Altair 8800 Simulator

- Power supply must be center-positive 7V-12V DC (typical usage 200-500mA, 1A max)
- Runs at about the same speed as the original (maybe a bit faster)
- Emulated memory size (RAM) is 64KB
- ALTAIR extended BASIC ROM (16k) can be mapped to addresses 0xC000-0xFFFF
- Emulates one serial device of an ALTAIR 88-2SIO serial board with control channel at 0x10 and data channel at 0x11. This device is mapped to the Arduino's serial input/output. To use, connect a serial terminal to Arduino RX1/TX1 pins at 9600 baud 8n1. Does some special processing (based on memory location of IN/OUT operations) to simplify interaction with ALTAIR BASIC (see below).
- A serial-to-Bluetooth dongle is connected to Arduino serial pins, Bluetooth id "ALTAIR", passcode 1234
- Emulates 4 Altair 88-DCDD disk drives
- Emulates a cassette interface at I/O addresses 0x06 and 0x07. There are 256 virtual tapes that can be used for storage by setting SW15-13 to 010, selecting the virtual tape number on SW7-0 and pressing AUX2 UP (to capture) or AUX2 DOWN (to replay captured data).
- Emulates an ALTAIR 88-SIO serial board with control channel at 0x00 and data channel at 0x01. Data sent to this device can be captured and replayed (as if a Teletype tape punch were connected). Capturing is the same as for tape but with SW15-13 set to 001.
- The cassette interface also supports using the CSAVE/CLOAD commands in extended BASIC. Each file will automatically be saved to a separate file specified by the file name in the CSAVE command. For CLOAD, all tapes will automatically be played back until the file is found (or not).
- Currently set up using Arduino Due, can also work on Arduino Mega 2560. In that case, it works at about 25% original speed and has 6k emulated RAM.

Front Panel

Lights:

A15-A0	Shows the current address bus state (i.e. PC during normal operation)
D7-D0	Shows the current data bus state

Switches:

SW15-SW0	Address/Data entry switches
RUN	Execute instructions starting at current PC location
STOP	Stop program execution
SINGLE STEP	Execute single instruction at current PC location and increment PC
EXAMINE	Set PC to address from SW15-0 and show content of that address on D7-0
EXAMINE NEXT	Increment PC and show content of that address on D7-0
DEPOSIT	Store value of D7-0 at current PC address
DEPOSIT NEXT	Increase PC by one and store value of D7-0 at that address
CLR	Stop serial capture/replay activated by AUX2 (see below) Holding CLR down during power-up will initialize memory and CPU registers to 0
RESET	Reset processor (set PC to 0)
PROTECT	Mark current memory page as write-protected (cannot be written to)
UNPROTECT	Remove write-protect status of current page
AUX1/AUX2	See detailed information below

Auxiliary switch functions

AUX1 down:

Function depends on current setting of SW7-0:

0…0000: Print this list to serial interface

0…0001: Calculator (David Hansel, 2015)

- Flipping SW15 (on->off or off->on) executes operation selected by SW14-13 between operand currently showing on A15-8 lights and operand selected by SW12-8 switches. Result is shown on A15-8 lights.
- Operations (SW14-13): 00=Add, 01=Subtract, 10=Multiply, 11=Divide

0…0010: Kill-the-bit (Dean McDaniel, 1975)

- Kill the rotating bit. If you miss the lit bit, another bit turns on leaving two bits to destroy.
- Quickly toggle the switch, don't leave the switch in the up position.
- Before starting, make sure all the switches are in the down position.

0…0011: Pong game using front panel (http://altairclone.com/downloads/pong.pdf)

- Left player quickly toggles SW15 to hit the "ball."
- Right player toggles SW8. Score is kept in memory locations 0x80 and 0x81 (left and right).
- Score is missed balls, so the lower number wins.

0…0100: Pong game using serial terminal (David Hansel, 2015)

- Needs terminal connected to serial interface. Terminal must understand escape sequences for cursor movement (“ESC-[“)
- Left player uses A/Z keys (paddle up/down), right player uses K/M keys (paddle up/down)

0…0101: 4k Basic (Bill Gates, Paul Allen, Monte Davidoff 1975)

- Before loading, set SW15-8 to 00001000
- Connect serial terminal to Arduino RX/TX pins, 9600 baud 8n1. The built-in serial emulation automatically translates between back-space and ‘_’ which is the back-space character in ALTAIR BASIC. It also converts lower-case to upper-case letters since 4k BASIC only uses upper-case.
- Answering N to questions about SIN/RND/SQR increases available BASIC memory

0…0110: MITS Programming System II

- Editor, Assembler and Debugger
- See quick intro at the end of this document, google for full documentation
- Before loading, set SW15-8 to 10001010

0…0111: ALTAIR Turnkey Monitor

0…1000: Disk Drive Boot ROM (see *Disk Drive Support* section below)

0…1001: Music (“Daisy, Daisy…”) (Steve Dompier, 1975)

- An AM radio held on top of Simulator, close to D0-D7 lights will pick up the song
- See: http://www.digibarn.com/collections/weirdstuff/altair-sheetmusic/

0…1010: 8080 CPU Diagnostic (Microcosm Associates, 1980)

- Basic test, takes about a second to finish
- Outputs to serial interface (“CPU IS OPERATIONAL”)

0…1011: 8080 CPU Exerciser (Frank D. Cringle 1994, Ian Bartholomew 2009)

- Very thorough test that generates a CRC code for each group of tests.
- Outputs to serial interface.
- Full test takes about 4 hours. The "aluop <b,c,d,e,h,l,m,a>" section takes especially long

010…0: Start file system manager (to manage storage files)

10……1: Save the 256-byte memory page currently selected on the SW15-8 switches

10……0: Load the 256-byte memory page currently selected on the SW15-8 switches

AUX1 up:

- Maps ROM extended BASIC to addresses 0xC000-0xFFFF (until next RESET)
- Sets PC to 0xC000 and starts execution (i.e. starts BASIC)
- Before activating, set SW15-8 switches to 00001000
- Connect serial terminal to Arduino RX/TX pins, 9600 baud 8n1. The built-in serial emulation automatically translates between back-space and '_' which is the back-space character in ALTAIR BASIC.
- When asked MEMORY SIZE, just press enter.
- When asked LINEPRINTER, enter capital O and press enter
- Available memory for BASIC programs is 48101 bytes (~3000 bytes on Arduino Mega)
- CSAVE and CLOAD commands can be used to save/load programs to/from internal storage

AUX2 down:

Function depends on current setting of SW15-12 (see below for details):

- 0000: Load BASIC example
- 1000: Play back data captured from the emulated 88-2SIO serial board
- 0100: Play back data captured from the emulated tape (cassette)
- 0010: Play back data captured from the emulated 88-SIO serial board
- 0001: Mount disk in disk drive (see *Disk Drive Support* section below)

Loading BASIC examples:

- Works by emulating a paper tape reader on the serial I/O ports
- SW7 must be down
- If SW6-0 are all 0 then a list of available BASIC examples will be transmitted to BASIC
- Setting SW7-0 according to the example number in the list and pushing AUX2 DOWN transmits an example to BASIC

Loading assembly code examples:

- SW7 must be up
- If SW6-0 are all 0 then a list of available assembly code examples will be transmitted
- Setting SW6-0 according to the example number in the list and pushing AUX2 DOWN transmits an example

Playing back captured serial data:

- Play back serial data previously captured via AUX2 up
- The file number to be played back must be selected via SW7-0
- Playback will stop automatically when the end of captured data is reached
- Playback can be stopped manually by pushing CLR

Playing back captured cassette data:

- Same as playing back serial data but using cassette I/O ports (0, 1)

AUX2 up:

Function depends on current setting of SW15-12:

- 1000: Captured serial data (on I/O ports 16/17)
- 0100: Captured tape (cassette) data (on I/O ports 6/7)
- 0010: Captured serial data (on I/O ports 0/1)
- 0001: Unmount disk from disk drive (see *Disk Drive Support* section below)

For both functions, SW7-0 specify the file number under which the captured data will be saved. Therefore, up to 256 files are possible. Capturing continues until AUX2 is again pressed UP.

