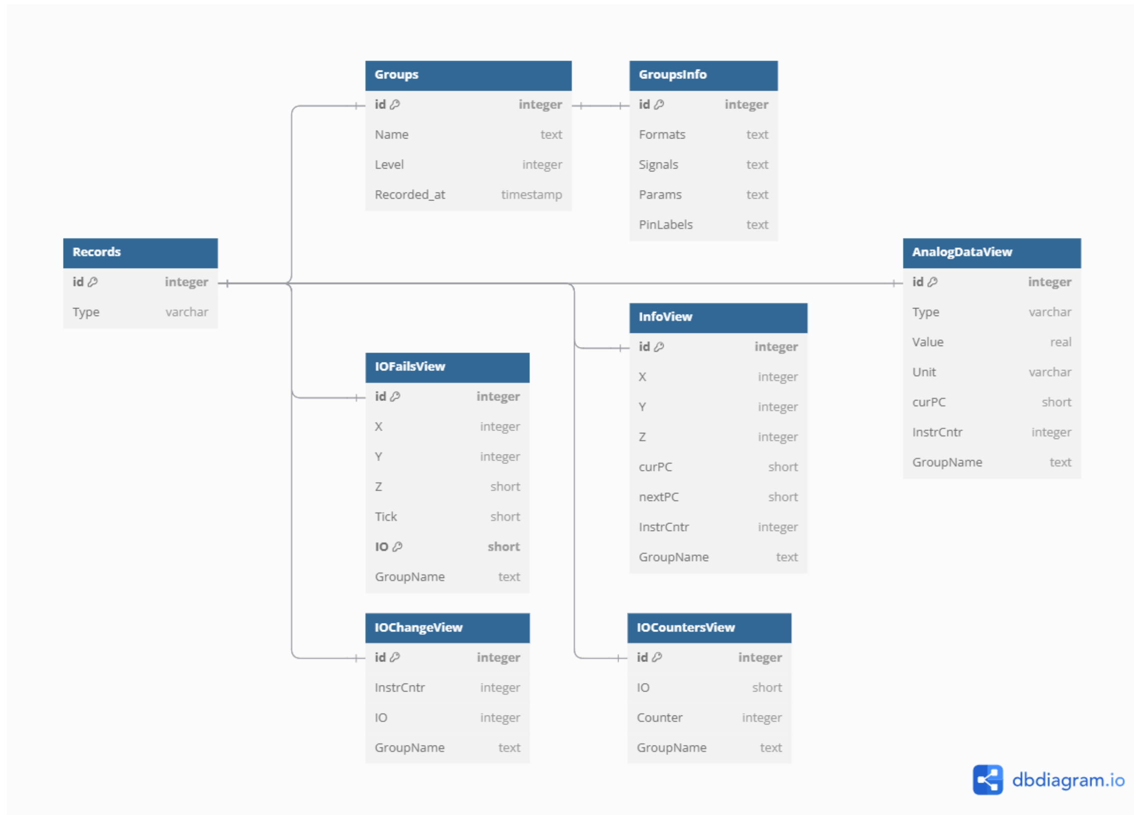


Figure 7.2: LT16M Database View Structure. Note the additional field for GroupName.

7.6 Database Tables

Now, here are the full details for each of the tables and their columns.

7.6.1 Records Table

Every reported record will have a unique ascending id and a type. This id will *connect* to other tables' data.

Table	Field	Description
Records	id	A unique ID for each record in the database
	Type	The type of record. Valid types are IOFails, Info, IOCounters, AnalogData, IOChange, and Groups.

Table 7.1: "Records" table. id is the primary key.

Please refer to subsection 12.5.1 for sample data.

7.6.2 Groups Table

When a pattern is launched, a new "level 1" record is created with the test name and test number.

Please refer to subsection 12.5.2 for sample data.

7.6.3 GroupsInfo Table

A record in GroupsInfo captures the run selections of test pattern including formats, signals, params and pin labels.

Please refer to subsection 12.5.3 for sample data.

Table	Field	Description
Groups	id	A unique ID for each group in the table
	Name	Group name
	Level	Group level. Level 1 is for patterns when they start, where level 2 is for sub-groups within a pattern. Sub-groups are defined by users for data grouping.
	Recorded_at	A time stamp for creating this group.

Table 7.2: "Groups" table. id is the primary key.

Table	Field	Description
GroupsInfo	id	A unique ID for a pattern in groups table.
	Formats	The selected format for the current pattern.
	Signals	The selected signals for the current pattern.
	Params	The parameters for the current pattern.
	PinLabels	A comma-seperated list of all 16 DIOs labels.

Table 7.3: "GroupsInfo" table. id is the primary key.

7.6.4 IOFailsView Table

A record in this table indicates that a *read* instruction failed, meaning the compare operation did not pass.

Table 7.4: "IOFailsView" table. (id,IO) is the primary key.

Table	Field	Description
IOFailsView	id	The ID for the current record.
	X	16 bit X for the current instruction.
	Y	16 bit Y for the current instruction.
	Z	The least significant 8 bits of Y for the current instruction.
	Tick	Bit vector for the failing tick.
	IO	IO number where fail is observed.
	GroupName	Group name which consists of test name (and data group name if exists).

Please refer to subsection 12.5.7 for sample data.

7.6.5 IOChangeView Table

A record in this table indicates that a read instruction detected a *change* in the IO state from the previous read.

Table	Field	Description
IOChangeView	id	The ID for the current record.
	InstrCntr	44 bit of instruction counter since the start of the pattern.
	IO	IO mask where *change* is observed.
	GroupName	Group name which consists of test name (and data group name if exists).

Table 7.5: "IOChangeView" table. id is the primary key.

Please refer to subsection 12.5.8 for sample data.

7.6.6 IOCountersView Table

A record reports the total number of fails for an IO.

Table	Field	Description
IOCountersView	id	The ID for the current record.
	IO	IO number
	Counter	24 bit fail counter
	GroupName	Group name which consists of test name (and data group name if exists).

Table 7.6: "IOCountersView" table. id is the primary key.

Please refer to subsection 12.5.6 for sample data.

7.6.7 InfoView Table

A record reports the state of X, Y, Z, ... etc. for an instruction.

Table	Field	Description
InfoView	id	The unique ID for the current record.
	X	16 bit X for the current instruction.
	Y	16 bit Y for the current instruction.
	Z	16 bit Z for the current instruction.
	curPC	Current Program Counter (PC).
	nextPC	Next Program Counter (PC).
	InstrCntr	44 bit instruction counter
	GroupName	Group name which consists of test name (and data group name if exists).

Table 7.7: "InfoView" table. id is the primary key.

Please refer to subsection 12.5.4 for sample data.

7.6.8 AnalogDataView Table

A record reports the analog value for a certain measurement.

Table	Field	Description
AnalogDataView	id	The unique ID for the current record.
	Type	Analog value type. Valid types are adc0, adc1, ppsmi, ppsmv, and dac1mi.
	Value	Analog value (floating point format).
	Unit	Measurement unit. Valid options are V or mA.
	curPC	Current Program Counter (PC).
	InstrCntr	44 bit instruction counter
	GroupName	Group name which consists of test name (and data group name if exists).

Table 7.8: "AnalogDataView" table. id is the primary key.

Please refer to subsection 12.5.5 for sample data.

