Application Note 001
LCD and 4 button keypad driver.

This paper contains and compares four programs:

1) An appnote for the BASIC Stamp I.
2) AmrBASIC adaptation of same program.
3) Same AmrBASIC program, but modularized for interactive testing.
4) Same program in AmrForth.

The programs are listed at the end of the paper, with line numbers for referencing. The original program, #1, is a simple application which demonstrates the power of small microcontrollers. It can be found at http://www.parallax.com.

Current executable versions of programs 2 and 3 can be found in the directory ~/amrforth/v6/017/basic/ for Linux or \amrforth\v6\017\basic for Windows.

Program 4 can be found in its current executable form in ~/amrforth/v6/017/example for Linux and\amrforth\v6\017\example for Windows.

----- PROGRAMS #1 AND #2. -----

First, let's compare the original, Program #1, with the first AmrBASIC version, Program #2. Program #2 is intended to be as similar to program #1 as possible, given the differences between the BASIC Stamp and the amr gadget.

In each program lines 6-16 define symbols, with the same syntax. Note that the switches use different pins because the UART on the amr gadget is hardwired to ports 0 and 1 for the 017 chip. Also, since the amr gadget has a UART, there is no need for the S_in and S_out symbols.

On line 20 of program #1 pins are set to 0. On line 19 of program #2 we set pins to %00000011, clearing all but the UART pins. 0 probably would have worked as well, since the UART has control of those pins anyway. On lines 26-43 of program #1 the LCD is initialized. Lines 24-46 of program #2 do the same thing. The differences are that we changed the name of E to En for Enable to avoid conflicting with the 'e' command in the interpreter, to locate source code. Where the original has 'pause 10' #2 has 'pause 100' because the amr gadget is much faster. Also some 'pause 10' lines were added for the same reason. The numbers were discovered by trial and error.

Both programs fall through after initialization to the main routine on lines 47 and 50 respectively, and the next four lines are identical for each.

The next major routine is ' wr_LCD' , at lines 53 and 56. Where the original program calls out b2 directly, the amrBASIC program defines the symbol ' nibble' for b2. The both programs clear the data pins in the first line, though data is on the low nibble for program #1 and on the high nibble for program #2, as mentioned before. Program #1 shifts the high nibble of ' char' into the low nibble of ' nibble' before oring that onto the data pins. Program #2 simply masks off the high nibble of ' char' , moving that into ' nibble' , before oring the result onto the data pins. Then each program, at lines 56 and 59, sends a pulse out the E or En line. Since ' pulsout' does its timing with machine cycles, the amr gadget goes for 10 instead of 1, being at least that much faster.

The ' bksp' routine starts at line 64 and 67 respectively. Note that this is not a subroutine and therefore not something that can be tested easily from the interactive command line, but on the BASIC Stamp of course there is no interactive command line. In both ' modular.bas' and the forth example ' lcdkey.fs' this routine is renamed ' handler' and is a callable subroutine. In both programs #1 and #2 ' bksp' is almost the same. The variable ' char' has already been filled with a character from the serial port. That character is compared to a series of values case by case to decide what to do with the character. In program #1 if ' char' is greater than 13, the value of a carriage return, then the character is output on the LCD. In program #2 we only output the character if it is a space or greater. It didn' t make sense to try to put control characters on the LCD, with possibly unexpected results. In each program if the ' char' is 3, control C, then the display is cleared. If the ' char' is 13, carriage return, then the buttons are watched until one is pressed and its number is sent out the serial port. If ' char' is 8, backspace, then a backspace is performed, otherwise nothing is done and the program loops back to the main routine.

Starting at lines 73 and 77 we have several more subroutines and code fragments. First is the ' back' subroutine. It is a subroutine because it needs to be called twice for each backspace. ' low RS' puts the LCD into instruction mode. $10 is the instruction for moving the cursor back one space. It is send to the LCD via the ' wr_LCD' subroutine. ' high RS' puts the LCD back into data or character mode again. ' return' returns from the subroutine to its caller. The only difference between the two versions is that the original used decimal 16 as the instruction, and #2 uses $10, the same number but in hexadecimal. We' re simply used to thinking in hexadecimal in cases like this. It doesn' t really matter.

On lines 79 and 83 is the clear routine. The difference is the ' pause 100' on line 87 of program #2. The amr gadget is so fast we needed to have it wait on the LCD before trying to do anything else. As in the ' back' subroutine, the RS line is pulled low to initiate instruction mode. The instruction, 1 in this case, is written to the LCD via the 'wr_LCD' subroutine. The RS line is pulled back high for future character writes, and finally it jumps back to the main routine.

On lines 89 and 90 we have the 'cret' routine, which might better be called 'read-buttons'. In each program the 'dir' register is used to turn the data lines of the LCD into inputs. This is so that they can be reused to read the four buttons. On the BASIC Stamp the low nibble of pins is used. On the amr gadget the high nibble is used. On the amr gadget the data pins need to be written high as well, though this is not needed on the BASIC Stamp. I will admit that I do not know why this is the case. After the port is setup, both programs identically read each switch in turn until one is found to be depressed, then its number is send out the serial port as an ascii character. The 'xmit' routine on lines 100 and 105 do the sending, then set the port bits back to outputs for the operation of the LCD. Once again the amr gadget has the data pins set back to zero. This shouldn't be necessary, as the 'wr_LCD' routine sets the data pins before enabling the LCD.

