Communications & Event Driven Programming

Prof. John McLeod

ECE9047/9407, Winter 2021

The basics of microcontroller serial communications are discussed. There are a lot of different protocols for communications, and most of them are too complicated to be discussed at this level, but universal asynchronous receiver-transmitters (UART) and the inter-integrated circuit protocol (I^2C) are simple enough to be discussed in some detail. Event-driven programming is also introduced, and a detailed example outlines various approaches.

Communications

Outside of toys, or teaching gimmicks, there are few uses for microcontrollers that do not involve some form of **communication** with other systems.

Definition: In the context of this course, **communication** refers to the exchange of information between two otherwise independent CPU-based systems.

It is the *independence* of the systems that makes communications challenging.

- The CPU exchanges information between peripherals and memory chips, but this is usually in the form of the CPU giving orders, and the other components listening attentively for those orders.
- When attempting to exchange information between two independent systems, it is quite likely that both CPUs have other things to do than to just wait for the information, and it is also possible that both CPUs may decide to send information (or *talk*) simultaneously on the same channel.

To address these challenges, computer systems communicate using well-defined protocols. The most common ones used today (USB, Bluetooth, WiFi, etc.) are all too complicated, so we will ignore them for now. [1]

[1] Eventually we will have to discuss some of these at least a little bit when we move on to wireless sensor networks.

Serial Communications and Shift Registers

From a hardware perspective, the simplest communication method is **serial**, using a single wire to transmit one bit of information at a time. The basic hardware element for serial communication is a **shift register**. A simplified 4-bit shift register is shown in Figure 1, a more practical 8-, 16-, or 32-bit shift register can be constructed by extending the pattern of flip flops. An appropriately-designed shift register can act as a **serial receiver** and a **serial transmitter**, with the functionality controlled by software.

- On each clock cycle the contents of each flip flop are shifted to the flip flop immediately on the right.
- New data from **serial in** can be read in to the left-most flip flop, synchronized by the clock.
- Data is transmitted from the right-most flip flop onto **serial out**, synchronized by the clock.
- A block of data can also be loaded in parallel to the flip flops from the data bus.
- Subsequently, this data will be sent bit-by-bit to the serial out port, synchronized with the clock.
- A block of data can also be read in parallel from the flip flops to the data bus.

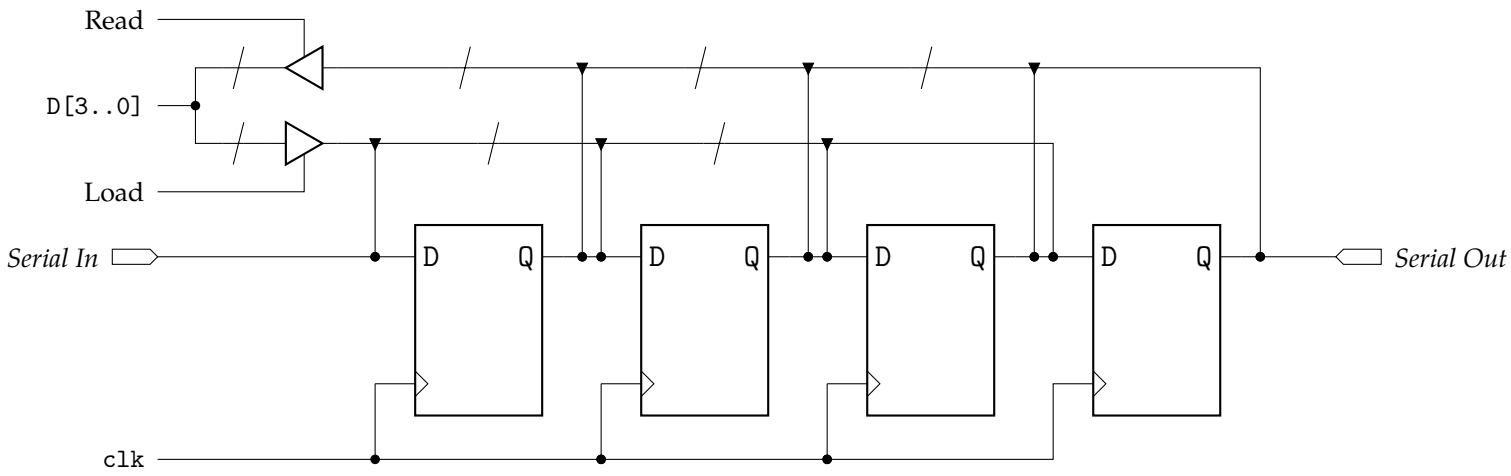


Figure 1: A simplified 4-bit shift register with parallel load and read.

If the serial out port of an n-bit shift register is connected to the serial in port of another n-bit shift register, then an n-bit block of data, loaded in parallel into the first shift register will by transmitted to the second shift register in n clock cycles. An example of sending a 4-bit number in this manner is shown in Figure 2.

- A 4-bit number `0b1011` is loaded into the first register from the data bus, synchronously with the clock.
- This number is shifted right bit-by-bit, as each successive LSb is transferred to the second shift register (appearing as the MSb there).
- When the transfer is complete, the result can be read from the second shift register by the data bus.

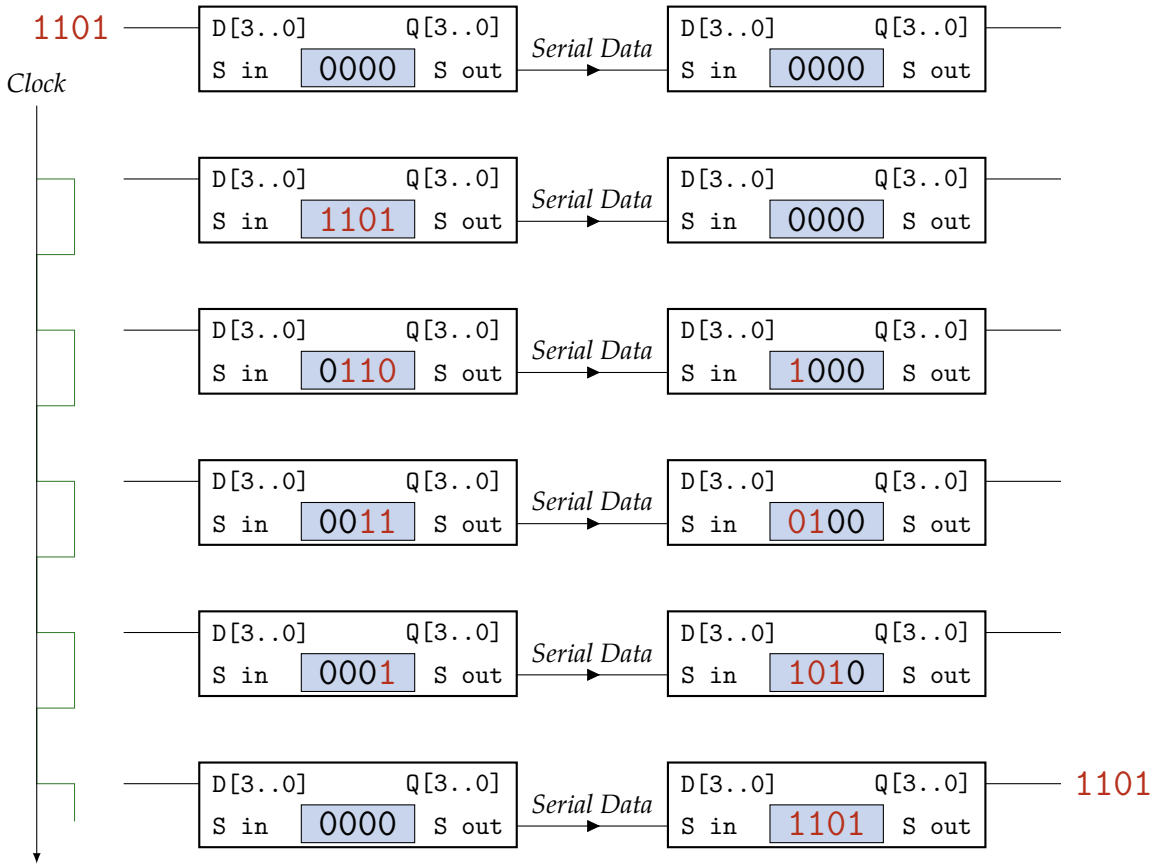


Figure 2: A schematic representation of transmitting a 4-bit value using shift registers. Each "row" in this diagram represents a subsequent clock cycle.

