

Electronics for Embedded Systems

Experimental Project

Fabio Scatozza (315216)

July 29, 2025

Contents

1	Introduction	3
2	Theory of Operation	3
3	System Design	7
3.1	Angular Velocity Acquisition	8
3.2	Angular Displacement Acquisition	9
3.2.1	PS/2 Peripheral	10
3.3	7-Segment Displays Peripheral	13
3.3.1	UAR(T) Core	14
3.4	Bipolar Stepper Motor Drive Circuit	15
4	Software Design	17
4.1	The Buildsystem	18
4.2	Virtual File System	19
4.3	Virtualized Resources	20
4.3.1	Object Abstractions	20
4.3.2	Device Drivers	25
4.4	Main Application	27
A	PS/2 Controller	29
B	Show-Ahead Single-Clock FIFO	31
C	SPI Slave Controller	31
D	UAR(T) Core	33

References

- [1] Analog Devices. *AD7928 8-Channel, 1 MSPS, 12-Bit ADC with Sequencer in 20-Lead TSSOP*. Datasheet. Version E. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/AD7908_7918_7928.pdf.
- [2] Anhui Vico Technologies. *Metal Film Fixed Resistors*. Datasheet. Dec. 2019. URL: https://www.lcsc.com/datasheet/lcsc_datasheet_2409272302_V0-MF2WS-0R68-1--6T63_C2903178.pdf.
- [3] IBM Corporation. *Personal System/2 Hardware Interface Technical Reference*. Oct. 1990. Chap. Keyboard and Auxiliary Device Controller. URL: http://www.mcamafia.de/pdf/ibm_hitrc07.pdf.
- [4] IBM Corporation. *Personal System/2 Hardware Interface Technical Reference*. Oct. 1990. Chap. Keyboard 101- and 102-Key. URL: http://www.mcamafia.de/pdf/ibm_hitrc11.pdf.
- [5] Intel Corporation. *Embedded Memory (RAM: 1-PORT, RAM: 2-PORT, ROM: 1-PORT, and ROM: 2-PORT) User Guide*. URL: <https://www.intel.com/content/www/us/en/docs/programmable/683240/17-0/about-embedded-memory-ip-cores.html>.
- [6] Intel/Altera Corporation. *Cyclone V SE SoC FPGA*. URL: <https://www.intel.com/content/www/us/en/products/details/fpga/cyclone/v/se.html>.
- [7] Linear Technology Corporation. *LTC2308 Low Noise, 500kps, 8-Channel, 12-Bit ADC*. Datasheet. Version C. URL: <https://www.analog.com/media/en/technical-documentation/data-sheets/2308fc.pdf>.
- [8] Mean Well Enterprises. *LRS-50-12 50W Single Output Switching Power Supply*. Datasheet. Feb. 2024. URL: <https://www.meanwell-web.com/content/files/pdfs/productPdfs/MW/LRS-50/LRS-50-spec.pdf>.
- [9] Stepperonline. *17HE15-1504S E-Series Nema 17 Bipolar Stepper Motor*. Datasheet. May 2023. URL: https://www.omc-stepperonline.com/index.php?route=product/product/get_file&file=2838/17HE15-1504S_Full_Datasheet.pdf.
- [10] STMicroelectronics. *Description of STM32F4 HAL and low-layer drivers*. User Manual. Mar. 2023. URL: https://www.st.com/resource/en/user_manual/um1725-description-of-stm32f4-hal-and-lowlayer-drivers-stmicroelectronics.pdf.
- [11] STMicroelectronics. *L298 Dual Full-Bridge Driver*. Datasheet. Oct. 2023. URL: <https://www.st.com/resource/en/datasheet/l298.pdf>.
- [12] STMicroelectronics. *L6505 Current Controller For Stepping Motors*. Datasheet. July 2003. URL: <https://www.st.com/resource/en/datasheet/l6506.pdf>.
- [13] STMicroelectronics. *Nucleo-64 Development Board with STM32F401RE MCU*. Datasheet. Jan. 2015. URL: <https://www.st.com/resource/en/datasheet/stm32f401re.pdf>.
- [14] STMicroelectronics. *STM32CubeMX Initialization Code Generator*. URL: <https://www.st.com/en/development-tools/stm32cubemx.html>.
- [15] Terasic. *DE1-SoC Board*. URL: <https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=165&No=836>.
- [16] Vishay General Semiconductor. *SB260 Schottky Barrier Plastic Rectifier*. Datasheet. Apr. 2020. URL: <https://www.vishay.com/docs/88717/sb220.pdf>.
- [17] XSD. *6063-T5 Aluminum Heatsink*. Datasheet. Mar. 2017. URL: https://www.lcsc.com/datasheet/lcsc_datasheet_1811100911_XSD-30-30-25-2P_C32487.pdf.

1 Introduction

The *Electronics for Embedded Systems* course explores the analog and digital aspects of embedded system design, focusing on the key components of their typical architecture. While sensor and actuator specifics are addressed in other courses, this course emphasizes the interface between the analog and digital domains, particularly A/D and D/A conversion, as well as the power electronics of actuator drivers and power supply devices. In the digital domain, the course spans software implementations running on microprocessors to hardware synthesized on programmable devices, with an emphasis on the underlying technology enabling computation, communication, and storage. Specifically, this involves analyzing the theory of programmable logic devices, microprocessor embedded peripherals, wired communication protocols, and memories.

In this context, the document presents a small-scale project aimed at addressing the key topics of the course, demonstrating competency aligned with the expected learning outcomes.



2 Theory of Operation

The system implements an interface to program the movement pattern for a bipolar stepper motor and execute it on demand. The pattern is committed to non-volatile storage and consists of a sequence of data points, each representing a circular motion segment defined by angular velocity and displacement. During execution, the data points are retrieved in a circular fashion from memory, and are converted into motor-driving stimuli. Referring to the system block diagram in fig. 1, the interface is operated as follows.

- The movement pattern is cleared by pressing and holding the user push button for at least 1 s. Successful clearing is acknowledged by displaying *CLEAR* on the 7-segment displays.
- Data points are entered using the linear potentiometer and the PS/2 keyboard. The potentiometer sets the angular velocity in the range from 2.5 rpm to 400 rpm, while the angular displacement is specified in degrees with the keyboard, according to the following regex pattern.

$$[+-]?[0-9]{1,3}(\.[0-9])?$$

The sign defaults to + if omitted, which corresponds to a counterclockwise rotation. The integer part is specified with up to three digits, whereas the fractional part is optionally specified with one digit. Due to the inherently discrete nature of the stepper motor rotation and accounting for the two-phases-on driving method, the target angular displacement is rounded to the nearest value that minimizes error.

When the data point entry begins, triggered by pressing some keys, the 7-segment displays change to *InPUt*. The operation is terminated by pressing the return  or  keys, upon which data is validated, and the potentiometer output is converted. The outcome is notified as follows: *Err-2 bAd PAttErn*, if the parsing of the keyboard input failed; *Err-3 FuLL*, if the maximum number of data points have already been entered. Otherwise, if no error has occurred, the data point is displayed in the format below, where the data point number starts at zero, the angular velocity is expressed in revolutions per minute, and the angular displacement in degrees.

$$[<\text{data point number}>] <\text{angular velocity}> <\text{angular displacement}>$$

- The execution of a valid movement pattern, that is, one containing at least one data point, begins by pressing the user push button. An empty pattern results in the display of *Err-1 no dAtA*, while a valid pattern displays *PLAy*. The execution continues until the push button is pressed again, after which the displays revert to printing *IdLE*.

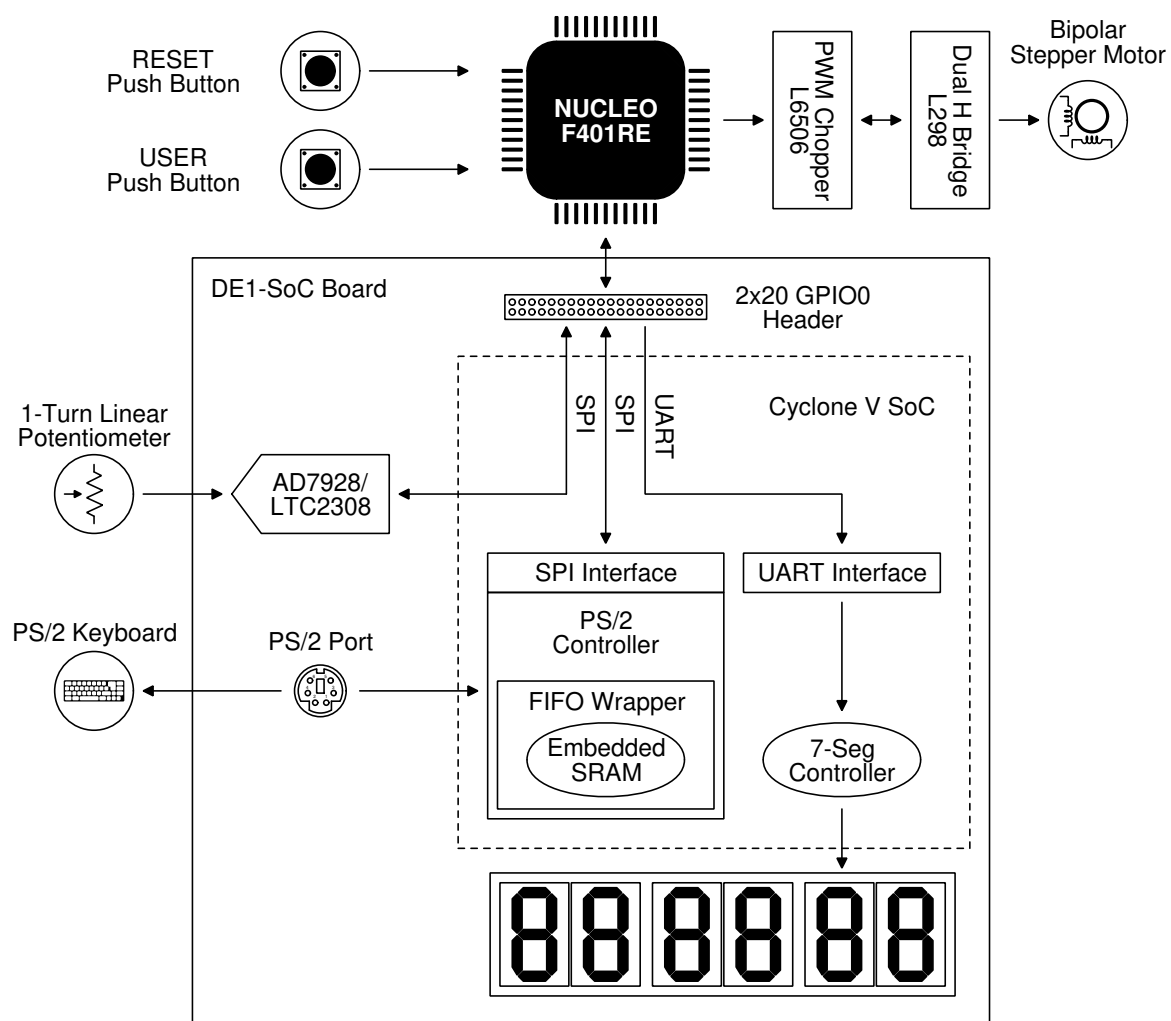
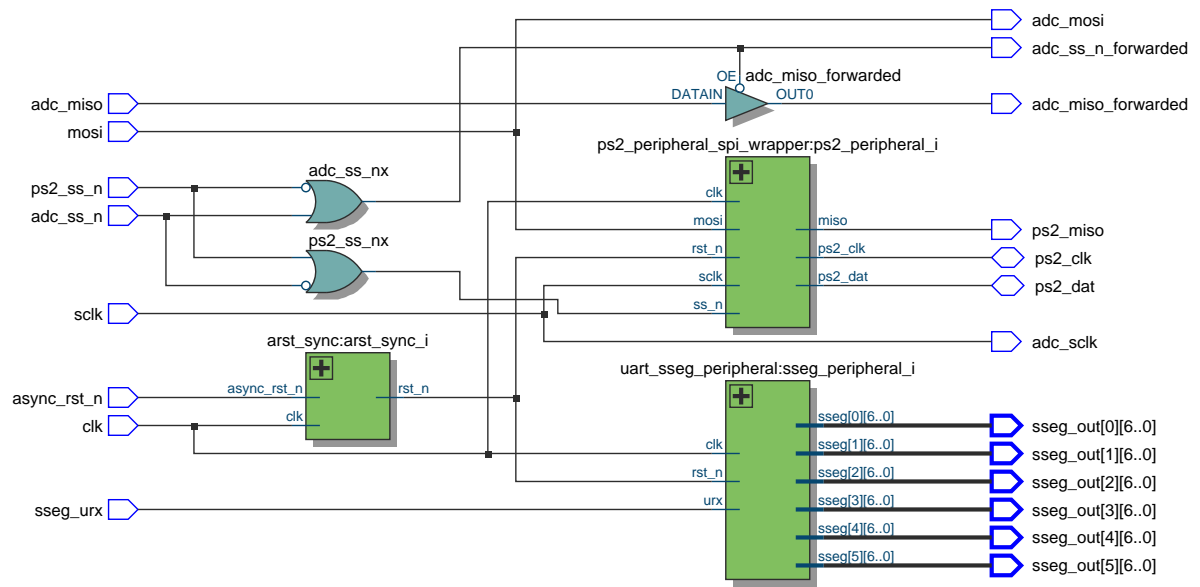
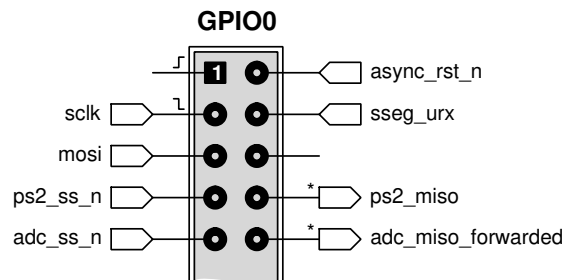


Figure 1: System block diagram



(a) Quartus Prime RTL view



(b) Signals mapped to the 2x20 header for interfacing with the MCU

Figure 2: DE1-SoC board subsystems

Figure 4: Drive circuit for the bipolar stepper motor. The L6506 senses and controls the current in the motor windings, which are energized with the dual full-bridge driver L298.