IV

Practical Examples

8 Quick Start Examples 67
8.1 LED Blinking
8.2 Seven Segment Display

9 Debugging Examples 71
9.1 Detecting Stuck-at Faults
9.2 Open Circuit Detection
9.3 Short Detection Between Pins
9.4 Using IOCounters
9.5 Channel snoop
9.6 Oscilloscope
9.7 Logic Analyzer

10 Production Examples 75
10.1 Automated Test Sequence
10.2 Reusable Pattern with Parameters
10.3 Logging and Exporting Results
10.4 Pass/Fail Summary
10.5 PWM
10.6 SPI Master Controller
10.7 SPI Slave Device
10.8 I^2C Master Controller
10.9 Duty Cycle measurement
10.10 Frequency counter
10.11 Analog waveform generator
10.12 Melody generator
10.13 NPN Transistor Characterization
10.14 MOSFET Transistor Characterization
10.15 Flash programmer
10.16 EEPROM reader

8. Quick Start Examples

8.1 LED Blinking

This example demonstrates how to blink a single LED connected to a DIO pin. We will try using different ways to achieve that in order to explore the tool's capability and flexibility.

Setup

- Connect an LED (with series resistor) to `DIO[0]` or rely on the DIO integrated LED
- Set jumper to 3.3V logic level

8.1.1 LED Blinking by changing the cycle

In this example, we define a single format with two cycles. One cycle, called `led_off`, is configured as an **o**utput with all four ticks set to **L**ow. On the other hand, the `led_high` cycle is set to **H**igh.

When an LED is connected to the pin `DIO[0]` of the LT16M, the Signals **mySig** define what data source should be used for this pin and what format.

When the pattern myTest starts executing, the first instruction uses the cycle `led_off` which sets the `DIO[0]` to 0 for all 4 ticks (40 ns). The second instruction uses the cycle `led_on` which sets the connected pin 0 to 1 for all 4 ticks (40ns). In addition, it jumps to the instruction at label `AGAIN`. In summary, this will cause the pattern to toggle infinitely. Unfortunately, the toggle at 12.5MHz is too fast for humans to see.

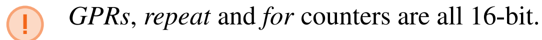
```
1  Formats(myFrmt){
2          # CYCLE         0       1
3      # FORMAT
4        cycle_sel  = [ led_off , led_on];
5        LED_FRMT   = [  oLLLL ,   oHHHH ];
6      }
7
8  Signals(mySig){
9      A1    = dio(pin=0, map=0, format=LED_FRMT);
10     }
11
12 Pattern(led_blinking){
13     AGAIN: cycle=led_off;                 # 1 Cycle ON  takes 40ns
```

```
        cycle=led_on, jmp(AGAIN);    # 1 Cycle OFF takes 40ns
    }
```

Listing 8.1: Toggling an LED forever, 12.5MHz!

8.1.2 Slowdown LED Blinking using Repeats

We will use repeat micro-instruction to slowdown LED blinking.



GPRs, *repeat* and *for* counters are all 16-bit.

```
Pattern(led_blinking_repeat){
    AGAIN: cycle=led_off, repeat(0xffff); # 0xffff*40e-9 --> 2.6214ms
           cycle=led_off, repeat(0xffff); # 0xffff*40e-9 --> 2.6214ms
           cycle=led_off, repeat(0xffff); # 0xffff*40e-9 --> 2.6214ms
           cycle=led_off, repeat(0xffff); # 0xffff*40e-9 --> 2.6214ms

           cycle=led_on, repeat(0xffff); # 0xffff*40e-9 --> 2.6214ms
           cycle=led_on, repeat(0xffff); # 0xffff*40e-9 --> 2.6214ms
           cycle=led_on, repeat(0xffff); # 0xffff*40e-9 --> 2.6214ms
           cycle=led_on, repeat(0xfffe); # 0xfffe*40e-9 --> 2.62136ms
           cycle=led_on, jmp(AGAIN);
    }
```

Listing 8.2: Toggling an LED forever, barely visible!

8.1.3 Slowdown LED Blinking using Loops

Using a for loop with nested repeats to achieve a total of 500ms of toggle time.

```
Pattern(led_blinking_loop){
    AGAIN: cycle=led_off, for(2500);        # (4998+1+1)*2500*40e-9 --> 500.000ms
              cycle=led_off, repeat(4998);
           cycle=led_off, endfor;

           cycle=led_on, for(2500);         # (4998+1+1)*2500*40e-9 --> 500.000ms
              cycle=led_on, repeat(4998);
           cycle=led_on, endfor;

           cycle=led_on, jmp(AGAIN);
    }
```

Listing 8.3: Toggling an LED forever, normal toggle

8.1.4 Slowdown LED Blinking by mapping to `x[0]`

Toggling the LED using data mapped from one of the GPRs in the pattern generator.

```
Formats(myFrmt){
        # CYCLE        0
    # FORMAT
      cycle_sel = [ led_ctrl];
      LED_FRMT  = [  oDDDD  ]; # Source data is from pattern
    }

Signals(mySig){
    LED    = dio(pin=0, map=x[0], format=LED_FRMT);  # LED -mapped-> bit 0 of x
    }

Pattern(led_blinking_x0){
          cycle=led_ctrl, r0=0, x=r0;        # Initializing r0 to 0, LED is off
                                             # x is pointing at r0
    AGAIN: cycle=led_ctrl,        x=r0; for(2500);
           cycle=led_ctrl,        x=r0, repeat(4998);
           cycle=led_ctrl,        x=r0, endfor;
```

```
        cycle=led_ctrl, r0+=1, x=r0, jmp(AGAIN); # Toggle LED by adding 1 to r0!
    }
```

Listing 8.4: Toggling an LED forever using `x[0]`

8.1.5 Slowdown LED Blinking by mapping to `x[15]`

Toggling an LED connected to pin 0 using bit 15 as the source data.

Pattern generator is reset before running any test, so there is no need to initialize `r0` to 0.

```
Formats(myFrmt){
        # CYCLE         0
    # FORMAT
      cycle_sel  = [ led_ctrl];
      LED_FRMT   = [   oDDDD   ];
    }

Signals(mySig){
    LED  = dio(pin=0, map=x[15], format=LED_FRMT);   # LED -mapped-> bit 15 of x
    }

Pattern(led_blinking_x15){
    AGAIN: cycle=led_ctrl, x=r0, repeat (400); # (400e-9)*(40)*2**15 = 0.524288
           cycle=led_ctrl, r0+=1, x=r0, jmp(AGAIN);
    }
```

Listing 8.5: Toggling an LED forever using `x[15]`

8.1.6 Slowdown LED Blinking by using Keep and Toggle ticks

Toggling an LED connected to pin 0.

```
Formats(myFrmt){
        # CYCLE         0
    # FORMAT
      cycle_sel  = [ led_togg, led_keep];
      LED_FRMT   = [   oTTTT,    oKKKK    ]; # oTTTT: toggle the state of the pin
    }

Signals(mySig){
    LED  = dio(pin=0, map=0, format=LED_FRMT);
    }

Pattern(myTest){
    AGAIN:
        cycle=led_keep, for(2500);   # 2500*(4998+1+1)*40e-9)=500ms
        cycle=led_keep, repeat (4998);
        cycle=led_keep, endfor;
        cycle=led_togg, jmp(AGAIN);
    }
```

Listing 8.6: Toggling an LED forever using Keep and Toggle ticks

8.2 Seven Segment Display

This example cycles through digits 0–9 on a common-cathode 7-segment display. It will start with 0 blinking at 2 Hz rate. If user press and hold the UBTN, the next number will blink.