In terms of hardware, Figure 1 is basically a functional communications device. The challenge now lies in software, as is indicated by the process of transmitting a 4-bit number, described above in Figure 2. Namely:

- How does the second microcontroller know when first microcontroller wants to send data?
- How does the second microcontroller know when the first microcontroller is finished sending data?
- How does the second microcontroller know the clock speed of the first microcontroller?

Both microcontrollers need to agree on a set of standards for serial communication in order for one to receive the same information that the other transmitted.

Definition: A communications **protocol** is a set of standards that describe how information is transmitted.

The communications protocol needs to have the answers to the above questions. Furthermore, for some communication networks, the protocol needs to guide out how conflicts are resolved if two or more systems try to transmit at the same time.

Universal Asynchronous Receiver-Transmitters

HARDWARE

Universal asynchronous receiver-transmitters (UARTs) are the hardware components that handle a particular form of serial communication. Technically, the actual communication **protocol** has a different name, like "RS-232" or "RS-485". But almost everyone uses **UART** as the general descriptor for both hardware and software protocols, so we'll stick with that. In this protocol the **baud rate** and **communication frame** need to be pre-determined.

Definition: The **baud rate** is the number of pulses transmitted per second. The baud rate is not exactly the same as the "bit rate" of information transfer, as not all communication pulses correspond to bits of data.

No every pulse corresponds in the communication corresponds to a bit of data

Definition: The **communication frame** is the set of transmission pulses the comprise the minimal transmission packet.

Definition: A **pulse** is holding the transmission line at a high or low voltage for a length of time determined by the baud rate.

Baud rates are always multiples of 300; 9600 is a popular choice. This is one of those bizarre historical artefacts that are held to ensure backwards-compatibility with archaic equipment that basically no longer exists. [2] A **communication frame** consists of the following pulses: [3]

multiples of 300 to preserve backwards compatibility

- Before sending data, the UART transmission line is **idle** at a high voltage.

[2] In this case, a teletypewriter from the 1960s!

[3] I am deliberately reserving the word "bit" for pulses that correspond to data transmission. A lot of other texts and written documentation do *not* do this, so maybe I am just looking for trouble...

- The **start signal** is always a single, low voltage pulse.
- The next n pulses transmit bits of data. n is usually 7, 8, or 9. The value of n must be agreed upon by all systems in the communications network. The data can be sent LSb-to-MSb, or MSb-to-LSb.
- A **parity pulse** (or "parity bit") for error-checking may come next.
- Finally, the line is held at a high voltage for m pulses as the **stop signal**. Usually $m = 1$, but sometimes it may be different.

in the old days 7 bits was the data length because that is enough to encode all ASCII characters

The receiver must always receive a **stop signal** for each frame, otherwise it will discard the result as corrupted junk. A stop signal must therefore be sent even if the transmitter will continue to immediately send more information.
The **parity bit** is another form of error checking. The communications protocol may define data transmission as even or odd parity.

- Before sending the frame, the transmitter will sum the n bits of data. The result will be either an even or an odd number.
- The parity bit will then be set to make sum of the entire $n+1$ bit sequence even or odd, as agreed upon by the communications protocol.
- After receiving the frame, the receiving system sums the n bits of data and the parity bit, and checks the result against the even or odd protocol. This helps inform the receiver whether the transmission was corrupted.

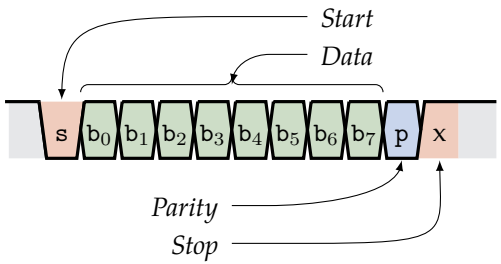


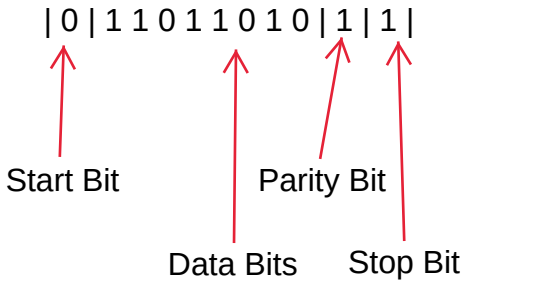
Figure 3: A UART communications frame for transmitting 8 bits of data, with one parity and one stop bit.

Of course, a parity check will only detect single (or triple, or quintuple, etc.) pulse transmission errors. A double (or quadruple, or hextuple) pulse transmission error will fool the system into thinking the data was correctly transmitted.

adds up to 4 ones--> 0

Example: With an even parity check, an 8-bit transmission of `0b11011000` should have a parity bit of zero, and an 8-bit transmission of `0b11011010` should have a parity bit of 1. adds up to 5 ones --> 1

Example: The UART protocol implemented by an arbitrary system calls for 8-bit transmissions, an odd parity check, and a single stop pulse. If communications frame of `01101101011` is received (written with the start pulse on the left and the stop pulse on the right), the receiver will recognize that the data is corrupted — omitting the stop pulse, the data has even parity.



- Data bit are odd parity bit should be 0

Example: The UART protocol implemented by an arbitrary system calls for 8-bit transmissions, an odd parity check, and a single stop pulse. The communications frame `01101101001` is transmitted through a noisy channel and the frame `01100101101` is received. As a double error occurred, the receiver does not recognize that the data has been corrupted. This is one of the limitations with UART.

UART communications frames are abbreviated as n-P-m, where n is the number of bits of data transmitted, P is the parity (`0` for even, `1` for odd, or `N` if there is no parity check), and m is the number of stop pulses. For low-noise transmission lines, a `8-N-1` frame is popular.

Because the transmitter and receiver have independent clocks, there is no guarantee that these clocks will be synchronized — in fact, it is very likely that they are not.

- Even if the communications medium can support very high baud rates, the baud rate used for communications should still be significantly slower than the clock speed for the transmitter and receiver.
- When a start pulse is detected by a receiver, sophisticated receivers will start sampling the communications channel at a higher frequency than the baud rate (often by a factor of $\times 8$ or $\times 16$).
- These *oversampled* are averaged by the baud rate. This allows correcting for clock mismatch, and will also smooth out high-frequency noise.
- In a simpler system communications system, the receiver may merely wait for half a pulse-length before sampling.
- This attempts to sample each pulse in the middle, allowing a relatively large clock mismatch to be accommodated without introducing errors in the data.

UART is pretty great, with only one flaw: it is not that great.

- UART doesn't have a fool-proof method for preventing multiple systems from trying to transmit at the same time. Because of the communications frame width and the stop pulses, each

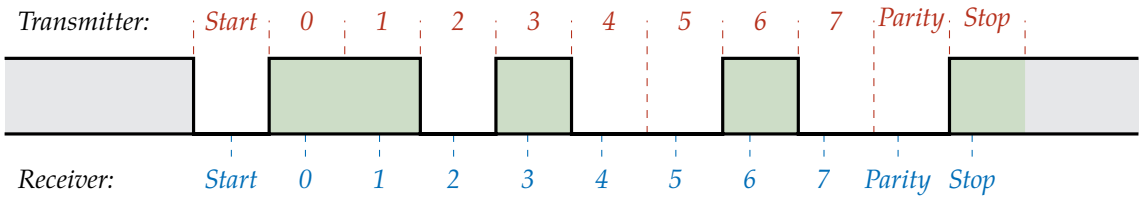


Figure 4: A receiver attempting to read transmitted pulses at an interval that is offset by half the pulse width. Although the baud rate is known to both the receiver and transmitter, in this case, the receiver's clock is slightly faster than the transmitter's clock. However by attempting to read the middle of each pulse, the data is still transmitted correctly. Here transfer is shown starting with the LSb and proceeding to the MSb.

system knows when a transmitter has finished sending a data packet, but that does not provide a mechanism for deciding which system gets to "talk" next.