That's it, the two programs are very similar. Program #1 was used as a model for program #2 and changes were made only when deemed necessary because of hardware differences.

----- PROGRAMS #2 AND #3 -----

Both programs #2 and #3 are written for the amr gadget. The difference is that program #3 is intended to be tested interactively. Each routine is a subroutine instead of a code fragment to be jumped to. The programmer can then initialize the LCD and get control back. Then each routine can be tested separately, 'back', 'clear', 'cret', and 'handler', as we're used to doing in forth. The main routine in program #3 has just three lines:

```
main:  serin char
       gosub handler
       goto main
```

which pretty much sums up what the program does, read a character over the serial link and do something with it. The original program's main routine is not so clear:

```
main:  serin char
       goto bksp
out:   gosub wr_LCD
       goto main
```

It is not obvious that this is a single routine given the 'goto' right in the middle of it. It just happens that the 'bksp' routine jumps back to 'out' when the character is to be displayed, but it takes some reading to determine that.

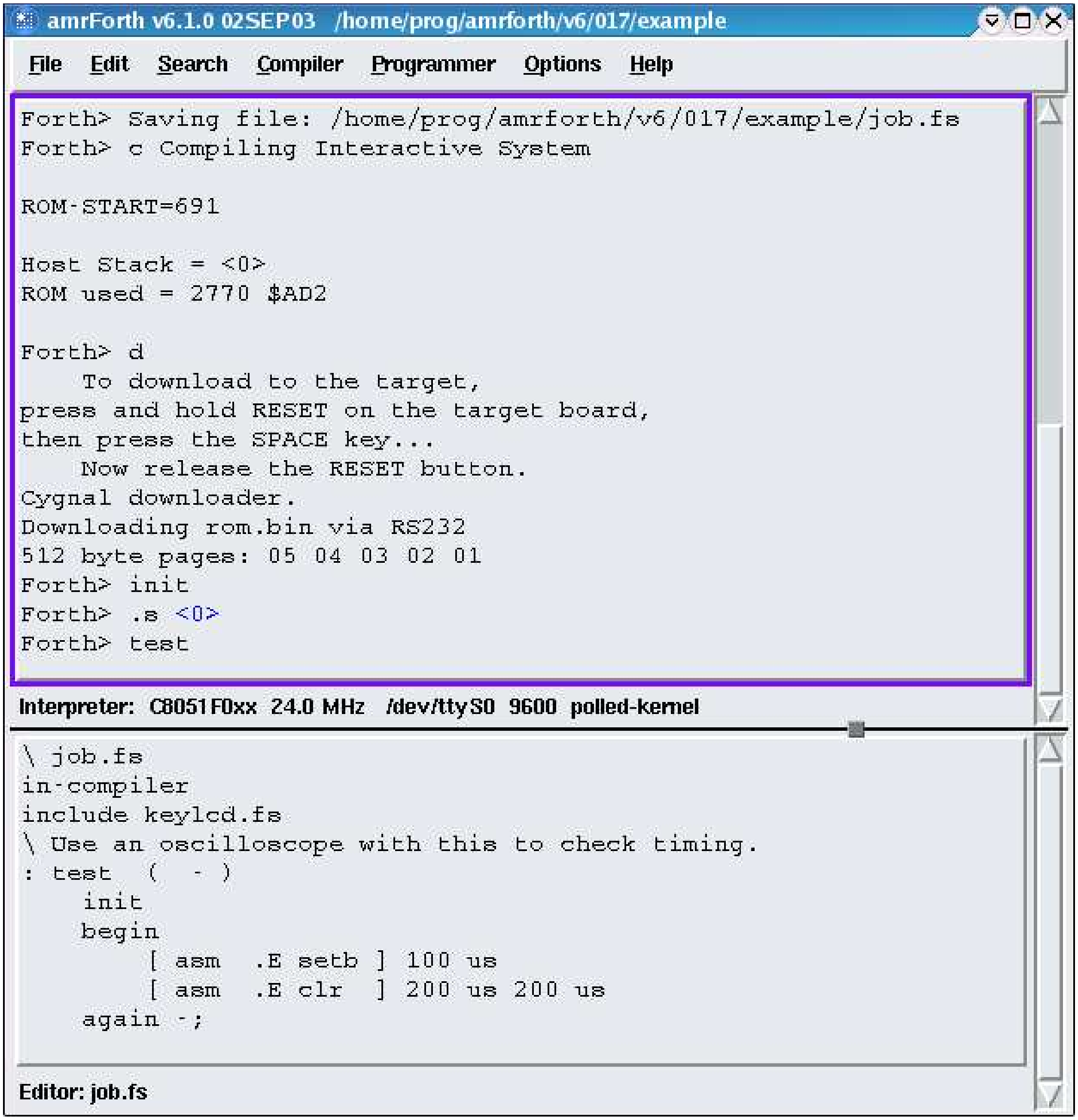
----- PROGRAM #4 -----

Our final example is done in forth. You might think of forth as the operating system underlying amrBASIC. AmrForth has access to an inline assembler for maximum efficiency. On lines 7-10 are some assembler macros. ' .E' becomes the name of the i/o pin at P0.2, ' .RS' is the name of pin P0.3, ' pins' is the name of P0, and ' dirs' is the name of the PRT0CF register, the Port 0 Config Register. These names help the forth program look more like the BASIC program and hide some of the underlying hardware details.

At line 16 we come to the first actual code. In forth a new assembler language subroutine is started with the word ' code' . A high level forth definition starts with the word ' :' . The first code word is ' instruction' , which clears the RS bit on the LCD to put it into instruction mode. A code word in forth ends with the macro ' next' and the word ' c;' . The ' next' macro just assembles an RET instruction, return from subroutine. When the word ' instruction' is used later in the program, it will compile a call to the ' instruction' subroutine. On line 17 the word ' data' is defined to set the RS bit to put the LCD back into data mode. Assembler instructions are used here because high level forth doesn' t easily handle bits directly in registers, but assembler does. This is not a criticism of forth. Programming in assembler is an essential and powerful part of programming in forth for embedded systems.

On line 22 we have the high level forth definition of ' pulseout-E' . We count cycles to delay for about 100 microseconds inside the pulse. The BASIC program used a timer interrupt. This is simpler. Note that us is defined such that it takes a byte as input. You can only delay for 255 microseconds at a time. That' s why ms is defined to call 250 us four times in a row, instead of just saying 1000 us. The loop inside us is simpler and more predictable using a byte instead of using two bytes.

Here is an example of a short test program written in the job.fs file. It can be used both to test the ' us' word and the .E macro.