Setup

- Connect segments A–G to DIO0–DIO6
- Use current-limiting resistors

Pattern Code

Use a lookup table or define formats for each digit. Example for digit "0":

```
Formats(myFrmt){
        # CYCLE         0
    # FORMAT
      cycle_sel  = [ led_set ,  led_off];
      LED_FRMT   = [  oDDDD  ,  oLLLL   ];
    }

Signals(mySig){
    A    = dio(pin=0, map=x[0], format=LED_FRMT);
    B    = dio(pin=1, map=x[1], format=LED_FRMT);
    C    = dio(pin=2, map=x[2], format=LED_FRMT);
    D    = dio(pin=3, map=x[3], format=LED_FRMT);
    E    = dio(pin=4, map=x[4], format=LED_FRMT);
    F    = dio(pin=5, map=x[5], format=LED_FRMT);
    G    = dio(pin=6, map=x[6], format=LED_FRMT);
    }

Pattern(myTest){
    @auto cycle=led_set, x=r0;
#
# GFEDCBA             # +---- A ------+
# 1111011 --> 0       # |             |
# 1001000 --> 1       # B             D
# 0111101 --> 2       # |             |
# 1101101 --> 3       # +---- C ------+
# 1001110 --> 4       # |             |
# 1100111 --> 5       # E             G
# 1110111 --> 6       # |             |
# 1001001 --> 7       # +---- F ------+
# 1111111 --> 8
# 1101111 --> 9
    _0_:    cycle=led_set, r0=0b1111011, call(DRAW_NUM); # Number 0
    _1_:    cycle=led_set, r0=0b1001000, call(DRAW_NUM); # Number 1
    _2_:    cycle=led_set, r0=0b0111101, call(DRAW_NUM); # Number 2
    _3_:    cycle=led_set, r0=0b1101101, call(DRAW_NUM); # Number 3
    _4_:    cycle=led_set, r0=0b1001110, call(DRAW_NUM); # Number 4
    _5_:    cycle=led_set, r0=0b1100111, call(DRAW_NUM); # Number 5
    _6_:    cycle=led_set, r0=0b1110111, call(DRAW_NUM); # Number 6
    _7_:    cycle=led_set, r0=0b1001001, call(DRAW_NUM); # Number 7
    _8_:    cycle=led_set, r0=0b1111111, call(DRAW_NUM); # Number 8
    _9_:    cycle=led_set, r0=0b1101111, call(DRAW_NUM); # Number 9

    cycle=led_off, service(pattern_stop(hw));

DRAW_NUM:
    cycle = led_set , for (2500) ; # (4998+1+1) *2500*40e -9 --> 500.000 ms
    cycle = led_set , repeat (4998) ;
    cycle = led_set , endfor;
    # Blink for 0.5s
    cycle = led_off , for (2500) ; # (4998+1+1) *2500*40e -9 --> 500.000 ms
    cycle = led_off , repeat (4998) ;
    cycle = led_off , endfor;
    # Proceed to next number when user button is pressed
    cycle = led_off, jmp(NUF, DRAW_NUM);
    cycle = led_off, return;
}
```

Listing 8.7: Count up seven segment LED

9. Debugging Examples

9.1 Detecting Stuck-at Faults

This example checks whether a pin is stuck at logic high or low. We use the programmable power supply (PPS) in a FVMV mode with low current mode.

Setup

- Connect DIO2 to the node under test
- Ensure the node is driven externally

Pattern

```
1  CUR_MODE = '25uA'
2  VOL = 1
3  THRESHOLD = VOL/2
4  hw.configPPS('FVMV', CUR_MODE, 'BIP_6V')
5
6  print("Press UBTN to stop")
7
8  while(not hw.getUSERBUTTON()):
9      # Force high and check if it is high
10     hw.setPPSV(VOL)
11     hw.sleep(0.01)
12     meas_vol_h = hw.getPPS()[0]
13     # Force low and check if it is low
14     hw.setPPSV(0)
15     hw.sleep(0.01)
16     meas_vol_l = hw.getPPS()[0]
17
18     diff = abs(meas_vol_h-meas_vol_l)
19     PASS = 1 if diff > THRESHOLD else 0
20     if PASS: hw.setUSERLED(0,int(diff*255),0)
21     else:    hw.setUSERLED(int((1-diff)*255),0,0)
22
23 hw.setUSERLED(0,0,0)  # Turn ULED off
```

Expected Behavior

If DIO2 reads the same value repeatedly, the pattern jumps to 'Fault' and calls a Python service to report the issue.

9.2 Open Circuit Detection

This example verifies whether a pin is floating or disconnected.

Pattern

```
Pattern(open_test){
# TODO:
cycle=low;
cycle=low;
cycle=low;
cycle=low, service(stop_pattern(hw));
}
```

Expected Behavior

If DIO3 fails to follow the driven value, the pattern logs an open fault.

9.3 Short Detection Between Pins

This example checks for shorts between DIO4 and DIO5.

Pattern

```
Pattern(short_test){
# TODO:
cycle=low;
cycle=low;
cycle=low;
cycle=low, service(stop_pattern(hw));
}
```

Expected Behavior

If DIO5 reads high while driven low, a short is detected and logged.

9.4 Using IOCounters

Python services can access per-pin counters to detect excessive toggling or inactivity. Check the example in section 10.9.

9.5 Channel snoop

Snooping a channel

9.6 Oscilloscope

enabling oscilloscope-like features

```
def stop_pattern(hw):
    return hw.STOP
```

```
1  Formats(miniFrmts){
2      cycle_sel   = [ low ];
3      OUT_LOW     = [oLLLL];
4  }
5
6  Signals(miniSigs){
7      DIO_0    = dio(pin=0,    map=0,    format=OUT_LOW );
8  }
9
10 Pattern(ADC_test){
11     @param SAMPLES=2048, PERIOD=250 # 40ns * 250 = 10usec --> 100KS/s
12     cycle=low, log(ADC), for(SAMPLES);
13     cycle=low, repeat(PERIOD-2);
14     cycle=low, endfor;
15     cycle=low, service(hw.plotall(['adc0', 'adc1']));
16     cycle=low, service(stop_pattern(hw));
17 }
```

Listing 9.1: Capturing a signal using ADC0 and ADC1

9.7 Logic Analyzer

Here is a pattern for reading the UART signal without knowing the baud rate. The UART RX signal is connected to DIO[5]. The pattern is set to log 4000 IO changes using the fail limit flag. On pattern completion, the user can query the data using the database operation window, figure 9.1. User can view the captured waveform using GTKwave, figure 9.2. For decoding the captured signal using UART protocol, the user can view the generated VCD file using PulseView (figure 9.3) or any other tool.

```
1  Formats(uart_frmt){
2        # CYCLE      0       1
3    # FORMAT
4      cycle_sel = [idle ,   rd   ];
5      RX_FRMT   = [iHHHH, iZMZZ];
6  }
7
8  Signals(uart_sigs){
9      RX = dio(pin=5,   map=x[0],    format=RX_FRMT );
10 }
11
12 Pattern(uart_read){
13     @auto cycle=idle, x=r0;
14     cycle=idle, r0=0;
15     cycle=idle, FLIMIT=4000;
16
17 MORE:
18     cycle=rd, iomask=0x20, log(CHANGE);
19     cycle=idle, call(NFLE, MORE);
20
21     cycle=idle, x=r0, y=r0, z=r0, r0=0xdead, service(pattern_stop(hw));
22 }
```

