AtariVox Programmer's Guide

By Alex Herbert

Last revised: 16th November 2004

--- DRAFT ---

AtariVox

Pinout:

Pin	
1	SpeakJet DATA
2	SpeakJet READY
3	EEPROM SDA
4	EEPROM SCL
5	
6	
7	+5v
8	GND
9	

SpeakJet Speech Synth

The AtariVox uses a SpeakJet chip to synthesize speech. Speech is constructed from strings of allophones – the individual sounds that make up speech. Refer to the SpeakJet User Manual PDF from **www.speakjet.com** for details of the allophone/control codes.

Communication with the SpeakJet

The SpeakJet receives speech data via asynchronous serial interface, pre-configured to operate at 19,200 baud (8n1, inverted). The DATA line is connected to pin 1 of the 2nd joystick socket (bit 0 of **SWCHA**) and the READY line is connected to pin 2 (bit 1 of **SWCHA**).

The READY line may be read to detect whether the SpeakJet is able receive data. If this reads 0 then the chip is not ready (i.e. the input buffer is full) and data should not be sent at this time.

The data is sent in 8n1 format (1 start bit, 8 data bits, no parity, 1 stop bit) but is inverted compared to normal RS-232. The DATA line should be held high whilst idle (between data

transmissions) either by setting the pin to input mode, or to output 1 (same thing).

To transmit a byte, each bit (1 start bit, 8 data bits and 1 stop bit) are output on the DATA line at intervals of 1/19,200th of a second. For the Atari 2600, an interval of 62 CPU clock cycles is a close enough match. First is the start bit, which is always 0. Then the 8 data bits, least significant bit first. And lastly the stop bit which is always 1 (returning the DATA line to idle). The stop bit must be present on the DATA line for at least 62 cycles.

Driver Code

The include file **speakjet.inc** contains 2 macros which will simplify adding speech to your project.

The code requires a 2-byte RAM variable **speech_addr** to be defined which will hold the address of the next byte to be transmitted to the SpeakJet. Speech data is organised as strings of allophone/control codes. Each string must be terminated with the byte **$ff**.

The **SPKOUT** macro defines code which checks the status of the READY line, reads the next byte from the speech string and sends it to the SpeakJet. It requires one additional byte of RAM for use as a temporary scratchpad. The following subroutine outputs one byte using the variable **temp** as it's temporary workspace:

```
output_speech
      SPKOUT      temp
      rts
```

Calling **output_speech** once per frame (during the vertical blank or overscan periods) is normally sufficient to keep the SpeakJet adequately supplied with speech data. Handy, because it takes roughly 8 TV scanlines to transmit a single byte.

Then to start some speech it's just a case of pointing **speech_addr** to the start of a speech string. The **SPEAK** macro sets this pointer:

```
say_hello
      SPEAK       hello_speech
      rts

hello_speech
      dc.b        20,96                   ; volume = 96
      dc.b        21,114                  ; speed = 114
      dc.b        22,88                   ; pitch = 88
      dc.b        23,5                    ; bend = 23
      dc.b        183,7,159,146,164       ; \HE \FAST \EHLE \LO \OWWW
      dc.b        $ff                     ; terminator byte
```

Note: **speech_addr** must be pointing to a valid speech string before **SPKOUT** is executed.

Also see the included example program: **vox_test.asm**

EEPROM

The memory device used in the AtariVox is the Microchip 24LC256 (or compatible) which provides 32KBytes of non-volatile storage.

Like the SpeakJet, the EEPROM also communicates via serial interface but uses the I2C protocol which is quite different. Rather than transmitting bits at a pre-determined time interval, data is clocked in/out of the chip by the controlling device (the console).

The EEPROM's SDA (**S**erial **DA**ta) line is connected to pin 3 of the joystick port (bit 2 of **`SWCHA`**) and SCL (**S**erial **CL**ock) is connected to pin 4 (bit 3 of **`SWCHA`**).