Capturing serial data can be used to save a BASIC program in BASIC (note that in extended BASIC the CSAVE/CLOAD commands provide an easier way of doing this):

1. Set SW7-0 to the desired storage file number
2. Set SW15-13 to 100 (or 001 if capturing from 88-SIO on address 0/1)
3. activate AUX2 up to start capture
4. type "list"
5. wait until listing is finished
6. activate AUX2 up again to finish capture

To later restore a program:

1. Set SW7-0 to a storage file number under which serial data has been captured before
2. Set SW15-13 to 100 (or 001 if replaying to 88-SIO on address 0/1)
3. type "new" to clear the current program
4. activate "AUX2 down" to start replaying the captured data
5. Ignore the "SYNTAX ERROR" at the end
   (reported because the final "ok" of the "list" function was also captured)

Capturing tape data can be used to save BASIC variable contents in Extended BASIC or saving programs from MITS Programming System II

1. Set SW7-0 to the desired storage file number
2. Set SW15-13 to 010
3. activate AUX2 up to start capture
4. enter CSAVE*v [where v is the variable name that is supposed to be saved]
5. Repeat the previous step if more variables need to be saved
6. wait until listing is finished
7. activate AUX2 up again to finish capture

To later load the data:

1. Set SW7-0 to a storage file number under which tape data has been captured before
2. Set SW15-13 to 010
3. activate AUX2 down to start replaying the captured tape data
4. enter CLOAD*v [where v is the variable name that is to be loaded]
5. Repeat the previous step if more variables need to be loaded
6. Activate AUX2 down to stop replay

Disk Drive Support

Disk drive support is optional and requires a SD card attached to the Arduino Due's SPI header (the 2-row, 6-pin header marked "SPI"). See the *Wiring SD card to Arduino DUE* section at the end of this document about how to physically hook up the card.

Disk drive support is **not** available in the Arduino Mega build, mainly for two reasons: 1. The SPI pins on the Mega are directly connected to some general I/O pins which are already used for the front panel and 2. The Mega only provides 6k of emulated RAM. Most disk based programs require more than that.

If you do not want to connect a SD card to the Due then you may want to change the `NUM_DRIVES` setting in `config.h` to 0. If `NUM_DRIVES` is greater than 0 then the Due spends about 1-2 seconds during startup looking fort the SD card (it will continue normally but without drive support if no card is found).

If an SD card is detected, the following files are expected to be found in the root directory of the (FAT format) card:

- **DISKxx.DSK** (where xx is a 2-digit hexadecimal number): Disk images that the simulator can mount.
- **DISKDIR.TXT**: A text file whose contents will be sent to the serial connection (i.e. shown to the user) if front panel switches are set to 0001xxxx00000000 and the AUX2 switch is pressed down. This should contain information about each of the DISCxx.DSK files

To mount disks in the drive, use the AUX2 down switch:

- Set SW15-0 to: 0001nnnnDDDDDDDD where nnnn is a 4-bit number selecting the drive (i.e. drive 0-15) and DDDDDDDD is an 8-bit number selecting the disk number and press AUX2 down.
- The 8-bit disk number corresponds to the xx in the DISKxx.DSK files on the SD card.
- For example, setting SW15-0 to 0001 0010 0000101 and pressing AUX2 down will mount disk number 5 in drive 2.
- Selecting disk number 0 is a special case. If disk 0 is selected for mounting, it will not be mounted but instead the contents of the DISKDIR.TXT file will be sent to the serial output (i.e. shown to the user). Note that that means a file named DISK00.DSK can not be mounted.
- If a disk is already mounted in the drive the mounted disk will be unmounted before mounting the new disk
- If the disk file does not exist, it is like inserting an empty disk in the drive. If the operating system writes to the disk, the selected disk file will be created. So inserting a non-existent disk and then formatting that disk via the operating system (e.g. CP/M) will create a new empty disk.
- It is possible to mount the same disk in multiple drives. The simulator has no problem with that but the running operating system may get confused.

To unmount a disk from a drive, use the AUX2 up switch:

- Set SW15-0 to: 0001nnnnxxxxxxxx where nnnn is a 4-bit number selecting the drive (i.e. drive 0-15) and press AUX2 up.
- It is not necessary to unmount disks before turning off the computer. Each write operation to a disk gets flushed to the SD card immediately so turning the computer off with disks mounted will not lose data.

To run a bootable disk image, first mount the disk and then start the Disk Boot ROM:

- Set SW0-7 to 00001000 (to select Disk Boot ROM)
- Press AUX1 down
- This will install the Disk Boot ROM at 0xFF00 and immediately start it. If a bootable disk has been mounted it should automatically start now

File system manager

The emulator includes a very simple mini file system to store the different types of data that can be saved/captured. A simple file system manager can be invoked by setting SW7-0 to 01000000 and pressing AUX1 down (the ALTAIR must be in WAIT mode).

The file system manager supports the following commands:

- **F**: Format file system (erases all files)
- **d**: Delete a file from the file system
- **r**: Read a file and show contents on screen

Debugging capabilities

During reset:

SW7 on	Print processor status and disassembled opcode serial during single-stepping
SW6 on	Print status of control panel to serial during single-stepping
SW5 on	read (single-character) debugging inputs from serial

When stopped, the following keys have a function (if debugging inputs were enabled via SW5):

0-9,a-f	Toggle SW0-15 (only if stand-alone mode enabled in setup.h)
/	Prompt for value to set SW0-15 (only if stand-alone mode enabled in setup.h)
r	Run
o	Stop
t	Step
R	Reset
!	Hard reset (STOP+RESET)
X/x	Examine/examine next
P/p	Deposit/deposit next
U	AUX1 up
u	AUX1 down
s	AUX2 up
l	AUX2 down
Q	Protect
q	Unprotect
B	Add breakpoint (only if breakpoints enabled in setup.h)
C	Delete last breakpoint
D	Disassemble (will prompt for start address, space bar continues, any other key exits)
M	Dump memory (will prompt for start address, space bar continues, any other key exits)
F	Enable/disable profiling (only if profiling enabled in setup.h)
n	change number system (hexadecimal/octal/decimal)
L	Load a program from serial First value is start address, second value is length, followed by data bytes (all values separated by spaces). Can be used to load a program via the terminal.