3 System Design

The system is devised to target the hardware available on the design platform employed for the laboratory activities of the course, the DE1-SoC development kit by Terasic [15]. The platform is built around Altera's Cyclone V SE SoC device [6], which integrates an ARM-based Hard Processor System (HPS) tied to the FPGA fabric with an on-chip interconnect. Potentially, this kit alone allows to experiment with digital processing in the form of software running on a hard or soft microprocessor to program hard and soft peripherals, as well as by synthesizing dedicated hardware. Despite this flexibility, the processor system is implemented with an external microcontroller development board, the NUCLEO-F401RE by STMicroelectronics [13], thus creating the opportunity to investigate communication protocols for wired interconnections. As illustrated in the block diagram of fig. 1, the system comprises:

- Input devices.
 - The user and reset push buttons are part of the microcontroller development board.
 - The linear potentiometer is wired to the ADC available on the DE1-SoC board, through the dedicated 2x5 header. This is discussed in section 3.1.
 - The PS/2 keyboard is attached to the corresponding port on the DE1-SoC board, through the fitted 6-pin mini-DIN connector.
- Peripherals and glue logic implemented on the FPGA target, as more accurately shown in fig. 2a. The interface to the microcontroller consists of the signals mapped to the 2x20 expansion header GPIO0, as depicted in fig. 2b.
 - The SPI-compatible port of the converter is routed through the FPGA fabric, up to the GPIO0 header.
 - An asynchronous reset signal driven by the microcontroller is brought into the FPGA 50 MHz clock domain, to be used as a synchronously deasserted system-wide reset.
 - The PS/2 peripheral implements the PS/2 protocol to communicate with the keyboard device, while it exposes an SPI port to the microcontroller. The design of the module is presented in section 3.2.
 - The display peripheral drives the 7-segment displays available on the DE1-SoC board with the character stream received from the microcontroller via UART. This is discussed in section 3.3.

The slave select inputs are internally pulled up, to ensure an inactive level while the microcontroller resets.

The blocks are modeled in SystemVerilog and are functionally simulated with testbenches that make use of the verification subset of the language. The approach to verification is directed testing, with non-constrained randomized stimuli. As a result, the testbenches can be run with the freeware edition of Questa Intel Starter FPGA Edition. The synthesis flow is carried out with the Quartus Prime Lite toolchain, using bash and Tcl automation scripts.

- Microcontroller and bipolar stepper motor drive circuit, as illustrated in greater details in figs. 3 and 4. The microcontroller development board, the L6506 controller [12], and the logic section of the L298 driver [11], are powered at 5 V from the DE1-SoC board; the microcontroller power supply is further regulated down to 3.3 V.
 - As the FPGA I/Os are configured in the 3.3 V LVTTTL standard, the microcontroller directly interfaces to the signals on the 2x20 header GPIO0. Notably, the PS/2 and ADC MISO signals are wired as a shared line into the microcontroller, and the input is internally pulled down to ensure a well-defined logic level while neither slave is selected.Several peripherals are configured to achieve the desired functionality, most importantly:

- * The GPIO, to detect push button presses, as well as for generating reset and enable signals for the FPGA subsystem and its peripherals.
- * The SPI3, USART1, and DMA2 to communicate with the peripherals synthesized in the FPGA.
- * The TIM1 and DMA2, to generate two pairs of complementary square waveforms, with one pair in quadrature to the other, to drive the bipolar stepper motor.

Furthermore, one sector of the embedded NOR flash memory is used as a non-volatile circular buffer to store the motor movement pattern, and play it on demand. The architecture of the firmware is further explained in section 4.

- The L6506 inputs accept 5 V TTL levels, with $V_{IL} = 0.8 \text{ V}$ and $V_{IH} = 2 \text{ V}$. Consequently, the microcontroller can reliably drive them, without requiring any additional translation logic. During reset, the enable signal is pulled down with an external resistor.
- The L298 is permanently enabled. The datasheet prescribes that the enable signals of the H-bridges are to be driven low before turning on or off the power stage, which is achieved by means of the resistive divider (R_5 , R_6).

The design of the circuit, for the bipolar stepper motor 17HE15-1504S [9], is discussed in section 3.4.

3.1 Angular Velocity Acquisition

Depending on the board revision, the ADC is either the AD7928 [1] or the LTC2308 [7], which feature 12 bit of resolution with a successive approximation topology, and an SPI-compatible serial interface. The analog input range changes slightly between the two devices: for the AD7928, the full range can be digitally selected equal to the external 2.5 V nominal reference or double its value. For the LTC2308 instead, it is fixed to 4.096 V. In either case, the 2x5 dedicated header only provides access to the eight input channels, to the ground, and to the 5 V supply rail, thus preventing measurements in a ratiometric configuration.

Accordingly, for the AD7928, a 10 k Ω linear potentiometer is wired between the 5 V supply rail and ground, with the wiper node connected to a single-ended input channel. The 500 μA nominal current through the voltage divider makes the 1 μA leakage current negligible. In the worst case scenario for the sample and hold circuitry, the wiper of the potentiometer sits at halfway position, such that the output resistance of the thevenin equivalent R_s is at its maximum value of 2.5 k Ω . In this condition, with a typical hold capacitance C_h of 20 pF and an equivalent input resistance R_{in} of 400 Ω , a fully discharged hold capacitor charges up to the 2.5 V input V_s following

$$v_h(t) = V_s \left(1 - e^{-t/\tau}\right), \quad \text{with} \quad \tau = (R_s + R_{in}) C_h$$

and the time required for this voltage to settle within half LSB of the final value is

$$t_{\text{set}} = \tau \ln \frac{V_s}{0.5 \text{ LSB}} \quad (1)$$

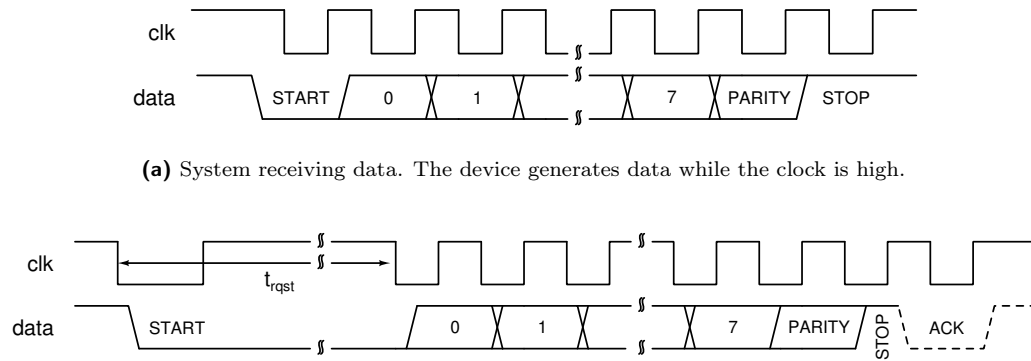
Hence, $t_{\text{set}} = 482 \text{ ns}$ bounds the minimum acquisition time for achieving full resolution conversions.

Considering the narrower analog range of the LTC2308, the 10 k Ω potentiometer is wired at the bottom of a voltage divider, with the resistor R_1 on the upper branch sized to limit the voltage drop on the potentiometer. Denoting with V_s the supply voltage and with V_{fr} the ADC full range, it must hold

$$V_s \frac{R_{\text{pot}}}{R_{\text{pot}} + R_1} \leq V_{fr}$$

Therefore, in the typical case for the supply and full range voltages, but considering a 10 % uncertainty on the potentiometer resistance and a 1 % uncertainty for R_1 , it is required that

$$R_{1,\text{min}} \geq R_{\text{pot,max}} \left(\frac{V_s}{V_{fr}} - 1 \right)$$



(b) System sending data. The system generates data while the clock is low; following the parity bit, the device detects the stop bit and pulls the data line low while the clock is high (dashed portion of the waveform).

Figure 5: PS/2 timing diagrams. The clock period ranges from 60 μ s to 100 μ s; the maximum request-to-send duration is 15 ms.

which yields $R_1 \geq 2.43 \text{ k}\Omega$. Accordingly, R_1 can be chosen in the E12 series with a nominal value of 2.7 $\text{k}\Omega$. Finally, the observation on the minimum acquisition time for the AD7928 applies here too. Given the different configuration of the voltage divider, the output resistance is maximized when the wiper reaches a position where the total divider resistance is split in half, hence R_s is 3.18 $\text{k}\Omega$. Considering the equivalent model of the analog input, with a typical hold capacitance C_h of 55 pF and an equivalent input resistance R_{in} of 100 Ω , eq. (1) yields $t_{set} = 1.53 \text{ }\mu\text{s}$.

The four signals of the ADC serial interface are routed through the FPGA fabric up to the 2x20 GPIO0 expansion header, where they connect to the microcontroller pins assigned to the SPI peripheral. The peripheral is configured as a full-duplex master, with software-controlled slave select signals.

3.2 Angular Displacement Acquisition

The PS/2 protocol [3, 4] regulates the point-to-point bidirectional communication between the system and the keyboard device. It is a synchronous serial protocol in which data is time-multiplexed over a single data line, while a dedicated line carries the clock signal. Accordingly, the communication is half-duplex and both clock and data lines are shared with an open-drain architecture to avoid damage in case of conflicts.

The clock is always generated by the PS/2 device, but the system can pull the clock line low to inhibit the bus, potentially causing collisions that are used to abort ongoing transmissions in either direction. When the bus is not inhibited, two configurations are possible prior to a transmission: if the data line is also pulled high, the bus is in its idle state, which is the only one in which the device can initiate a transmission to the system. Alternatively, if the data line is held low by the system, the configuration signals the device that the system is ready to transmit data and awaiting clock generation. The transmission is character-based and the frame comprises:

- Start bit, always 0
- 8 bit character, transmitted LSB first
- Odd parity bit, either 0 or 1 such that the overall count of bits with value 1 in the character concatenated with the parity bit is odd
- Stop bit, always 1

The timing diagram for receiving and sending data are shown in fig. 5.

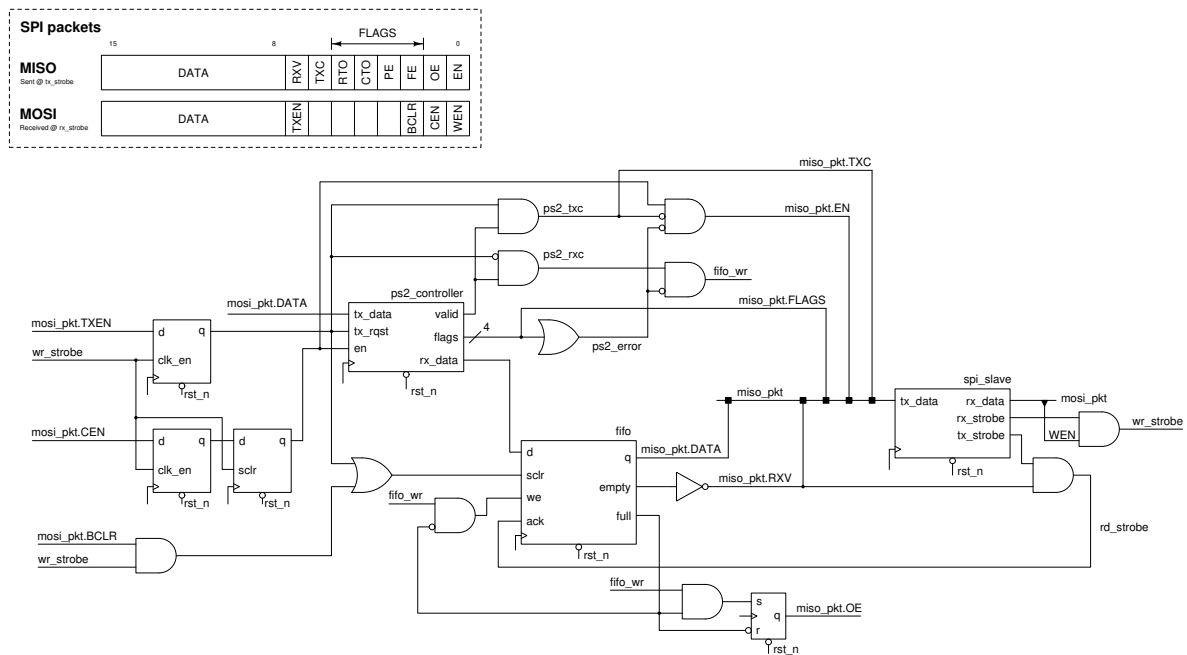


Figure 6: SPI-compatible PS/2 controller peripheral design. The module implements the PS/2 protocol for exchanging data with a PS/2 device, exposing its control, status, and data registers through SPI. Data received from the device is buffered in an SRAM FIFO. The PS/2 and SPI buses are not shown.