Listing 9.2: UART Signal

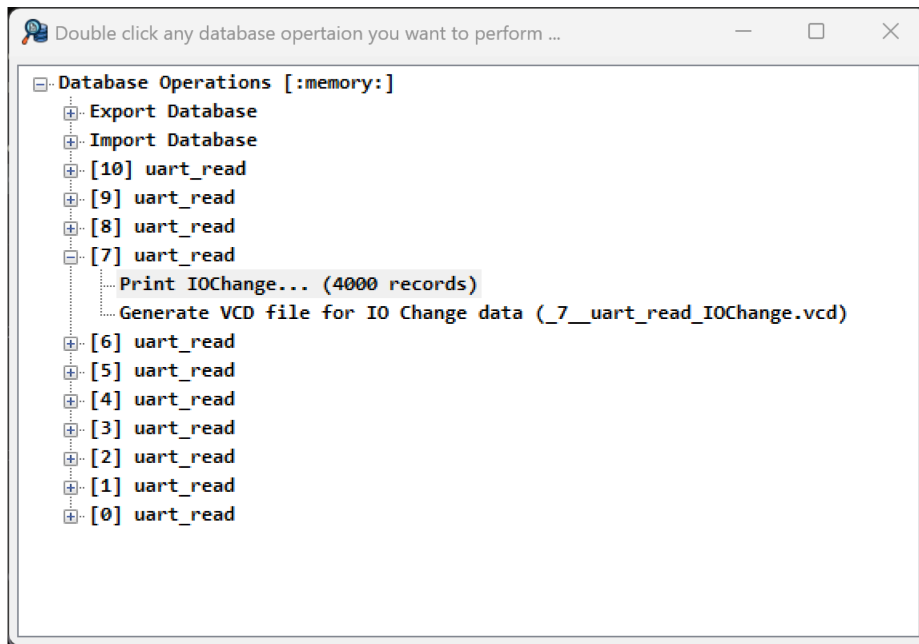


Figure 9.1: Database operations

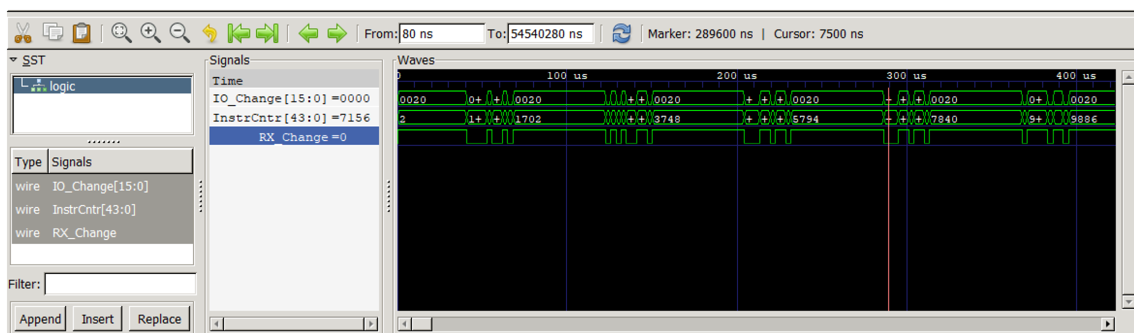


Figure 9.2: Database operations

Figure 9.3: Database operations

10. Production Examples

10.1 Automated Test Sequence

This example runs a full test cycle using a batch script that loads patterns, logs results, and calls services.

Batch Script

```
1 # Placeholder
```

Expected Behavior

Executes a power-on test, followed by a functional test, then logs and exports results.

10.2 Reusable Pattern with Parameters

This example uses a parameterized pattern to test multiple devices with different configurations.

Pattern

Batch Invocation

```
1 # Placeholder
```

10.3 Logging and Exporting Results

This example shows how to log data and export it to Excel for reporting.

Service Snippet

```
1 # Placeholder
```

10.4 Pass/Fail Summary

This example generates a summary of test outcomes across multiple units.

Service Snippet

```
# Placeholder
```

10.5 PWM

```
# Define Formats
Formats(LOW_HIGH){
    cycle_sel = [drv_l, drv_h];
    DRV_FRMT  = [oLLLL, oHHHH];
}

# Define Signals
Signals(MySigs){
    DIO_0 = dio(pin=0, map=0, format=DRV_FRMT);
}

# Define the Pattern
# Create a pattern to generate PWM
Pattern(PWM){
    @param DUTY = 70, CYCLES=10;
    cycle=drv_l, for(CYCLES)
# We need to drive high for DUTY cycle and low for 100-DUTY
RELOAD:
    cycle=drv_h, r0 = DUTY-1;
STAY_HERE:
    cycle=drv_h, r0-=1, jmp(NZ1, STAY_HERE);
    cycle=drv_l, repeat((100-DUTY)-1);
    cycle=drv_l, endfor; # loop to the next cycle

    # Keep looping until user press UBTN
    cycle=drv_l, jmp(NUF, RELOAD);
    cycle=drv_l, service(pattern_stop(hw));
}
```

Listing 10.1: PWM

10.6 SPI Master Controller

10.7 SPI Slave Device

10.8 I^{2}C Master Controller

10.9 Duty Cycle measurement

```
def printDutyCycle(hw):
    print("Duty Cycle: %%%.01f"%(hw.runQuery("select Counter from IOCounters where
    IO=0")[-1][0]*100/0xffff))
    return hw.CONTINUE
```

```
Formats(MyFormats){
      # CYCLE     0       1
  # FORMAT
    cycle_sel = [zero , cntr   ];
    RD_FRMT   = [iLLLL, iZZLZ];
    DRV_FRMT = [oLLLL, oDDDD ];
}

Signals(MySignals){
   # To be used with GeneralFormats
    DIO_0    = dio(pin=0,  map=x[0],   format=RD_FRMT );
```

```
    DIO_4    = dio(pin=4,  map=y[0],    format=DRV_FRMT ); # Used for testing
    DIO_5    = dio(pin=5,  map=y[1],    format=DRV_FRMT ); # Used for testing
    DIO_6    = dio(pin=6,  map=y[2],    format=DRV_FRMT ); # Used for testing
    DIO_7    = dio(pin=7,  map=y[3],    format=DRV_FRMT ); # Used for testing
}

Pattern(DutyCycle){
    @auto cycle=zero, x=r0, y=r1;
DO_AGAIN:
    cycle=zero, r1=0x0f, clr(FCNTR);
    cycle=cntr, r1=r1>>>1, for(0xffff);
    cycle=cntr,  iomask=0x1;  # Read for 0;
    cycle=cntr, endfor;
    cycle=cntr, log(FCNTRL);  # Number of fails means how many ones were read
    cycle=zero, service(printDutyCycle(hw));
    cycle=zero, jmp(NUF, DO_AGAIN);    # Press UBTN to stop
    cycle=zero, service(pattern_stop(hw));
}
```