I2C Protocol

Without going into great detail about the actual state changes on the I/O pins (for full details refer to the 24LC256 datasheet from **www.microchip.com**) the I2C protocol can be broken down into 4 basic signals: Start, Stop, Write Bit and Read Bit. The Start signal is used to start a read/write operation and Stop is used to terminate a read/write. Data is read/written a byte at a time (most significant bit first). After each byte transmitted, the receiving device responds by transmitting an acknowledge bit (normally 0).

Start signals are always followed by a command byte: **`$a0`** to enable writing to the EEPROM or **`$a1`** to enable reading.

The Write command is always followed by 2-bytes representing the 16-bit start address (high byte first). Subsequent bytes are then written to the EEPROM's internal buffer and will be committed to non-volatile memory on receipt of the next Stop signal.

Note: The 26LC256 memory is segmented into 64-byte pages and it is only possible to write to a single page at a time. Attempting to perform a write operation that crosses a page boundary will cause the internal memory pointer to loop back to the start of the current page.

The Read command does not set the start address and reading starts immediately using the current memory pointer. To set the start address for reading, a "fake" Write is used. This consists of a Start signal, **`$a0`** Write command byte, 2 address bytes and a Stop signal – i.e. no data. Then it's just a case of issuing a Start signal followed by the Read command **`$a1`** to enable reading from the newly set address.

An acknowledgement bit (0) must be transmitted after each byte read, except for the last byte where a no-acknowledge (1) should be used. (This is to prevent the chip from transmitting the next byte, possibly obscuring the Stop signal.) There is no limit on the number of byte than can be read in one go.

EEPROM Driver Code

The **i2c.inc** include file defines macros/subroutines that take care of the low level signalling. The subroutines defined in the **I2C_SUBS** macro require 1 byte of RAM as a temporary workspace which should be specified when the macro is invoked. For example:

```
      I2C_SUBS    temp
```

The subroutines are:

i2c_startwrite Outputs a Start signal and transmits the byte **$a0** to initiate a write sequence.

i2c_startread Outputs a Start signal and transmits the byte **$a1** to initiate a read sequence.

i2c_txbyte Transmits the byte in A, returning the acknowledge bit in C. The carry flag will be clear (0) if the byte was received ok, or set (1) if the device wasn't ready or is not connected.

i2c_rxbyte Reads one byte and returns it in A.

i2c_stopwrite Outputs a Stop signal terminating the write sequence and commits the data to the EEPROM.

i2c_stopread Outputs a no-acknowledge bit followed by a Stop signal, terminating a read sequence.

Writing to the EEPROM

Writes to the EEPROM begin with a Start signal followed by the command byte **$a0**. Then 2 bytes representing the start address are written (high byte first) followed by the actual data and a Stop signal.

For example, the following code writes 8 bytes from **buffer** to EEPROM addresses **$0140-$0147**:

```
            jsr   i2c_startwrite    ; Start signal and $a0 command byte
            bcs   eeprom_error      ; exit if command byte not acknowledged

            lda   #$01              ; upper byte of address
            jsr   i2c_txbyte
            lda   #$40              ; lower byte of address
            jsr   i2c_txbyte

            ldx   #$00
      write_loop
            lda   buffer,x          ; get byte from RAM
            jsr   i2c_txbyte        ; transmit to EEPROM
            inx
            cpx   #$08              ; 8 bytes sent?
            bne   write_loop
```

```
        jsr   i2c_stopwrite     ; terminate write and commit to memory
```

Note: If no EEPROM is detected, this code jumps to **`eeprom_error`** which should call **`i2c_stopwrite`** (to restore the I/O port to it's normal state) and perform any other functions (such as loading **`buffer`** with default values) that may be required in such event.

Reading the EEPROM

Unlike the Write command, no address may be specified with the Read command and reading starts from the current address. To read from a specific address we must first issue a fake Write sequence (i.e. without any data) in order to set the memory pointer.

The actual read itself begins with a Start signal followed by the command byte **`$a1`**. We can now read as many bytes as we like - there are no restrictions on crossing page boundaries when reading.