3.2.1 PS/2 Peripheral

The PS/2 protocol is implemented by the custom peripheral shown in fig. 6, which enables data exchange with an external PS/2 device, under the control of an SPI-capable system. It includes three key modules: a PS/2 controller, a show-ahead single-clock FIFO, and an SPI slave controller. Additional glue logic implements the desired behavior for the control and status register bits that are exposed through SPI.

PS/2 Controller It implements the PS/2 protocol specifications and provides an higher-level simplified interface to the PS/2 device, as presented in listing 3.1. It supports bidirectional communication and provides several reliability features. The data and clock lines are synchronized to the system clock domain with a flip-flop chain: the depth defaults to 2, which achieves a reasonable trade-off of latency and Mean Time Between Failures (MTBF) due to metastable events. The controller detects:

- Frame errors. On reception, if the start or stop bit is missing at the expected time. On transmission, if the acknowledgment bit is similarly absent.
- Parity error, if the parity check fails during reception.
- Clock timeouts, if the falling edges of the clock are separated by more than 150 μ s during either reception or transmission.
- Request timeouts. On transmission, if the device does not generate the clock within a 15 ms window after the system inhibits the PS/2 bus (see fig. 5).

The controller has been designed according to the ASM methodology, starting from a pseudocode description of the receive and transmit operations; the resulting ASM chart and data path are provided in appendix A. In summary, the module operates as follows.

```
module ps2_controller #(
    parameter int unsigned SYNC_STAGES = 2,
    parameter real FCLK_HZ = 50e6,
    parameter int unsigned TIM_PSC = 249 // tim_fclk = fclk/(PSC+1) = 200 kHz, for 50 MHz clock
) (
    input var logic clk,
    input var logic rst_n,
    input var logic en,

    input var logic tx_rqst,
    input var logic [7:0] tx_data,

    output logic valid,
    output logic [7:0] rx_data,
    output ps2_pkg::flags_t flags,

    inout tri ps2_clk,
    inout tri ps2_dat
);
```

Listing 3.1: SystemVerilog interface of the PS/2 controller. The number of flip-flop synchronizing stages for the PS/2 lines is parameterized and defaults to 2. The timing thresholds are computed at compile-time, based on the specified clock frequency and prescaler value.

- On reset or while not enabled, the controller inhibits the bus by driving the clock line low.
- While enabled, with no pending transmission request, the controller monitors the bus for transmissions from the device. In case of successful reception, the valid signal pulses for one clock cycle and the received character appears on the data output. Conversely, in case of failure, the valid signal is held high while the flags signal the cause. While in this failure state, the PS/2 bus is inhibited; resuming normal operation requires a falling transition on the enable signal.
- While enabled, with no ongoing transmission to the device, the controller samples the data to be transmitted in the same clock cycle where the transmission request gets asserted. In case of successful transmission, the valid signal is held high until the request signal becomes inactive, and the bus is inhibited. Conversely, the failure behavior is the same as in reception.

Show-Ahead Single-Clock FIFO Data received from the PS/2 device is buffered in a synchronous FIFO structure with a show-ahead interface. Provided that the empty signal is inactive, this interface automatically outputs the first valid word, and asserting the read-acknowledge signal causes the FIFO to output the next valid word.

The module is designed to infer embedded dual-port SRAM blocks if available, without depending on a specific read-during-write behavior. On reset, the read and write pointers are cleared, and the FIFO is empty. With the first write operation, the data is stored at the write pointer location, which is the same one being read: due to the read-during-write event, the SRAM may output the old data, thus the latency for the data to appear at the output is 2 clock cycles. Accordingly, the empty flag is cleared with the same latency. The full design is provided in appendix B.

SPI Slave Controller It implements the SPI protocol in the slave role, with programmable frame length, clock polarity (CPOL) and clock phase (CPHA).

To decrease clock oversampling requirements, the receive and transmit logic operate within the domain of the external SPI clock, generated by the master device; the slave select signal crosses into the system clock domain with a dual flip-flop synchronizer. While the slave is not selected,

the data to be transmitted is sampled in the system clock domain up to one clock cycle before the transmit strobe signal pulses high, which marks the start of communication. At the end of the communication, when the slave is deselected, the received data is loaded onto the data output and becomes valid in the system clock cycle when the receive strobe signal pulses high. Appendix C contains the design of the module and the manual timing analyses.

The peripheral exposes its control, status, and data registers as packets exchanged through SPI, according to the format summarized in fig. 6. A detailed description of the packets format is provided below.

MOSI Packet

- Bit 0* **WEN:** Write Enable
0: The packet is discarded
1: The packet is decoded
- Bit 1* **CEN:** Controller Enable
0: Disables the controller and inhibits the PS/2 bus (clock line low)
1: Clears the flags (FE, PE, CTO, and RTO), and enables the controller in the operation mode set by TXEN.
If the controller is already running (EN bit is set), the current operation is aborted. Aborting a receive operation while the PS/2 device is transmitting a character generates a communication error (FE, PE, or CTO set).
- Bit 2* **BCLR:** Buffer Clear
0: The receive FIFO buffer is not flushed
1: The receive FIFO buffer is flushed. If both CEN and BCLR are set, the flush operation is performed prior to enabling the controller.
- Bit 7* **TXEN:** Transmitter Mode Enable
This bit determines the operation mode.
0: The controller receives data from the PS/2 device. Incoming packets are stored in the FIFO buffer, whose state is represented by the RXV and OE status bits. When the FIFO is empty, the RXV bit is cleared and the DATA field is undefined. Otherwise, the RXV bit is set, and the DATA field is valid. If the FIFO buffer is full and new packets are received, the OE status bit is set. Each SPI transfer removes one element from the FIFO, which may also clear the OE bit. Alternatively, the OE bit is cleared by flushing the FIFO buffer when the BCLR control bit is set.
1: The FIFO buffer is cleared, and the controller sends data to the PS/2 device. When the operation completes, the TXC bit is set, the flags are updated (FE, CTO, and RTO), the EN bit is cleared and the PS/2 bus is inhibited (clock line low).
- Bits 15:8* **DATA:** Transmit Data
If TXEN is set, the data to be sent to the PS/2 device.

MISO Packet

- Bit 0* **EN:** Controller Enabled
This bit is set by the master when enabling the controller and is cleared by the hardware in one of the following cases:

- The communication has failed and a flag is set (FE, PE, CTO, or RTO).
- The TXEN control bit enabled the controller in transmitter mode, and the transmission has terminated (TxC is set).

With the EN bit cleared, the PS/2 bus is inhibited (clock line low).

Bit 1 **OE:** Overrun Error

This bit is set by the hardware when the FIFO buffer is full and a new packet has been received and lost. It is automatically cleared when removing elements from the FIFO with SPI transfers, when flushing the buffer by setting the BCLR bit, or when the controller is enabled in transmitter mode.

Bit 2 **FE:** Frame Error

This bit is set by the hardware if the communication has failed in one of the following ways. While receiving data from the PS/2 device, either the START or STOP bits are not recognized at the expected time. While transmitting data, the ACK bit is not recognized at the expected time.

Once the error is raised, the controller halts and inhibits the PS/2 bus (clock line low). Resuming normal operation requires enabling the controller again.

Bit 3 **PE:** Parity Error

This bit is set by the hardware if the parity check on the data received from the PS/2 device has failed.

Once the error is raised, the controller halts and inhibits the PS/2 bus (clock line low). Resuming normal operation requires enabling the controller again.

Bit 4 **CTO:** Clock Timeout

This bit is by the hardware if the falling edges of the clock generated by the PS/2 device are separated by more than 150 μ s, during either reception or transmission.

Once the error is raised, the controller halts and inhibits the PS/2 bus (clock line low). Resuming normal operation requires enabling the controller again.

Bit 5 **RTO:** Request Timeout

This bit is set by the hardware if, on transmission, the PS/2 device has failed to generate the clock within a 15 ms window after the PS/2 bus was inhibited.

Once the error is raised, the controller halts and inhibits the PS/2 bus (clock line low). Resuming normal operation requires enabling the controller again.

Bit 6 **TxC:** Transmission Complete This bit is set by the hardware when the controller was enabled in transmitter mode and the transmission has completed (with or without errors).

Bit 7 **RxV:** Receive Data Valid This bit is set by the hardware when the DATA field is valid and contains one character received from the PS/2 device. The read operation from the FIFO is automatically acknowledged.

Bits 15:8 **DATA:**

If RxV is set, one character received from the PS/2 device.

3.3 7-Segment Displays Peripheral

The display peripheral is devised to serially control a set of 7-segment displays. It processes a stream of UART characters, where each character can either instruct the peripheral to clear all displays or update the input for a single display; one character after the other, the displays can be all updated in a rotating

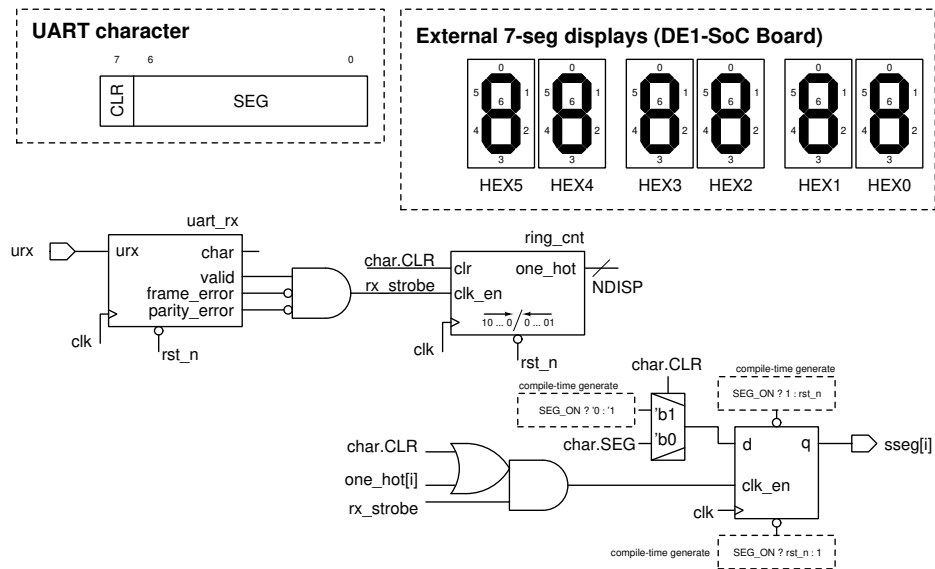


Figure 7: Display peripheral design. The module circularly displays an UART character stream on a set of 7-segment displays.

manner. The design, shown in fig. 7, makes use of a custom UART receiver module. When a transmission completes without frame or parity errors, the character appears on the output while the valid signal pulses high: the most significant bit of the character is decoded as the clear command. If the bit is set, the registers driving the display segments are forced into the off-state specified at compile-time, and the one-hot counter is cleared. Otherwise, the one-hot counter state is used to mask all clock-enable signals of the registers driving the display segments, except for one. This register is loaded with the received character data, while the one-hot counter shifts to point to the next display in the set.

3.3.1 UAR(T) Core

The core implements a flexible asynchronous serial receiver for the NRZ-L encoding. It supports several compile-time programmable parameters:

- Frame format:
 - Character length
 - Presence and type of parity (even or odd)
 - Number of STOP bits
- Baud rate, by specifying system clock, prescaler, and target clock frequency
- Serial line synchronizer depth, which determines the trade-off between data latency and MTBF due to metastable events

In addition, it features clock synchronization on all transitions, besides the UART-mandated synchronization on the START bit. Upon reception, the valid signal pulses high for one clock cycle, while the received character and error flags are latched on the outputs.