Listing 10.2: Arbitrary Waveform Generator using User Memory

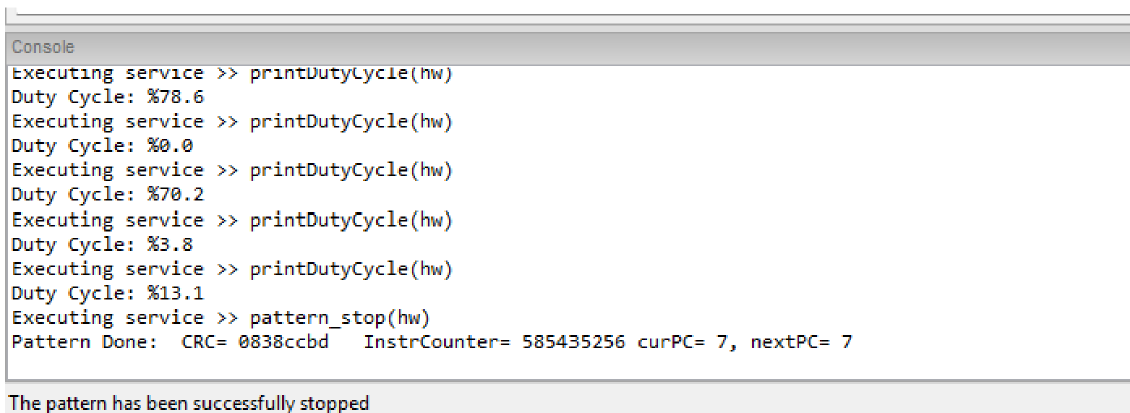


Figure 10.1: Output showing the Duty Cycle printed by the `printDutyCycle` service.

10.10 Frequency counter

10.11 Analog waveform generator

```
def sinewave(hw, SAMPLES=1024):
    from math import sin
    # We scale the data so that it's 0 to 10000 which can set the DAC to 0V to 10V
    for addr, val in enumerate( [(sin(i * 2 * 3.14/SAMPLES)+1)*5000  for i in range(SAMPLES)] ):
        hw.setUSERMEM(addr, int(val))
    return hw.CONTINUE

def sawtooth(hw, SAMPLES=1024):
    # We scale the data so that is it 0 to 10,000 which can set the DAC to 0V to 10V
    for addr, val in enumerate( [ (i/SAMPLES)*10000  for i in range(SAMPLES)] ):
        hw.setUSERMEM(addr, int(val))
    return hw.CONTINUE

def squarewave(hw, SAMPLES=1024):
    # We scale the data so that is it 0 to 10,000 which can set the DAC to 0V to 10V
    for addr, val in enumerate( [ ((i&16)>>4)*10000  for i in range(SAMPLES)] ):
        hw.setUSERMEM(addr, int(val))
    return hw.CONTINUE
```

```
# Define Formats
Formats(miniFrmts){
        # CYCLE     0
  # FORMAT
    cycle_sel   = [ zero ];
    OUT_LOW     = [oLLLL];
}

# Define Signals
Signals(miniSigs){
    DIO_0    = dio(pin=0,   map=0,   format=OUT_LOW );
}

# A pattern to generate arbitrary waveforms using user memory and DAC.
Pattern(waveform){
    @param TIMES=1024, DELAY=250, MASK=0x1ff;
    # Lets give the data a name
    cycle = zero, service(hw.setGroupName ("SINEWAVE"))
    # we need to fill the user memory with some arbitrary data
    cycle = zero, service (sinewave(hw, 512));
    cycle = zero, r0=0, call(SETDAC0);
    #cycle = zero, jmp(END);

    cycle = zero, service(hw.setGroupName ("SAWTOOTH"))
    # we need to fill the user memory with some arbitrary data
    cycle = zero, service (sawtooth(hw, 512));
    cycle = zero, r0=0, call(SETDAC0);

    cycle = zero, service(hw.setGroupName ("SQUAREWAVE"))
    # we need to fill the user memory with some arbitrary data
    cycle = zero, service (squarewave(hw, 512));
    cycle = zero, r0=0, call(SETDAC0);

END:
    cycle = zero, r1=0;
    cycle=zero, dac0=r1, service(pattern_stop(hw));

    # Lets capture it by ADC1
SETDAC0:
    cycle=zero, r1 = mem[r0], for(TIMES);
    cycle=zero, dac0=r1;
    cycle=zero, repeat(DELAY-4);
    cycle=zero, r0=r0+1, log(ADC);
    cycle=zero, r0=r0&MASK, endfor; # We make sure r0 goes from 0 to 1023 and
    wrapps around
    cycle=zero, return;

}
```

Listing 10.3: Arbitrary Waveform Generator using User Memory

10.12 Melody generator

10.13 NPN Transistor Characterization

```
# BC546: BJT NPN
Rb=10000 # 10K Ohm at Base
VBE=0.750

# Calls to sweep & measure currents/voltages
def setIB(cur_uA):
    hw.setDAC0( cur_uA * Rb * 1e-6 + VBE)

setVCC=hw.setPPSV
getVCE=hw.getADC0
def getIC(): return hw.getPPS()[0]

# Configure the PPS for Force Voltage Measure Current
```

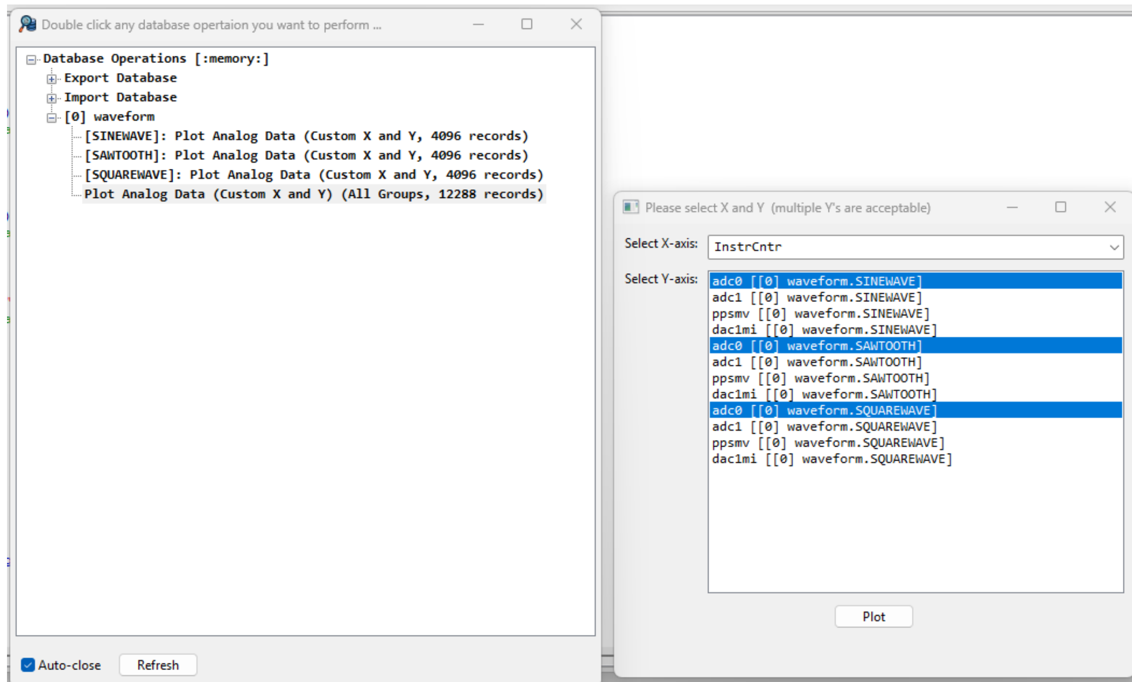


Figure 10.2: Using database operations to plot the DAC0 output captured by ADC0

```
14 hw.configPPS(opmode="FVMI", cur_range="500mA", volt_range="UNI_P10V")
15
16 # to plot you need to provide data in this format
17 # data=[["VCE", "IB=50uA"], [0,0], [1, 123] ... ]
18 # You can overlay curves
19
20 #1- Set IB --> 50uA, 100uA ...400uA in 50uA steps
21 for IB in range(50, 401, 50):
22     setVCC(0)
23     setIB(IB)
24     data=[["VCE", f"IB={IB}uA"]]
25     #2- Sweep VCE and measure Ic  (0....10V)
26     for vcc in range(11):
27         setVCC(vcc)
28         data.append([getVCE(), getIC()])
29     hw.plot(data, title="VCE vs IC for NPN Transistor", win=1, ylabel1="IC (mA)")
30
31 # Put supplies in safe state
32 hw.setPPSV(0)
33 hw.setVCC(0)
34 hw.setDAC0(0)
```

