20 Peripherals

A system-on-chip has a variety of peripherals and memories in addition to the core. This chapter presents a variety of basic peripherals that are common for embedded systems or necessary to boot Linux. Fully-featured RISC-V systems require peripherals to generate the timer, software, and external interrupts. Embedded systems usually need the ability to control individual I/O pins with general-purpose I/O (GPIO). Many systems need a serial port to print to a console or control serial peripherals. Most processors access peripherals using memory-mapped I/O using loads and stores to memory locations associated with the peripherals.

In RISC-V, the SiFive Core Local Interruptor (CLINT) has become the de facto standard timer and software interrupt controller [FE310]. The Platform-Level Interrupt Controller (PLIC) is the standard RISC-V external interrupt controller that arbitrates among external interrupts and routes them to the appropriate harts [PLIC22]. The SiFive GPIO peripheral [FE310] is a de-facto standard for general-purpose I/O. Many Linux system serial ports are compatible with the classic Texas Instruments PC16550D Universal Asynchronous Receiver/Transmitter (UART) [PC16550]. On Wally, the peripherals are in the uncore module and connect to the bus via an APB bridge, as was shown in Figures 6.28XREF and 6.14XREF. The memory-mapped peripherals are tested by a sequence of reads and writes that test their basic functions.

Commented [1]: Add SDC, SPI, PWM

Wally contains the peripherals described in this chapter. Figure 1.29XREF showed the default Wally memory map, including the base addresses of all of these peripherals. The RAM and ROM were discussed in Section 6.6.3XREF, and the other peripherals are discussed here.

When testing peripherals, it can be helpful to connect their outputs back to their inputs so a test program can drive known values and read them back. This is known as *loopback testing*.

The `uncore` uses the following configuration parameters to specify whether a peripheral exists and where it goes in the memory map.

- CLINT_SUPPORTED / BASE / RANGE
- GPIO_SUPPORTED / BASE / RANGE
  - GPIO_LOOPBACK_TEST — Connect GPIO outputs back to inputs
- UART_SUPPORTED / BASE / RANGE
  - UART_PRESCALE — Reduce HCLK by $2^{\text{UART\_PRESCALE}}$
- PLIC_SUPPORTED / BASE / RANGE
  - PLIC_NUM_SRC — Number of interrupt sources
  - PLIC_GPIO_ID — Interrupt ID for GPIO
  - PLIC_UART_ID — Interrupt ID for UART
- SDC_SUPPORTED / BASE / RANGE

The test plan for each peripheral is described in the given peripheral's section below. The test programs are in `tests/wally-riscv-arch-test`. Each is an assembly language program that performs memory-mapped I/O accesses and checks for expected results. They use the same WALLY-TEST-LIB as in Chapters 5 and 8, so that test cases can be expressed as a series of

memory accesses. The library logs the results to the signature. Memory-mapped I/O addresses are defined by a peripheral base address and register offset so that a peripheral could be relocated by changing only the base address in `wally-config.vh` and the peripheral's test program. Interrupt testing leaves the global interrupts disabled and just checks the MIP pending interrupt bits; interrupt handlers were tested in Chapter 8XREF.

Some of the peripherals are too complex to describe here in full detail. They are worth reading about if you wish to learn about peripheral design and bus interfacing. See the SystemVerilog code in `src/uncore` for more details.

20.1 GPIO

The GPIO peripheral controls up to 32 GPIO pins on a chip. It can configure each pin individually to be an input or output and can generate interrupts when a particular pin is 0, 1, rises, or falls.

Figure 19.1 shows a block diagram of the GPIO peripheral top-level interface. The module connects to the APB bus, PLIC, and GPIO pins on the chip. Each pin driver receives `GPIOPinsIn`, `GPIOPinsOut`, and `GPIOPinsEn` signals to read or write (drive) the GPIO pin and control whether the pin is an input or output (i.e., enable/disable the tristate output driver). The GPIO can send interrupts to the PLIC when specified events occur.

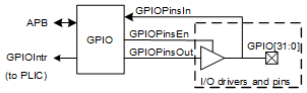

Figure 19.1 GPIO top-level interface

20.1.1 Memory Map

Table 19.1 lists the memory-mapped GPIO registers and their address offsets. For example, if the GPIO base address is 0x10060000 then a write to 0x1006000C puts a value in the output_val register. All GPIO registers are 32-bits, even on RV64. The registers are cleared on reset, making all pins inputs.

