SlowSoftSerial design notes

Paul Williamson, paul@mustbeart.com

On the Teensy 3.5 (used on Cadetwriter[1]), the hardware baud rate generator has a relatively high minimum rate, such that 1200 baud is the slowest standard baud rate it can handle. The typewriter mechanism peaks out at about 16cps, which would be 160 baud (using 8N1), but the average rate for a mix of characters is lower. A standard baud rate of 150 or even 110 would be a better match for the Cadetwriter's speed.

In order to support slower baud rates on the Cadetwriter, we have the following requirements:

- interrupt latency of no more than a handful of microseconds, worst case, to avoid interfering with the keyboard matrix scan and "fake" keypress generation
- use of pin 0 for RX and pin 1 for TX, to work with the existing board layout and cabling
- simultaneous transmit and receive
- support for data word length, parity, and stop bits settings of 7O1, 7E1, 8N1, 8O1, 8E1, 8N2, 8O2, and 8E2.

Unfortunately, the only software serial library that's officially supported[2] under Teensyduino is AltSoftSerial, which meets the first and third requirements handily but requires the use of fixed pins 20 and 21 for RX and TX (because it gets fancy with the timer hardware) and only supports 8N1. The normal SoftwareSerial library (based on NewSoftSerial) doesn't receive on Teensy 3.5, and the ancient OldSoftSerial adds way too much interrupt latency.

So, we need a new serial implementation. It doesn't need to perform well at high baud rates, since we have the hardware UART for that. It just needs to handle slow baud rates without impacting interrupt latency much. For our purposes, it only needs to work on the Teensy 3.5 hardware, which is the MK64FX512. We have pin change interrupts, and we have a crazy variety of timer hardware to work with. We can use the simple Periodic Interrupt Timers (PIT) to do what we need.

We'll use one timer for transmit and another for receive, keeping them independent of each other. Teensyduino[3] provides a nice IntervalTimer abstraction that does what we need, and more.

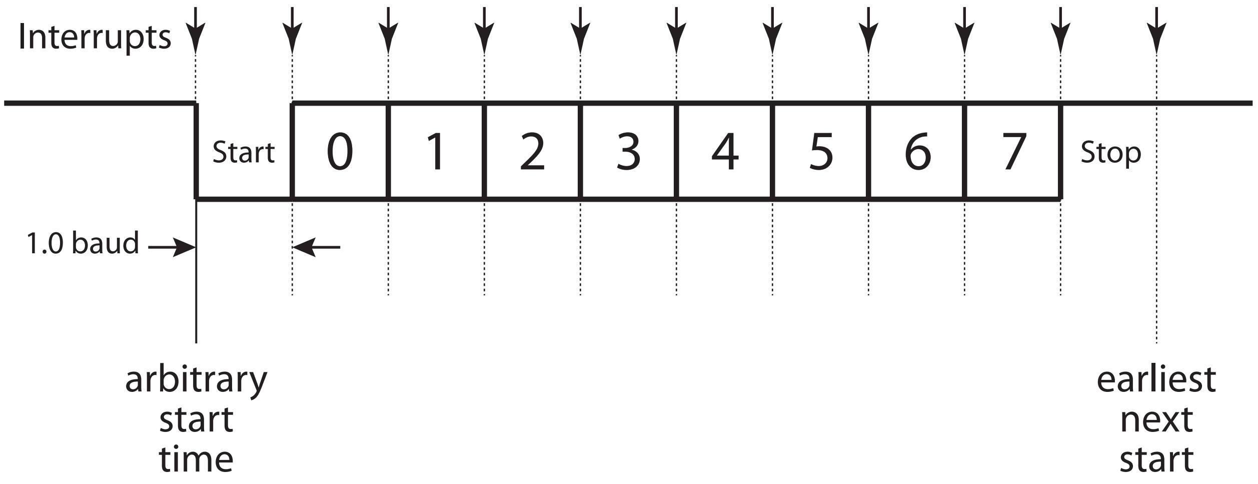
We do not need to support receive flow control handshaking for the Cadetwriter application, since there is already application-layer support for that. The application layer cannot easily implement transmit flow control handshaking, though, since it has no way to keep characters already buffered for transmit from being transmitted. There's no handshaking in the standard Arduino serial API, but Teensyduino defines functions for hardware flow control. We will do the same.