10.14 MOSFET Transistor Characterization

10.15 Flash programmer

10.16 EEPROM reader

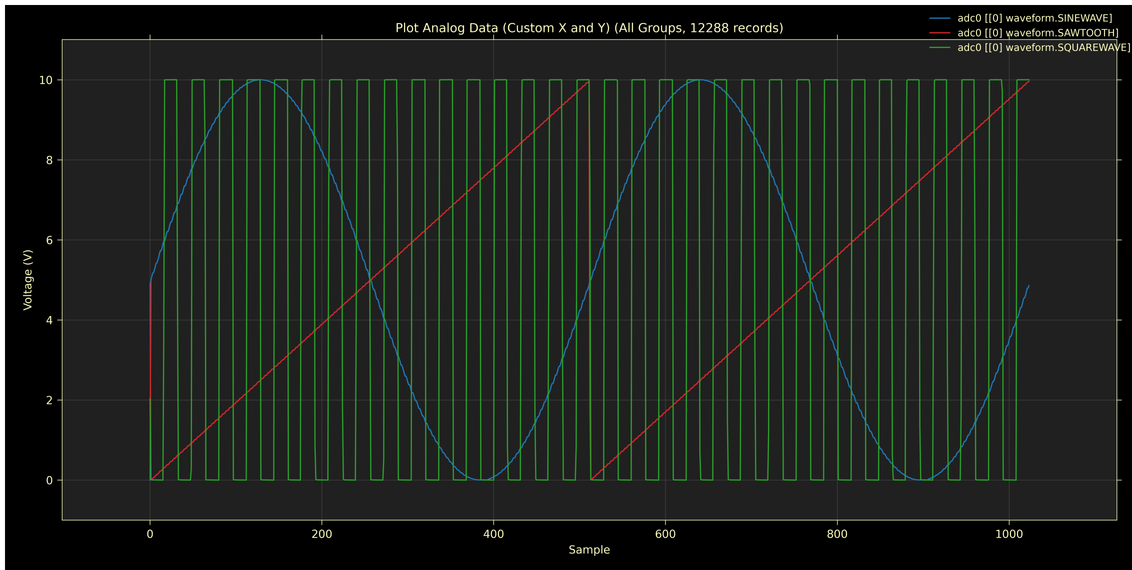


Figure 10.3: Plotting the ADC0 data groups.

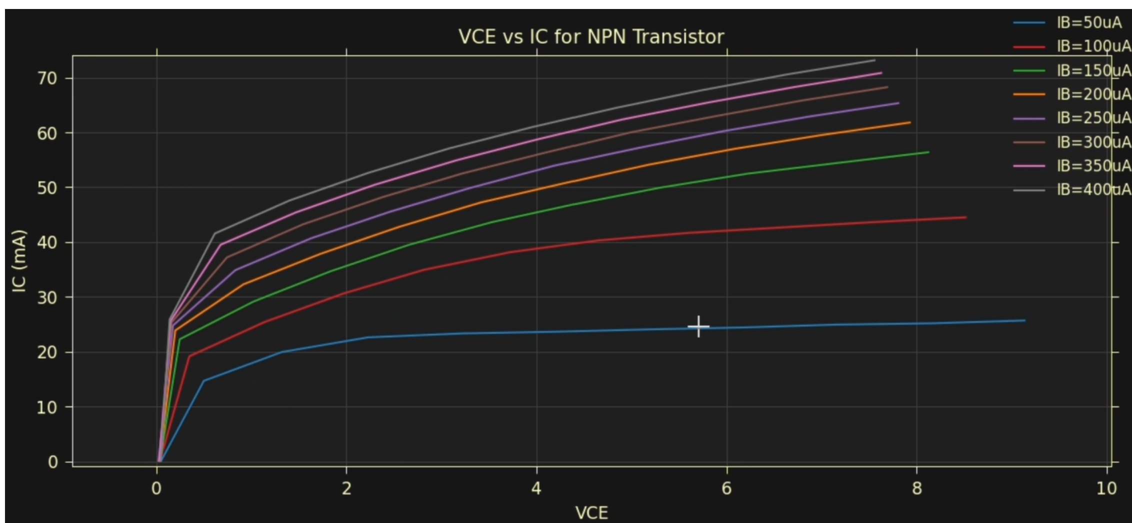


Figure 10.4: Plotting the NPN Transistor IV curves.

Reference

11 Troubleshooting and Optimization 82
11.1 Connection Issues
11.2 Pattern Execution Failures
11.3 Signal Integrity
11.4 Performance Optimization
11.5 Best Practices

12 Definitions 84
12.1 Object Types
12.2 Formats Syntax
12.3 Signals Syntax
12.4 Pattern Syntax
12.5 Database Tables

13 Descriptions and Definitions 89
13.1 Descriptions and Definitions
13.2 Glossary

Index 89

14 Version History 92
14.1 Version History

11. Troubleshooting and Optimization

11.1 Connection Issues

If LTStudio fails to connect to LT16M:

- Verify USB-C cable and port (use USB 2.0 if possible)
- Confirm power is on (status LED should be red or green)
- Check Device Manager for correct drivers:
  - Interface 1: `libusb-win32`
  - Interface 0: `USB Serial Converter A`
- Reinstall drivers using Zadig if needed
- Restart LTStudio and reconnect

11.2 Pattern Execution Failures

Common causes:

- Syntax errors in pattern code
- Missing signal or format definitions
- Invalid register usage or branching
- Unreachable service calls

Tips

- Use LTStudio's compiler messages to locate errors
- Test patterns incrementally
- Validate signal mappings before execution
- Verify with known patterns

11.3 Signal Integrity

To improve digital signal quality:

- Use short, shielded cables

- Avoid floating inputs — always drive or pull
- Match voltage levels using jumper settings
- Use series resistors for LEDs and displays

11.4 Performance Optimization

- Minimize WAIT instructions for faster execution
- Call services to offload pending data
- Limit service calls inside tight loops
- Preload reusable patterns and services

11.5 Best Practices

- Modularize patterns and services
- Use descriptive names and comments
- Validate hardware connections before execution
- Keep firmware and LTStudio updated

12. Definitions

12.1 Object Types

- **Format** – Defines timing and signal behavior
- **Signal** – Maps logical bits to physical pins
- **Pattern** – Sequence of instructions executed on LT16M
- **Service** – Python function executed on host PC
- **Batch** – Python script automating multiple objects