- The worst-case scenario is if two systems try to transmit at exactly the same time — this will lead to each system competing to try and drive the channel to high or low voltage, creating a short circuit.

UART works best as a *full-duplex communication* protocol, where separate transmission/receiving channels are used by each device (see Figure 5). UART also works best when the communication network is limited to two systems: a two-system full duplex connection allows both systems to transmit and receive simultaneously.

- The second system is sometimes a hub that uses a different communication standard to communicate with a broader network.

- This is how dial-up internet works — the computer communicates with a modem using UART, while the modem converts information to/from an analog signal from the wired phone network.

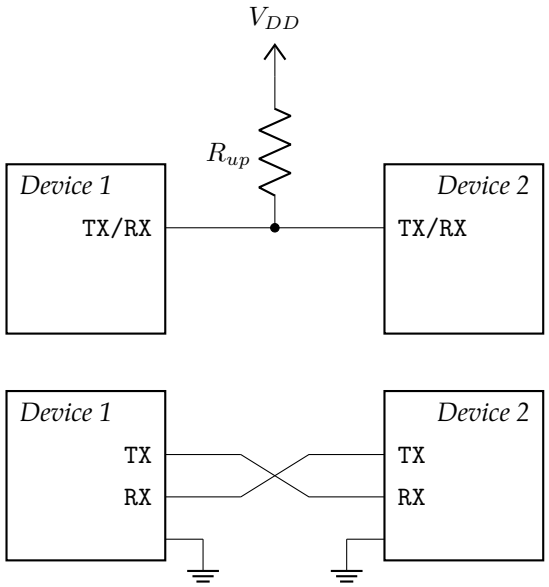


Figure 5: *Half-duplex* (top) and *full-duplex* (bottom) UART connections. The half-duplex channel must be pulled up to high voltage externally, and a short circuit will occur if both devices try to transmit at once.

Almost every microcontroller has the capability for UART communications. Using these communications systems is as simple as interacting with any other memory-mapped peripheral: simply write or read data to the relevant control registers.

Example: The DE1-SoC has a JTAG UART [4] that is implemented in the simulator. This peripheral is for communicating with a computer system. In the simulator, ASCII characters can be written or read from the JTAG window. The JTAG UART is memory-mapped to address `0xff201000`, and has the structure shown in Figure 6. In the simplest method of operating, data is written or read to the **data register**. For reading data, bit 15 of the **data register** indicates when a **frame** of data was successfully read by the UART.

[4] JTAG stands for "joint test action group", which is a protocol for testing and debugging microcontroller systems.

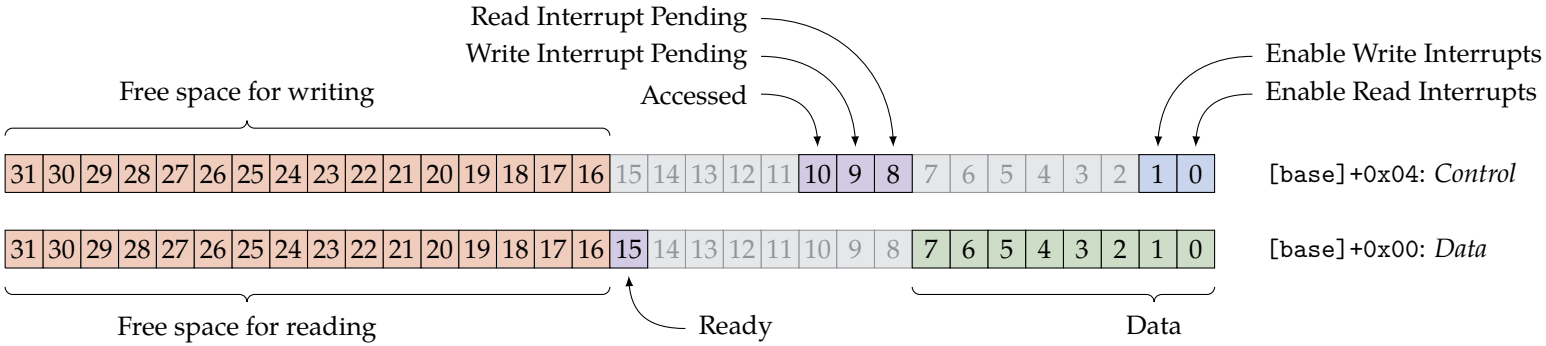


Figure 6: Structure of the JTAG UART controller on the DE1-SoC board. The base address is `0xff201000`. This UART reads/writes in units of bytes.

The JTAG UART contains on-chip memory, organized as a **queue** [5] This internal data structure makes transmitting/receiving data from the UART particularly simple.

[5] A **queue** is a "first-in, first-out" (FIFO) data structure for anyone who is rusty with these terms. A queue is in contrast to a **stack**, which is a "last-in, first-out" (LIFO) data structure.

- To send a reasonably large amount of data we can just continually write to the JTAG UART.
- The JTAG UART will worry about all the details in actually transmitting this data, and will do so in the order that it was received - each byte of data written to the JTAG UART is popped off the queue after it is successfully transmitted. We do not need to synchronize write operations to the UART with the baud rate for transmission, for example.
- The only thing we need to pay attention to is the "free space for writing" available in the JTAG UART write queue. This information is stored in the top 16 bits of the **control register**.
- If we write too much data to the JTAG UART, such that there is no memory space left in the write queue, any subsequent data written to the JTAG UART will be silently ignored.

If you want the microcontroller to receive data from the JTAG UART using the simulator, just click on the JTAG UART window and hammer away on your keyboard. The corresponding 8-bit ASCII values for each keypress will be transmitted into the read queue. [6]

A possibly more interesting piece of code is shown below: the Assembly language equivalent of a "hello world" program. If you run this code, you should see the text appear in the JTAG UART window on the simulator.

[6] Until the read queue runs out of space, as indicated by the top 16 bits of the **data register**.

```
.data
/* data structure for storing array
   of characters */
chars:
  .word 72 @H in ASCII
  .word 69 @E
  .word 76 @L
  .word 76 @L
  .word 79 @O
  .word 32 @space
  .word 87 @W
  .word 79 @O
  .word 82 @R
  .word 76 @L
  .word 68 @D

.text
.global _start

_start:
  ldr r0, =0xff201000 @JTAG UART address
  ldr r1, adr_chars @get array
  mov r4, #11 @loop counter

_loop:
  /* read next character from array, then
     increment array address */
```

```
  ldr r2, [r1], #4

  str r2, [r0] @write to JTAG UART
  subs r4, #1 @increment counter
  bne _loop

/* address of start of array */
adr_chars: .word chars
```

The above program uses an entire word to store each character. It would be more memory efficient to use just a byte. Notice, however, that writing a word (or a byte, for that matter) to the **data register** of the JTAG UART does not affect the data stored in that register — rather the bottom 8 bits of the data written to the UART **data register** are instead inserted into the write queue.

Inter-Integrated Circuit Protocol

The inter-integrated circuit protocol (I^2C or I2C) is a relatively simple communications protocol that addresses some of the problems with UART.

- I^2C still uses serial communication over a single-bit data channel (called SDA).
- I^2C also sends a common clock signal over a second channel (called SCL).
- I^2C allows any number of devices to be connected to these channels.
- I^2C operates in "master-slave" mode: one system in the network is designated as the **master**, all other systems are **slaves**. The master controls which system gets to send or receive data.
- The role of **master** can be transferred between systems.