Table 19.1 GPIO register map

Offset	Name	Description
0x00	`input_val`	Pin value
0x04	`input_en`	Pin input enable
0x08	`output_en`	Pin output enable
0x0C	`output_val`	Output value
0x10	`pue`	Internal pullup enable
0x14	`ds`	Pin drive strength
0x18	`rise_ie`	Rise interrupt enable
0x1C	`rise_ip`	Rise interrupt pending

0x20	`fall_ie`	Fall interrupt enable
0x24	`fall_ip`	Fall interrupt pending
0x28	`high_ie`	High interrupt enable
0x2C	`high_ip`	High interrupt pending
0x30	`low_ie`	Low interrupt enable
0x34	`low_ip`	Low interrupt pending
0x38	`iof_en`	I/O function enable
0x3C	`iof_sel`	I/O function select
0x40	`out_xor`	Output XOR

Each register has 32 1-bit fields corresponding to the 32 GPIO pins. The `output_en` (output enable) register enables the tristate drivers when a bit is 1 to make the corresponding pin an output. The `output_val` register contains the values to drive onto the output pins. The `input_val` register contains values read from the input pins. The `input_en` register resets the corresponding bits in `input_val` to 0 when not enabled. The `out_xor` register conditionally inverts the output value going to the pin.

The inputs are generally asynchronous to the APB clock, so they are synchronized to the peripheral's clock domain with a 2-stage synchronizer (see Fig 3.XREF) to prevent metastability. The inputs control the interrupt pending register. When an input is 1, the corresponding high interrupt pending `high_ip` bit is set. Interrupt pending bits are sticky so that the bit will remain 1 even if the input becomes 0. Writing a 1 to an interrupt pending bit clears it. Similarly, the `low_ip`, `rise_ip`, and `fall_ip` bits are set by an input being 0, rising, or falling, and will remain set until a 1 is written to that bit of the interrupt pending (ip) register.

The interrupt enable registers (`high_ie`, `low_ie`, `rise_ie`, `fall_ie`) control whether particular GPIO pins generate an interrupt when the interrupt pending bit is set. If any of the high, low, rise, or fall interrupts are both pending and enabled, the `GPIOIntr` signal raises to inform the interrupt controller.

Some chips have configurable drive strength for output pads and configurable internal pullup resistors to prevent input pads from floating. These features are controlled by the `pue` and `ds` registers.

Some embedded processors with limited numbers of pins multiplex the GPIO pins with other peripherals such as serial ports. Optional `iof_en` and `iof_sel` GPIO registers control whether the pins are driven by an alternate I/O function and select which function.

20.1.2 Example

Commented [2]: None of the code examples in this chapter are tested.

Example 19.1 Consider the system in Figure 19.2 in which two pushbutton switches and an LED are connected to GPIO pins. The pulldown resistors set a default value of 0 when the switches are not pressed. The series current-limiting resistor sets the brightness of the LED and prevents damage to the pin overheating by driving too much current. Write a program to turn on the LED when either switch is pressed.

Solution: The following program initializes the input and output pin enables, then repeatedly reads the input pins, and turns on the LED if either input is nonzero.

```
         la t0, 0x10060000          # GPIO base address
         li t1, 3                   # Enable input pins 0 and 1 for switches
         sw t1, 4(t0)               #  using input_en register
         li t1, 4                   # Enable output pin 2 for LED
         sw t1, 8(t0)               #  using output_en register
loop:    lw t2, 0(t0)               # Read input_val (switches, other bits=0)
         beq t2, zero, led_off      # Are both switches 0?
         sw t1, 0xC(t0)             # No: turn on LED
         j loop                     # repeat
led_off: sw t0, 0xC(t0)             # Yes: turn off LED
         j loop                     # repeat
```

The same program is written in C below:

```
volatile uint32_t *GPIO_INPUT_VAL  = (uint32_t*)0x10060000;
volatile uint32_t *GPIO_INPUT_EN   = (uint32_t*)0x10060004;
volatile uint32_t *GPIO_OUTPUT_EN  = (uint32_t*)0x10060008;
volatile uint32_t *GPIO_OUTPUT_VAL = (uint32_t*)0x1006000C;

void gpio_switch_led(void) {
      int switches;

      // activate two switch inputs and one LED output
      *GPIO_INPUT_EN = (1 << 1) | (1 << 0);     // switches on pins 1 and 0
      *GPIO_OUTPUT_EN = (1 << 2);               // LED on pin 2

      while (1) {                               // loop forever
            switches = *GPIO_INPUT_VAL;         // read switches
            if (switches) *GPIO_OUTPUT_VAL = 4; // turn on LED
            else          *GPIO_OUTPUT_VAL = 0; // turn off LED
      }
}
```

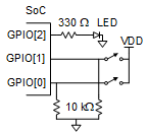

Figure 19.2 GPIO pins connected to switches and LED

20.1.3 Loopback Testing

On the GPIO peripheral, `GPIOPinsIn` and `GPIOPinsOut` are normally connected at the top-level of the chip to the I/O pin drivers. When the Wally `GPIO_LOOPBACK_TEST` configuration is set, the GPIO module connects the signals internally, so `GPIOPinsIn` gets its value from `GPIOPinsOut` when `GPIOPinsEn` is set. This facilitates testing at lower levels of the SoC hierarchy when there are no pin drivers.

