NASCOM 2 UART Performance

NAS-SYS uses a documented block-based binary format ("CAS format") for saving and loading data to/from tape. The hardware consists of a 6402 UART and an analogue front-end. On the NASCOM 2, the standard speeds for the interface are 300 baud and 1200 baud.

NAScas is an Arduino-based "digital tape recorder" for the NASCOM. It connects to the SERIAL connector. Appropriate LKSW settings allow the Arduino to connect to the digital side of the NASCOM serial interface and to drive the serial bit clock (which runs at 16x the bit rate) for both Tx and Rx. Bit clock control allows the Arduino to set an arbitrary baud rate.

I expected NAScas to work reliably at much higher speeds than 1200 baud. However, I found that I could not achieve reliable operation above 2400 baud. This paper documents my attempts to understand the limitations of the NASCOM and the Arduino hardware/software.

As a reference point for timing discussions, 1 byte sent at 4800 baud takes 2.08ms (1 start bit, 8 data bits and 1 stop bit). For serial transfers, the *maximum* transfer rate is achieved when the bytes are sent back-to-back; any idle time on the line between bytes will, of course, reduce the transfer rate.

The 6402 UART is unbuffered: as soon as a byte has been shifted in, it is transferred to the UART data register and flagged as being available to be read (by the Z80). If another byte arrives before the Z80 has read this one, it will overwrite the older byte and set the Overflow status bit. NAS-SYS never checks or responds to the Overflow (or the Framing Error) status bits and there is no hardware or software flow control. NAS-SYS relies on the header and body checksums that are part of the CAS format to detect overflow/framing errors.

When NAS-SYS reads data from tape, it polls both the UART and the keyboard for input. The keyboard poll involves a software-scan of the keyboard matrix. At the start of each tape data block, NAS-SYS reports the block number and load address to the screen and, as the data payload is read, it flashes each byte in turn on one location of the screen. All of this takes CPU cycles. At the end of a block it may need to perform a carriage return (quite simple, involving the calculation of the next line start address) and it may also need to perform a scroll of the screen (involving a LDIR block copy of ~1Kbytes).

Experiment 1: scroll time.

A program[1] on the NASCOM moved the cursor to the penultimate screen position, bottom right, then flipped the Drive LED to "ON" then output 1 character (initiating a scroll) then flipped the Drive LED to "OFF" and finally did a scal MRET. The duration of the pulse on the drive LED (measured using a digitising scope) provides a direct indication of the scroll speed. I measured a time of 5.61ms for the

1 All programs were coded in Z80 assembler. This program is included at the end of the paper; the others are available from the author on request.

scroll (4MHz NASCOM, 0 wait-state). This is significantly longer than 2.08ms and so incoming characters at 4800 baud could cause UART overflow during a scroll.

However, the CAS format includes 10 NUL characters between blocks. The protocol allows any number of these to be dropped. Therefore, scrolling does not seem to be the root-cause of the speed restriction.

Experiment 2: Data transfers NASCOM → Arduino

On the Arduino, hardware counters are used to generate the 16x bit clock and the UART is implemented in software. I wondered whether my timings were correct and whether there were any problems at the Arduino end. A program on the NASCOM accessed the UART directly and transmitted 3072 characters in a modulo-256 incrementing pattern at as high a rate as possible (unlike the NAS-SYS code, the program did its wait-for-empty at the top of the loop rather than the bottom). A program on the Arduino received the data stream and checked for errors by verifying that the received data incremented modulo 256. This test ran with no reported errors using baud rates of 2400, 4800, 9600 and 19200. It's possible that the Z80 was not able to use the UART at full utilisation (i.e., that there was idle time between transmitted characters). It would be necessary to *time* the transfer to determine this. Conclusions: (1) both ends can keep up with one another for NASCOM data transmit. (2) The generated bit clock matches the baud rate. (3) There are no signal integrity problems on the interconnect.

Experiment 3: NASCOM input poll time

A program on the NASCOM flipped the Drive LED to "ON" then called scal ZIN to poll for input (from keyboard and UART) then flipped the Drive LED to "OFF" and finally did an MRET. I measured a time of 836us for the poll. Another program without the scal ZIN timed at 106us, giving a time of 730us for scan ZIN. The keyboard/UART poll time is significantly less than the character time at 4800 baud (2.08ms). It is close to the limit for 9600 baud and longer than the time for 19200 baud. Conclusions: the standard NASCOM input routines do not seem to be the limiting factor.

Experiment 4: Data transfers Arduino → NASCOM

A program on the NASCOM polled the UART directly, checking for character available but also checking for overflow and framing errors. On character available, the input character was counted then read/discarded. On error, the program reported the (byte transfer) count and halted. A program on the Arduino sent 2Kbytes of data. This test ran with no reported errors using baud rates of 2400, 4800, 9600 and 19200. Again, It's possible that the Arduino was not able to use the UART at full utilisation (i.e., that there was idle time between transmitted characters). It would be necessary to *time* the transfer to determine this. Conclusions: (1) both ends can keep up with one another for NASCOM data receive. (2) The generated bit clock matches. (3) There are no signal integrity problems on the interconnect.