8080 Instruction set

```
Conventions in instruction source:
    D   = Destination register (8 bit)
    S   = Source register (8 bit)
    RP  = Register pair (16 bit)
    #   = 8 or 16 bit immediate operand
    a   = 16 bit Memory address
    p   = 8 bit port address
    ccc = Conditional

Conventions in instruction encoding:
    db  = Data byte (8 bit)
    lb  = Low byte of 16 bit value
    hb  = High byte of 16 bit value
    pa  = Port address (8 bit)

Dest and Source reg fields:
    111=A   (Accumulator)
    000=B
    001=C
    010=D
    011=E
    100=H
    101=L
    110=M   (Memory reference through address in H:L)

Register pair 'RP' fields:
    00=BC   (B:C as 16 bit register)
    01=DE   (D:E as 16 bit register)
    10=HL   (H:L as 16 bit register)
    11=SP   (Stack pointer, refers to PSW (FLAGS:A) for PUSH/POP)

Condition code 'CCC' fields: (FLAGS: S Z x A x P x C)
    000=NZ  (Zero flag not set)
    001=Z   (Zero flag set)
    010=NC  (Carry flag not set)
    011=C   (Carry flag set)
    100=PO  (Parity flag not set - ODD)
    101=PE  (Parity flag set - EVEN)
    110=P   (Sign flag not set - POSITIVE)
    111=M   (Sign flag set - MINUS)
```

Inst	Encoding	Flags	Description
ACI #	11001110 db	ZSCPA	Add immediate to A with carry*
ADC S	10001SSS	ZSCPA	Add register to A with carry
ADD S	10000SSS	ZSPCA	Add register to A
ADI #	11000110 db	ZSCPA	Add immediate to A
ANA S	10100SSS	ZSCPA	AND register with A
ANI #	11100110 db	ZSPCA	AND immediate with A
CALL a	11001101 lb hb	-	Unconditional subroutine call
Cccc a	11CCC100 lb hb	-	Conditional subroutine call
CMA	00101111	-	Complement A
CMC	00111111	C	Complement Carry flag
CMP S	10111SSS	ZSPCA	Compare register with A
CPI #	11111110	ZSPCA	Compare immediate with A
DAA	00100111	ZSPCA	Decimal Adjust accumulator
DAD RP	00RP1001	C	Add register pair to HL (16 bit add)*
DCR D	00DDD101	ZSPA	Decrement register*
DCX RP	00RP1011	-	Decrement register pair
DI	11110011	-	Disable interrupts
EI	11111011	-	Enable interrupts
HLT	01110110	-	Halt processor
IN p	11011011 pa	-	Read input port into A
INR D	00DDD100	ZSPA	Increment register
INX RP	00RP0011	-	Increment register pair
Jccc a	11CCC010 lb hb	-	Conditional jump*
JMP a	11000011 lb hb	-	Unconditional jump*
LDA a	00111010 lb hb	-	Load A from memory
LDAX RP	00RP1010 *1	-	Load indirect through BC or DE
LHLD a	00101010 lb hb	-	Load H:L from memory*
LXI RP,#	00RP0001 lb hb	-	Load register pair immediate*
MOV D,S	01DDDSSS	-	Move register to register*
MVI D,#	00DDD110 db	-	Move immediate to register*
NOP	00000000	-	No operation
ORA S	10110SSS	ZSPCA	OR register with A
ORI #	11110110	ZSPCA	OR immediate with A
OUT p	11010011 pa	-	Write A to output port
PCHL	11101001	-	Jump to address in H:L
POP RP	11RP0001 *2	*2	Pop register pair from the stack
PUSH RP	11RP0101 *2	-	Push register pair on the stack
RAL	00010111	C	Rotate A left through carry*
RAR	00011111	C	Rotate A right through carry
Rccc	11CCC000	-	Conditional return from subroutine
RET	11001001	-	Unconditional return from subroutine
RLC	00000111	C	Rotate A left
RRC	00001111	C	Rotate A right
RST n	11NNN111	-	Restart (Call n*8)
SBB S	10011SSS	ZSCPA	Subtract register from A with borrow
SBI #	11011110 db	ZSCPA	Subtract immediate from A with borrow
SHLD a	00100010 lb hb	-	Store H:L to memory*
SPHL	11111001	-	Set SP to content of H:L
STA a	00110010 lb hb	-	Store A to memory
STAX RP	00RP0010 *1	-	Store indirect through BC or DE
STC	00110111	C	Set Carry flag
SUB S	10010SSS	ZSCPA	Subtract register from A
SUI #	11010110 db	ZSCPA	Subtract immediate from A
XCHG	11101011	-	Exchange DE and HL content
XRA S	10101SSS	ZSPCA	Exclusive OR register with A
XRI #	11101110 db	ZSPCA	Exclusive OR immediate with A
XTHL	11100011	-	Swap H:L with top word on stack

*1 = Only RP=00(BC) and 01(DE) are allowed for LDAX/STAX
*2 = RP=11 refers to PSW for PUSH/POP (cannot push/pop SP).
When PSW is POP'd, ALL flags are affected.

Encoding	Inst	Flags	Description
00000000	NOP	-	No operation
00000111	RLC	C	Rotate A left
00001111	RRC	C	Rotate A right
00010111	RAL	C	Rotate A left through carry*
00011111	RAR	C	Rotate A right through carry
00100010 lb hb	SHLD a	-	Store H:L to memory*
00100111	DAA	ZSPCA	Decimal Adjust accumulator
00101010 lb hb	LHLD a	-	Load H:L from memory*
00101111	CMA	-	Complement A
00110010 lb hb	STA a	-	Store A to memory
00110111	STC	C	Set Carry flag
00111010 lb hb	LDA a	-	Load A from memory
00111111	CMC	C	Complement Carry flag
00DDD100	INR D	ZSPA	Increment register
00DDD101	DCR D	ZSPA	Decrement register*
00DDD110 db	MVI D,#	-	Move immediate to register*
00RP0001 lb hb	LXI RP,#	-	Load register pair immediate*
00RP0010 *1	STAX RP	-	Store indirect through BC or DE
00RP0011	INX RP	-	Increment register pair
00RP1001	DAD RP	C	Add register pair to HL (16 bit add)*
00RP1010 *1	LDAX RP	-	Load indirect through BC or DE
00RP1011	DCX RP	-	Decrement register pair
01110110	HLT	-	Halt processor
01DDDSSS	MOV D,S	-	Move register to register*
10000SSS	ADD S	ZSPCA	Add register to A
10001SSS	ADC S	ZSCPA	Add register to A with carry
10010SSS	SUB S	ZSCPA	Subtract register from A
10011SSS	SBB S	ZSCPA	Subtract register from A with borrow
10100SSS	ANA S	ZSCPA	AND register with A
10101SSS	XRA S	ZSPCA	Exclusive OR register with A
10110SSS	ORA S	ZSPCA	OR register with A
10111SSS	CMP S	ZSPCA	Compare register with A
11000011 lb hb	JMP a	-	Unconditional jump*
11000110 db	ADI #	ZSCPA	Add immediate to A
11001001	RET	-	Unconditional return from subroutine
11001101 lb hb	CALL a	-	Unconditional subroutine call
11001110 db	ACI #	ZSCPA	Add immediate to A with carry*
11010011 pa	OUT p	-	Write A to output port
11010110 db	SUI #	ZSCPA	Subtract immediate from A
11011011 pa	IN p	-	Read input port into A
11011110 db	SBI #	ZSCPA	Subtract immediate from A with borrow
11100011	XTHL	-	Swap H:L with top word on stack
11100110 db	ANI #	ZSPCA	AND immediate with A
11101001	PCHL	-	Jump to address in H:L
11101011	XCHG	-	Exchange DE and HL content
11101110 db	XRI #	ZSPCA	Exclusive OR immediate with A
11110011	DI	-	Disable interrupts
11110110	ORI #	ZSPCA	OR immediate with A
11111001	SPHL	-	Set SP to content of H:L
11111011	EI	-	Enable interrupts
11111110	CPI #	ZSPCA	Compare immediate with A
11CCC000	Rccc	-	Conditional return from subroutine
11CCC010 lb hb	Jccc a	-	Conditional jump*
11CCC100 lb hb	Cccc a	-	Conditional subroutine call
11NNN111	RST n	-	Restart (Call n*8)
11RP0001 *2	POP RP	*2	Pop register pair from the stack
11RP0101 *2	PUSH RP	-	Push register pair on the stack