20.1.4 Test Plan

GPIO is tested by checking the reset state, then driving outputs and checking inputs in loopback mode. Each of the interrupt enable and pending bits is also exercised. The cvw repository enumerates all of these tests.

20.1.5 Wally Implementation

Figure 19.3 shows a schematic of the GPIO module on RV32 with an APB interface. The module contains a 14-register × 32-bit register file containing the GPIO registers (excluding the `input_val`, `pue`, and `ds` registers). `input_val` is a synchronized register at the top of the schematic, and `pue` and `ds` are not implemented in Wally. An address decoder examines the bottom bits of the address to determine which register to read or write. On the access phase of a write when the GPIO is selected, the memwrite signal asserts to enable the appropriate entry. The read multiplexer also reads the entry selected by the address decoder. The loopback mux takes the input from `GPIOPinIn` when the output is disabled and from `GPIOPinOut` when the output is enabled. If the input is enabled, the input goes through a two-stage synchronizer and one more delay register. Output logic selects from `output_val` or optional alternative I/O functions (`iof`) and optionally inverts the polarity with an XOR. Interrupt set/clear logic detects rise/fall/high/low input conditions and writes to the interrupt pending registers and clears or sets the sticky registers. If any interrupt is both pending and enabled, the `GPIOIntr` is raised.

Commented [SH3]: ***This should be MemWrite (also in the figure). David or others, I can change the figure if you're ok with this change.

Commented [h4R3]: I believe it is memwrite in the code.

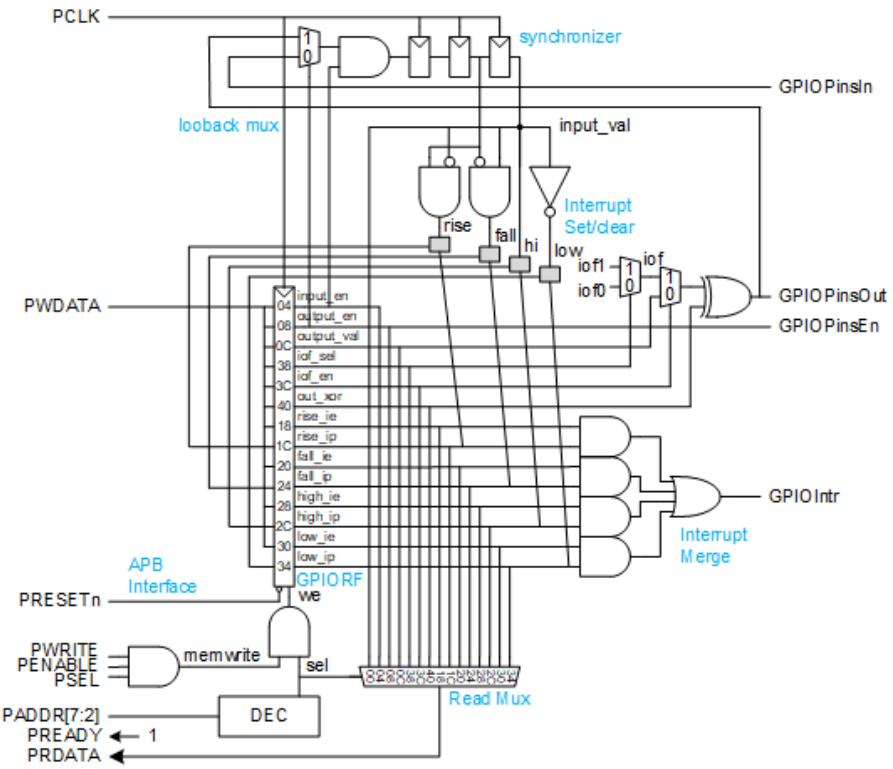

Figure 19.3 GPIO implementation

Commented [SH5]: ***Need to redo this figure with correct coloring. Also fix slanted vertical wires.

Also APB Interface should refer to all of those signals.

<SIDEBAR>
The RV64 design is identical except that it adds multiplexers on PRDATA and PWDATA to use the upper or lower 32 bits of a 64-bit word.
</END>

Figure 19.4 shows a schematic of the gray box that sets and clears pending interrupts. If an interrupt register is written, ones in the write data (PWDATA) clear corresponding pending inputs. Otherwise, pending interrupts are set when their condition (rise/fall/high/low) occurs, indicated by signal cond.

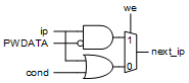

Figure 19.4 Interrupt pending set/clear logic

Wally does not rely on I/O pads having internal pull-up resistors or adjustable drive strength, so it does not implement the pue and ds registers.