[1] https://github.com/IBM-1620/Cadetwriter
[2] Library support according to https://www.pjrc.com/teensy/td_libs.html
[3] https://www.pjrc.com/teensy/teensyduino.html

For the most elegant integration with Cadetwriter, and for use on other Teensy projects, SlowSoftSerial is packaged as an Arduino library. For other projects, we support a wider range of serial parameters and much of the rest of the Arduino serial API[4].

Transmit processing

Transmit processing is relatively easy. We can start whenever we want, and count whole bauds to time all the necessary signal transitions. Here it is for the 8N1 case:



When idle, nothing is happening. No interrupts are enabled.

The transmitter's interface to the rest of the software is through the transmit buffer. Characters in the transmit buffer are stored in an as-transmitted format that includes the parity bit (if any) and the stop bit(s) but not the start bit. The transmit buffer supports the following operations, some of which map directly onto Arduino API methods:

- void write(byte chr)
  Add a character to the end of the transmit buffer. If space is not available immediately, wait until space becomes available[5]. If the serial settings call for parity, translate the character to include the required parity bit. Add stop bit(s). If transmitting is enabled but not currently running, begin transmitting.
- int availableForWrite()
  Returns the number of characters of space currently available in the transmit buffer.

[4] https://www.arduino.cc/reference/en/language/functions/communication/serial/
[5] Blocking write functions are standard in Arduino libraries.

To begin transmitting, we simply assert the Tx line (creating the start bit) and set a repeating timer with a duration of 1.0 baud. Initialize the bit count in accordance with the current serial word size and stop bit setting. Here is pseudocode for that operation, starting with a little helper macro that we will use again in the interrupt handler:

```
initialize_transmit_variables:
      data_word = character from transmit buffer
      bit_count =  9 for 7E1, 7O1, or 8N1
                  10 for 8E1, 8O1, or 8N2
                  11 for 8E2 or 8O2, etc.

begin_transmitting:
      set a periodic timer of duration 1.0 baud
      assert Tx line
      initialize_transmit_variables()
```

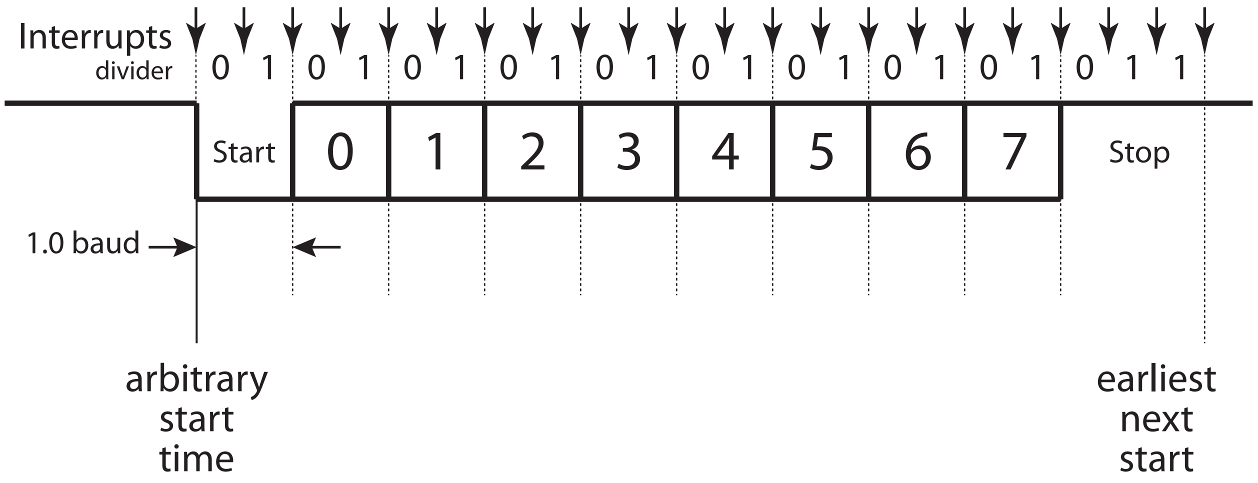
Here's pseudocode for the transmit timer interrupt handler:

```
tx_timer_interrupt_handler:
      if bit_count > 0
            set Tx line to the least significant bit of data_word
            shift character right one bit
            decrement bit_count
      else if transmit_enabled
            if (chr = next_tx_char()) != 1
                  assert Tx line (making a new start bit)
                  initialize_transmit_variables()
      else
            disable timer interrupt
```

In the actual implementation, the write function does not directly assert the Tx line. It places the character into the transmit buffer and starts up the periodic timer interrupt. The transmission actually begins when the periodic timer expires for the first time. This small extra delay is a small price to pay for the resulting cleanliness of the code.