The key to the I^2C protocol is that clock is not always "ticking": rather the master starts the clock to activate communication, and stops the clock when communication is completed. In this manner, the "start" and "stop" signals are clearly distinct from "data" signals: [7]

- Like UART, the data channel SDA is idles at a high voltage. The clock channel SCL also idles high.

[7] Unlike UART, where the start and stop pulses were only distinguished by their location in the communications frame. Any system that somehow misses a start or stop pulse in UART is unable to determine the state of the frame.

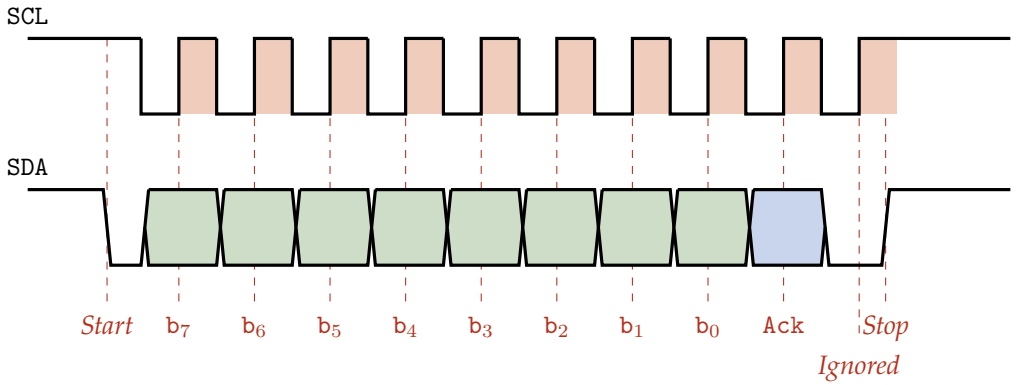


Figure 7: A byte of data transmitted by I^2C from MSb to LSb. The master only starts the clock (SCL) after the start signal, and stops the clock after the stop signal. After the acknowledgement pulse (ACK), any extra clocked pulses are ignored.

- A **falling edge** in SDA when SCL is high indicates the **start signal**.
- A **rising edge** in SDA when SCL is high indicates the **stop signal**.
- Data is otherwise transmitted on **rising edges** in SCL.

Data in I^2C is generally transmitted in **bytes**, and an acknowledgement from the receiver is expected after each transmission.

- After sending 8 bits, the transmitting system idles for the 9^{th} clock cycle.
- The receiving system is then expected to transmit a single bit of 0 to acknowledge that data was received. If SDA remains high, this indicates the receiving system did not receive the data.

- Another byte of data and an acknowledgement can then be sent, either resending the original information if reception was not acknowledged, or sending the next byte of data.
- If the stop signal is received before 9 clock cycles are completed, any excess pulses are discarded.

This transfer is shown in Figure 7.

As I^2C can involve several systems all using the same channels, it is organized similar to a microcontroller.

- Each system has a pre-defined address.
- The **master** sends the start signal and starts the clock, all the **slave** systems prepare to receive data.
- The **master** then transmits the address of a **slave** system and an instruction on whether that slave should transmit or receive data (either with the master or another specified slave).
- The data is then transmitted according to the master's instructions, after which the master sends the stop signal and stops the clock.

If the communications network consists of one microcontroller and several "dumber" devices (perhaps devices that primarily only receive data, like displays or printers; or devices that primarily only transmit data, like sensors or input devices) then the above describes a complete communications network. If the communications network has multiple microcontrollers, then often the **master**

can send the instructions for one of the **slaves** to take control of the clock, thus passing the role of master to another device.

- As long as there is only one **master**, all of the **slaves** follow instructions, and each device has a unique address, then multiple systems trying to transmit data simultaneously should never occur.

Like UART, almost every microcontroller has the capability for I^2C communications. Again, communicating with I^2C is a simple matter of looking up the **structure** of the I^2C controller and reading/writing data to that peripheral. The DE1-SoC development board has an I^2C bus connected to an audio and a video chipset. Decoding/encoding audio or video is a bit beyond the scope of this course, so we won't discuss the I^2C capabilities of the DE1-SoC any further.

Event-Driven Programming

The programming that we have learned in this course and (probably) in your other courses is mostly **sequential**:

- Program has a clear "beginning" and "end".
- Program can be mapped out step-by-step in a flow chart.
- Program computes *outputs* based on supplied *inputs*.
- Almost all functionality is implemented in software.

Since embedded systems deal primarily with hardware, we need to change things a bit. The programs for embedded systems typically involve simpler computational tasks, but a more complex set of inputs and outputs. This is best dealt with using **event-driven programming**.

Definition: An **event** is some input to the system generated by an external source. Exactly when this input occurs is often unpredictable.

The design strategies behind **event-driven programming** are somewhat different than those used for **sequential programming**.

- Program has a clear "beginning", in which all components are initialized.
- Program does not have an "end", it cycles through an endless loop.

- Continuously updates *outputs* based on frequently, and sometimes sporadically, changing *inputs*.
- With multiple *inputs*, several processes may occur almost simultaneously, or in an unpredictable order.
- Some, maybe even most, functionality is implemented in hardware. The CPU sometimes just needs to check that hardware is working correctly, and shuttle the *output* of one peripheral to the *input* of another.

Before going into detail on how **event-driven programming** is implemented, we will first introduce some notation and discuss the importance of **scheduling**.

Scheduling Formalism

Processing an event or a **task** requires three basic steps:

1. Start (or initialize) the task. Usually this is accomplished by writing the appropriate bits to a **control register**. This may be in software, for example by initializing a variable, or in hardware by turning on a peripheral.

2. Wait for the task to complete. If the task is entirely in software, there is no waiting. However if a task is done in hardware, it can take time (sometimes a variable amount of time) for an input to be acquired or an event to occur. Usually waiting is accomplished by monitoring some **status register** of the peripheral.

3. Process the result. If the task is entirely in software, this step can occur immediately after starting the task. However in hardware the result can only be processed once the hardware is finished — i.e., once waiting is complete. Usually processing is accomplished by reading from a **data register** of the peripheral. Often the data acquired will be used to start a new task.

We will use the following notation, which I have shamelessly cribbed from Prof. Ken McIsaac.

Notation: Each particular **task** is identified by a label `T`.

Notation: The **starting** (or initializing) phase of task `T` is labelled **S**[`T`].

Notation: The **waiting** phase of task `T` is labelled **W**[`T`].

Notation: The **processing** phase of task `T` is labelled **P**[`T`].

We need to implement **S**[`T`], **W**[`T`], and **P**[`T`] for every possible task `T`, and when the program is running **S**[`T`], **W**[`T`], and **P**[`T`] will occur strictly in that order. However:

- We do not necessarily have to code **S**[`T`], **W**[`T`], and **P**[`T`] in that particular order.
- Sometimes we do not need to code one or more of these at all: they may occur naturally as a consequence of some other task, or may be accomplished entirely by hardware.
- We can usually assume that anything implemented in software will complete "instantly" in comparison to anything in hardware: peripherals often respond on the order of second to microseconds, while the CPU clock is considerably faster.

For a desired functionality and a given set of peripherals, our job is to:

- Enumerate all the tasks that can occur.
- Implement **S**[`T`], **W**[`T`], and **P**[`T`], as necessary, for each task `T`.
- **Schedule** how, and when, each stage for each task will proceed (wherever possible and predictable).

Example: Stating the Problem

We are given two analog signals, v_1 and v_2. Our job is to design a system that will provide the following:

- The time-average value of v_1 ($\langle v_1 \rangle$, output at $1\,\mathrm{Hz}$.
- The truth-value of the comparison $v_1 > v_2$, output at $1\,\mathrm{kHz}$.

We are provided with the following hardware:

- A suitable microcontroller (presumably an ARM Cortex®A9).
- One timer with a $100\,\mathrm{MHz}$ clock.
- One analog-to-digital converter (ADC) with at least 2 channels.