*1 = Only RP=00(BC) and 01(DE) are allowed for LDAX/STAX
*2 = RP=11 refers to PSW for PUSH/POP (cannot push/pop SP).
When PSW is POP'd, ALL flags are affected.

Example program: Kill-The-Bit game

To enter the kill-the-bit game below into the ALTAIR:

1. Activate RESET
2. Set SW7-0 to first byte of program (041 octal, 00 100 001 binary)
3. Activate DEPOSIT
4. Set SW7-0 to next byte of program
5. Activate DEPOSIT NEXT
6. Repeat steps 4-5 until all bytes have been entered
7. If running the simulator on an Arduino MEGA, the game runs slow.
   To compensate, change the content of memory location 006 from 016 to 056 (all octal).

To run the game:

1. Activate RESET
2. Set SW15-8 switches to 0
3. Activate RUN

To save the game to Arduino EEPROM:

1. Set SW15 switch to 1, SW14-0 switches to 0
2. Activate AUX1 down

To load the game from Arduino EEPROM:

1. Set SW15 switch to 0, SW14-0 switches to 0
2. Activate AUX1 down

```
                ; Kill the Bit game by Dean McDaniel, May 15, 1975
                ;
                ; Object: Kill the rotating bit. If you miss the lit bit, another
                ;         bit turns on leaving two bits to destroy. Quickly
                ;         toggle the switch, don't leave the switch in the up
                ;         position. Before starting, make sure all the switches
                ;         are in the down position.
                ;
0000                        org   0
0000 210000                 lxi   h,0     ; initialize counter
0003 1680                   mvi   d,080h  ; set up initial display bit
0005 010E00                 lxi   b,0eh   ; higher value = faster
0008 1A       beg:          ldax  d       ; display bit pattern on
0009 1A                     ldax  d       ; ...upper 8 address lights
000A 1A                     ldax  d
000B 1A                     ldax  d
000C 09                     dad   b       ; increment display counter
000D D20800                 jnc   beg
0010 DBFF                   in    0ffh    ; input data from sense switches
0012 AA                     xra   d       ; exclusive or with A
0013 0F                     rrc           ; rotate display right one bit
0014 57                     mov   d,a     ; move data to display reg
0015 C30800                 jmp   beg     ; repeat sequence
0018                        end

Here is the program in octal for easier entry into the Altair:
000: 041 000 000 026 200 001 016 000
010: 032 032 032 032 011 322 010 000
020: 333 377 252 017 127 303 010 000
```

MITS Programming System II

[adapted from altairclone.com]

The MITS Programming System II allows development of 8080 assembly language applications on the Altair 8800 using just paper tape or cassette for mass storage. The package consists of an editor, assembler, debugger and a monitor that allows execution of these programs as well as the programs you may develop.
To use the programming package, the monitor program is loaded from paper tape or cassette in the same manner as loading BASIC. Once loaded, the monitor is then used to load the editor, assembler, debugger or user programs.
The monitor loads from paper tape or cassette using the same bootstrap loader as would be used for BASIC version 3.2. Sense switch settings are the same with the addition of A9 functionality:
A9 up – the monitor does NOT use serial input interrupts
A9 down – the monitor uses serial input interrupts

Loading the Monitor, Editor and Assembler

1) Raise A15 to select the simulated cassette interface as program source.
2) Raise A11 to select the simulated 88-SIO2 board as the console.
3) Raise A9 for no interrupts. The simulator currently only supports interrupts for serial I/O but not for cassette, so loading components from cassette will fail if A9 is down.
4) Set A7-A0 to 00000110 and then push AUX2 down to load the monitor.
5) The monitor prompt is two spaces and "?"
6) STOP! Do not type any commands to see "what happens." If what is typed is not a command, the monitor tries loading a program of that name from the cassette and hangs there until you provide that program. If the ABS device has been set to audio-cassette (AC), then Ctrl-C will return to the monitor prompt if interrupts were enabled during the boot process. Otherwise, follow the instructions below to restart the monitor from the front panel.
7) Assign the program load device to the cassette, type: "OPN ABS,AC<cr>"
8) Load the editor, type: "EDT<cr>" and then send the editor .bin file through the tape port. When loaded, the editor's prompt "*" is displayed. Type "E<cr>" to return the monitor.
9) Move the editor's buffer location into high memory so the assembler can reside in memory at the same time as the editor. For larger programs, more than the default 2K of buffer space will probably be needed as well. Here are settings for an 8K buffer:
   Type "DEP 5124<cr>" and enter "0<cr>" then "100<cr>" then ctrl-z.
   This specifies the 16-bit octal address 40000 (0x4000) for buffer start.
   Type "DEP 5530<cr>" and enter "0<cr>" then "140<cr>" then ctrl-z.
   This specifies the 16-bit octal address 60000 (0x6000) for buffer end.
10) Load the assembler, type: "AM2<cr>" and the send the assembler version 2 .bin file through the tape port. When loaded, the assembler prompt is "*ASM*<cr>". Type "EOA<cr>" (end of assembly) to return to the monitor.
11) DBG, EDT and ASM share the same memory space (see Memory Allocation section below). If two of them are loaded sequentially, the system will just start the second one loaded, even if invoking the first. For example, if EDT is loaded first and then ASM, the monitor will invoke ASM if EDT is typed as a command. To force reloading of a component, use CLR to remove it from the program table, i.e. in the situation mentioned before, typing CLR EDT and then EDT will invoke the editor.

The computer is now ready to iteratively edit, assemble and run/test a program.