20.2 CLINT

The Core Local Interruptor (CLINT) contains memory-mapped registers to generate software and timer interrupts and read the current time. Specifically, it has a 64-bit time counter, a 64-bit time compare register that produces a timer interrupt when the timer exceeds the compare value, and a 1-bit register to create a software interrupt. In a multi-core system, it gives cores a common time reference and provides a timer compare and software interrupt for each core.

<SIDEBAR>
Cores can send messages to each other with software interrupts. Operating systems use timer interrupts to schedule one of several user-mode processes.
</END>

Figure 19.5 shows a block diagram of the CLINT peripheral top-level interface. The module connects to the APB bus and sends machine-mode timer and software interrupts to the core(s).

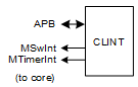

Figure 19.5 CLINT top-level interface

20.2.1 Memory Map

Table 19.2 lists the memory-mapped CLINT registers and their address offsets. For example, if the CLINT base address is 0x02000000 then a 64-bit read from 0x0200BFF8 reads the `mtime` register. `mtime` and `msip` are zeroed on reset.

Table 19.2 CLINT Register Map [FE310]

Offset	Name	Description
0x00	`msip hart 0`	1-bit software interrupt pending
…	`msip hart i`	Additional bits if multiple harts exist
0x4000	`mtimecmp hart 0`	64-bit timer compare value
…	`mtimecmp hart i`	Additional bits if multiple harts exist
0xBFF8	`mtime`	Shared 64-bit timer

The `mtime` timer typically runs continuously at a constant frequency, even if some or all of the cores are sleeping with their clocks stopped. It may use a different frequency clock than the core, especially if the core frequency is variable. If the timer is asynchronous to the cores, the CLINT requires clock domain crossing circuitry.

Recall from Section 6.XREF that a hart has a read only `time` CSR that returns the value of `mtime` from the CLINT.

20.2.2 Example

Example 19.2 Write functions to set the CLINT software interrupt to 0 or 1, and to create a delay of a specified number of timer ticks.

Solution: The following program defines the CLINT register addresses. `setSWInt()` turns the software interrupt on or off. `delay()` causes the timer interrupt to rise after the specified number of ticks.

```
volatile uint64_t *CLINT_MSIP     = (uint64_t*)0x20000000;
volatile uint64_t *CLINT_MTIMECMP = (uint64_t*)0x20004000;
volatile uint64_t *CLINT_MTIME    = (uint64_t*)0x2000BFF8;

void setSWInt(int val) {
      *CLINT_MSIP = val;
}

void delay(uint64_t ticks) {
      // create a delay of the specified number of timer ticks
      uint64_t doneTime = *CLINT_MTIME + ticks;

      while (*CLINT_MTIME < doneTime); // spin until time has elapsed
}
```

20.2.3 Test Plan

The CLINT is tested by toggling software interrupts, `mtime` and `mtimecmp`, and checking that the `mip.MSIP` and `mip.MTIP` (software and timer interrupt pending) bits respond accordingly. The cvw repository enumerates all of these tests.

20.2.4 Wally Implementation

Figure 19.6 shows a schematic of the CLINT module on RV64 with an APB interface.
The CLINT has a 64-bit `mtime` counter, a 64-bit `mtimecmp` register, 1-bit `msip` register, and a 64-bit timer interrupt comparator, along with circuitry to read and write these registers. The `mtime` counter increments on each cycle unless it is being written.

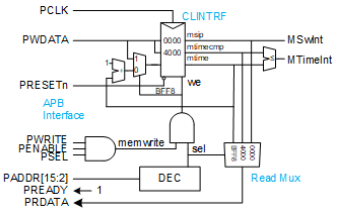

Figure 19.6 CLINT implementation

20.3 PLIC

The Platform-Level Interrupt Controller (PLIC) takes interrupts from multiple peripherals, prioritizes them, and routes them to the appropriate hart's machine or supervisor-mode external interrupt pin [PLIC22]. In general, the PLIC standard supports up to 1024 *interrupt sources* and 15872 *hart context* destinations, although a system normally has only a few of each.

<SIDEBAR>
Early SiFive processors have 54 interrupt sources, of which only a fraction connect to peripherals. Wally does the same.
</END>

The PLIC has *interrupt pending* bits and programmable *interrupt priorities* for each of the interrupt sources. Source 0 does not exist. A priority of 0 means never interrupt.

The *hart context* is a specific privilege mode on a specific hart. The PLIC drives one external interrupt pin for each hart context. As discussed in Section 5.XREF, machine mode has a hart context, and supervisor mode, when implemented, has a second hart context. The PLIC has a programmable *priority threshold* and *claim/complete* register for each context. It also has an *interrupt enable* bit for each source/context pair, so each context can receive interrupts from specific sources.