Implementation of 1.5 Stop Bits Case

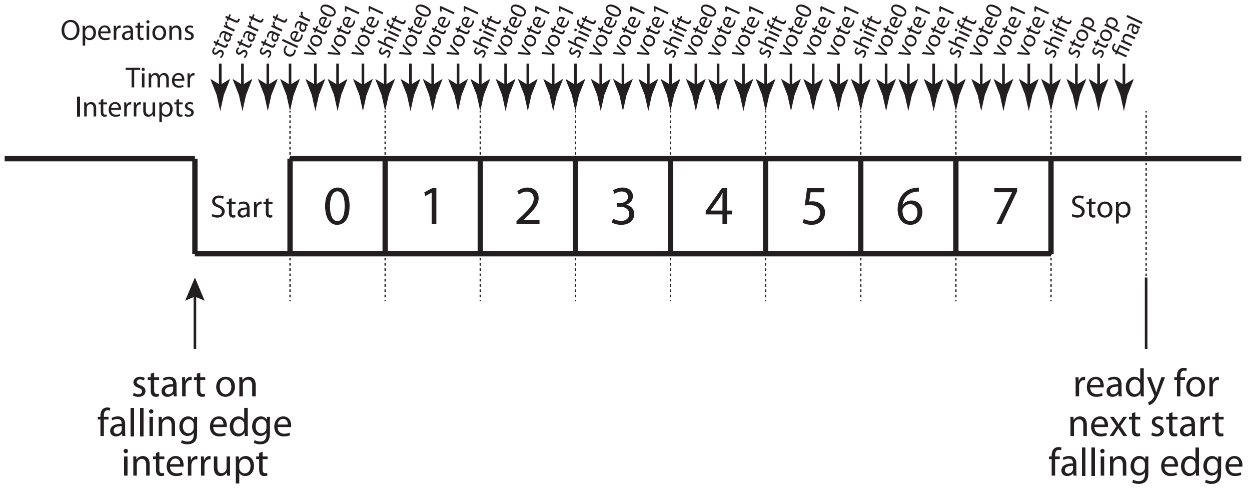
In order to implement the configurations with 1.5 stop bits, we need to run the periodic timer twice as fast. Here it is for the 8N1.5 case:



Receive processing

Receive processing is a little bit more involved, because we don’t control the byte timing and we want to be a little bit robust to noise and framing errors.

When idle, we're just waiting for a falling edge pin change interrupt on RX. The exact time when that falling edge occurs will be the reference point for timing the reception of that one character. Starting at the falling edge, we will take a periodic timer interrupt at exactly four times the baud rate (duration of 0.25 bauds). This will enable us to sample each bit three times, at the 25%, 50%, and 75% marks during the nominal timing of the bit. The interrupts that fall on a bit transition can be used for other bookkeeping. Here is the 8N1 case:



The operation on each individual interrupt is controlled by a schedule kept in a table, for both simplicity and speed at interrupt time. There are only a few operation types. Exactly one runs on each timer interrupt, and each one is pretty simple and fast to execute. The slowest operation defines the worst-case impact on interrupt latency for other tasks, so we work hard to keep the operations quick.

Much of the logic is involved with detecting framing errors: Rx high during a start bit or low during a stop bit, or Rx changing during a data bit. In all those cases, we simply stop working on the current invalid character, disable the timer interrupts, and re-enable the falling edge interrupt. When that falling edge arrives, we start afresh. The same procedure applies after successful reception of a character.