The following code reads 8 bytes from EEPROM addresses **`$0140-$0147`** and stores them in **`buffer`**.

```
        jsr   i2c_startwrite    ; Start signal and $a0 command byte
        bcs   eeprom_error      ; exit if command byte not acknowledged

        lda   #$01              ; upper byte of address
        jsr   i2c_txbyte
        lda   #$40              ; lower byte of address
        jsr   i2c_txbyte

        jsr   i2c_stopwrite     ; end of "fake" write

        jsr   i2c_startread     ; Start signal and $a1 command byte

        ldx   #$00
read_loop
        jsr   i2c_rxbyte        ; read byte from EEPROM
        sta   buffer,x          ; store in buffer
        inx
        cpx   #$08              ; 8 bytes read?
        bne   read_loop

        jsr   i2c_stopread      ; terminate read
```

Note: **`i2c_startread`**, **`i2c_rxbyte`** and **`i2c_stopread`** use the oVerflow flag to track whether an acknowledge bit is needed.

Read/Write Speed

As you have probably worked out already, reading/writing the serial EEPROM is much slower than accessing RAM. In the process of developing the driver code, I chose to perform EEPROM reads/writes during the 192-line visible display period. Part of the reason for this was so that I can change the background colour during EEPROM access, giving a visual

representation of the data transfer and time taken. (Set **I2C_DEBUGCOLORS** in **i2c.inc** to enable.) It is possible to read/write 32-bytes (maybe a few more?) during one 192-line display period.

Memory Partitions

```
0000-3fff         Static Allocation Area
4000-7fff         File Area
```

Static Allocation Area

The Static Allocation Area occupies the lower first 16K of the address range. Memory is pre-allocated to specific programs. Once a memory range has been allocated it becomes exclusive to that program and should not be overwritten by other programs. This memory is intended for use by Atari 2600 programs where minimal code is required.

Static Allocation is obviously finite. It is possible, in theory at least, that at some point in time all the memory will have been allocated, leaving no room for new games to store their data. It's unlikely that many Atari 2600 programs will need much space though so this should keep us going for a while.

Memory is allocated in 64 byte pages:

Page	Address Range	Assignment
00	0000-003f	System Settings (TV mode)
01	0040-007f	Man Goes Down
02	0080-00bf	
03	00c0-00ff	
...		
fc	3f00-3f3f	
fd	3f40-3f7f	
fe	3f80-3fbf	
ff	3fc0-3fff	

To allocate memory for your project, please contact: **richard.hutchinson@dsl.pipex.com**

System Settings Page

The first EEPROM page is reserved for the System Settings which is currently just the 8 byte ASCII signature “ATARIVOX” followed by the TV Mode byte.

Bit 7 of the TV Mode byte = 0 for PAL, 1 for NTSC. Bit 6 = 0 for 60Hz, 1 for 50Hz.

See **syspage.inc** for code to read/write the TV Mode.

File Area

The File Area occupies the upper 16K of the EEPROM and is more flexible than the Static Allocation Area.

Proposed Format for File Area

Note: No software currently uses the File Area so the following is only a proposed format and is not set in stone.

- 64-byte pages are paired to form 128-byte Blocks.
- Each block starts with an 8 character ASCII filename (7-bit character codes) and the remaining 120 bytes are free for data.
- Bit 7 of the first character is used to determine if the block is allocated or available. 0 indicates the block is in use, a 1 means it is free.

Even implementing something as simple as this gets quite fiddly on the 2600. The time taken to search for a file/free block is variable and may take several TV frames to complete and care must be taken to ensure that VSYNC still happens at the right time. Also, any software that uses the File Area should provide the user with some way to delete files as well as create them. This will possibly involve text routines, a character set, etc., all adding further bulk to the program.

Contact:

Richard Hutchinson - Creator of the AtariVox.
Web: http://www.vectrex.biz
Email: richard.hutchinson@dsl.pipex.com

Alex Herbert – Author of this document and driver code.
Web: http://www.herbs64.com
Email: herbs64@yahoo.co.uk