<SIDEBAR>
For example, a 4-core system with 2-way multithreading has 8 harts. Each hart has machine and supervisor mode, so the system has 16 hart contexts. Wally has one hart and two hart contexts.
</END>

<SIDEBAR>
Section 5.2.5.2XREF described external interrupts from a hart's perspective.
</END>

The PLIC identifies the highest priority pending interrupt. If multiple interrupts with the same priority are pending, it selects the lowest source number. The PLIC broadcasts that interrupt to all hart contexts whose interrupts are enabled for that source and whose priority threshold is less than the interrupt priority.

A hart context claims an interrupt by reading its claim/complete register. The register returns the source number of the highest priority pending interrupt, or 0 if no interrupt is pending. Reading the register also clears the pending bit within the PLIC. The hart context should respond to the claimed interrupt and cause the peripheral to stop issuing the interrupt. For example, if the GPIO signals an interrupt on a rising input, the hart should clear the `rise_ip` bit. Then the hart context writes the interrupt source number back to the claim/complete register to indicate that the interrupt has been processed and that the PLIC can raise the pending bit again the next time that interrupt source is active.

20.3.1 Memory Map

Table 19.3 lists the memory-mapped PLIC registers and their address offsets, assuming 54 sources and 2 contexts. All PLIC registers are 32 bits wide.

Table 19.3 PLIC register map

Offset	Name	Description
0x000004	intPriority1	Priority for interrupt source 1
0x000008	intPriority2	Priority for interrupt source 2
…	…	…
0x0000DB	intPriority53	Priority for interrupt source 53
0x001000	intPending0	Pending bits for interrupts 31:0
0x001004	intPending1	Pending bits for interrupts 53:32
0x002000	intEn00	Interrupt enables for context 0 sources 31:1
0x002004	intEn01	Interrupt enables for context 0 sources 53:32
0x002080	intEn10	Interrupt enables for context 1 sources 31:1
0x002084	intEn11	Interrupt enables for context 1 sources 53:32
0x200000	threshold0	Priority threshold for context 0
0x200004	claimComplete0	Claim/Complete register for context 0
0x201000	threshold1	Priority threshold for context 1
0x201004	claimComplete1	Claim/Complete register for context 1

Wally's PLIC receives interrupts from two sources (UART and GPIO) and two hart contexts (Machine and Supervisor modes of the core). Like the SiFive FE310, it supports up to 54 interrupt sources with 3-bit priorities (1-7). `PLIC_NUM_SRC` configures the actual number of sources. The interrupt numbers are defined in `wally-config.vh`, with `GPIOIntr` going to source `PLIC_GPIO_ID` = 3 and `UARTIntr` going to source `PLIC_UART_ID` = 10. Because GPIO has a lower source number than the UART, GPIO takes precedence when the `intPriority` values are equal.

20.3.2 Example

Example 19.3 Write a function to configure the PLIC to route UART and GPIO interrupts to Wally's machine mode external interrupt pin `MExtInt`, with the GPIO receiving higher priority. Write an external interrupt handler to determine which peripheral caused the interrupt and handle it accordingly.

Solution: The following program defines the PLIC register addresses. Context 0 is machine mode for Wally's only hart. `plicInit()` sets the GPIO to have an arbitrary priority higher than that of the UART, and the threshold for context 0 to accept both sources. It also enables interrupts from both sources. When an external interrupt occurs, the hart enters the machine mode interrupt handler and saves registers. It should then call `mextIntHandler()`, which reads the `claimComplete` register to determine which peripheral caused the external interrupt. It handles that interrupt (e.g., by reading data waiting on the UART) and ensures the

peripheral is no longer asserting the interrupt. It then writes the peripheral source number back to `claimComplete` to inform the PLIC that the interrupt has been handled.

```
volatile uint32_t *PLIC_INTPRIORITY3   = (uint32_t*)0x0C00000C;
volatile uint32_t *PLIC_INTPRIORITY10  = (uint32_t*)0x0C000028;
volatile uint32_t *PLIC_INTEN00        = (uint32_t*)0x0C002000;
volatile uint32_t *PLIC_THRESHOLD0     = (uint32_t*)0x0C200000;
volatile uint32_t *PLIC_CLAIMCOMPLETE0 = (uint32_t*)0x0C200004;

void plicInit(void) {
      *PLIC_INTPRIORITY3 = 5;  // GPIO has priority 5
      *PLIC_INTPRIORITY10 = 4; // UART has lower priority 4
      *PLIC_INTEN00 = (1 << 3) | (1 << 10); // enable GPIO and UART
      *PLIC_THRESHOLD0 = 2;    // interrupts for any source with priority > 2
}

void mextIntHandler(void) {
      uint32_t *source = *PLIC_CLAIMCOMPLETE0; // what caused the interrupt?
      if (source == 10)     handleUARTInt();
      else if (source == 3) handleGPIOInt();
      *PLIC_CLAIMCOMPLETE0 = source;           // signal completion
}
```

20.3.3 Test Plan

The general method of testing the PLIC is to go through various permutations of interrupts from the UART and GPIO having different priorities and enables and being routed to different contexts. The cvw repository enumerates all of these tests.