Falling Edge Interrupt Handler

Between incoming characters, no processing takes place. The Rx line is enabled for a falling edge pin change interrupt. This handler sets up the timing for receiving the character.

```
falling_edge_interrupt_handler:
      start a periodic timer with period 0.25 baud
      disable falling edge interrupt
      next_operation_index = 0
```

Timer Master Interrupt Handler

When the timer interrupt fires, we look up the appropriate operation handler in the schedule by next_operation_index and run it. We advance next_operation_index so the operation handlers don't have to, even though that's wasted effort in some cases.

```
receive_timer_master_interrupt_handler:
      op = schedule[next_operation_index++]
      *op()
```

Start Operation

The first three timer interrupts occur at the 25%, 50%, and 75% marks during what we assume is the start bit. All three must find the Rx line low, or else there's been a timing error that we need to recover from.

```
start_op:
      if Rx line is high
            disable timer interrupt
            enable falling edge interrupt
```

Clear Operation

The timer interrupt at the end of the start bit is used to set up initial conditions for receiving the data bits to follow.

```
clear_op:
      data_word = 0
```

Vote0 Operation

The first timer interrupt during each data bit records the value of the Rx line for that bit.

```
vote0_op:

      bit_value = Tx line state
```

Vote1 Operation

The second and third timer interrupts during each data bit check that the value of the Rx line still matches the value recorded by the Vote0 operation. If a mismatch is detected, there has been a timing error that we need to recover from.

```
vote1_op:
      if Rx line state != bit_value
            disable timer interrupt
            enable falling edge interrupt
```

Shift Operation

Each timer interrupt that falls at the end of a data bit takes care of shifting the received bit value into the data word.

```
shift_op:
      shift data_word right by 1 bit
      if bit_value
            data_word |= (0x80 for 8-bit words, 0x100 for 9-bit words)
```

Stop Operation

The first two (or four, for 1.5 stop bits, or six, for 2 stop bits) timer interrupts during the stop bit just check that the Rx line is high, as required during a stop bit. If not, there has been a timing error that we need to recover from.

```
stop_op:
      if Rx line is low
            disable timer interrupt
            enable falling edge interrupt
```

Final Operation

The last timer interrupt during the stop bit checks one last time that the Rx line is high, as required during a stop bit. If not, there has been a timing error that we need to recover from. If so, we have successfully received a character without any detected timing errors.

First thing, we enable the falling edge interrupt to catch the beginning of the next character, and disable the timer interrupt.

If there's space in the receive buffer, we put the received character there. If the buffer is full, we simply drop the character.

```
Final_op:
      disable timer interrupt
      enable falling edge interrupt
      if space is available in the receive buffer
            add char to the receive buffer
```

This would be the place to add automatic handshaking support for receive.

Note that we do not check or strip parity in the Final op. The parity bits are stored in the buffer along with the data bits. The time to check parity bits is when the characters are read from the buffer by the API read() call. An error flag could be made available at that time. However, the standard Arduino serial API has no provision for any kind of error detection.

Handshaking

Serial flow control handshaking is handled entirely at the application layer in Cadetwriter. Transmit flow control probably doesn't come up often, since Cadetwriter only transmits keystrokes in real time. Receive flow control is critical, though, since the Cadetwriter's print mechanism is very slow compared to, well, almost anything else with a serial port.

Cadetwriter currently (as of version 5R6) maintains a rather large receive buffer: 1000 characters in the serial buffer and 100,000 print codes in the print buffer. The flow control thresholds are 750 and 45,000, respectively. That could amount to over an hour of buffered printing. The big buffer is useful when the host does not support flow control, but allowing an hour of backlog is inconvenient for interactive use. These buffer sizes should be greatly reduced when flow control is available and working.

Automatic flow control within SlowSoftSerial is yet to be designed in detail in the receive direction.

Port Switching

Cadetwriter already has application layer multiplexing for multiple "serial" ports, since it supports both USB and the hardware UART for RS-232. To integrate SlowSoftSerial, we just need to visit each of those places in the code and add a case as applicable. No architectural or user interface changes should be required. The updated Cadetwriter will automatically switch between the hardware UART and SlowSoftSerial based on the baud rate configured.

Non-Standard Baud Rates

SlowSoftSerial doesn't care about which baud rates are standard. It can handle a wide range of values. The user interface for selecting baud rates in Cadetwriter only permits entry of a limited subset of standard baud rates. That makes it easier to enter standard baud rates correctly, even if the user can't remember the exact figure, but it also makes it impossible to experiment with non-standard baud rates. I propose to enhance the baud rate UI to allow the user to type an equals sign followed by any number within range for SlowSoftSerial. That adds flexibility without compromising usability for the common case.