12.2 Formats Syntax

```
Formats Object Syntax:
======================
    FormatsObj    := Formats(<validName>){<format_header>; <formatSel_def; ...>}
    validName     := [a-zA-Z_][a-zA-Z0-9_]*
    list(<items>) := [<item0>, <item1>, ...]

    format_header := cycle_sel=list(<validNames>);
    formatSel_def := validName=list(<cycle_formats>);
    cycle_format  := <dir><tick><tick><tick><tick>
    dir           := [io]
    tick          := [LHDVKTMZ]

    Reserved Keywords:
         Formats cycle_sel
```

12.3 Signals Syntax

```
Signals Object Syntax:
======================
    SignalsObj    := Signals(<validName>){<[PinAssignment], ...>}
    validName     := [a-zA-Z_][a-zA-Z0-9_]*
    PinAssignment := PinLabel=pinType(pin=<PinNum>, map=<DataSource>, format=<FormatSel>);
    PinLabel      := validName
                  := Must be a valid variable name, e.g., SEL, CS1, WR_EN, _PIN1.
    pinType       := [dio]
```

```
   PinNum         := [0-9]|[1[0-5]]
                  := Options for LT16M are 0 to 15.
   DataSource     := [01]|x[0-9]|x[1[0-5]]|y[0-9]|y[1[0-5]]|z[0-9]|z[1[0-5]]
                  := Options are 0, 1, x[0], ...x[15], y[0], ...y[15], z[0], ...z[15].
   FormatSel      := validName   (Options are dependent on the current Format)
```

12.4 Pattern Syntax

```
Pattern Object:
===============
   pattern               := Pattern(<patternName>){[compilerInstruction][Instruction]...}
   compilerInstruction   := @using <symbolName>;
                            @auto cycle_uInstr<,uInstr>;
                            @param <parameterName>=<ParameterValue>, ...;
   Instruction           := [(Label)cycle_uInstr[<,ALU1_uInstr><,ALU2_uInstr><,XYZ_uInstr>
                                                 <,iomask_uInstr><,cntrl_uInstr><,log_uInstr>
                                                 <,do_uInstr><,branch_uInstr>];
   validName             := [a-zA-Z_][a-zA-Z0-9_]*
   rx                    := r0, r1 ... r15
   imm_16                := 16-bit integer number
   Label                 := <validName>:
   cycle_uInstr          := cycle=<validName>
   ALU1_uInstr           := NOP|REG_OP_REG|REG_OP_IMM|REG_ASSIGN|RAND_OP|MEM_RD|MEM_WRT|
                            AUXREG_ASSIGN|DAC_ASSIGN
   XYZ_uInstr            := (x|y|z)=rx
   iomask_uInstr         := iomask=imm_16b
   cntrl_uInstr          := clr([PF|T0|T1|FCNTR])
   log_uInstr            := log([FAIL|INFO|FCNTRL|FCNTRH|CHANGE|ADC])
   do_uInstr             := do=imm_4b
   branch_uInstr         := [<null>|jmpCallOp|cond_jmpCallOp|forRepeatOp|serviceOp|endfor_returnOp]
```

12.4.1 Cycle Syntax

```
Cycle Micro-Instruction:
========================
    cycle_uInstr  := cycle=<cycleName>
    cycleName     := Valid cycle name defined in selected Format object (in cycle_sel list).
```

12.4.2 ALU Syntax

```
ALU1/2 Micro-Instruction:
=========================
    rx                     := r0, r1 ... r15
    imm                    := 16-bit integer number
    imm_32b                := 32-bit integer number
    math_expr              := combination of numbers and math operation (Python style)
    ALU1_uInstr|ALU2_uInstr := NOP|REG_OP_REG|REG_OP_IMM|REG_ASSIGN|RAND_OP|MEM_RD|MEM_WRT|
                              AUXREG_ASSIGN|DAC_ASSIGN
    NOP                    := <null>
    REG_OP_REG             := rx=rx+rx
                              rx=rx-rx
                              rx=rx|rx
                              rx=rx&rx
                              rx=rx^rx
                              rx=rx>rx
                              rx=rx<rx
                              rx=rx*rx
                              rx=rx**rx
                              rx=rx>>>rx
                              rx=rx<<<rx
    REG_OP_IMM             := rx=rx+imm
                              rx=rx-imm
                              rx=rx|imm
                              rx=rx&imm
                              rx=rx^imm
                              rx=rx>>imm
                              rx=rx<<imm
```

```
                             rx=rx*imm
                             rx=rx**imm
                             rx=rx>>>imm
                             rx=rx<<<imm
    REG_ASSIGN           := rx=(rx|imm|math_expr|io)
    RAND_OP              := rx=rand(rx)
    MEM_RD               := rx=mem[rx|imm|math_expr]
    MEM_WRT              := mem[rx|imm|math_expr]=rx
                             mem[rx]=imm
    AUXREG_ASSIGN        := [T0|T1|SEED|FLIMIT]=imm_32b
    DAC_ASSIGN           := [DAC0|DAC1]=rx
```

12.4.3 Output Assignment Syntax

```
X/Y/Z Micro-Instruction:
========================
    X_uInstr      := x=rx
    Y_uInstr      := y=rx
    Z_uInstr      := z=rx
```

12.4.4 Control Syntax

```
Control Micro-Instruction:
========================
    cntrl_uInstr  := clr([PF|T0|T1|FCNTR])
```

12.4.5 IO Mask Syntax

```
IO Mask Micro-Instruction:
==========================
    imm_16b       := 16 bit integer
    iomask_uInstr := iomask=imm_16b
```

12.4.6 Log Syntax

```
Log Micro-Instruction:
======================
    log_uInstr    := log([FAIL|CHANGE|INFO|FCNTRL|FCNTRH|ADC])
```

12.4.7 Drive-only Syntax

```
Drive-Only Micro-Instruction:
===================================
    imm_4b        := 4 bit integer
    do_uInstr     := do=imm_4b
```

12.4.8 Branch Syntax

```
Branch Micro-Instruction:
=========================
    branch_uInstr  := [<null>|jmpCallOp|cond_jmpCallOp|forRepeatOp|serviceOp|endfor_returnOp]
    conditionFlag  := [NUF | UF | NT1 | T1 | NT0 | T0 | NFLE | FLE | NPF | PF | NF | F | NZ1 | Z1]
    jmpCallOp      := [jmp|call](<label>)
    cond_jmpCallOp := [jmp|call](<conditionFlag>, <label>)
    forRepeatOp    := [for|repeat]([imm|rx])
    serviceOp      := service(<serviceFuncName>)
    endfor_returnOp:= [endfor|return]
```

12.4.9 Compiler Instructions Syntax

```
Pattern Object:
===============
    pattern              := Pattern(<patternName>){[compilerInstruction][Instruction]...}
    compilerInstruction  := @using <symbolName>;
                            @auto cycle_uInstr<,uInstr>;
                            @param <parameterName>=<ParameterValue>, ...;
```

12.5 Database Tables

- `Records` – Raw signal logs
- `Groups` – Logical test segments
- `GroupsInfo` – Metadata for each group
- `Info` – Metadata for each group
- `AnalogData` – ADC samples
- `IOCounters` – Pin-level toggle counters
- `IOFails` – Fault detection logs
- `IOChange` – Fault detection logs