The module has been designed according to the ASM methodology, starting from the pseudocode in listing 3.2; the resulting ASM chart and data path are provided in appendix D.

```

frame[1 + NCHAR + PARITY_NONE?0:1 + NSTOP] = {1, ..., 1};
pbit = PARITY_ODD ? 1 : 0;
tim_reload_val = HALF_BIT_TIME;
tim_auto_reload_val = BIT_TIME;

while(!urx_nedge);
tim_reload();

do {
    // wait sampling instant
    while(!tim_tc) {
        // if there are transitions, resync clocks
        if (urx_edge)
            tim_reload();
    }
    // sample
    frame = {urx_sync, frame[MSB:1]};
    // update parity (start, parity xor'd too)
    if (!PARITY_NONE & (&frame[NSTOP-1:0]))
        pbit ^= urx_sync;
} while (frame[0]);

frame_error = ! &frame[MSB -: NSTOP];
parity_error = PARITY_NONE ? 0 : pbit;
rx_data = frame[1 +: NCHAR];
valid = 1;

```

Listing 3.2: Pseudocode of the asynchronous receive operation for the ASM design methodology

3.4 Bipolar Stepper Motor Drive Circuit

For educational purposes, the drive circuit does not employ fully integrated bipolar motor drivers. Instead, the L298 integrates two full-bridges to energize and reverse the phase currents. Additionally, the L6506 addresses the limitations of the simpler L/nR drive topology, in which the time constant of the inductive load is reduced by adding external resistances in series with the coils, while the desired asymptotic current is restored by increasing the drive voltage. To avoid the large power dissipation of the external resistances, the approach involves increasing the slew rate of the phase currents with an higher drive voltage, and limiting the phase currents by switching the driver. As shown in fig. 4, the phase current is sensed on the return path from the H-bridge. During the low time of the oscillator period, the drive signals pass through. However, if the phase current exceeds the limit set by the reference voltage, the drive signals are gated for the remaining part of the period. The energy stored in the inductor is dissipated as the current recirculates in the lower half of the bridge: through one snubber diode, the coil resistance, the transistor in the opposite branch, and the sense resistance. Notice that, being possible to independently set the peak phase currents, this chopping circuit supports a microstepping drive technique.

The power output stage of the L298 operates from a fixed 12 V switching power supply [8]. By means of the (R_5, R_6) divider, the enable signal pulls low as the supply voltage is about to turn on or off [11, p. 9]. Considering the input characteristics and accounting for a sensible noise margin, it must hold:

$$V_{IH} + N_{MH} \leq V_s \frac{R_6}{R_5 + R_6} \leq V_{ss}$$

$$\frac{V_s}{R_5 + R_6} \gg I_{IH}$$

To ensure that the enable signal pulls low in the widest range of the voltage supply dynamic, and assuming

a 5 % tolerance on the resistances, the nominal values in the E12 series can be chosen as:

$$R_5 = 5.6 \text{ k}\Omega \quad R_6 = 1.8 \text{ k}\Omega$$

The bipolar stepper motor 17HE15-1504S is rated for a phase current of 1.5 A, and is characterized by a coil resistance of $(2.30 \pm 0.23) \Omega$ and a coil inductance of $(4.0 \pm 0.8) \text{ mH}$. Working with a 10 % derating factor, the current driven by the L298 is limited to 1.35 A per channel: in the worst case of a total saturation voltage $V_{\text{CEsat}} = 4.9 \text{ V}$, the total power dissipated by the driver amounts to 13.23 W. Now, for the multiwatt15 package in free air, the temperature at the junction would rise above the ambient temperature by:

$$\Delta T = R_{\theta\text{ja}} P_{\text{tot}} \approx 463^\circ\text{C}$$

hence, the need to lower the thermal resistance with an heat sink. Common heat sinks for this form factor are made of 6063-T5 aluminum alloy, which has a typical thermal conductivity κ of 209 W/mK (for instance, [17]). From the mechanical data of the package, the minimum contact area S to the heat sink is 338.1 mm²; therefore, the thermal resistance across the contact area, for an average thickness Δl of 5.52 mm is:

$$R_{\theta\text{sa}} = \frac{\Delta l}{\kappa S} \approx 78.2 \times 10^{-3}^\circ\text{C/W}$$

Finally, by using a thermal compound to improve the thermal coupling between the case and the heat sink, it is reasonable to achieve:

$$R_{\theta\text{cs}} \approx 0.5^\circ\text{C/W}$$

In conclusion, the driver is expected to work at a maximum ambient temperature of:

$$T_{\text{a,max}} = T_{\text{j,max}} - (R_{\theta\text{jc}} + R_{\theta\text{cs}} + R_{\theta\text{sa}}) P_{\text{tot}} \approx 83^\circ\text{C}$$

which covers the broadly accepted range for consumer electronics from 0 °C to 70 °C.

For the current recirculation paths, the datasheet advises selecting fast-switching snubber diodes, preferably of Schottky type. A family of suitable devices is the SB2x0 [16], which are rated for a maximum average forward current of 2 A. As a first approximation, the supply voltage of the power output stage, plus one forward voltage drop of the snubber diode during clamping, bounds the maximum reverse voltage. As a consequence, the SB220 can be selected for the circuit.

The sense resistors must be sized to comply with the maximum allowable voltage at the sensing pins, equal to 2 V. Considering a $(680.0 \pm 6.8) \text{ m}\Omega$ resistor rated for 2 W [2], the nominal voltage drop for the chosen maximum phase current of 1.35 A is 918 mV. In this condition, the resistor dissipates 1.24 W: computing the thermal resistance from the power derating curve in the datasheet:

$$R_\theta = \frac{155^\circ\text{C} - 70^\circ\text{C}}{2 \text{ W}} = 42.5^\circ\text{C/W}$$

it is possible to evaluate the effect of self-heating. In the worst case of the 0 °C to 70 °C temperature range, the resistor heats to:

$$T_{\text{r,max}} = T_{\text{a,max}} + R_\theta P \approx 123^\circ\text{C}$$

Consequently, from the temperature coefficient $\alpha = 300 \text{ ppm}/^\circ\text{C}$:

$$\Delta R_{\text{sh}} = R_0 \alpha (T_{\text{r,max}} - T_{\text{r}0}) \approx 19.9 \text{ m}\Omega$$

which shows that the self-heating effect dominates the uncertainty. Accordingly:

$$R_7 = R_8 = (680 \pm 27) \text{ m}\Omega$$

The sensed voltage is used in the L6506 to implement the constant current drive. In the worst case, the phase current reaches the maximum value of 1.35 A and the sensed voltage at the comparator input

is 882 mV. As a consequence, the major constraints are:

$$V_{ss} \frac{R_4}{R_3 + R_4} \leq 882 \text{ mV}$$

$$\frac{V_{ss}}{R_3 + R_4} \gg I_{IB}$$

Having already derated the maximum phase current, the constraint can be evaluated for the nominal resistance values. An optimal selection is:

$$R_3 = 22 \text{ k}\Omega \qquad R_4 = 4.7 \text{ k}\Omega$$

In fact, with 1 % uncertainties, the worst case voltage reference would be 895 mV, which corresponds to a maximum phase current of 1.37 A. In addition, the phase current may exceptionally increase above the chosen peak value during the active time of the oscillator waveform, as the driving inputs cannot be masked. With the same selection of (R_2, C_4) as in the application circuit [12, p. 5], the nominal frequency of the oscillator is 16.9 kHz and the nominal active time lasts 2.57 μ s.

4 Software Design

Prototyping with STM32 microcontrollers is facilitated by the STM32Cube project, which aims to simplify development by providing a comprehensive embedded software platform for each STM32 series. This software kit is complemented by the STM32CubeMX tool [14], which enables the graphical configuration of the peripherals and the clock tree of the microcontroller, as well as the automatic generation of the corresponding initialization code in the C language.

Given the opportunity to gain a deeper understanding of the tools involved in the cross-platform build process, but also to achieve greater control and flexibility over the firmware architecture, the software project is structured from scratch using CMake, without relying on STM32CubeMX. The details of the buildsystem are discussed in section 4.1.

To benefit from the object-oriented and generic programming paradigms, the firmware is developed in C++. The software architecture follows a bare metal yet layered design, with a clear separation between virtualized hardware resources and application logic. Depending on the resource, the virtualization is implemented in one of two ways:

- As an object, with public methods directly invoked by the application.
- As a file, manipulated through I/O library functions or system calls.

This latter approach is typical of UNIX-like systems, where special files are interfaces to device drivers, and the Virtual File System (VFS) makes the association between the low-level operations invoked internally by file-related system calls and their implementation in the drivers. Accordingly, the firmware includes a custom implementation of a minimal virtual file system, which is presented in section 4.2. The device drivers and the simpler objects abstracting hardware resources are covered in section 4.3.

The firmware also relies on the following generic abstraction layers:

- The CMSIS Core(M) component, which provides a standardized API for all Arm Cortex-M processor cores. It consists of two complementary parts:
 - The Core(M) Standard, delivered as a set of vendor-independent C header files directly by Arm. It declares the core exception vectors and defines the functions for accessing core and core peripheral registers, special CPU instructions, and the SIMD instructions.
 - The Core(M) device, developed by the silicon vendor, in this case, STMicroelectronics. It declares interrupt vectors and defines the functions for startup and system configuration.
- Note that the startup assembly code provided by STMicroelectronics is not used in this project; it is replaced by a C++ version tailored to a custom linker script.

```
MEMORY
{
    FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 16K*4 + 64K + 128K*2
    NVS (w!x) : ORIGIN = 0x08060000, LENGTH = 128K
    RAM (w!x) : ORIGIN = 0x20000000, LENGTH = 96K
}
```

Listing 4.1: Location and size of the memory blocks available in the STM32F401xE target, described in the linker command language.

- The STM32Cube Low-Layer (LL) drivers [10], which offer a lightweight and portable C abstraction to operate directly on memory-mapped peripheral registers. Whenever convenient, this abstraction is extended in the project with utility functions at the same abstraction level, organized by peripheral in dedicated namespaces (e.g., `dma::`, `exti::`, `flash::`).
- The newlib nano and GNU standard C++ libraries. Newlib nano is a size-optimized variant of newlib, an ANSI C standard library implementation intended for use on embedded systems; it is prebuilt and distributed with the Arm GNU Toolchain.

4.1 The Buildsystem

CMake acts as a meta-build tool that translates an abstract buildsystem written in the CMake language into configuration files for the native build tool of the host platform. To support cross-compilation, CMake is told about the target platform via a toolchain file. For this project, this is found split into:

arm-none-eabi.cmake It provides basic configuration settings for an AArch32 bare metal target. It establishes the cross-compilation environment by automatically locating the Arm GNU Toolchain installation in the host platform and defines compiler and linker options optimized for embedded development. In addition, it introduces interface targets corresponding to different C/C++ runtime variants:

- **ARM::FreeStanding** for standalone applications which do not use the standard startup files and libraries.
- **ARM::NoSys** for bare metal applications to weakly link system calls with stubs that implement graceful failure.
- **ARM::Newlib** for linking with newlib C and GNU C++ standard libraries.
- **ARM::Nano**, **ARM::Nano::FloatPrint**, **ARM::Nano::FloatScan** to refine the size-optimizations of the newlib C standard library being linked.

armv7em-hard-fpv4sp.cmake It specializes the generic configuration of AArch32 bare metal targets for the Armv7-M architecture profile, also enabling the Digital Signal Processing (DSP) and single-precision Floating Point (FP) extensions. The settings are encapsulated in the interface target **ARM::V7EM-HARD-FPV4SP**.

At the top level, the **CMakeLists.txt** configures project-specific settings. It loads the toolchain file, specifies the linker script, and collects the project sources and the external libraries to link with. The linker script is a key component of the build process, as it describes how the sections in the input object files are mapped into the output executable file, and what its memory layout is. The STM32F401xE microcontroller is equipped with 96 KiB of static RAM and 512 KiB of NOR flash memory, which is partitioned in eight erasable sectors as follows: four 16 KiB sectors, one 64 KiB sector, and three 128 KiB sectors. To allow for the non-volatile retention of the movement pattern, the load script reserves a dedicated memory region matching the last 128 KiB flash sector, as shown in listing 4.1.

The linker script is also tightly connected to the startup file. On system reset, the vector table is expected at address `0x00000000`, with the stack pointer reset value in its first entry. The linker ensures that this placement is satisfied and provides the stack pointer reset value, and several other addresses referenced by the code, by means of symbol definitions. Specifically, the linker script manages all the space left in the SRAM region as a shared ascending-heap/descending-stack: the start address of the region initializes the variable used to track the top of the heap in the `_sbrk()` system call, which handles heap allocation; whereas the end address of the region is put in the first vector table entry. Other symbols are referenced while setting up the runtime context: the `Reset_Handler()` initializes the `data` segment in the SRAM by copying from the flash, according to the addresses provided by the linker; similarly, the C/C++ initialization code makes use of these addresses to clear the `bss` segment, and to locate the initialization and de-initialization routines.