In this example, the ARM LTC2308 will not be used. Rather, we will use a model ADC that only has three registers:

- A **control register** at the ADC base address. Writing a number to this register will start ADC sampling from the corresponding channel.
- A **status register** at one word higher than the ADC base address (`[base] + 0x04`). The LSb of this register is set whenever the ADC is done converting.
- A **data register** two words higher than the ADC base address (`[base]+0x08`). This stores the converted value from whichever channel was selected by the **control register**. Reading this value automatically clears the **status register**.

There are ADCs available that are similar to this model. Why are we using it? Because this example is for emphasizing scheduling events, and this kind of ADC requires v_1 and v_2 to be sampled separately. With the LTC2308, all channels can be sampled simultaneously.

- Furthermore, the ADC on the simulator isn't particularly useful so this model can't be meaningfully tested.

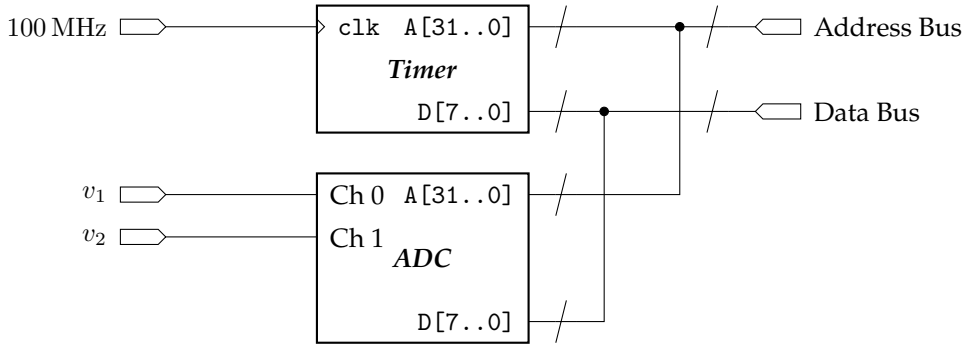


Figure 8: Simplified schematic of the peripherals used in this example.

From the system requirements we have two explicit **waiting** times: a $1\,\mathrm{s}$ time for the average of s_1, and a $1\,\mathrm{ms}$ time for the $s_1 > s_2$ comparison. There is also a third wait time: the time for the ADC to operate. [8] This last wait time is unknown, but we can assume it is significantly faster than $1\,\mathrm{ms}$. [9]

[8] Remember from our previous studies that *all* implementations of ADC require time to complete.

[9] Otherwise the design requirements cannot be met with the hardware provided.

Example: Defining the Tasks

With this in mind, one way to design the system is to base it around four **tasks**:

Task 1*:* Measure the passage of each $1\,\mathrm{ms}$. This is straightforward; it is clear we should use the timer.

S[1]*:* Since the timer runs at $100\,\mathrm{MHz}$, and we want to measure $1\,\mathrm{kHz}$, we write $100\,000$ to the **control register** of the timer.

W[1]*:* Monitor the **status bit** of the timer until the countdown is complete.

P[1]*:* Compute $v_1 > v_2$ and write the result to a **output register** (really we only need 1 bit).

Task 2*:* Measure the passage of each $1\,\mathrm{s}$. This would be straightforward if we had a second timer, but since we only have one and it is in constant use, we need to implement this in software.

S[2]*:* Initialize a **counter** and a **running sum** to zero.

W[2]*:* Monitor 1, each time it completes increment the **counter** and add the current value of v_1 to the **running sum**. Stop once the **counter** reaches 1000 $(1000\,\mathrm{ms} = 1\,\mathrm{s})$.

P[2]*:* Compute $\langle v_1 \rangle$ by dividing the **running sum** by 1000 and writing the result to a **output register**.

Task 3: Sample v_1 at least once each $1\,\mathrm{ms}$, in order to compute $v_1 > v_2$ and $\langle v_1 \rangle$. Since we have only one data register on the ADC, we must have separate tasks for reading v_1 and v_2.

S[3]: Write the appropriate flag to the **control register** of the ADC to start sampling v_1.

W[3]: Monitor the **status register** of the ADC until it indicates the conversion is complete.

P[3]: Store the digitized version of v_1 in a register or memory.

Task 4: Same as task 3 but for v_2. Sample v_2 at least once each $1\,\mathrm{ms}$, in order to compute $v_1 > v_2$.

S[4]: Write the appropriate flag to the **control register** of the ADC to start sampling v_2.

W[4]: Monitor the **status register** of the ADC until it indicates the conversion is complete.

P[4]: Store the digitized version of v_2 in a register or memory.

It is clear that **P**[3] and **P**[4] are needed for **P**[1], and **P**[3] is also needed for **P**[2]. However this does not mean that we must complete tasks 3 and 4 before **S**[1] and **S**[2], since each task has a significantly different waiting time. Furthermore:

- The timing and order of sampling v_1 and v_2 within the $1\,\mathrm{ms}$ window is ambiguous, so we have significant leeway when scheduling the tasks.

[10] Here "slow" is relative to all other tasks.

- **W**[2] is implemented in software, but it is slow: the microcontroller does not have to continuously monitor **W**[1]. [10]
- **W**[1] and **W**[3] (or **W**[1] and **W**[4]) occur in hardware, and can happen simultaneously.

Example: Coding the Tasks

We can implement the tasks in the above examples as subroutines. This is convenient because we will discuss several different ways of scheduling these tasks, and if each is written as a portable subroutine then the only difference will be the way each is called from the main program. To interact with hardware, we need:

- The code for **S**[1], to initialize and start the timer for counting the 1 ms interval.

```
/* start the timer, single countdown
 * assume timer period is in r0 */
_timer_start:
  /* save state */
  push {r1, r2}
  /* initialize timer */
  ldr r1, adr_timer @load timer address
  str r0, [r1] @clear timeout
  str r0, [r1, #8] @set low period
  lsr r0, #16
  str r0, [r1, #12] @set high period
  /*start timer*/
  mov r2, #4
  str r2, [r1, #4] @start timer
  /* restore state and return */
  pop {r1, r2}
  bx lr
```

Here r0 is not pushed to the stack as it is unchanged by the subroutine, and is expected as a parameter. Previously I mentioned that "good programming practice" was to only push registers r4 and higher to the stack, and to assume that r0 to r3 would be modified by a subroutine (as possible outputs). However I can't be bothered with that in this simple example.

- The code for **W[1]**, to wait until the timer has completed a 1 ms interval.

```
/* wait for the timer to run down */
_timer_wait:
  /* save state */
  push {r0, r1}
  ldr r1, adr_timer @load timer address
  /* loop until finished */
_timer_wait_loop:
  ldr r0, [r1] @get timer status
  cmp r0, #2 @check if it is still running
  beq _timer_wait_loop
  /* restore state and return */
  pop {r0, r1}
  bx lr
```

Note: If the status register of the timer is #2, the timer is still counting down. If the status register is #1, it has finished (a "timeout" occurred). Assuming the timer was properly initialized and started, these should be the only two options. However if the timer was set to continuous counting, or the timer is turned off, the status register

might be #3 or #0, respectively. In those cases a timeout will never occur, so the subroutine should return immediately. This is why I am using a "branch if equal to #2" rather than a "branch if not equal to #1" condition.

- The code for **S**[3] and **S**[4], to initialize and start the ADC to sample v_1 or v_2.

```
/* start the ADC
 * assume channel is in r0 */
_adc_start:
  /* save state */
  push {r1, r2}
  ldr r1, adr_adc @load adc address
  /* start ADC */
  ldr r2, [r1, #8] @clear any previous junk
  str r0, [r1] @start sampling channel
  /* restore state and return */
  pop {r1, r2}
  bx lr
```

- The code **W**[3] and **W**[4], to wait until the ADC has completed sampling the channel, and obtain the sampled value.

```
/* wait for the ADC to complete
 * then store result in r0 */
_adc_wait:
  /* save state */
```

```
  push {r1}
  ldr r1, adr_adc @load adc address
  /* loop until finished */
_adc_wait_loop:
  ldr r0, [r1, #4] @get adc status
  cmp r0, #1 @check if it is still converting
  bne _adc_wait_loop
  ldr r0, [r1, #8] @get sampled value
  /* restore state and return */
  pop {r1}
  bx lr
```

Note that this subroutine returns the sampled value in `r0`.