12.5.1 Records Table

```
Sample Records Table:
#  id*  | Type
#   1     Groups
#   2     Groups
#   3     Info
#   4     IOFails
#   5     IOFails
#   7     IOFails
#   10    Info
#   161   IOCounters
#   162   IOCounters
#   163   IOCounters
#   164   IOCounters
#   170   Groups
#   171   Groups
#   176   AnalogData
#   177   AnalogData
#   181   Groups
#   185   Info
#   192   AnalogData
#   194   AnalogData
#   195   AnalogData
#   200   Groups
#   201   IOChange
#   202   IOChange
#   203   IOChange
#   204   IOChange
```

12.5.2 Groups Table

```
Sample Groups Table:
#  id*   |  Name      |    Level    | Recorded_at
#   1      [1] faillog      1       1622547800.0
#   2      VI_Trendline     2       1622557800.0
#   160    [2] pps_test     1       1622657800.0
#   161    HighVolt_plot    2       1622557900.0
#   175    LowVolt_plot     2       1622857800.0
#   181    LowVolt_plot     2       1622857800.0
#   200    [2] vcd_test     1       1622957800.0
```

12.5.3 GroupsInfo Table

```
Sample GroupsInfo Table:
#   id*    |     Formats     |     Siganls    |     Params      |     PinLabels
#    1      GeneralFormats  GeneralSignals  {REPEAT=1000}   CS_,SCLK,MOSI,MISO,,,,,,,,,,,,
#   200     DefaultFormats  GeneralSignals                  RST_N,SCL,SDA,SEL0,SEL1,,,,,,,,,,,
```

12.5.4 InfoView Table

```
Sample InfoView Table:
# id*   |  X  |  Y   |  Z   |  curPC    |  nextPC   |InstrCntr  | GroupName
#  3       12     20     61     60          61         101923      [1] faillog.VI_Trendline
```

```
#  10     2     12     50     100        101       102010     [1] faillog.VI_Trendline
#  185    15    0      7      2          7         30         [2] pps_test.LowVolt_plot
```

12.5.5 AnalogDataView Table

```
Sample AnalogDataView Table:
#   id*      |  Type    |  Value     | Unit | curPC   | InstrCntr  | GroupName
#    165        adc1       12.6         V       12      120342       [2] pps_test.HighVolt_plot
#    167        adc0       12e-3        V       12      212423       [2] pps_test.HighVolt_plot
#    186        dac1mi     0.3          mA      12      212515       [2] pps_test.LowVolt_plot
#    187        ppsmi      1.232        mA      12      212513       [2] pps_test.LowVolt_plot
#    189        ppsmv      5.0          V       12      212514       [2] pps_test.LowVolt_plot
```

12.5.6 IOCountersView Table

```
Sample IOCountersView Table:
#  id*     |  IO | Counter    | GroupName
#   163       0      102       [2] pps_test.HighVolt_plot
#   164       1      654       [2] pps_test.HighVolt_plot
#   176       2      100       [2] pps_test.LowVolt_plot
#   177       3      7         [2] pps_test.LowVolt_plot
```

12.5.7 IOFailsView Table

```
Sample IOFailsView Table:
#  id*   |      X     |      Y     |      Z     |     Tick    |      IO   | GroupName
#   4           12           20          60           3            7         [1] faillog.VI_Trendline
#   5           12           20          61           0            7         [1] faillog.VI_Trendline
#   7           15           0           7            1            1         [1] faillog.VI_Trendline
#   204         12           20          60           3            7         [2] vcd_test
#   205         12           20          61           0            7         [2] vcd_test
#   207         15           0           7            1            1         [2] vcd_test
```

12.5.8 IOChangeView Table

```
Sample IOChangeView Table:
#  id*    | InstrCntr |  IO       | GroupName
#   201      10001       127        [2] vcd_test
#   202      10102       60         [2] vcd_test
#   203      10134       12         [2] vcd_test
#   204      10207       54         [2] vcd_test
```

13. Descriptions and Definitions

13.1 Descriptions and Definitions

ADC – Analog-to-digital converter
ATPG – Algorithmic Test Pattern Generator
DAC – Digital-to-analog converter
DIO – Digital input/output pin
DO – Digital Drive-only pin
LT16M – LogicTerm Mixed-signal 16-pin box
LT16D – LogicTerm analog carrier board
LT16A – LogicTerm digital base board
PPS – Programmable power supply
UBTN – User button
ULED – User RGB LED

13.2 Glossary

- **Pattern** – A sequence of instructions executed on LT16M
- **Format** – Defines signal behavior per cycle
- **Signal** – Maps logical bits to physical pins
- **Service** – Python function executed on host PC
- **Batch** – Python script that automates multiple objects
- **log** – Micro-instruction to capture data

Index

A

ADC . . . 12
ALU Operations . . . 29
AnalogDataView Table . . . 65

B

Batch examples . . . 57
Batch Scripts . . . 56
Branching . . . 38

C

call . . . 39
Compiler Instructions . . . 41

D

DAC . . . 12
Database Structure . . . 61
DO Assignment . . . 37

F

FLIMIT . . . 30
for . . . 40
Formats Object . . . 23
Formats Syntax . . . 25

G

Groups Table . . . 63
GroupsInfo Table . . . 63

H

hw.configPPS . . . 51
hw.getADC0 . . . 50
hw.getADC1 . . . 50
hw.getDAC1MI . . . 50
hw.getDbName . . . 45
hw.getFLIMIT . . . 49
hw.getGPR . . . 42
hw.getInstrCounter . . . 42
hw.getPC . . . 42
hw.getPPS . . . 51
hw.getUSERBUTTON . . . 44
hw.getUSERLED . . . 43
hw.getUSERMEM . . . 47
hw.getUSERWORD . . . 43
hw.plot . . . 52
hw.plotall . . . 52
hw.PrintDB . . . 45
hw.printGPRs . . . 42
hw.runQuery . . . 45
hw.setDAC0 . . . 50
hw.setDAC1 . . . 50
hw.setFLIMIT . . . 49
hw.setGroupName . . . 47

hw.setPPSV . . . 51
hw.setUSERLED . . . 43
hw.setUSERMEM . . . 47
hw.setUSERWORD . . . 43
hw.sleep . . . 55
hw.waitUSERBUTTON . . . 44

I

InfoView Table . . . 65
Input Format Ticks . . . 25
IO Mask . . . 33
IOChangeView Table . . . 64
IOCountersView Table . . . 65
IOFailsView Table . . . 64

J

jmp . . . 38

L

Log(ADC) . . . 36
Log(CHANGE) . . . 34
Log(FAIL) . . . 34
Log(FCNTRH) . . . 36
Log(FCNTRL) . . . 36
Log(INFO) . . . 35
LT16M Box . . . 11
LTStudio IDE . . . 15

M

Memory Access . . . 31
Micro-Instructions . . . 29

O

Output Format Ticks . . . 24

P

Patterns Object . . . 28
PPS . . . 13

R

Random assignment . . . 31
Records Table . . . 63
repeat . . . 40

S

SEED . . . 30
service . . . 40
Service Objects . . . 41
Signals Object . . . 26
SignalSyntax . . . 26

U

UBTN . . . 12
ULED . . . 12
USB Driver . . . 16

W

Wrappers for Batch Scripts . . . 56
Wrappers for services . . . 42

14. Version History

14.1 Version History

- v1.0 – Initial release with core features
- v1.1 – Added PPS current clamping and ADC enhancements
- v1.2 – Improved LTStudio UI and batch scripting support
- v1.3 – Peer review and document consolidation