Using the Editor and Assembler

1. To start the editor, type: “EDT<cr>” This starts the editor and clears the edit buffer. To subsequently re-edit a program, type “EDT(R)<cr>” This leaves the existing source code in memory. If the “(R)” is left off, the program source will be erased.
2. At the editor prompt type “I” for insert. Ctrl-Z exits the insert mode. “P” prints (displays) the file with line numbers. “W” displays the file without line numbers. “D line[,line]” deletes line(s). “R line” replaces a line. “I line” inserts after the specified line. “I” by itself inserts before the first line. “E” exits the editor and returns control to the monitor.
3. End programs with:
   - BEG start label
   - END program name
   - EOA

   Where “start label” is the program entry point, “program name” is a three character program name. “EOA” means end of assembly.
4. Assign source file input to come from the edit buffer instead of a device and declare file type of ASCII (text file): Type, “OPN FIL,EB,A<cr>” This only has to be done once during a session (or if the FIL device was subsequently assigned to a different device during the session).
5. Run the assembler: Type “AM2<cr>”
6. Tell the assembler to take input from a file: Type “FILE<cr>” The assembler will run and show errors and undefined symbols. Note: The “Undefined Symbols” heading is displayed even when there are no undefined symbols. Control is returned to the monitor.
7. Run the program by jumping to the starting address: Type “JMP xxxxxx” where xxxxxx is the program starting address in octal.
8. You can enter the program into the monitor’s program table as follows: After a successful assembly, re-enter the assembler and preserve symbols: Type “AM2(P)<cr>”
9. Type “RUN name<cr>” where name is the three character name for the program. The program will run. From here on, the program can be run by typing the “name” specified at the monitor prompt. This step does not have to be repeated after subsequent assemblies if the entry address has not changed.

Memory Allocation

- The monitor is about 2.5K in length and uses RAM up through 0x0A3F
- The editor is about 2K in length and resides just above the monitor at 0xA40 – 0x11B1, followed by the default 2K edit buffer space from 0x11B2 – 0x19B1.
- Two versions of the assembler are available. Each are about 3K in length. ASM (assembler version 1) loads at the same address as the editor (0xA40). This is inconvenient for the iterative cycle of edit, assemble and test. As an alternative, AM2 (assembler version 2) loads just above the editor at 0x1350 – 0x1D78. The assembler’s symbol table grows up from 0x1D78. Note that the default location of the edit buffer conflicts with the load address of AM2 as the edit buffer grows. Before using the editor for longer programs, the location of the edit buffer should moved to a free area in memory by using the monitor DEP command to patch the buffer start address into octal locations 5124-5125 and the buffer end address+1 into octal locations 5530-5531.
- When using the AM2 assembler, a program loaded at 0x2000 leaves about 650 bytes of symbol table space (0x1D78 – 0x1FFF). This is enough for small demo programs of 100 lines or less. Otherwise, a higher starting address should be used for the target program
- The debugger is is about 2K in length and overlays the editor at 0xA40 – 0x133F. The debugger and AM2 assembler can both reside in memory at the same time. This makes it easy to patch programs with the assembler while debugging the program.

Loading and Saving Files

1. Assign the FIL device to the audio cassette and specify ASCII files:
   Type “OPN FIL,AC,A<cr>”
2. In the editor, type “L<cr>” to load a source file. Type “S<cr>” to save a source file.
3. After loading a source file into the editor, you can re-assign FIL to the edit buffer for use as the assembler input: Type “OPN FIL,EB,A<cr>”
4. When you want to save an edited file after the edit and assemble process, be sure to set the FIL device back to AC before using the “S” command from within the editor.
5. Optionally, you can take assembler source file input from the audio cassette. After assigning FIL to the AC (see step 2), in the assembler, type “FILE name<cr>” where “name” is the three character source file name for the assembler to read. The source file is then read directly from the audio cassette instead of the edit buffer.

Additional Notes

- When using the edit buffer as the source for the assembler, the source file must be “rewound” by editing the source file in between successive assemblies.
- To restart the monitor, stop the machine, set all switches off except A6 (i.e., set address of octal 100). Raise examine, set front panel switches back the way they were at load time, then depress run.
- When sending large amounts of text (e.g. pasting source code into the serial terminal), it may come out corrupted because by the simulator serial rate is 9600 baud but the editor/assembler cannot process the arriving characters at that speed, so some get lost. You may need to drop the serial speed to 1200 baud for this to work (requires re-compiling the simulator). Note that this is not a problem when piping in examples (via AUX2 down) because in that case a new character is only read when the system is ready (i.e. executing an IN instruction).

Example usage #1: assemble directly from input

1)	*[STOP]+[RESET]*	reset Altair
2)	*[A15, A11, A9 up]*	A15 (load from tape), A11 (2SIO terminal), A9 (no interrupts),
3)	*[A2, A1 up]*	select program #6 (PS2 monitor)
4)	*[AUX1 down]*	load and run PS2 monitor and mount PS2 tape
5)	OPN ABS,AC	assign tape to ABS device (to load editor/assembler)
6)	AM2	start assembler (to load it into memory)
7)	*[CLR]*	un-mount the PS2 tape
8)	*[A15 down, A7 up, A2 down, A1 up]*	select ASM example #2 (“dump” example from PS2 documentation)
9)	*[AUX2 down]*	start ASM source code example replay
10)	RUN DUMP	runs DMP example directly from assembler

Example usage #2: insert into edit buffer, then assemble

1)	*[STOP]+[RESET]*	reset Altair
2)	*[A15, A11, A9 up]*	A15 (load from tape), A11 (2SIO terminal), A9 (no interrupts),
3)	*[A2, A1 up]*	select program #6 (PS2 monitor)
4)	*[AUX1 down]*	load and run PS2 monitor and mount PS2 tape
5)	OPN ABS,AC	assign tape to ABS device (to load editor/assembler)
6)	EDT	start editor (to load it into memory)
7)	E	end editor
8)	AM2	start assembler (to load it into memory)
9)	EOA	exit assembler
10)	DEP 5124	move edit buffer to 0x4000-0x6000 = 8K length
11)	0	start address 100-000 = 40000 octal = 0x4000
12)	100	

13) *[CTRL-Z]* end of input
14) DEP 5530
15) 0 end address 140-000 = 60000 octal = 0x6000
16) 140
17) *[CTRL-Z]* end of input
18) EDT start editor again (with new buffer)
19) I enter "insert" mode
20) *[CLR]* un-mount the PS2 tape
21) *[A15 down, A7 up, A2 down, A1 down, A0 up]* select ASM example #1 (PONG)
or [A15 down, A7 up, A2 down, A1 up] select ASM example #2 (DUMP)
22) *[AUX2 down]* start ASM source code example replay
23) *[CTRL-Z]* exit "insert" mode after example is loaded
24) E exit editor
25) OPN FIL,EB,A select edit buffer as input device
26) AM2(S) go into assembler (S parameter to print symbol table at end)
27) FILE load input file (from edit buffer)
28) EOA return to monitor
29) JMP 20000 run program