- The code **P[1]**, which outputs according to whether or not $v_1 > v_2$.

```
/* process ADC values
 * assume v1 is in r0 and v2 in r1
 * don't modify r0 as it is needed later*/
_process_1:
  /* save state */
  push {r2}
  ldr r2, adr_output_gt @address for output
  /* check v1 > v2 */
  cmp r0, r1
  movhi r1, #1
  ldrhi r1, [r2] @output 1 if v1>v2
  movls r1, #0
```

```
  ldrls r1, [r2] @output 0 if v1<v2
  /* restore state and return */
  pop {r2}
  bx lr
```

- The code **P[2]**, which outputs the average $\langle v_1 \rangle$. Note that this uses the division mnemonic `udiv` (for "unsigned division"), which has the same format as addition and subtraction.

```
/* process average of v1
 * assume running total of v1 in in r0 */
_process_2:
  /* save state */
  push {r1, r2}
  ldr r2, adr_output_avg @address for output
  ldr r1, =0x3e8 @hex for 1000
  /* compute average of v1 and output*/
  udiv r0, r1
  str r0, [r2]
  /* restore state and return */
  pop {r1, r2}
  bx lr
```

In all of the above subroutines, it is expected that the memory addresses for the timer, ADC, and whatever is used to output $v_1 > v_2$ and $\langle v_1 \rangle$ are already assigned to the labels `adr_timer`, `adr_adc`, `adr_output_gt`, and `adr_output_avg`, respectively. The remaining code can be done in the main program.

Example: Scheduling

It should be fairly clear that task 1 sets the pace of the program, as **W**[1] is essentially used to clock reading v_1 and v_2, and outputting $v_1 > v_2$ and $\langle v_1 \rangle$. For task 2, **W**[2] only needs to be checked after **P**[1], as **W**[2] is just incrementing a counter. Completing a step of **W**[2] (i.e., after incrementing the counter) or completing **P**[1] can be used to begin **S**[3], and since both tasks 3 and 4 share the same hardware, **S**[4] can begin no earlier than **P**[3].

One possible schedule for this program is shown in Figure 9. The size of each phase is only a qualitative representation of the execution time; the diagram is not to scale.

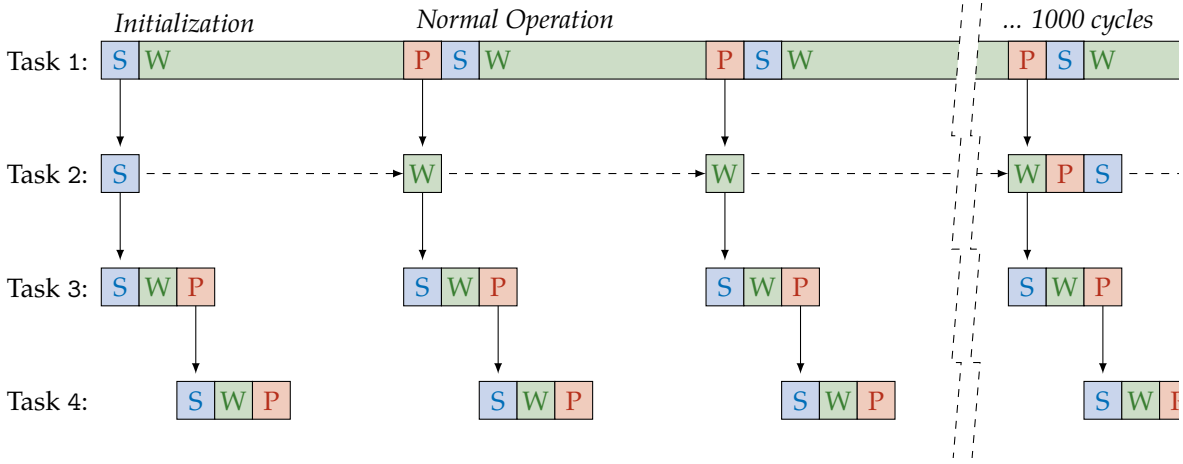


Figure 9: One way to schedule the tasks in this example. During the initialization phase, completing **S**[1] triggers **S**[2], which then triggers **S**[3]. During normal operation, **P**[1] always triggers **W**[2], and **W**[2] always triggers **S**[3]. **S**[4] is always triggered by **P**[3].

This schedule shows the order the phases in each task are executed, and how completing a phase in one task can trigger the start of a phase in a different task. This diagram does not show how data is passed back between tasks: for example, we know the data read in

from v_1 and v_2 as a result of **P**[3] and **P**[4] is necessary to complete **P**[1], but this is not shown in the diagram.
When determining these schedules, we usually assume (as was done here) that everything in software executes almost instantly. Unless the software is rendering HD graphics or some similar computationally intensive task, [11] this is a good approximation: the CPU clock speed is almost always several orders of magnitude faster than any peripherals.

[11] In which case, perhaps a microcontroller is not the best architecture to use?

- This is the reason why I drew **S**[4] concurrent with **P**[3] in Figure 9, for example. Technically, **S**[4] cannot begin until after **P**[3] is finished, so there should be horizontal offsets in Figure 9.

- In fact, as the CPU can only do one thing at a time, technically there should be only *one* CPU job at a given time (i.e., most **S**[n] and **P**[n] stages, and any non-hardware controlled **W**[n] stage).

- But I think that kind of precise spacing makes the schedule harder to read.

The following source code implements this schedule using the subroutines previously defined. I assume the appropriate number of timer cycles for a $1\,\mathrm{s}$ interval is associated with the label `count_time`.

```
/* initialization */
ldr r0, count_time
ldr r3, =0x3e8 @r3 = 1000 cycles, S[2]

/* program loops endlessly */
_main_loop:

  /* start some stuff */
  bl _start_timer @start the timer, S[1]
  mov r4, #0 @v1 accumulator, S[2]

  /* get v1 from ADC ch0 */
  mov r0, #0 @ch0 for ADC = v1
  bl _adc_start @start conversion, S[3]
  bl _adc_wait @wait for completion, W[3]
  mv r2, r0 @save v1, P[3]

  /* get v2 from ADC ch1 */
  mov r0, #1 @ch1 for ADC = v1
  bl _adc_start @start conversion, S[4]
  bl _adc_wait @wait for completion, W[4]
  mov r1, r0 @save v2, P[4]

  /* wait for rest of 1ms */
  bl _timer_wait @W[1]
```

```
/* process v1 > v2 */
mov r0, r2 @shift v1 back to r0
bl _process_1 @ P[1]

/* accumulate sum of v1 */
add r4, r1  @sum of v1, W[2]
/* check if 1000 intervals */
subs r3, #1  @1ms intervals, W[2]
bne _main_loop

mov r0, r4 @shift accumulated v1 to r0
bl _process_2 @ P[2]
ldr r3, =0x3e8 @reset 1000 cycle counter

/* always loop endlessly */
b _main_loop
```

I supposed based on this code that **S**[2] happens before **S**[1], which is not quite as indicated in Figure 9, but since all of these are steps in software which we assumed happen simultaneously, it doesn't really matter.

Example: Simplifying the Schedule

As the parts of a given task that are implemented code are completed almost instantly, and assuming that the ADC conversion wait time (**W**[3] and **W**[4]) is much shorter than $1\,\mathrm{ms}$, it is not actually necessary to code **W**[4]. Instead, we can run **S**[4] after the digitized v_1 is read off the ADC in **P**[3], then forget about it until **W**[1] has finished an the digitized v_2 is needed.