20.3.4 Wally Implementation

See the Verilog code in `src/uncore/plic_apb.sv`.

20.4 UART

Computers historically communicated with teletype machines or VT100 text consoles over phone lines using a modem. A modem (short for modulator/demodulator) operates asynchronously because it is impractical to send a clock over a phone line. It communicates information with audio pulses at tens to thousands of bits per second traveling both directions along the line. The modem converts these audio pulses to electrical pulses and conveys them to a computer or device using a Universal Asynchronous Receiver/Transmitter (UART). Many devices still use this standard for backward compatibility or historical reasons, despite the existence of simpler and faster SPI and I^2C serial links.

<SIDEBAR>
The Digital Equipment Corporation VT100 series of terminals, introduced in 1978, was a wildly successful way to communicate over a modem serial port with mainframe computers before the

PC revolution. Over 6 million VT terminals were sold. Linux terminal windows still use serial communication, now called a COM port.

Figure 19.7 VT100 Terminal[1] [Jason Scott, 2013-08-30, CC BY 2.0]

</END>

Figure 19.8 shows the top-level interface of a UART. Serial communication takes place over the TX and RX transmit and receive pins. The UART optionally has RTS, CTS, DSR, DTR, RI, and DCD handshake signals to control a modem.

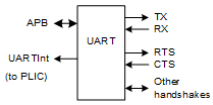

Figure 19.8 UART top-level interface

Figure 19.9 shows the waveforms to transmit a character over a UART. The TX line idles high. The start of the transmission is indicated by TX going low for one bit period; this is called the *start bit*. The UART then sends *data bits* (typically 8), optionally followed by a *parity bit* used to detect an error in the transmission. Finally, it raises TX for at least one bit period; this is called the *stop bit*. If the UART uses one start bit, one stop bit, no parity bits, and an 8-bit character, it takes 10 bit times to send 8 data bits. Hence, the UART speed is called the *baud rate* rather than the bit rate. The bit time is the inverse of the baud rate. For example, 9600 baud means 9600 bit times per second, or 960 characters per second (9600 bit times/second × 8 data bits/10 bit times × 1 character/8 data bits) after accounting for the start and stop bits.

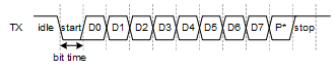

Figure 19.9 UART timing

The UART is asynchronous, so the baud rate is set by mutual agreement between the transmitter and receiver rather than by a common clock. The existence and polarity of the parity bit and the number of data and stop bits is also set by agreement. Some systems need the optional handshaking signals, while others ignore them. TX of the UART is hooked up to RX of the device it is speaking to, and vice versa. In a normal cable, RTS of the UART connects to RTS of

[1] https://en.wikipedia.org/wiki/VT100#/media/File:DEC_VT100_terminal.jpg

the device, and CTS connects to CTS, but in a null modem cable, RTS connects to CTS and vice versa. Sometimes the 0s and 1s are once converted to +/- 12 V and sent over a bulky RS-232 serial cable. Alternatively, they may be sent across a circuit board at a standard digital logic level, such as 3.3 V. As you can see, UART communication has many fussy parameters that must be set exactly right on each end, or the system will fail and be hard for ordinary users to troubleshoot.

<SIDEBAR>
USB was developed to end these troublesome UART configurations as well as to greatly increase the data rate, reduce the connector size, and reduce cable cost. Unfortunately, USB has inherent security holes. For example, the firmware on a USB drive can be configured to pose as a keyboard device instead and type malicious keystrokes into a computer.
</END>

A UART must generate signals to sample the received data at the right time. It typically begins with a high-speed peripheral clock PCLK operating at the system clock frequency (clk). The UART divides the peripheral clock down by a *baud rate divisor* × $2^{\text{UART\_PRESCALE}}$ to 16x the desired baud rate. The UART oversamples the RX line at this 16x clock. The first high-to-low transition indicates going from idle to the start bit. The UART can then wait 8 clock periods and sample every 16 clocks thereafter to measure the data bits in the middle of their window.

For example, suppose the system clock runs at 320 MHz and UART_PRESCALE = 5. If the baud rate is 9600, the 16x clock is at 9600 × 16 = 153.6 KHz, and its period is 1/153600 = 6.51 μs. The baud rate divisor should be the baud period / ($2^{\text{UART\_PRESCALE}}$ / f_{clk}) = 6.51×10^{-6} s / (2^5 / 320×10^6 Hz) = 65.1, so the UART uses a baud rate divisor of 65 to generate a 16x clock with a 6.5 μs period. The error is small enough that the UART can sample all of the bits in the character accurately, and then synchronize again to the next start bit.

