Appendix C

ArdEx Command Reference

In ArdEx, everything entered by the user is either a command or an instruction. Commands are used to control the ArdEx environment, providing ways to load, save or display memory, or to execute programs.

Instructions are executed by the processor. Anything preceded by a number is treated as an instruction to be stored at that step address for later execution. Instructions entered interactively will be executed immediately.

Instructions are documented in Appendix D.

- **BASE 2 | 10 | 16, num**
  BASE 10 | 16
  In the first form, *num* will be displayed in the selected number base. Num can be a valid number including `0x` or `0b` prefix.

  In the second from, subsequent `DUMP`, `LIST`, `STAT` and `STEP` commands will show certain numbers in the selected number base. The default number base is 10.

- **BOOT flag(, step)**
  Sets behaviour of ArdEx on hardware reset. By default, or if *flag* is 0, ArdEx boots with all steps initialised to `STOP` instructions and all RAM initialised to 0. If bit 0 of *flag* is set, all instructions will be automatically loaded by the `LOAD` command. If bit 1 of *flag* is set, all RAM will be automatically loaded by the `RAML` command. If bit 2 of *flag* is set, execution will be started at *step* (default 0) by the `RUN` command.

- **BRK (brkid(, step))**
  Manages user-defined breakpoints. With no arguments, active breakpoints are listed showing their *brkid* and *step*. With one argument, breakpoint *brkid* is disabled. With two arguments, breakpoint *brkid* is enabled at program step *step*. When execution reaches this step, execution will halt and the breakpoint number and the register contents will be displayed. There are only two breakpoints available: *brkid* 0 or 1.

- **CONT**
  Resumes execution following a breakpoint or a user interrupt with `<Ctrl>-C`.

- **DUMP addr(, nbytes)**

Displays memory in the currently selected number base starting at *addr*. If *nbytes* is specified, that many bytes will be displayed. The default is 16.

- **LIST (step1(, step2))**
  Displays instructions stored in program memory. With no arguments, all 256 steps are listed. With one argument, just the instruction at that step is listed. With two arguments, all instructions from *step1* to *step2* are listed.

- **LOAD (step(, step))**
  Loads instructions from EEPROM into program memory. With no arguments, all 256 steps are loaded. With one argument, just the instruction at that step is loaded. With two arguments, all instructions from *step1* to *step2* are loaded.

  Resulting program steps may be illegal if EEPROM has not been written by a previous `SAVE` command.

- **NEW**
  Sets all 256 program steps to the `STOP` instruction. Sets all registers to zero.

- **OVER**
  If the instruction about to be executed is a `CALL`, execution continues until the corresponding `RET` has been executed or a breakpoint occurs. It then reports register values. If a breakpoint does occur, execution will not be stopped by the `RET` instruction.

  For all other instructions, `OVER` single-steps just like `STEP`.

- **RAML**
  Load all user RAM from EEPROM. Results are unpredictable if the EEPROM has not been written by a previous `RAMS` command.

- **RAMS**
  Save all user RAM to EEPROM.

- **RELO start,end,newstart**
  Relocate a block of instructions to a different step. All three arguments are required. The first two arguments, *start* and *end*, define a block of instructions. The third argument, *newstart*, is the step to which the instruction at *start* is to be moved, with the other instructions following.

  Steps vacated by the move will be filled with `STOP` instructions. Source and destination blocks may overlap. Moves are allowed upwards or downwards in program space. A `RELO` command will be rejected if the moved block would extend past step 255.

  All *branch constant* branches (anywhere in program space) to an instruction in the original block will be adjusted to jump to the instruction's new location.

  All *branch register* branches will have their step numbers listed. A human will need to check if these can be affected. If *start* and *newstart* are the same, this command will simply list all steps with *branch register* instructions.

  - `RELO 30,30,50` – move the instruction at step 30 to step 50. The original instruction at step 50 will be lost. The instruction at step 30 will be replaced with `STOP`.
  - `RELO 30,40,10` – move all instructions at steps 30 to 40 inclusive to steps 10 to 20. The 11 instructions at 30 to 40 will be replaced with `STOP`.

– `RELO 30,40,31` – move all instructions at steps 30 to 40 inclusive up by one step number. This makes room for one new instruction at step 30.

The following example shows how to make space for two instructions to be inserted into a block of code.

```
ArdEx> LIST 0,5
0        JMP      2
1        MOV      #0,PORTB
2        WAIT     1000
3        ADD      #1,PORTB
4        JMP      2
5        STOP
ArdEx> RELO 2,4,4
ArdEx> LIST 0,7
0        JMP      4
1        MOV      #0,PORTB
2        STOP
3        STOP
4        WAIT     1000
5        ADD      #1,PORTB
6        JMP      4
7        STOP
```

Note in the above example that the branches at step 0 and step 4 are both updated though only the branch at step 4 is part of the block of instructions being moved.

- **RUN (step)**
  Begin free execution. With no arguments, begin at step 0, otherwise start at step *step*.

- **S**
  Processor executes one instruction then displays the next instruction to be executed and returns control to the user,

- **SAVE (step(, step))**
  Saves instructions from program memory to EEPROM. With no arguments, all 256 steps are saved. With one argument, just the instruction at that step is saved. With two arguments, all instructions from *step1* to *step2* are saved.

- **SS | STEP**
  SS | STEP nsteps
  SS | STEP step, nsteps
  Much like the `S` command. With no arguments, a single instruction is executed. With one argument, *nsteps* instructions are executed. With two arguments, *nsteps* instructions are executed starting with the instruction at *step*.

  In all cases, register values and the next instruction to be executed are displayed just before control is returned to the user.

- **STAT**
  Displays all register values in the current number BASE along with the values in Carry and Zero flags and the `CALL` nest depth. The next instruction to be executed is also shown.

D.3 Instruction Set Summary

Instruction		Action	Z	C
ADD	*src,dst*	$dst = src + dst$	•	•
ADDC	*src,dst*	$dst = src + dst + Carry$	•	•
AND	*src,dst*	$dst = src \cap dst$	•	
BIC	*src,dst*	$dst = \overline{src} \cap dst$		
BIS	*src,dst*	$dst = src \cup dst$		
BIT	*src,dst*	DISCARD$(src \cap dst)$	•	•
CALL	*step*	SAVE$(current_step)$; $jump\ step$		
CMP	*src,dst*	DISCARD$(dst - src)$	•	•
IN	*dst*	$dst =$ INPUT()	•	
JC	*step*	$if\ Carry\ jump\ step$		
JEQ	*step*	$if\ Zero\ jump\ step$		
JMP	*step*	$jump\ step$		
JNC	*step*	$if\ \overline{Carry}\ jump\ step$		
JNE	*step*	$if\ \overline{Zero}\ jump\ step$		
MOV	*src,dst*	$dst = src$		
OR	*src,dst*	$dst = src \cup dst$	•	
OUT	*val16*	OUTPUT$(val16 \cap$ `0xFF`$)$		
RET		$jump\ last\ saved\ step$		
ROL	*dst*	$dst = 2 \times dst + Carry$	•	•
ROR	*dst*	$dst = dst/2 +$ `0x80` $\times\ Carry$	•	•
SL	*dst*	$dst = 2 \times dst$	•	•
SR	*dst*	$dst = dst/2$	•	•
STOP		HALT()		
SUB	*src,dst*	$dst = dst - src$	•	•
SUBC	*src,dst*	$dst = dst - src - Carry$	•	•
WAIT	*val16*	PAUSE$(val16 \times 100\mu s)$		