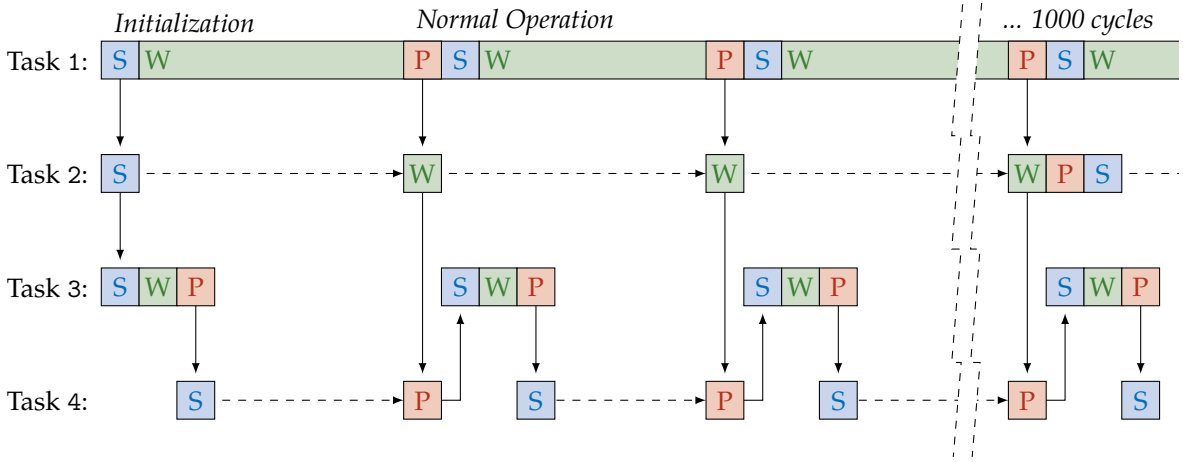


Figure 10: The second method of scheduling the tasks in this example. **W**[4] has be removed, and **P**[4] is now synchronized with task 2.

This revised schedule, shown in Figure 10, requires some feedback between **P**[1] and **P**[4], since the **P**[1] is used indirectly to start **P**[4], but data from **P**[4] is needed to complete **P**[1] (calculate $v_1 > v_2$; the value v_1 is already available from **P**[3]). As this is all done in software, it is trivial (just passing parameters). In code, this revised schedule only requires idle looping while checking the ADC for the conversion of v_1 (**W**[3]), as shown below.

```
/* initialization */
ldr r0, count_time
ldr r3, =0x3e8 @r3 = 1000 cycles, S[2]

/* program loops endlessly */
_main_loop:

  /* start some stuff */
  bl _start_timer @start the timer, S[1]
  mov r4, #0 @v1 accumulator, S[2]

  /* get v1 from ADC ch0 */
  mov r0, #0 @ch0 for ADC = v1
  bl _adc_start @start conversion, S[3]
  bl _adc_wait @wait for completion, W[3]
  mov r2, r0 @save v1, P[3]

  /* start converting v2 from ADC ch1 */
  mov r0, #1 @ch1 for ADC = v1
  bl _adc_start @start conversion, S[4]

  /* wait for rest of 1ms */
  bl _timer_wait @W[1]

  /* now get v2 */
  ldr r0, adr_adc
  ldr r1, [r0, #8] @P[4]
```

```
/* process v1 > v2 */
mov r0, r2 @shift v1 back to r0
bl _process_1 @ P[1]

/* accumulate sum of v1 */
add r4, r1  @sum of v1, W[2]
/* check if 1000 intervals */
subs r3, #1  @1ms intervals, W[2]
bne _main_loop

mov r0, r4 @shift accumulated v1 to r0
bl _process_2 @ P[2]
ldr r3, =0x3e8 @reset 1000 cycle counter

/* always loop endlessly */
b _main_loop
```

The only difference between this code and the previous one is that there was no second `bl _adc_wait` for the second ADC conversion, and instead the value was read directly from the ADC without checking whether conversion had completed.

Example: Introducing an Extra Task

Since this program requires shared use of a single ADC, and has output clocked at $1\,\mathrm{ms}$, another option for scheduling is to introduce a new task:

Task `0`*:* Measure the passage of each $0.5\,\mathrm{ms}$ using the timer.

S[`0`]*:* Since the timer runs at $100\,\mathrm{MHz}$, and we want to measure $2\,\mathrm{kHz}$, we write $50\,000$ to the **control register** of the timer.

W[`0`]*:* Monitor the **status bit** of the timer until the countdown is complete.

P[`0`]*:* Increment a **half-ms counter** for monitoring $0.5\,\mathrm{ms}$ ticks.

This new task requires us to redefine task `1`:

Task `1`*:* Measure the passage of each $1\,\mathrm{s}$. This is now implemented in software.

S[`1`]*:* Initialize a **half-ms counter** to zero.

W[`1`]*:* Monitor task `0`, each time it completes increment the **half-ms counter**. Stop once the **half-ms counter** reaches 2.

P[`1`]*:* Compute $v_1 > v_2$ and write the result to a **output register**.

This extra task simplifies how we use the ADC: we can sample v_1 and v_2 each after one millisecond, but separated by a half-millisecond.

Example: Scheduling at the Half-Millisecond

One possible schedule for this program with the $0.5\,\mathrm{ms}$ counter implemented in task `0` is shown in Figure 11.

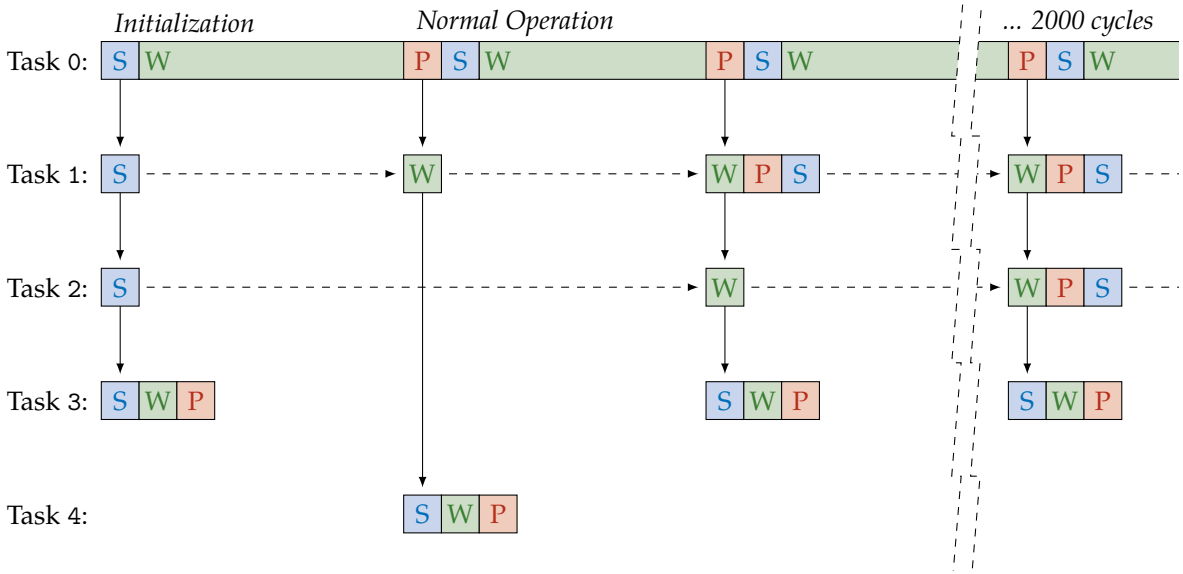


Figure 11: Another way of scheduling the tasks in this example, this time using a $0.5\,\mathrm{ms}$ timer. Now ADC for v_1 (**S**[3]) is triggered by the "even ticks" of the $0.5\,\mathrm{ms}$ timer, and ADC for v_2 (**S**[4]) is triggered by the "odd ticks".