A UART classically has a *Transmit Holding Register* (THR) and a *Receiver Buffer Register* (RBR), along with control signals to indicate whether each of these registers is full. A host processor sends a character by waiting until the THR is empty and then writing a byte to the THR. The UART marks THR full, generates the appropriate start, stop, parity, and data bits at suitable times and sends them out the TX line, then marks THR empty again. Similarly, a host processor receives a character by waiting until RBR is full and reading it out. If a second character arrives before the host reads the first from the RBR, a *buffer overflow* occurs, and data is lost. The system must use RTS/CTS or software flow control to prevent buffer overflow. More advanced UARTs have a first-in first-out (FIFO) on the transmitter and receiver so that the system can send and receive many characters before having to wait for buffers to empty. UARTs may also generate interrupts when the buffers are empty so that the host can do other work and get notified when the next character is ready.

The original IBM PC used the 8250 UART in its COM port. At speeds above 9600 baud, the PC had trouble servicing interrupts fast enough to avoid buffer overflows. The 16550 UART [PC16550] added FIFOs while maintaining backward compatibility, and the register set for the 8250/16550 remains a widely used standard. National Semiconductor initially manufactured the 16550 used in IBM PS/2 computers. Texas Instruments and others soon developed compatible

products. RISC-V Linux has device drivers for UARTs compatible with the classic 16550, as well as the simpler SiFive FU540 UART [FU540].

20.4.1 Memory Map

The PC16550 has a memory-mapped interface with eight 8-bit registers at offsets of 0-7 from the base address. Figure 19.10 shows the PC16550 register map. A read from offset 0 reads the RBR, while a write to offset 0 writes the THR. Offset 1 is the Interrupt Enable Register (IER), which controls what events generate interrupts. Reads from offset 2 return the Interrupt Identification Register (IIR) providing information about the interrupt, while writes to offset 2 control the FIFO. The Line Control Register (LCR) configures the number of start bits, the parity, the number of bits in a character, etc. The Modem Control Register (MCR) generates outputs such as DTR and RTS. Bit 4 is the Loop field, which configures the transmit and modem control outputs to feed back to the receive and modem status inputs for loopback testing. The Line Status Register (LSR) indicates error conditions such as parity or buffer overrun. The Modem Status Register (MSR) has more handshaking signals. The scratch register (SCR) holds one byte of information; it has little practical purpose now. Bit 3 of the LCR is the Divisor Latch Access Bit (DLAB). When DLAB is set, writes to address 0 and 1 set the low and high byte of the baud rate divisor, respectively, rather than the THR and IER. This divisor is normally programmed during initialization and then DLAB is cleared.

	REGISTER ADDRESS											
Bit No.	0 DLAB=0	0 DLAB=0	1 DLAB=0	2	2	3	4	5	6	7	0 DLAB=1	1 DLAB=1
	Receiver Buffer Register (Read Only)	Transmitter Holding Register (Write Only)	Interrupt Enable Register	Interrupt Ident. Register (Read Only)	FIFO Control Register (Write Only)	Line Control Register	MODEM Control Register	Line Status Register	MODEM Status Register	Scratch Register	Divisor Latch (LS)	Divisor Latch (MS)
	RBR	THR	IER	IIR	FCR	LCR	MCR	LSR	MSR	SCR	DLL	DLM
0	Data Bit 0[1]	Data Bit 0	Enable Received Data Available Interrupt (ERBFI)	"0" if Interrupt Pending	FIFO Enable	Word Length Select Bit 0 (WLS0)	Data Terminal Ready (DTR)	Data Ready (DR)	Delta Clear to Send (DCTS)	Bit 0	Bit 0	Bit 8
1	Data Bit 1	Data Bit 1	Enable Transmitter Holding Register Empty Interrupt (ETBEI)	Interrupt ID Bit (0)	RCVR FIFO Reset	Word Length Select Bit 1 (WLS1)	Request to Send (RTS)	Overrun Error (OE)	Delta Data Set Ready (DDSR)	Bit 1	Bit 1	Bit 9
2	Data Bit 2	Data Bit 2	Enable Receiver Line Status Interrupt (ELSI)	Interrupt ID Bit (1)	XMIT FIFO Reset	Number of Stop Bits (STB)	Out 1	Parity Error (PE)	Trailing Edge Ring Indicator (TERI)	Bit 2	Bit 2	Bit 10
3	Data Bit 3	Data Bit 3	Enable MODEM Status Interrupt (EDSSI)	Interrupt ID Bit (2) [2]	DMA Mode Select	Parity Enable (PEN)	Out 2	Framing Error (FE)	Delta Data Carrier Detect (DDCD)	Bit 3	Bit 3	Bit 11
4	Data Bit 4	Data Bit 4	0	0	Reserved	Even Parity Select (EPS)	Loop	Break Interrupt (BI)	Clear to Send (CTS)	Bit 4	Bit 4	Bit 12
5	Data Bit 5	Data Bit 5	0	0	Reserved	Stick Parity	0	Transmitter Holding Register (THRE)	Data Set Ready (DSR)	Bit 5	Bit 5	Bit 13
6	Data Bit 6	Data Bit 6	0	FIFOs Enabled [2]	RCVR Trigger (LSB)	Set Break	0	Transmitter Empty (TEMT)	Ring Indicator (RI)	Bit 6	Bit 6	Bit 14
7	Data Bit 7	Data Bit 7	0	FIFOs Enabled[2]	RCVR Trigger (MSB)	Divisor Latch Access Bit (DLAB)	0	Error in RCVR FIFO[2]	Data Carrier Detect (DCD)	Bit 7	Bit 7	Bit 15