4.2 Virtual File System

In general, hiding complexity behind simple and uniform interfaces is highly desirable. In UNIX-like systems, resources are abstracted as files, allowing user applications to interact with them using the same system calls as for regular files (e.g., `open()`, `read()`, `write()`, etc.). This framework relies on the VFS, which defines a common set of low-level operations supported by all file types. These operations are invoked internally by system calls, and dispatched to their implementation depending on the file type. In particular, for special files representing character devices, the implementation of the file operations is provided by the corresponding device driver, a C kernel module that exposes the hardware as a byte stream, hiding its actual behavior.

Given the number of peripherals behaving as character devices in the project, the firmware implements a lightweight abstraction layer to support file-related system calls. This is achieved through the `IFile` and `FileManager` classes.

class `IFile` It is the abstract base class that defines the interface to operate on files. A device driver is thus a derived class that overrides the supported operations; by default, all operations are implemented in `IFile` to return the error code `-ENOSYS`.

class `FileManager` It implements relevant file system services using the data structures illustrated in fig. 8. Special files are represented as `Node` objects, namely links between a file name and a driver object. The full list of special files is specified at compile time via the file manager constructor, which forwards it to initialize the node table member, as shown in fig. 8a. The open file table tracks open files at run time. As depicted in fig. 8b, each entry of this data structure holds a pointer to a node along with state data; file descriptors index the open file table and are considered free when the corresponding entries hold no valid node pointer.

The file system services roughly follow POSIX semantics:

- `open()` performs name resolution through the node table to locate the associated node object. It then allocates the lowest, non-reserved file descriptor shown free in the open file table and initializes the corresponding open file table entry with the address of the node, as well as the access mode and the flags passed as arguments. Finally, the driver implementation of the operation is invoked via the node object.
- `close()` reverses the actions of `open()`, provided that the file descriptor identifies an open file. The driver implementation of the operation is invoked via the node pointer, which is retrieved in the open file table entry indexed by the file descriptor. Afterward, the file descriptor is released by invalidating the node pointer.
- `lseek()`, `read()`, and `write()` follow a similar pattern: if the file descriptor refers to an open file, the corresponding driver operation is invoked via the node pointer from the open file table. The only exception is that `read()` and `write()` check that the access mode specified when opening the file is compatible with the requested operation.

Node _nodes [4]		OFileEntry _files [7]			
	const char * name	IFile & cdev	const Node * node	OFile & ofile	
			mode	flags	pos
[0]	"st_link_uart_tx"	St_Link_Uart_Tx()	[0]	nullptr	
[1]	"sseg_display"	SSeg_Display()	[1]	_nodes	FWRITE 0x00000000 0x00000000
[2]	"ltc_2308"	Ltc_2308()	[2]	_nodes	FWRITE 0x00000000 0x00000000
[3]	"kbd"	Kbd()	[3]	_nodes + 2	FREAD O_BINARY 0x00000000
			[4]	_nodes + 3	FREAD 0x00000000 0x00000000
			[5]	_nodes + 1	FWRITE O_TRUNC 0x00000000
			[6]	nullptr	

(a) Compile-time table of nodes: each entry links the name of the file representing a character device to its corresponding driver.

(b) Run-time table of open files, indexed by file descriptors: an open file entry holds a pointer to the associated node and some state data.

Figure 8: FileManager data members

- `select()` enables monitoring multiple file descriptors for readiness to perform an I/O operation without blocking. Internally, for each monitored file descriptor referring to an open file, the ready state is questioned with the driver `poll()` operation, invoked via the node pointer from the open file table.

4.3 Virtualized Resources

The firmware follows two approaches when dealing with hardware resources:

- Behavior close to microcontroller peripherals capabilities is abstracted through C++ classes that directly expose a convenient interface of public member functions to the application code. This is discussed in section 4.3.1.
- Behavior of complex hardware devices that fit a byte-stream access model is abstracted through device drivers, relying on the virtual file system layer. Note that, in all cases, device parameters are not manipulated through `ioctl()` calls, but directly with the methods exposed by the drivers. The device drivers are presented in section 4.3.2.

4.3.1 Object Abstractions

`class HwAlarm` It transforms a free-running general-purpose timer into an interrupt-based alarm scheduler and a delay generator.

The class is parameterized with the timer base address, allowing to refine its implementation at compile-time, based on the features of the chosen timer: namely, the width of the counter, and the number of independent capture/compare channels. The channels are abstracted as an array of `Alarm` objects, which hold the parameters representing a runnable alarm on the corresponding channel. The representation consists of: a pointer to a callback functor, which differentiates between running alarms when valid and free channels; the remaining number of repetitions, and the periodicity converted to counter ticks. Note that the implementation relies on the `ICallback` abstract base class to provide an interface to non-owning functor wrappers of function and member function pointers.

The setup is performed by the `init()` method. It computes the prescaler to achieve the desired counter clock resolution, then it enables the interrupt request in the Nested Vector Interrupt Controller (NVIC), and the timer in free-running mode.

When the `setAlarm()` method is invoked to set up a new alarm, the caller specifies how many times it will fire, the periodicity, and the callback functor. Provided that there are free channels and that

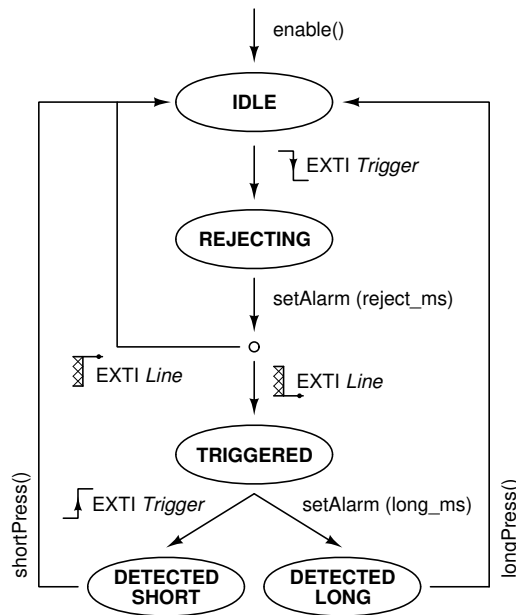


Figure 9: PushButton internal FSM for an active low push button

the periodicity is representable in the counter width, a free `Alarm` entry is initialized with the alarm parameters. The target compare value is computed starting from the current counter tick and is used to configure the corresponding capture/compare channel for interrupt generation on match events. When the interrupt fires and the handler starts executing, the triggered alarms are identified, their representation is updated, and lastly their callback functions are invoked for execution.

The `delay()` method provides traditional blocking delays, capable of exceeding the counter reload period. This is implemented through a polling loop that accumulates the elapsed time since function invocation, and compares it against the target delay.

class `PushButton` It implements an interrupt-based push button handler, with debouncing logic and short/long press detection.

The class behavior follows the FSM illustrated in fig. 9, where the key state transitions are driven either by external edge-sensitive interrupts or by alarm timeouts. The application is supposed to query the button state with the `shortPress()` and `longPress()` methods, which are non-blocking and return `true` when the corresponding gesture is detected.

class `BStepper` It generates the waveforms required to drive the windings terminals of a bipolar stepper motor, supporting both full-step and half-step modes. This is achieved with minimal CPU overhead by delegating the transfer of precomputed bit patterns onto the GPIO port via a DMA controller, synchronized by the advanced hardware timer.

Provided that the windings terminals belong to the same GPIO port, a bit pattern can be fully represented as a Bit Set/Reset Register (BSRR) mask on 32 bit. Given the motor phase states shown in fig. 10, the `Translator` class generates four arrays of bit patterns corresponding to clockwise (CW) and counter-clockwise (CCW) driving sequences in half-step and full-step modes. While the half-step and full-step sequences consist of eight and four states respectively, the corresponding bit pattern arrays are twice as long. This redundant data simplifies the DMA transfer, which occurs from a memory region handled as a circular buffer targeting the GPIO BSRR. This is because the `Translator` class tracks

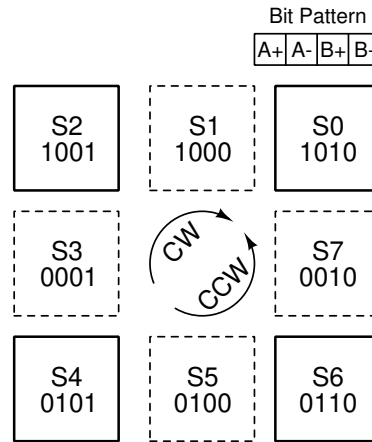


Figure 10: Bipolar motor phase states for CW and CCW rotations; the dashed states are skipped in full-step mode.

the motor phase state as an index within the half-step sequence, ranging from 0 to 7. Without this redundancy, locating the start address of the circular buffer by simply indexing the masks arrays with a state number for half-step movements, and half that value for full-step movements, could result in out-of-bounds accesses.

The initializations of GPIO pins, DMA controller, advanced timer, and NVIC are performed by the `init()` method. The advanced timer is configured in up-counting mode, with overflows as the only source of Update Events (UEVs).

Motor movements are initiated with the `rotate()` method and require no further intervention from the CPU, except when the number of steps minus one does not fit in the Repetition Counter Register (RCR) of the advanced timer. As a preliminary step, the `calcTimeBase()` method determines the values for the prescaler and auto-reload registers, such that the reload period matches the step duration, and the counter resolution is numerically lowest. That is

$$(\text{PSC} + 1)(\text{ARR} + 1) = \lfloor f_{\text{PSC}} T_s \rfloor$$

where the step duration T_s is expressed as

$$T_s = \frac{60 \text{ s/min}}{N \omega_{\text{rpm}}}$$

with N being the motor resolution in steps per revolution, and ω_{rpm} the rotational frequency in revolutions per minute.

Considering the simpler case when the number of steps minus one fits the RCR, the core logic is the following. The RCR is forcefully loaded with a software UEV and the counter is configured in one pulse mode. The `Translator::advance()` method updates the internal motor phase state to reflect the one at the end of the movement. In addition, it locates the start address of the circular buffer with the precomputed sequence of BSRR masks, which is used to reconfigure the DMA stream source. The last configuration step determines when to generate the DMA requests: one capture/compare channel of the advanced timer is configured to fire the requests on match events, with a compare value equal to the auto-reload register. Finally, the counter is enabled: as an example of the resulting execution diagram, see fig. 11.

If this approach is not feasible, the RCR is set to its maximum value and the hardware-counted repetitions are supplemented with software-counted ones (namely, RCR reload events). Except for the

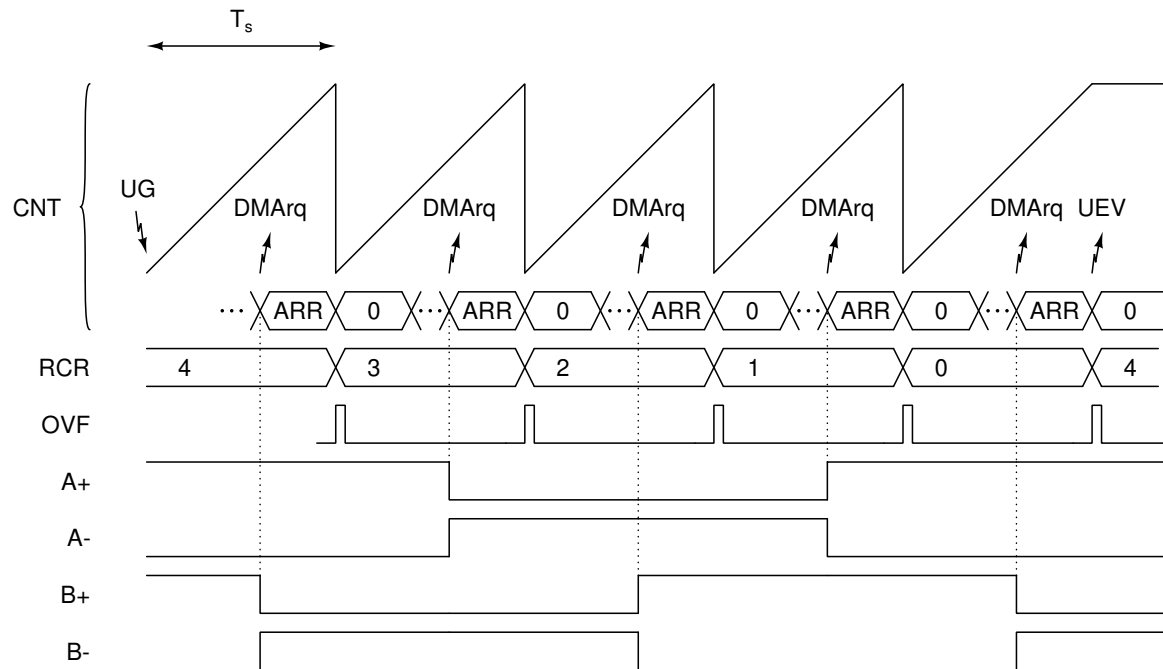


Figure 11: BStepper::rotate() execution diagram for the simpler case when the number of steps fits the RCR. Starting from the motor phase state S0, the motor takes five full steps in CCW direction.

corner case where exactly one software repetition must be counted, this is handled by starting the advance timer with the one pulse mode disabled. It is configured such that, upon counter overflow and RCR reload, the UEV generates an interrupt. The handler then updates the software repetitions count and, in the last software-counted cycle, either preloads the RCR or directly switches the counter to one pulse mode. In the former case, at the end of the software repetitions count, the counter is switched to one pulse mode as well.