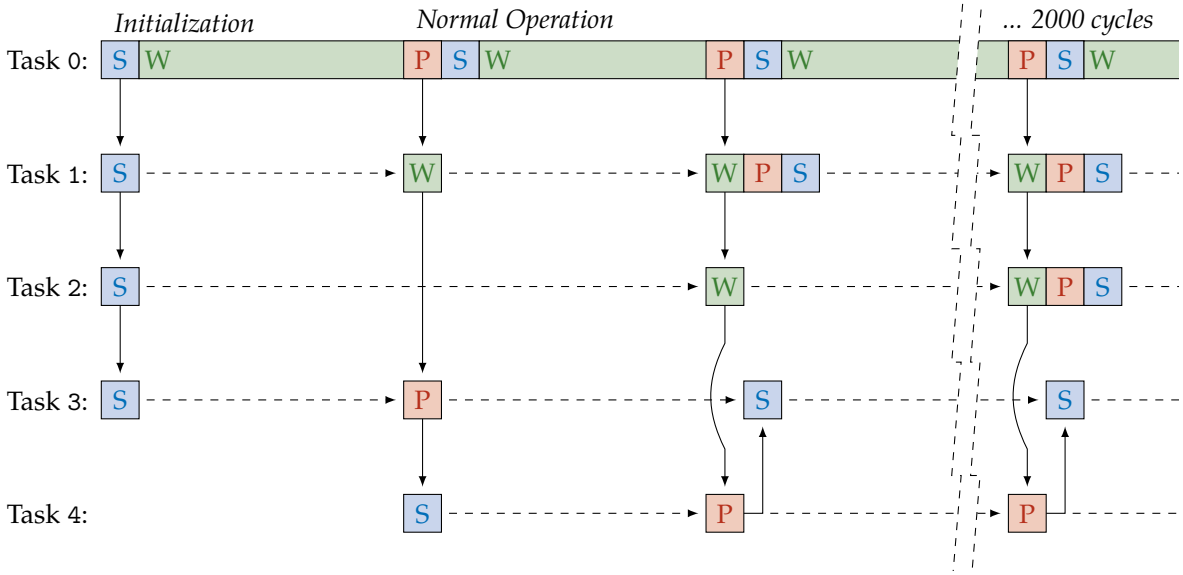
Here we need to program some logic to trigger **W**[1] and **S**[4] whenever the $0.5\,\mathrm{ms}$ counter is an odd number (the accumulating counter has LSb $= 1$), and to instead trigger **W**[2] and **S**[3] whenever the $0.5\,\mathrm{ms}$ counter is an even number (LSb $= 0$). Often we don't need to implement anything in **S**[0] beyond the first initialization of the timer, as many timers can count continuously (and roll over to restart).

```
/* initialization */
ldr r0, count_half_time
ldr r3, =0x7d0 @r3 = 2000 cycles, S[2]

/* program loops endlessly */
_main_loop:
  /* start some stuff */
  bl _start_timer @start the timer, S[1]
  mov r4, #0 @v1 accumulator, S[2]

  /* is this an even or odd tick */
  ands r1, r3, #1 @r1 is not important
  bnz _odd_tick @branch if non-zero
_even_tick:
  /* get v1 from ADC ch0 on even ticks */
  mov r0, #0 @ch0 for ADC = v1

  bl _adc_start @start conversion, S[3]
  bl _adc_wait @wait for completion, W[3]
  mv r2, r0 @save v1, P[3]

  /* wait for rest of 0.5ms */
  bl _timer_wait @W[1]

  b _clean_up
_odd_tick:
  /* get v2 from ADC ch1 on odd ticks*/
```

```
  mov r0, #1 @ch1 for ADC = v1

  bl _adc_start @start conversion, S[4]
  bl _adc_wait @wait for completion, W[4]
  mv r1, r0 @save v2, P[4]

  /* wait for rest of 0.5ms */
  bl _timer_wait @W[1]

  /* process v1 > v2 */
  mov r0, r2 @shift v1 back to r0
  bl _process_1 @ P[1]
_clean_up:
  /* accumulate sum of v1 */
  add r4, r1  @sum of v1, W[2]
  /* check if 2000 intervals */
  subs r3, #1  @0.5ms intervals, W[2]
  bne _main_loop
  mov r0, r4, lsr #1 @shift accumulated v1 to r0
  @ notice use of lsr #1 to divide by 2 so
  @ we can reuse old divide-by-1000 routine
  bl _process_2 @ P[2]
  ldr r3, =0x7d0 @reset 2000 cycle counter

  /* always loop endlessly */
  b _main_loop
```

Example: Removing Waits

With a $0.5\,\mathrm{ms}$ counter we can further simplify the schedule by removing the waiting phases for the ADC (**W**[3] and **W**[4]). As long as the ADC takes less than $0.5\,\mathrm{ms}$ to complete, [12] the digitized value will be ready and waiting at to completion of each $0.5\,\mathrm{ms}$ tick.



One of the benefits of this schedule is that there is only one **waiting** phase that requires frequent checking for completion: **W**[0]. The timing for all other tasks are based on these $0.5\,\mathrm{ms}$ ticks.

[12] Again, the ADC must be this fast or we cannot read in two analog values in series at $1\,\mathrm{kHz}$.

Figure 12: A final way of scheduling the tasks in this example, again using a $0.5\,\mathrm{ms}$ timer. Here the wait times for the ADC conversions are all removed, it is assumed that the ADC is finished by the end of each $0.5\,\mathrm{ms}$ "tick".

```
/* initialization */
ldr r0, count_half_time
ldr r3, =0x7d0 @r3 = 2000 cycles, S[2]

/* program loops endlessly */
_main_loop:
  /* start some stuff */
  bl _start_timer @start the timer, S[1]
  mov r4, #0 @v1 accumulator, S[2]

  /* is this an even or odd tick */
  ands r2, r3, #1 @r2 is not important
  bnz _odd_tick @branch if non-zero
_even_tick:
  /* read v2 from ADC from previous */
  ldr r0, adr_adc
  ldr r6, [r0, #8] @store in high register, P[4]

  /* start converting v1 from ADC ch0 */
  mov r0, #0 @ch0 for ADC = v1
  bl _adc_start @start conversion, S[3]

  b _clean_up
_odd_tick:
  /* read v1 from ADC from previous */
  ldr r0, adr_adc
  ldr r5, [r0, #8] @store in high register, P[3]
```

```
  /* start converting v2 from ADC ch1 */
  mov r0, #1 @ch1 for ADC = v2
  bl _adc_start @start conversion, S[4]
_clean_up:
  /* wait for rest of 0.5ms */
  bl _timer_wait @W[1]

  /* process v1 > v2 */
  mov r0, r5 @shift v1 back to r0
  mov r1, r6 @shift v2 back to r1
  bl _process_1 @ P[1]

  /* accumulate sum of v1 */
  add r4, r1  @sum of v1, W[2]
  /* check if 2000 intervals */
  subs r3, #1  @0.5ms intervals, W[2]
  bne _main_loop
  mov r0, r4, lsr #1 @shift accumulated v1 to r0
  @ notice use of lsr #1 to divide by 2 so
  @ we can reuse old divide-by-1000 routine
  bl _process_2 @ P[2]
  ldr r3, =0x7d0 @reset 2000 cycle counter

  /* always loop endlessly */
  b _main_loop
```

Example: Conclusions

The code provided above isn't necessarily the best implementation of the various schedules, and the reliance on short (and somewhat pointless) subroutines is of dubious value. The reason for using so many subroutines was to demonstrate how much of the actual code remains the same for the various methods of scheduling — the main thing that changes are the conditions which cause that code to execute.

It is not correct to say that one schedule is "better" than the other. There is something to be said for the last example: as it has only one *idle* state, [13] there are fewer ways the microcontroller can mysteriously freeze. However the downside is, that by *assuming* successful ADC completion, sometimes the values assumed to be v_1 or v_2 will actually be garbage.

- This does in fact happen in the final example given above, as the value of v_2 is read on even $0.5\,\mathrm{ms}$ "ticks". However the very first time the program runs, the ADC had not been started yet to sample v_2, so this initial value is garbage.
- Usually these things are more trouble to fix than they are worth. A stream of data at $1\,\mathrm{kHz}$ should be expected to have some bad values.

[13] I.e., waiting for the $0.5\,\mathrm{ms}$ timer to complete.