Example usage #3: insert into edit buffer, assemble, write to file, run file

1) *[STOP]+[RESET]* reset Altair
2) *[A15, A11, A9 up]* A15 (load from tape), A11 (2SIO terminal), A9 (no interrupts),
3) *[A2, A1 up]* select program #6 (PS2 monitor)
4) *[AUX1 down]* load and run PS2 monitor and mount PS2 tape
5) OPN ABS,AC assign tape to ABS device (to load editor/assembler)
6) EDT start editor (to load it into memory)
7) E end editor
8) AM2 start assembler (to load it into memory)
9) EOA exit assembler
10) DEP 5124 move edit buffer to 0x4000-0x6000 = 8K length
11) 0 start address 100-000 = 40000 octal = 0x4000
12) 100
13) *[CTRL-Z]* end of input
14) DEP 5530
15) 0 end address 140-000 = 60000 octal = 0x6000
16) 140
17) *[CTRL-Z]* end of input
18) EDT start editor again (with new buffer)
19) I enter "insert" mode
20) *[CLR]* un-mount the PS2 tape
21) *[A15 down, A7 up, A2 down, A1 down, A0 up]* select ASM example #1 (PONG)
22) *[AUX2 down]* start ASM source code example replay
23) *[CTRL-Z]* exit "insert" mode after example is loaded
24) E exit editor
25) OPN FIL,EB,A select edit buffer as input device
26) AM2(S,A) go into assembler (S parameter to print symbol table at end,
A to dump output binary to file)
27) FILE load input file (from edit buffer)

[wait for "SENSE SW 15 FOR DUMP" message]

28) *[A14 up, A7 down, AUX2 up]* start recording tape data (file #1 because A0 is still up)
29) *[A15 up, A15 down]* tells AM2 to start writing

[wait for "?" prompt from PS2 monitor]

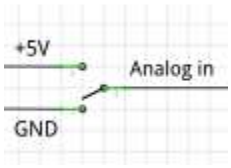
30) *[AUX2 up]* finish recording
31) *[AUX2 down]* start replay
32) PONG starts game

Hardware setup for Arduino Mega 2560

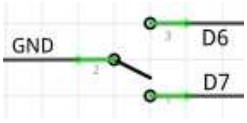
What makes the Arduino Mega 2560 perfect for this project is that it has exactly the right number of I/O pins to connect all the Altair's input/output elements:

Altair:	12	Status LEDs (INT,WO,STACK,HLTA,OUT,M1,INP,MEMR,PROT,INTE,WAIT,HLDA)
	8	Data bus LEDs (D0-D7)
	16	Address bus LEDs (A0-A15)
	16	Input switches (SW0-15)
	16	Function switches (RUN,STOP,EXAMINE,EXAMINE NEXT,DEPOSIT,DEPOSIT NEXT,RESET,CLR,PROTECT,UNPROTECT,AUX1 UP/DOWN,AUX2 UP/DOWN)
	2	Connections for serial RX/TX
	70	**(digital) inputs/outputs required**
Arduino Mega 2560:	54	Digital I/O pins
	16	Analog input pins (can be used for digital input)
	70	**I/O lines available**

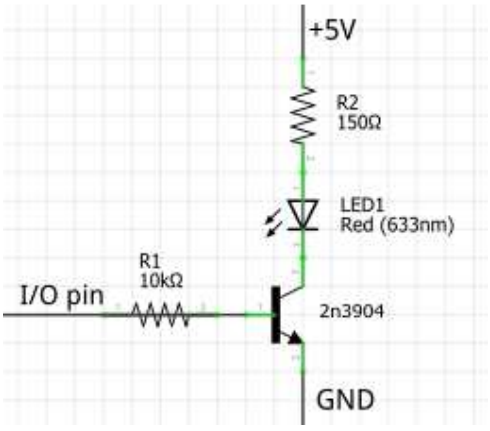
For simplicity, we connect the Altair's 16 input switches (SW0-15) to the Arduino's 16 analog inputs. These are SPDT On-On switches that alternate between two settings:



For the function switches we use the Arduino's internal pull-up resistors and just switch GND to the appropriate digital input. These are momentary SPDT (On)-Off-(On) switches. For example, the EXAMINE/EXAMINE NEXT switch is connected like this:



To drive the output LEDs we just use a simple transistor switch circuit for each LED (to prevent overload on the Arduino if too many of the LEDs are on at the same time):



See the following page for a mapping of exactly which front panel element maps to which Arduino I/O pin.

Pin mapping for Arduino Mega 2560

Mapping Arduino pin to Altair element

Arduino Pin	Altair Element	Direction	Arduino Pin	Altair Element	Direction
D0	Serial RX	in	**D35**	A10	out
D1	Serial TX	out	**D36**	A9	out
D2	AUX1 UP	in	**D37**	A8	out
D3	AUX1 DOWN	in	**D38**	INTE	out
D4	STEP	in	**D39**	PROT	out
D5	SLOW	in	**D40**	WAIT	out
D6	EXAMINE	in	**D41**	HLDA	out
D7	EXAMINE NEXT	in	**D42**	D7	out
D8	DEPOSIT	in	**D43**	D6	out
D9	DEPOSIT NEXT	in	**D44**	D5	out
D10	OUT	out	**D45**	D4	out
D11	M1	out	**D46**	D3	out
D12	INP	out	**D47**	D2	out
D13	MEMR	out	**D48**	D1	out
D14	AUX1 UP	in	**D49**	D0	out
D15	AUX1 DOWN	in	**D50**	HLTA	out
D16	PROTECT	in	**D51**	STACK	out
D17	UNPROTECT	in	**D52**	WO	out
D18	RESET	in	**D53**	INT	out
D19	CLR	in	**A0**	SW0	in
D20	RUN	in	**A1**	SW1	in
D21	STOP	in	**A2**	SW2	in
D22	A0	out	**A3**	SW3	in
D23	A1	out	**A4**	SW4	in
D24	A2	out	**A5**	SW5	in
D25	A3	out	**A6**	SW6	in
D26	A4	out	**A7**	SW7	In
D27	A5	out	**A8**	SW8	In
D28	A6	out	**A9**	SW9	In
D29	A7	out	**A10**	SW10	In
D30	A15	out	**A11**	SW11	In
D31	A14	out	**A12**	SW12	In
D32	A13	out	**A13**	SW13	In
D33	A12	out	**A14**	SW14	In
D34	A11	out	**A15**	SW15	In

Mapping Altair element to Arduino pin

Altair Element	Arduino Pin	Direction	Altair Element	Arduino Pin	Direction
SW0	A0	in	**A0**	D22	out
SW1	A1	in	**A1**	D23	out
SW2	A2	in	**A2**	D24	out
SW3	A3	in	**A3**	D25	out
SW4	A4	in	**A4**	D26	out
SW5	A5	in	**A5**	D27	out
SW6	A6	in	**A6**	D28	out
SW7	A7	In	**A7**	D29	out
SW8	A8	in	**A8**	D37	out
SW9	A9	in	**A9**	D36	out
SW10	A10	in	**A10**	D35	out
SW11	A11	in	**A11**	D34	out
SW12	A12	in	**A12**	D33	out
SW13	A13	in	**A13**	D32	out
SW14	A14	in	**A14**	D31	out
SW15	A15	in	**A15**	D30	out
STOP	D21	in	**D0**	D49	out
RUN	D20	in	**D1**	D48	out
SINGLE STEP	D4	in	**D2**	D47	out
SLOW	D5	in	**D3**	D46	out
EXAMINE	D6	in	**D4**	D45	out
EXAMINE NEXT	D7	in	**D5**	D44	out
DEPOSIT	D8	in	**D6**	D43	out
DEPOSIT NEXT	D9	in	**D7**	D42	out
RESET	D18	in	**INT**	D53	out
CLR	D19	in	**WO**	D52	out
PROTECT	D16	in	**STACK**	D51	out
UNPROTECT	D17	in	**HLTA**	D50	out
AUX1 UP	D14	in	**OUT**	D10	out
AUX1 DOWN	D15	in	**M1**	D11	out
AUX2 UP	D2	in	**INP**	D12	out
AUX2 DOWN	D3	in	**MEMR**	D13	out
Serial RX	D0	in	**PROT**	D39	out
Serial TX	D1	out	**INTE**	D38	out
			WAIT	D40	out
			HLDA	D41	out