`class MotionPattern` It implements an `std::vector`-like container for `MotionSegment` objects, that is allocated in a flash sector with wear-levelling and cached in SRAM over a statically allocated array.

The class is parameterized by the maximum number of elements it can contain. While the elements stored in the SRAM array are plain `MotionSegment` objects, the corresponding flash entries are instances of a larger type, `FlashChunkEntry`, which packs an additional byte attribute that indicates whether the entry is valid (`WRITTEN = 0xAA`) or not (`ERASED = 0xFF`).

As it happens in the application, suppose that the class allocates the non-volatile container in the 128 KiB sector starting at address 0x08060000 and that the `FlashChunkEntry` is represented on 8 B. If the maximum number of elements is taken to be four, then the sector can accomodate

$$N = \frac{128 \text{ KiB}}{\text{sizeof}(\text{FlashChunkEntry}[4])} = 4096$$

distinct containers, hence the logical partitioning of the sector into corresponding N chunks. The wearlevelling mechanism leverages this excess space to minimize the number of sector erasures. Clearing the non-volatile container does not immediately cause a sector erase; instead, it invalidates the current chunk and enables the next one available. This process continues until all chunks have been invalidated, only then the sector is erased. To illustrate, consider fig. 12:

	MotionSegment				
	milli_rpm	steps	dir	attr	
0x08060000	FFFFFFFF	FFFF	FF	(FF)	ERASED (*_fchunk)[_n]
0x08060008	FFFFFFFF	FFFF	FF	FF	
0x08060010	FFFFFFFF	FFFF	FF	FF	
0x08060018	FFFFFFFF	FFFF	FF	FF	
0x08060020	FFFFFFFF	FFFF	FF	FF	
0x08060028	FFFFFFFF	FFFF	FF	FF	
0x08060030	FFFFFFFF	FFFF	FF	FF	
0x08060038	FFFFFFFF	FFFF	FF	FF	
	:	:	:	:	
0x0807FFE0	FFFFFFFF	FFFF	FF	FF	
0x0807FFE8	FFFFFFFF	FFFF	FF	FF	
0x0807FFF0	FFFFFFFF	FFFF	FF	FF	
0x0807FFF8	FFFFFFFF	FFFF	FF	FF	

(a) Fully erased sector, empty container

	MotionSegment				
	milli_rpm	steps	dir	attr	
0x08060000	000249F0	0032	00	(AA)	WRITTEN (*_fchunk)[0]
0x08060008	FFF186A0	0019	10	AA	
0x08060010	FFF55730	0096	00	AA	
0x08060018	FFFFFFFF	FFFF	FF	FF	
0x08060020	FFFFFFFF	FFFF	FF	FF	
0x08060028	FFFFFFFF	FFFF	FF	FF	
0x08060030	FFFFFFFF	FFFF	FF	FF	
0x08060038	FFFFFFFF	FFFF	FF	FF	
	:	:	:	:	
0x0807FFE0	FFFFFFFF	FFFF	FF	FF	
0x0807FFE8	FFFFFFFF	FFFF	FF	FF	
0x0807FFF0	FFFFFFFF	FFFF	FF	FF	
0x0807FFF8	FFFFFFFF	FFFF	FF	FF	

(b) First chunk in use, container size is three

	MotionSegment				
	milli_rpm	steps	dir	attr	
0x08060000	FFF249F0	0032	00	(00)	DIRTY
0x08060008	FFF186A0	0019	10	AA	
0x08060010	FFF55730	0096	00	AA	
0x08060018	FFFFFFFF	FFFF	FF	FF	
0x08060020	FFFFFFFF	FFFF	FF	(FF)	ERASED (*_fchunk)[_n]
0x08060028	FFFFFFFF	FFFF	FF	FF	
0x08060030	FFFFFFFF	FFFF	FF	FF	
0x08060038	FFFFFFFF	FFFF	FF	FF	
	:	:	:	:	
0x0807FFE0	FFFFFFFF	FFFF	FF	FF	
0x0807FFE8	FFFFFFFF	FFFF	FF	FF	
0x0807FFF0	FFFFFFFF	FFFF	FF	FF	
0x0807FFF8	FFFFFFFF	FFFF	FF	FF	

(c) First chunk marked dirty, second chunk in use, container is empty

	MotionSegment				
	milli_rpm	steps	dir	attr	
0x08060000	FFF249F0	0032	00	(00)	DIRTY
0x08060008	FFF186A0	0019	10	AA	
0x08060010	FFF55730	0096	00	AA	
0x08060018	FFFFFFFF	FFFF	FF	FF	
	:	:	:	:	DIRTY
0x0807FFC0	FFF249F0	0032	00	(00)	
0x0807FFC8	FFF186A0	0019	10	AA	
0x0807FFD0	FFF55730	0096	00	AA	
0x0807FFD8	FFFFFFFF	FFFF	FF	FF	WRITTEN
0x0807FFE0	FFF249F0	0032	00	(AA)	
0x0807FFE8	FFF186A0	0019	10	AA	
0x0807FFF0	FFF55730	0096	00	AA	
0x0807FFF8	FFFFFFFF	FFFF	FF	FF	(*_fchunk)[_n]

(d) All chunks marked dirty but the last one in use. Clearing the container triggers a sector erase.

Figure 12: MotionPatter logic for managing the persistence of the MotionSegment[4] array.

- (a) Starting in the erased state, the container is located in the first chunk and has a zero size; entry zero is pointed to be the written next.
- (b) After three write operations, the container is still located in the first chunk. The fourth entry is pointed to be written next.
- (c) The container is cleared: the chunk is invalidated by marking the first entry as **DIRTY** = 0x00. The container is now located in the second chunk and entry zero is pointed to be written next.
- (d) After $N - 1$ clear operations, all chunks are marked dirty but the last one. A further clear operations triggers a sector erase.

class SpiMaster It provides a simple interface to communicate with multiple SPI slave devices, internally managing the select-signal generation via GPIO.

During initialization, the GPIO pins assigned to SCLK, MOSI, and MISO are configured in the alternate function corresponding to SPI. Since these pins are left floating when the controller is disabled, weak pull-downs are enabled. The SPI controller is configured as well, preparing for master-mode, full-duplex operation with software-managed slave selection.

The communication targets are abstracted as an array of `Slave` objects, each holding the protocol parameters and the GPIO pin to be used for slave selection. Targets are dynamically registered via the class interface: provided that there are free entries in the array of targets, the `addSlave()` method computes the lowest prescaler that satisfies the requested SCLK maximum frequency, and stores the SSN pin information and protocol parameters. It returns the index of the newly registered slave, which serves as its identifier. Although the STM32F401xE SPI peripheral supports only 8 bit and 16 bit transfers, the class accommodates arbitrary frame lengths up to 16 bit by adjusting the alignment of the transmitted and received data.

Full-duplex transfers are blocking and are performed with the `txrx()` method specifying the identifier of the target slave. If the retrieved slave-dependent configuration differs from the last active one, the SPI controller is reconfigured accordingly. Note that, in this case, the configuration of weak pull-up/down resistors on SCLK is changed to match the clock polarity required by the slave. Then the method asserts the SSN line, generates a pre-transfer delay via the `HwAlarm` abstraction, executes the transfer, and de-asserts the SSN line after a post-transfer delay.

4.3.2 Device Drivers

class UartTx It is a lightweight driver for the USART peripheral, implementing DMA-based UART transmission in blocking or non-blocking fashion.

The driver object is configured directly through its public methods, which includes assigning a GPIO pin, a DMA stream, and protocol parameters like frame settings and baud rate. Note that the character payload has a fixed length of 8 bit.

The initialization of the underlying peripherals is performed when opening the device file, which supports only the write mode. The DMA stream is prepared for a memory-to-peripheral direct byte transfer: the source address points to the start of the characters buffer and is auto-incremented by the DMA controller; the destination is fixed and corresponds to the data register of the USART peripheral. On the USART side, following the configuration of the protocol parameters, the DMA transmission mode is enabled.

The implementation of the `write()` operation is trivial. If there is an ongoing transfer, the open-file flags are queried to determine if it is acceptable to block while waiting for it to complete. Upon completion, the user data buffer is copied locally, the DMA stream is enabled, and the transmission continues with no further intervention from the CPU.

class SSegDisplay It is the driver for the display peripheral presented in section 3.3, implemented by extending the generic capabilities of the `UartTx` base class with functionalities such as character re-encoding, text buffering, and interrupt-based text scrolling.

In addition to the configuration of the base driver object, the method `setDisplay()` allows to configure the number of 7-segment displays composing the peripheral screen, the minimum number of times a string that does not fit on the screen should be scrolled, and the scrolling speed. The alphabet supported by the peripheral is built at compile time via the template function `fmtChar()` and made available for the re-encoding of an ASCII stream through the private method `encode()`.

Opening the device file performs the same initialization described for the base class, with the only peculiarity that truncation forces the driver to transmit a `CLEAR` command to the peripheral.

At the core of the `write()` method there is the agreement that the newline character indicates the termination of a string to be transmitted. It is expected that the text may exceed the screen length, hence the implementation of an interrupt-based mechanism to handle the scrolling of the displayed text. This is achieved in collaboration with the `HwAlarm` class, following the FSM illustrated in fig. 13, where the state transitions are driven either by application logic or by alarm timeouts.

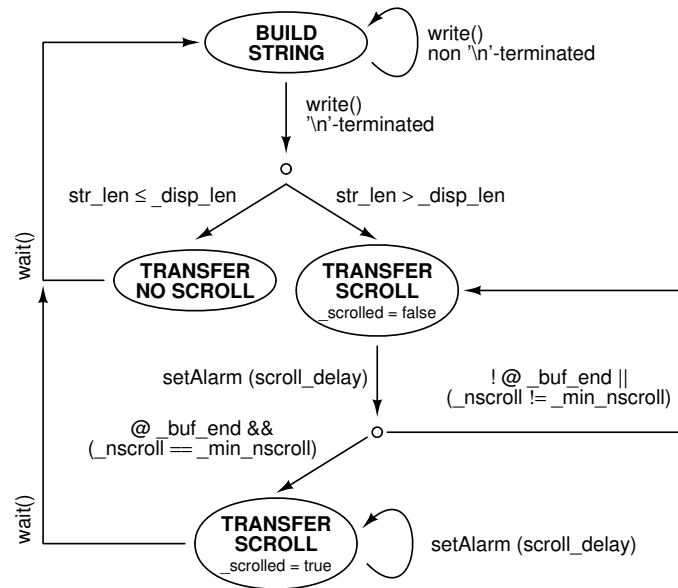


Figure 13: SSegDisplay internal FSM for text scrolling

It is worth noting that when the driver is scrolling a string, the alarm callback is executed by the `HwAlarm` object within an interrupt handler. Since the UART transfer is synchronized via DMA, the amount of processing required in CPU Handler Mode is minimal. This holds true except when the last contiguous substring has been transmitted to the peripheral, in which case a `std::rotate()` operation is performed to manipulate the underlying character buffer and produce the effect of continuous scrolling.

class LTC2308 It is a full driver for the ADC peripheral discussed in section 3.1, implemented according to the timing diagrams and the applications information found in the datasheet [7].

The class collaborates with the `SpiMaster`, in charge of the low-level details of SPI communication, and with the `HwAlarm`, that generates the delays required to satisfy timing constraints. The configuration of the driver object includes specifying the protocol-related parameters required for registering a slave with the `SpiMaster`: these are the Conversion Start (CONVST) pin, which triggers both the start of the analog-to-digital conversion process and the start of the SPI transfer, and the maximum `clk` frequency. In addition, the `setOptions()` method internally generates the programming word that configures the actual ADC peripheral, based on the input range type (unipolar or bipolar), the input stage type (single-ended or differential), the multiplexer channels, and the power management mode.