More Obscure Word Sizes, etc.

SlowSoftSerial supports all the serial word sizes, parities, and stop bit counts defined by the Arduino API: 5N1, 6N1, 7N1, 8N1, 5N2, 6N2, 7N2, 8N2, 5E1, 6E1, 7E1, 8E1, 5E2, 6E2, 7E2, 8E2, 5O1, 6O1, 7O1, 8O1, 5O2, 6O2, 7O2, and 8O2. It also supports arbitrary (floating point) baud rates, from an arbitrary lower bound of 1 baud up to its performance limit, currently somewhere around 28,800 baud. So you could emulate a 60wpm Baudot code teletype by choosing 5N1 or 5N2 and a baud rate of 45.45 baud. Of course, then you'd also need features like FIGS shift and unshift-on-space.

SlowSoftSerial also supports inverted signaling voltages.

Multiple SlowSoftSerial Ports

The Teensy 3.5 contains just four of the Programmable Interrupt Timers (PIT), and one SlowSoftSerial port uses two of them. If any other part of the application needs a PIT, only one serial port can be supported. If all of the PITs can be dedicated to serial ports, then two simultaneous ports could theoretically be supported.

The normal way to handle multiple ports is to instantiate a SlowSoftSerial class object for each port, so they have completely self-contained, independent resources. There is, however, a difficulty with this model for our design. In order to have access to the resources of the class instance, a function has to be a non-static member function of that class. Such functions cannot normally be used directly as callback functions. We need callback functions from the IntervalTimer library and from a pin change interrupt for receive.

A somewhat ugly workaround was found for this problem. As currently implemented, it only supports a single active instance of SlowSoftSerial. Multiple instances can exist, as long as only one is active. With more bookkeeping code, it could be generalized to support multiple active instances.

Performance Bounds

Interrupt Latency Impact

The path through the receive operation seems to take a fairly consistent 1.0 microsecond. That omits some of the overhead. An actual measurement of the whole impact on interrupt latency has yet to be attempted, but it should be less than 3 microseconds in total, which is plenty good enough for Cadetwriter and many other applications.

Maximum Baud Rate

Very preliminary testing shows that SlowSoftSerial can reliably transmit up to at least 115,200 baud, and receive up to about 28,800 baud, on a Teensy 3.5 running at 120 MHz with a test sketch not doing much else. Those limits are subject to change, of course. There's more work to be done to identify bottlenecks and optimize. See below for a discussion of interrupt latency sensitivity, which is a significant limitation for SlowSoftSerial.

Minimum Baud Rate

The lower bound on the baud rate is imposed arbitrarily at 1.0 baud. Without architectural change, the slowest rate would be limited by the Periodic Interrupt Timers. Those are 32-bit timers running at 60 MHz, for a maximum interval of 71.6 seconds, which would yield a baud rate of 0.014 baud. Here is where SlowSoftSerial deserves its name.

Baud Rate Accuracy

The Periodic Interrupt Timers are straight 32-bit counters, not small counters with switchable prescalers, so any baud rate is achievable. Within the speed range of SlowSoftSerial, systematic speed accuracy is essentially as good as the Teensy's oscillator. Edge jitter is dominated by uncertainty in interrupt latency, which depends on what else is running with interrupts or disabling interrupts. See next section.

Interrupt Latency Sensitivity

SlowSoftSerial is more sensitive to interrupt latency than AltSoftSerial, because of its simpler use of timer hardware. The AltSoftSerial author suggests[6] that a maximum interrupt latency due to other libraries of 15 microseconds is not unheard of. With that much latency, we'd see edge jitter of up to about 15% at a baud rate of 9600. Since there's no cumulative error, that much jitter is probably tolerable, but not much more, so 9600 might be the practical speed limit for a complex Teensy sketch with other interrupt activity.

Since Cadetwriter can switch to the hardware UART for baud rates of 1200 and above, this performance is plenty good for that application.

[6] https://www.pjrc.com/teensy/td_libs_AltSoftSerial.html