Figure 19.10 PC16550 register map [PC16550]

Figure 19.11 shows the manufacturer's block diagram of the PC16550, including each register. The serial output comes from the transmitter shift register, which comes from either the THR or transmitter FIFO. Similarly, the serial input goes to a receiver shift register and on to the RBR or receiver FIFO.

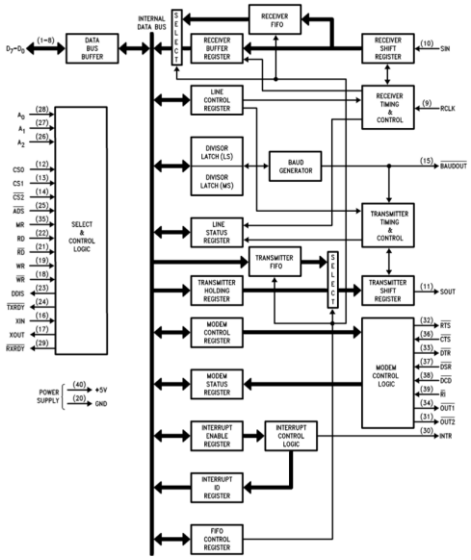

Figure 19.11 PC16550 block diagram [PC16550]

20.4.2 Example

Example 19.3 Write a function to configure the UART to operate at 9600 baud with eight data bits, one start bit, one stop bit, and no parity. Set the baud rate divisor to 65 as in the example above. Also write functions to send a character and receive a character.

Solution: The following program defines the UART register addresses. Note that the RBR, THR, and DLL are all at offset 0. When DLAB = 1, the UART accesses the DLL. Otherwise, writes go to the THR and reads come from the RBR. `uartInit()` sets the Line Control Register (LCR) to configure the data, stop, and parity bits. It sets DLAB, then writes the clock divisor to DLL, then clears DLAB. `uartSend()` and `uartReceive()` wait for the transmit holding register to be empty or the data ready signal to be asserted, then write or read the character.

```
volatile uint8_t *UART_RBR = (uint32_t*)0x10000000;
volatile uint8_t *UART_THR = (uint32_t*)0x10000000;
volatile uint8_t *UART_DLL = (uint32_t*)0x10000000;
volatile uint8_t *UART_LCR = (uint32_t*)0x10000003;
volatile uint8_t *UART_LSR = (uint32_t*)0x10000005;

void uartInit(void) {
```

```
      *UART_LCR = 0b1000011; // 8-bit characters, 1 stop bit, no parity, DLAB
      *UART_DLL = 65;        // write clock divisor with DLAB = 1
      *UART_LCR = 0b0000011; // turn off DLAB
}

void uartSend(char c) {
      while (!(*UART_LSR & (1<<5))); // wait for THRE (trans hold reg empty)
      *UART_THR = c;
}

char uartReceive(void) {
      while (!(*UART_LSR & (1<<0))); // wait for DR (Data Ready)
      return *UART_RBR;
}
```

20.4.3 Test Plan

The general method of testing the UART is to transmit data in loopback mode under various configurations and check the received data is correct. The tests check each configuration of the line control register. They verify that interrupts occur from the transmitter's line status register, the modem control register, and the FIFO. They also check that the receiver FIFO triggers thresholds and overflow. The cvw repository enumerates all of these tests.

20.4.4 Wally Implementation

Wally has a 16550-compatible UART. The logic is interesting to those who are curious how a serial port works, but it is too complex to describe in this chapter. Consult the Verilog (`src/uncore/uartPC16550D.sv`) for details.

20.5 SPI

<DH write after merging Naiche's SPI>

20.6 SDC

<RT write>

- **Summary**

- **Exercises**