Opening the device file has the effect of registering the slave target with the `SpiMaster` and transmitting the programming word to initialize the ADC peripheral with the selected options. The `read()` method implements the logic for acquiring a stream of 12 bit samples from the ADC peripheral and transforming it into a binary stream of 16 bit signed or unsigned integers, according to the selected options.

class Keyboard It is a full driver for PS/2-compatible keyboards using scan code set 2, interfaced through the custom PS/2-to-SPI peripheral described in section 3.2.1.

Communication with the SPI slave and timing synchronization are implemented similarly to the LTC2308 driver, relying on the `SpiMaster` and `HwAlarm` abstractions. The `SpiMaster::txrx()` function is further abstracted with private helper methods, tailored to the behavior of the peripheral device:

`_pollTxRx()` Sends a command to the peripheral and polls for an acknowledgment within a bounded number of SPI cycles. This approach is necessary because: first, command execution requires time;

second, the full-duplex nature of the transfer implies that data received from the slave cannot represent the response to the command being sent concurrently.

`_trySet()` Builds upon `_pollTxRx()` to implement the transmission logic of the PS/2 *Set/Reset Status Indicator* command to the keyboard, which consists of a command byte followed by an argument byte. First, an SPI command programs the PS/2 controller in transmit mode to send `0xED`; the controller is then reconfigured to receive the keyboard's acknowledgment. The process is repeated to transmit the argument byte.

Opening the device file registers the slave with the `Spimaster` and issues a reset sequence: inside the peripheral, the PS/2 controller is disabled and re-enabled, and the FIFO buffer is flushed. Then, `_trySet()` is invoked to turn off both the Num Lock and Caps Lock indicators.

The `poll()` operation must determine whether a subsequent `read()` call would return a character, ideally without blocking. In practice, one blocking SPI transfer is performed via the helper method `_tryRead()` to check if a scan code is available for processing. Note, however, that the presence of a scan code does not guarantee that a character is ready to be returned. This is where the parsing machine comes into play. It consists of:

`class ScanCodeParser` Implements the logic for parsing the scan codes belonging to set 2 with a FSM and a lookup table. Processing any scan code generates a `ParsedKey` object, which is differently interpreted based on its `Action` field. Some scan codes merely advance the FSM without yielding actionable information; these are represented with action `NONE`. Conversely, the `MAKE` and `BREAK` actions are paired with the layout-independent representation of the key for which the make or break code has been received; this is represented as an enumerator of `UniKey` type.

`_map()` Translates layout-independent `UniKey` keys into ASCII characters using a layout-specific lookup table, considering the current state of modifier keys (Left/Right Shift, Alt Gr, Caps Lock, and Num Lock).

Note that while the current implementation provides the Italian lookup table, the mapping can be generalized through template parameters and specialization.

`_step()` Overall, it implements the logic that translates scan codes into ASCII characters, returning `'\0'` when the scan code does not yield such information.

In details, the scan code is first parsed into a `ParsedKey` object. If the action is `MAKE` or `BREAK`, the associated key is processed as follows:

- If it corresponds to a modifier key, the internal state is updated accordingly. For Caps Lock and Num Lock, a small FSM determines whether the corresponding keyboard indicators should be updated, in which case `_trySet()` is invoked.
- Otherwise, only key press (corresponding to the `MAKE` action) requires processing. In this case, the `_map()` helper function is invoked.

Tying things together: if `_tryRead()` yields a scan code, it is processed by `_step()`, which may produce a non-zero character, which updates the internal `_peek` member. The `poll()` operation returns accordingly: a `_peek` value different from `'\0'` indicates read readiness.

The `read()` function complements this behavior. If `poll()` was previously called, the `_peek` value may already contain a character. If not, and the open file flags allow blocking, a character is fetched using the helper method `_getc()`, which repeatedly calls `_tryRead()` and `_step()` until a valid character becomes available.

4.4 Main Application

The application integrates all the described abstractions to implement the theory of operation presented in section 2.

Device drivers and objects requiring exception handling follow the Construct On First Use (COFU) idiom, which ensures proper initialization order before the `main()` is run. System initialization proceeds through several phases, notably:

- `systemClockConfig()` establishes a 64 MHz system clock using the internal RC oscillator and the PLL. The number of wait states for flash memory access is adjusted accordingly.
- The logging system is setup using the low-level driver `UartTx` over USART2, which is accessible through the on-board ST-LINK debugger/programmer. The standard input and standard error streams are redirected to the corresponding device file.
- The `HwAlarm` object is constructed over the general-purpose 16 bit timer TIM9, which features two independent capture/compare channels. The time base is configured with a counter clock frequency of 64 kHz.
- The ADC device file is opened for buffered binary reading. The buffer size is configured to 16 bit, so that each `read()` operation from the driver returns a complete sample.
- The display device file is opened for line-buffered writing: due to truncation, the screen is cleared. As the newline character both flushes the buffer and triggers a transmission to the display, the number of `write()` invocations is minimized.

If all initializations are successful, the system performs a sign of life sequence by printing *run...* on the peripheral display and rotating the motor shaft one full turn counterclockwise and back clockwise at 25 rpm.

The main loop is designed following a polling scheme:

1. Detection of `Push_Button().longPress()` is immediately handled to clear the movement pattern.
2. Detection of `Push_Button().shortPress()` with a non-empty movement pattern is immediately handled to initiate movement pattern execution. Similarly, the detection of `Push_Button().shortPress()` while executing the movement pattern immediately aborts movement pattern execution.
3. Detection of available input from the keyboard through the `select()` system call is immediately handled to attempt acquisition of a new motion segment.

Note that keyboard input validation is not implemented using the C++ standard regex library, which parses and compiles the regular expression pattern at runtime. Resorting to the *Compile Time Regular Expressions* library significantly reduced the program image size while also improving execution speed.