Hardware setup for Arduino Due

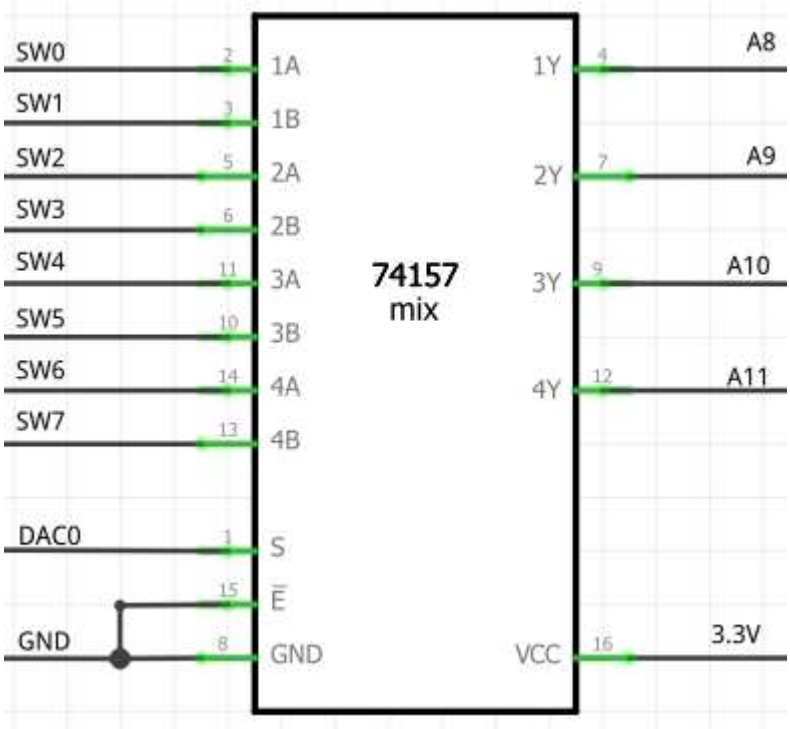
While the hardware setup for the Mega is very simple, there are drawbacks: the Mega has only 8k of RAM (leaving only 6k emulated RAM) and with its clock speed of 16MHz emulation runs at about 25% speed of the original.

Enter the Arduino Due: it has 96k RAM and a processor running at 84MHz. But it does have some shortcomings on the hardware side:

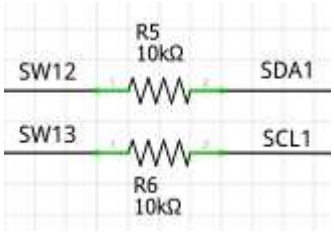
- Only 12 (instead of 16) analog inputs
- The serial TX/RX pins (digital pin D0 and D1) are hardwired to the USB controller, making them impossible to be used for serial communication that does **not** go through USB. I want the serial communication to go through a Bluetooth link, so pins D0/D1 can't be used for serial communication (or anything else for that matter).
- On the Mega, I/O pins that correspond to the same processor register are neatly located next to each other. On the Due, they are not.
- The reset circuit on (some) Due boards is not very reliable at power-up (this seems to be a common problem), leaving the Due sometimes after power-up in a blocked state.

This makes the hardware setup a bit more challenging for the Due. Here are the changes required (in relation to the Mega hardware setup) to address the differences:

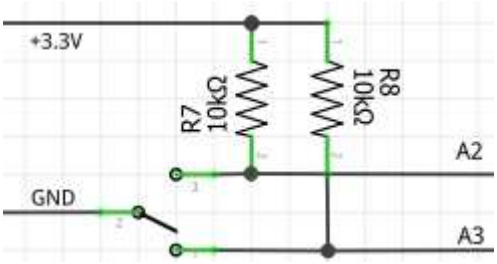
- The positive voltage input to SW0-15 needs to be connected to 3.3V instead of 5V because Due inputs can only accept a maximum of 3.3V.
- Using a 74HC157 quad 2-input multiplexer, we can duplicate analog inputs A8-A11, giving us 16 analog inputs again (the multiplexer is digital but we're using the analog inputs as digital inputs anyways). We use the Due's DAC0 output to drive the data select input of the multiplexer so we don't lose any other I/O pins:



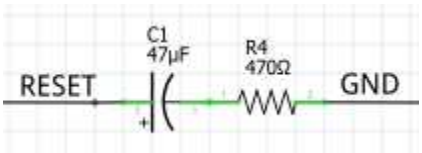
- To solve the serial I/O problem, we use the TX1/RX1 (D18/D19) pins for serial communication. We're still short two digital I/O pins though (because we can't use D0/D1). To compensate we use the SDA1/SCL1 pins (for the second I2C interface) as digital inputs by directly reading the processor register. To make sure we don't accidentally cause a short by grounding them when (for whatever reason) the Due decides to make them outputs, I added a 10k resistor (not sure if that's necessary but it doesn't hurt):



- Since the pin layout is already messed up and not as neat as with the Mega, I decided to make some changes that will help improve performance:
  - The SW8-15 address switches (which can be queried by executing an "IN 0xff" instruction in the Altair's program) are now connected to digital input pins (instead of analog inputs) because those are faster to query.
  - The SW0-7 address switches are connected to analog inputs A8-A11 (through the multiplexer). SW0-7 settings only have an effect when the Altair is stopped, so the fact that querying them is slow does not affect emulation performance.
  - Analog inputs 0-7 are connected to function switches that also only have an effect while the Altair is stopped. Since the function switches are momentary switches, this requires adding some pull-up resistors to get stable analog readings when the switches are open. For example, the EXAMINE/EXAMINE NEXT function switch is connected like this:



- To work around the reset problem, a capacitor and resistor keep the RESET line low for a bit longer at power-up. This has worked fine for me and not caused any side effects



See the next page for the complete pin mapping.