Experiment 5: Data transfers Arduino → NASCOM with echo

This used a similar program to Experiment 4 except characters received by the NASCOM were echoed to the screen using "rst ROUT" rather than discarded. At 2400 baud this ran with no errors. At 4800 baud, the program ran correctly until the character output reached the bottom of the screen and caused a scroll: at this point the program stopped with a UART overflow error. This is consistent with the character/scroll times reported in Experiment 1.

By introducing a short delay[2] after sending each character, I was able to get reliable data transfer at 4800baud – but, of course, with idle times between characters, so that the data throughput was comparable to 2400baud operation.

Experiment 6: Data transfers Arduino → NASCOM with tuned delays

On the Arduino I generated a data stream in CAS format, but with a delay inserted after sending each byte. On the NASCOM I used the R command to receive the data. The transfer was long enough that the NASCOM had to scroll as it reported progress with the blocks.

With a 1.5ms delay between bytes, this worked reliably at 9600 baud (though, with an effective data rate slightly slower than 4800 baud). Next, I removed the delays one by one to find which were necessary. I was left with 2 places where the Arduino had to introduce a delay. Both were associated with points in the protocol where the NASCOM had to do a little bit more "work" with the received byte, reducing its polling rate to the point where an overflow would otherwise occur.

As a benchmark, I timed a 16Kbyte transfer (slightly more once encoded in CAS format).

@2400 baud with no delays inserted:	1min 14s
@9600 baud with delays inserted[3]:	20s

Wow! I got quite excited to see the speed that those blocks flew by. This seemed a good and complete solution to the "Read" part of the problem.

Experiment 6: Writes to SDcard on Arduino

When doing a "Write" from the NASCOM to the Arduino at 9600 baud I found that the resultant files were corrupt. During a write, the Arduino receives bytes from the NASCOM and writes them out, byte-by-byte, to the SDcard. The SDcard write operation is blocking, but there is no byte-by-byte "flush". The Arduino software UART is interrupt driven with a 64-byte receive buffer. Adding an ".overflow()" call to the Arduino code showed that the Arduino receive buffer was overflowing. Conclusion: byte writes to the SDcard were (sometimes) taking too long, causing the incoming data stream from the NASCOM to back up in the receive buffer, leading to an overflow.

2 Using the standard Arduino delayMicroseconds() routine
3 Two, 1.5ms delays per 256-byte CAS block

I timed writes to SDcard and calculated a transfer rate of 10,498 bytes per second (95us per byte) – about 1/10th of the byte time of 9600 baud serial transmission. With a bit more instrumentation of the Arduino code, I saw the problem: the average transfer rate to SDcard is high, but sometimes a byte takes a lot longer to transfer – up to 13ms for one of the bytes. I assume this is a delay that occurs, for example, when a new sector needs to be allocated to the file.

At 9600 baud, 13ms is 13/1.04~13 byte-times. Therefore, the 64-byte receive buffer *should* be enough to even out these delays,: the buffer should fill during slow SDcard writes and then drain during fast SDcard writes. Apparently, though, this is not sufficient to prevent overflow. I consider these solutions:

- Increase the size of the software UART buffer (it's controlled by a #define, and therefore easy)
- Add a local buffer for data between UART read and SDcard write. Unless writes to SDcard are more efficient when done in lumps, this is an awkward way of achieving the same effect as increasing the size of the software UART receive buffer
- Dynamically change the software UART speed between NASCOM write operations and NASCOM read operations

Mystery 1: Changing the buffer from 64 to 128 and even to 256 did not seem to make any difference. Given that slowing the incoming serial rate from 9600 baud to 2400 baud avoids overflow, this is a surprising result.

Changing the software UART speed dynamically gave a better result. I ran the interface at 2400 by default (for the CLI and for writes) and then sped it up to 9600 baud for read operations. This seemed to work OK in general, but..

Mystery 2: reads from SDcard did not work (they were OK for flash and even for virtual disk, which itself reads its data from SDcard). That is a surprising result because it should be OK for the Arduino to arbitrarily slow down the read (due to slow SDcard access) and the test program shows that it works at maximum speed (with the two delays that are inserted in both cases). And, besides, it works for virtual disk?

Update 01Sep2019: Despite all of my investigations back in May I did not solve the basic problem: the released version of the Arduino code is still restricted to 2400baud operation. The next avenues to explore are to try to solve the two "mysteries" noted above. For mystery 1 it may be useful to record high-water-mark in the UART receive buffer as a route to learning more. For mystery 2, conforming the behaviour with virtual disk and looking for coding differences seems the best next step.

Example test program (for Experiment 1)

```
;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; scroll
;;; how long does it take the NASCOM to scroll the screen

;;; get to the bottom of the screen
        ld      a,$0d
        ld      b,16
lines:  rst     ROUT
        djnz    lines
;;; get to the end of the line
        ld      a,65
        ld      b,47
chars:  rst     ROUT
        djnz    chars

;;; turn on the DRIVE LED
        scal    ZMFLP
;;; cause a scroll
        ld      a,66
        rst     ROUT
;;; turn off the DRIVE LED
        scal    ZMFLP
        scal    ZMRET
```

Date	**Comments**
12 May 2019	First release
31 Aug 2019	Tighten up some of the wording