A PS/2 Controller

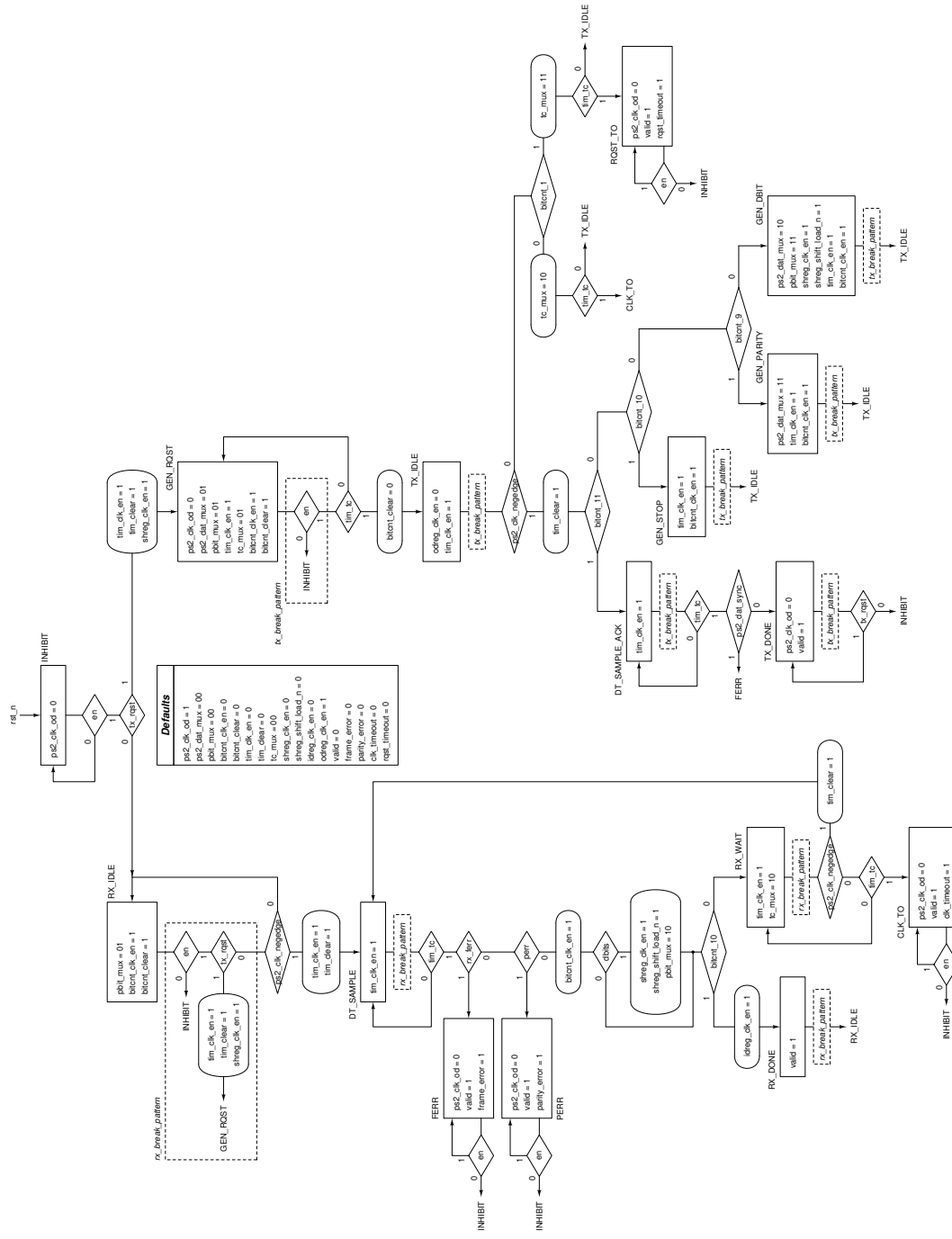


Figure 14: PS/2 controller detailed ASM chart

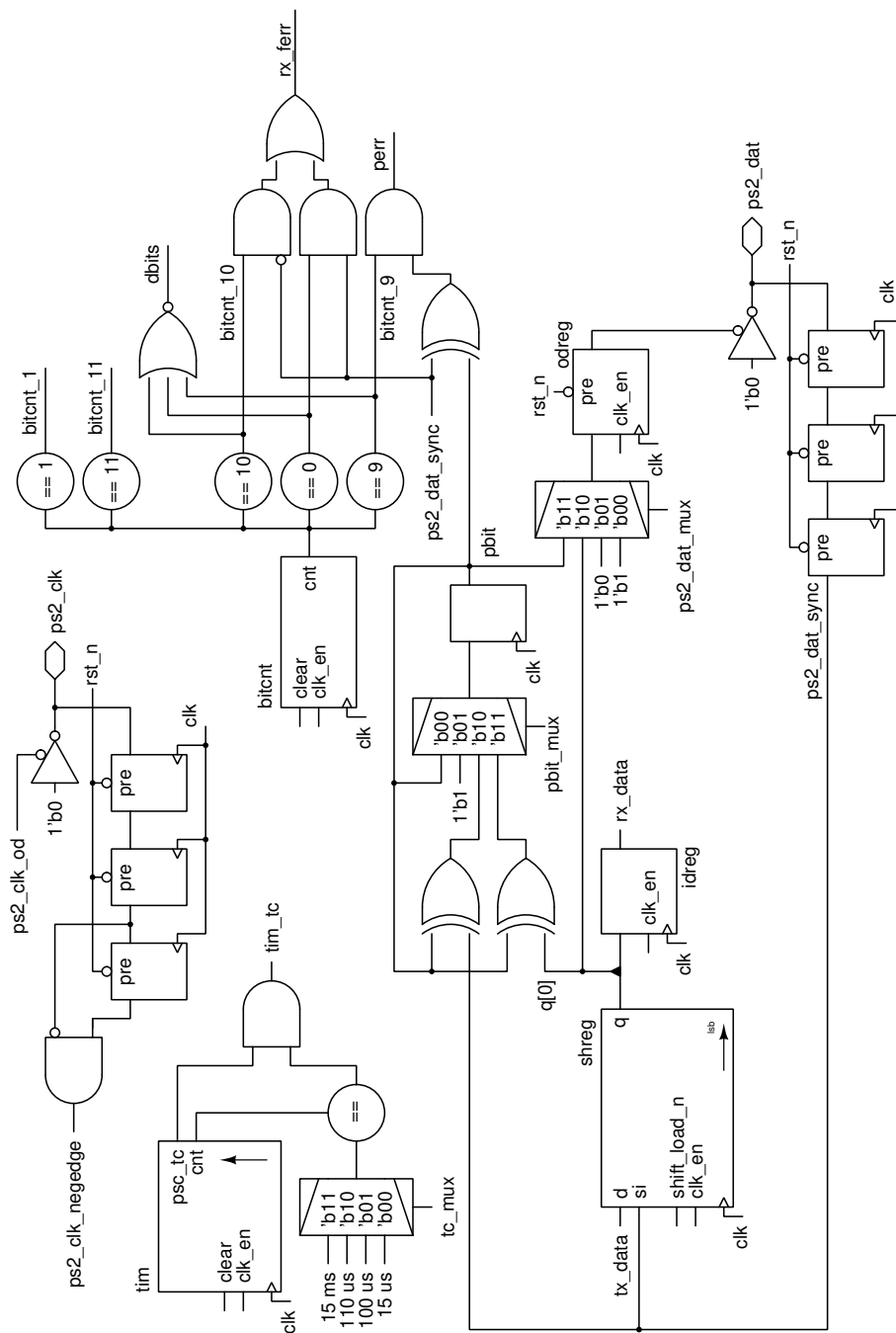


Figure 15: PS/2 controller data path

B Show-Ahead Single-Clock FIFO

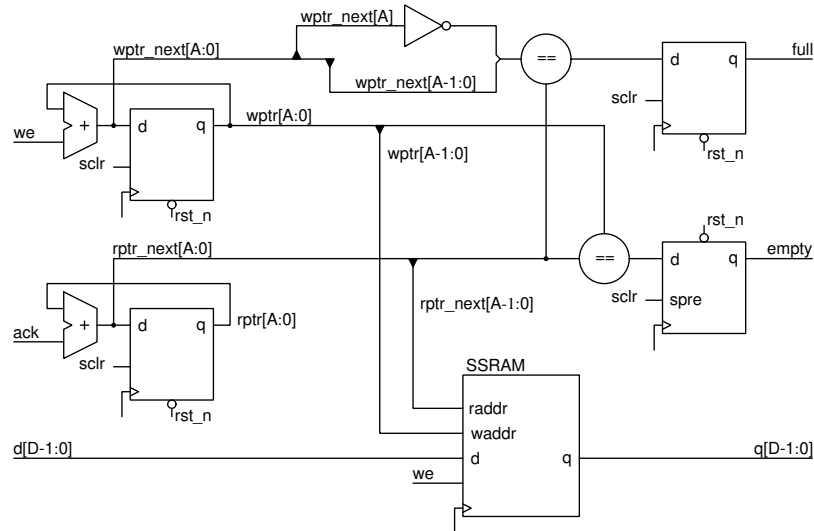


Figure 16: Show-ahead single-clock FIFO design

C SPI Slave Controller

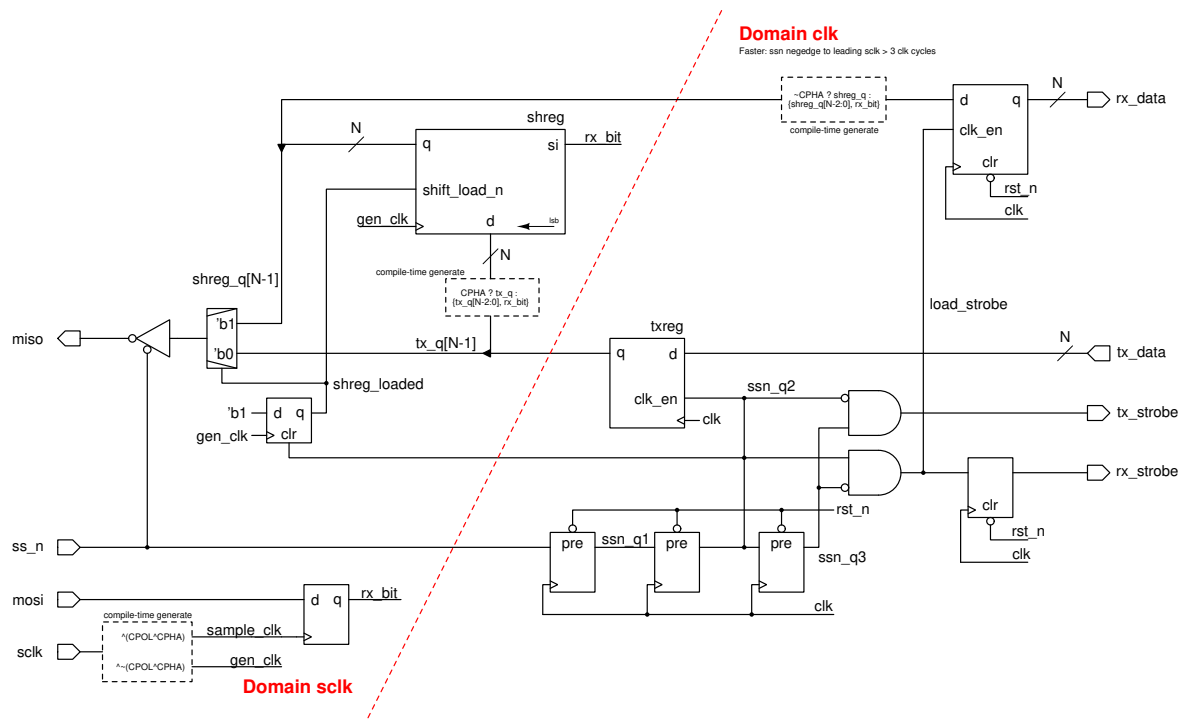
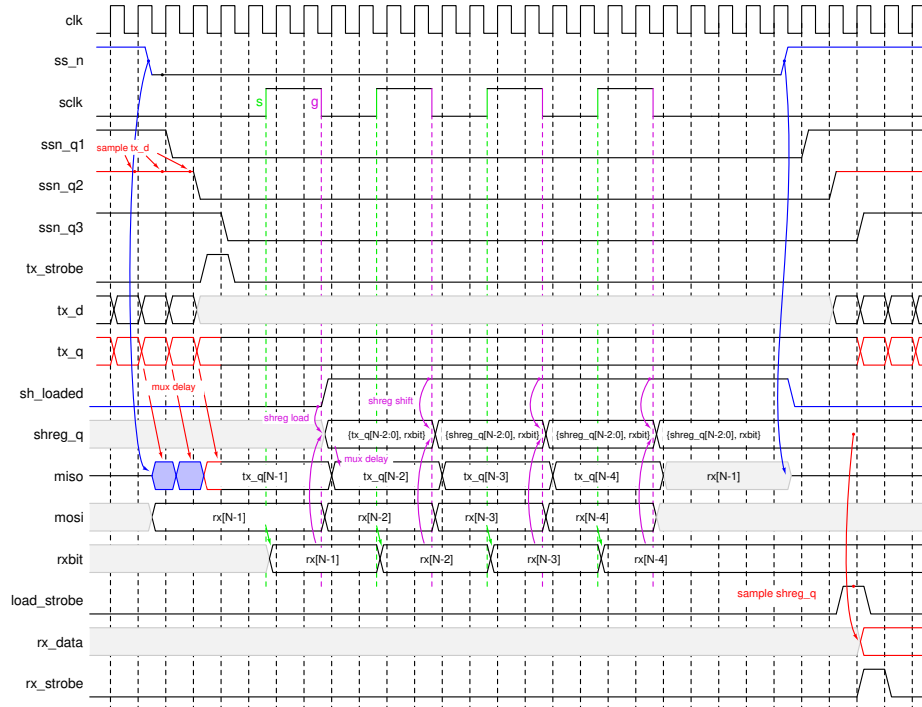
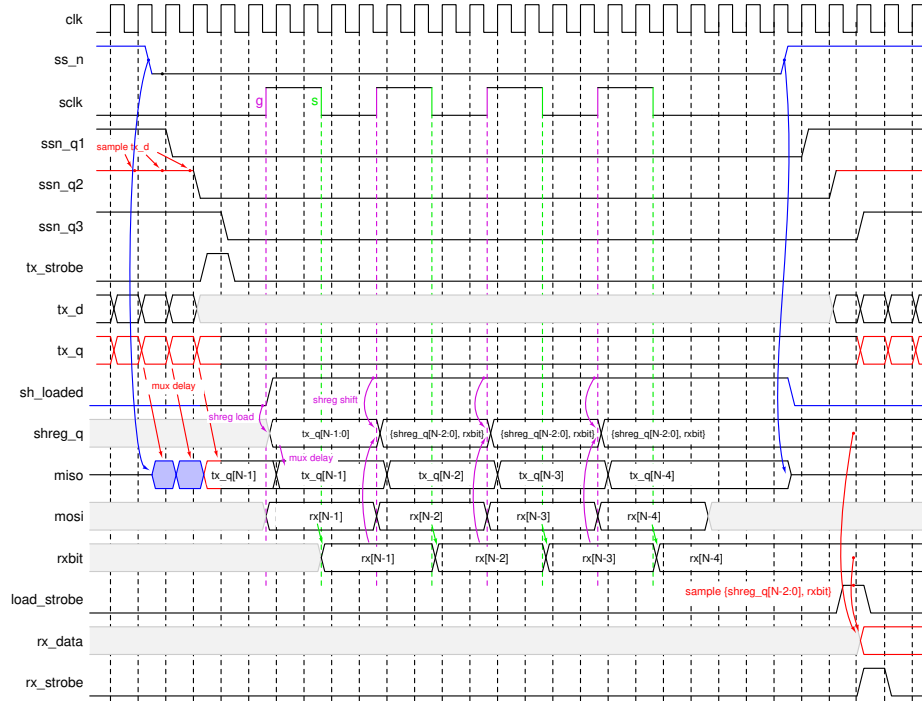


Figure 17: SPI slave controller



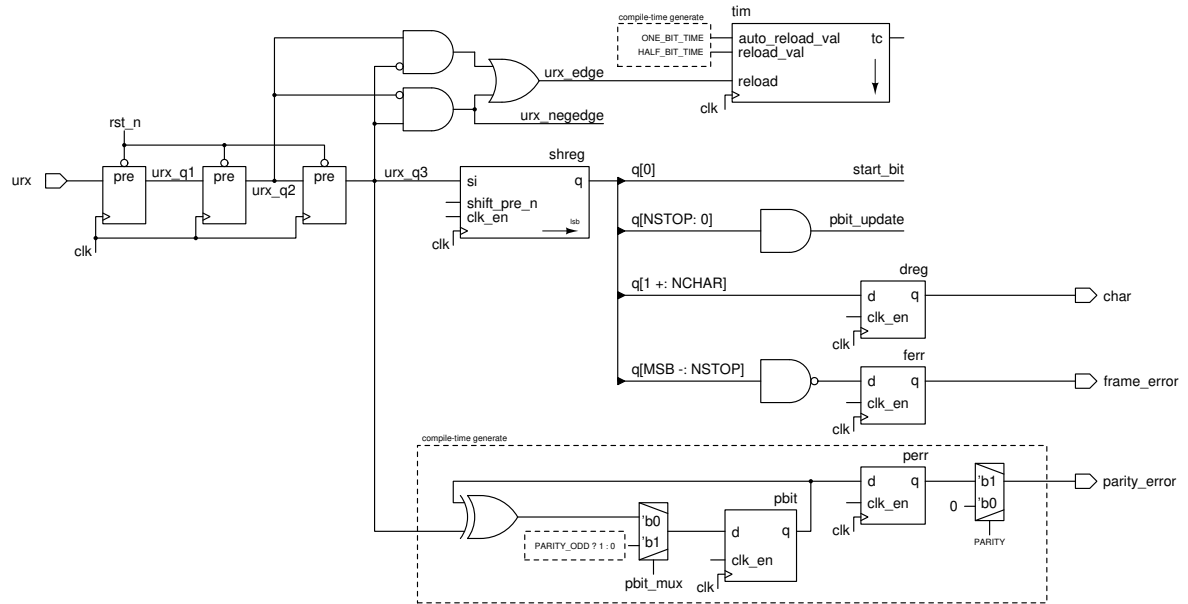
(a) 4 bit transmission, CPOL = 0, CPHA = 0



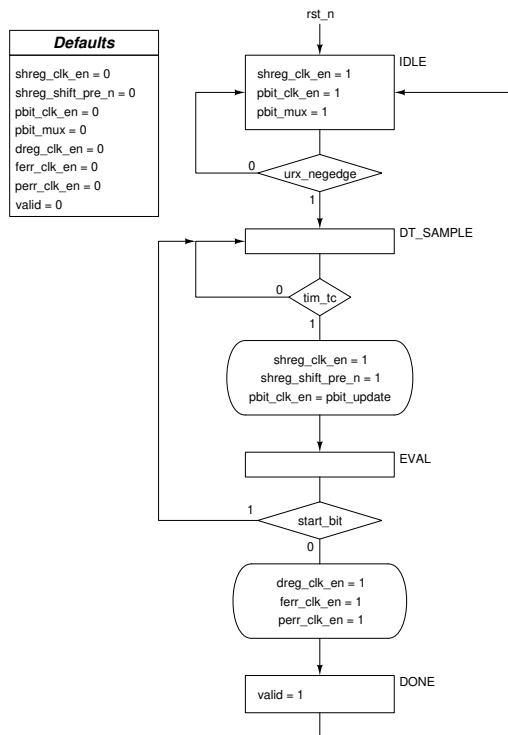
(b) 4 bit transmission, CPOL = 0, CPHA = 1

Figure 18: Manual timing analysis for the SPI slave controller

D UAR(T) Core



(a) Datapath



(b) ASM detailed chart

Figure 19: UART receiver-only core design