NASCOM 2 UART Performance

Under NAS-SYS, save to and load from tape uses a block-based binary format (“CAS format”) which is documented in the NAS-SYS manual. The data path consists of a 6402 UART and an analogue front-end.

NAScas is an Arduino-based “digital tape recorder” for the NASCOM. It interfaces a serial port on the Arduino to the NASCOM tape interface. It connects to the NASCOM on the digital interface and also supplies the serial bit clock (which runs at 16x the bit rate) for both Tx and Rx.

The standard tape speeds on the NASCOM 2 were 300 baud and 1200 baud.

When developing NAScas I expected to be able to run at considerably higher speed. However, the highest speed at which I could achieve reliable operation was 2400 baud. This paper documents my attempts to understand the limitations of the NASCOM and Arduino hardware/software.

As a reference point, consider 1 byte sent at 4800 baud. With 1 start bit, 8 data bits and 1 stop bit the byte takes 2.08ms. This time gives rise to the *maximum* transfer rate: 1 byte of user data every 2.08ms. If there is any idle time on the line between bytes, the transfer rate will, of course, be lower.

The 6402 UART is unbuffered: as soon as a byte has been shifted in, it is transferred to the data register, and flagged as being available to be read (by the processor). If another byte arrives before the processor 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. If an overflow error did occur, it is highly likely to be detected by the header and body checksums that are part of the CAS format.

When NAS-SYS reads data from tape, it checks both the UART and the keyboard for a character. The keyboard check involves a software-scans of the keyboard matrix. At the start of each data block, it 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 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 an MRET. I triggered a digitising scope on Drive and measured a time of 5.61ms for the scroll (4MHz NASCOM, 0 wait-state). This is significantly longer than the character time of 2.08ms and so incoming characters could cause UART overflow as the result of a scroll.

However, the CAS format includes 10 expendable NUL characters between blocks. It would be invisible and acceptable to the protocol to drop one or more of these. Therefore, I did not think this was the root-cause of my speed restriction.

Experiment 2: Data transfers NASCOM → Arduino

I was using a software UART on the Arduino and, independently, programming a counter in the Arduino to generate the 16x bit clock. 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 (it simply checked that the received data incremented modulo 256). This test was run using baud rates of 2400, 4800, 9600, 19200 with no reported errors. 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. This test shows that both ends can keep up with one another for NASCOM data transmit, that the generation of the bit clock matched the baud rate and that there were 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 triggered a digitising scope on Drive and 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. This is significantly less than the character time of 2.08ms (@4800 baud), close to the limit for 9600 baud and longer than the time for 19200 baud.

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, it read and discarded the input character and incremented a counter. On error, it reported the (byte transfer) count and halted. A program on the Arduino sent 2Kbytes of data. This test was run using baud rates of 2400, 4800, 9600, 19200 with no reported errors. 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. This test shows that both ends can keep up with one another for NASCOM data receive, that the generation of the bit clock matched the baud rate and that there were 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" ratehr 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 above.

At the Arduino end, I introduced a short delay (using the delayMicroseconds() library routine) after the transmission of each character. By increasing the delay 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 each byte was sent. On the NASCOM I used the R command to receive the data. I made sure that 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). I then removed the delays one by one until I found 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 sent 16Kbytes of user data (so, actually a bit more once encoded in CAS format).

@2400 baud with no delays inserted:	1min 14s
@9600 baud with delays inserted*:	20s

*two, 1.5ms delays per 256-byte CAS block

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 I tried using 9600 baud for saving data from the NASCOM to SDcard I found that the resultant files were corrupt. During a save, the Arduino receives bytes from the NASCOM and writes them out, byte-by-byte, to the SDcard. This SDcard write operation is blocking, but there is no byte-by-byte "flush". The software UART is interrupt driven and has a 64-byte receive buffer. By adding a ".overflow()" call to the Arduino code, I could see that the Arduino receive buffer was overflowing.

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 some bytes take a lot longer to transfer – upto 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.

I thought that 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 had 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 SD did not work (they were OK for flash and even for virtual disk). 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 SD??

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
```