Pin mapping for Arduino Due

Mapping Arduino pin to Altair element

Arduino Pin	Altair Element	Direction	Arduino Pin	Altair Element	Direction
D0	Unused		**D34**	A0	out
D1	Unused		**D35**	A1	out
D2	INT	out	**D36**	A2	out
D3	WO	out	**D37**	A3	out
D4	STACK	out	**D38**	A4	out
D5	HLTA	out	**D39**	A5	out
D6	OUT	out	**D40**	A6	out
D7	M1	out	**D41**	A7	out
D8	INP	out	**D42**	SW14	in
D9	MEMR	out	**D43**	SW15	in
D10	WAIT	out	**D44**	A15	out
D11	D7	out	**D45**	A14	out
D12	INTE	out	**D46**	A13	out
D13	PROT	out	**D47**	A12	out
D14	D4	out	**D48**	A11	out
D15	D5	out	**D49**	A10	out
D16	SW9	in	**D50**	A9	out
D17	SW8	in	**D51**	A8	out
D18	Serial TX	out	**D52**	RESET	in
D19	Serial RX	in	**D53**	CLR	in
D20	RUN	in	**A0**	STEP	in
D21	STOP	in	**A1**	SLOW	in
D22	HLDA	out	**A2**	EXAMINE	in
D23	SW10	in	**A3**	EXAMINE NEXT	in
D24	SW11	in	**A4**	DEPOSIT	in
D25	D0	out	**A5**	DEPOSIT NEXT	in
D26	D1	out	**A6**	PROTECT	in
D27	D2	out	**A7**	UNPROTECT	in
D28	D3	out	**A8**	Mux 1Y (SW0/1)	in
D29	D6	out	**A9**	Mux 2Y (SW2/3)	in
D30	AUX1 UP	in	**A10**	Mux 3Y (SW4/5)	in
D31	AUX1 DOWN	in	**A11**	Mux 4Y (SW6/7)	in
D32	AUX2 UP	in	**DAC0**	Mux selector for SW0,2,4,6 / 1,3,5,7	out
D33	AUX2 DOWN	in			
			SDA1	SW12	in
			SCL1	SW13	in

Mapping Altair element to Arduino pin

Altair Element	Arduino Pin	Direction	Altair Element	Arduino Pin	Direction
SW0	* Mux 1A	in	**A0**	D34	out
SW1	* Mux 1B	in	**A1**	D35	out
SW2	* Mux 2A	in	**A2**	D36	out
SW3	* Mux 2B	in	**A3**	D37	out
SW4	* Mux 3A	in	**A4**	D38	out
SW5	* Mux 3B	in	**A5**	D39	out
SW6	* Mux 4A	in	**A6**	D40	out
SW7	* Mux 4B	In	**A7**	D41	out
SW8	D17	in	**A8**	D51	out
SW9	D16	in	**A9**	D50	out
SW10	D23	in	**A10**	D49	out
SW11	D24	in	**A11**	D48	out
SW12	SDA1	in	**A12**	D47	out
SW13	SCL1	in	**A13**	D46	out
SW14	D42	in	**A14**	D45	out
SW15	D43	in	**A15**	D44	out
STOP	D21	in	**D0**	D25	out
RUN	D20	in	**D1**	D26	out
SINGLE STEP	A0	in	**D2**	D27	out
SLOW	A1	in	**D3**	D28	out
EXAMINE	A2	in	**D4**	D14	out
EXAMINE NEXT	A3	in	**D5**	D15	out
DEPOSIT	A4	in	**D6**	D29	out
DEPOSIT NEXT	A5	in	**D7**	D11	out
RESET	D52	in	**INT**	D2	out
CLR	D53	in	**WO**	D3	out
PROTECT	A6	in	**STACK**	D4	out
UNPROTECT	A7	in	**HLTA**	D5	out
AUX1 UP	D30	in	**OUT**	D6	out
AUX1 DOWN	D31	in	**M1**	D7	out
AUX2 UP	D32	in	**INP**	D8	out
AUX2 DOWN	D33	in	**MEMR**	D9	out
Serial RX	D19	in	**PROT**	D13	out
Serial TX	D18	out	**INTE**	D12	out
			WAIT	D10	out
			HLDA		out

* These are pins on the 74HC157 Multiplexer whose 1Y-4Y outputs are connected to pins A8-A11 on the Arduino

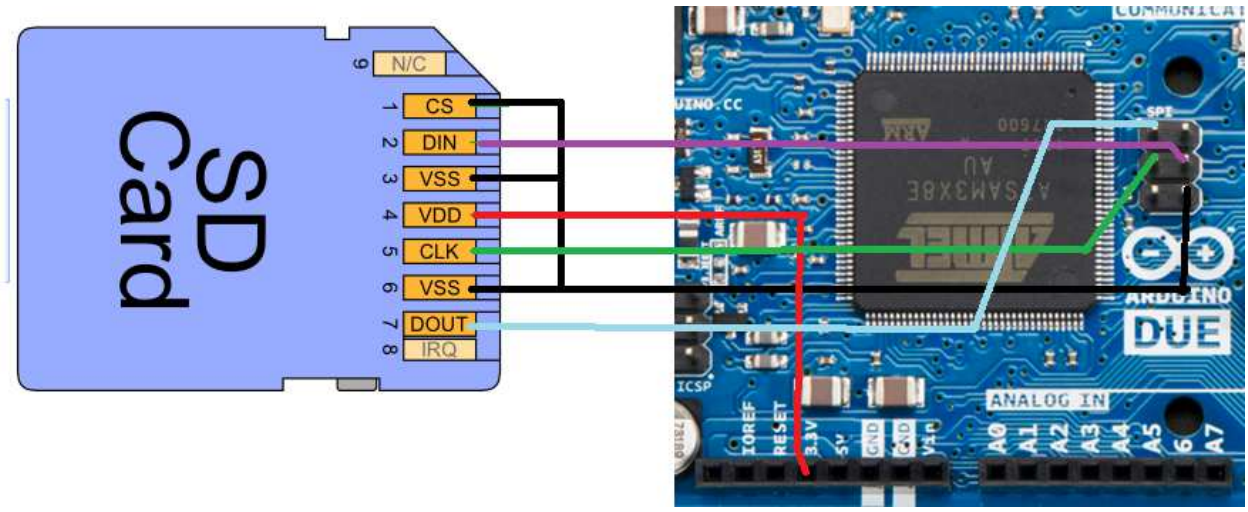
Wiring an SD card to Arduino Due

Standard Arduino SD card shields will not work with the Arduino Due because the SPI pins are not connected to the D13-D11 pins as in other Arduino board. That is a good thing because we're already using the D13-D11 pins for front panel elements.

On the DUE, the SPI pins are **only** available on the separate 2-row, 6-pin SPI header (labeled "SPI" on the board).

There are commercial products such as the SparkFun Level Shifting microSD Breakout board that provide an SD card slot. That board is certainly works but is overkill since the Arduino DUE (like the SD cards) operates on 3.3V so no level shifting is required.

In fact, an SD card can be wired directly (without any other required electronic elements) to the SPI header on the Due:



- Unfortunately, there is no 3.3V pin on the SPI header on the Arduino Due so that wire must go to the (separate) 3.3V pin. Do not wire the 5V output from the SPI header to the card. Doing so could damage the SD card and/or the Due.
- The CS (chip select) pin is wired directly to GND, so the SD card is always selected. That saves us from having to find another I/O pin on the Arduino to use for chip select. The SD library on the Arduino requires to specify a Chip Select output pin but the simulator software sets that to the HLDA light output pin which as a side effect gives a "sd card active" visual indicator.

I recommend getting a microSD card with a microSD-to-SD adapter. Take the microSD card out of the adapter and create a cable by soldering wires directly to the adapter's pins and connecting them to the SPI header using the wiring given above. The adapter now serves as the socket for the microSD card, which can be plugged in and taken out easily.