amrForth v6.1.0 02SEP03 /home/prog/amrforth/v6/017/example
File Edit Search Compiler Programmer Options Help

```
Forth> Saving file: /home/prog/amrforth/v6/017/example/job.fs
Forth> c Compiling Interactive System

ROM-START=691

Host Stack = <0>
ROM used = 2770 $AD2

Forth> d
    To download to the target,
press and hold RESET on the target board,
then press the SPACE key...
    Now release the RESET button.
Cygnal downloader.
Downloading rom.bin via RS232
512 byte pages: 05 04 03 02 01
Forth> init
Forth> .s <0>
Forth> test
```

Interpreter: C8051F0xx 24.0 MHz /dev/ttyS0 9600 polled-kernel

```
\ job.fs
in-compiler
include keylcd.fs
\ Use an oscilloscope with this to check timing.
: test  (  - )
    init
    begin
         [ asm  .E setb ] 100 us
         [ asm  .E clr  ] 200 us 200 us
    again -;
```

Editor: job.fs

You can see that the test word is an endless loop that starts by initializing the system, then toggling the .E pin high for 100 us and low for 400 us. Remember that ' us' only works up to 255 us, so we run ' 200 us' twice. If an oscilloscope is connected to the .E pin you should be able to see that the signal is high for 100 us and low for 400 us. Be sure to either delete or comment out such test code before shipping the final program.

The ' write-lcd' word starts on line 25. It is based very closely on the ' wr_LCD' word in the BASIC programs. It uses a new feature of amrForth which allows assembler code to be mingled with high level forth code in the same definition.

The ' [' word on lines 26, 29, and 35 place the forth compiler back into interpret mode. The word ' in-assembler' switches the main vocabulary to assembler instead of forth, and the following code is assembled directly into the word. ' %00001011 # pins anl' specifies pins in binary to be anded with the i/o port, pins. The result is that the four data pins are set to zero. Once again it is easier to directly handle registers in assembler than in forth. The word ']' puts us back into compile mode. The character on the stack needs to be split into separate nibbles for the four bit LCD interface, so we ' dup' the character. On line 29 we switch back to assembler. To read a byte from the data stack into the accumulator we increment the data stack pointer with ' SP inc' , then read the next byte into A with the predefined macro ' Apop' . We ' and' the accumulator with %11110000 to make all but the data pins zero. Then we ' or' the accumulator onto the pins with ' A pins orl' to get the data nibble ready. The word ' pulseout-E' clocks those four bits into the LCD with a pulse on the Enable pin. We' re back in compile mode now, so we can use high level ' 16 *' to shift the next four bits into place, switch back to assembler to place those bits on the LCD data pins, and back in forth we call ' pulseout-E' to clock those bits into the LCD as well. The same work could be done completely in forth or completely in assembler, but bit and register handling is easier in assembler, and multiplication and data passing is easier in forth.

Line 42 starts the definition of ' no-pullups' . This is a simple code word that tells the i/o system of the Cygnal chip not to apply weak pullups to its input ports, by setting a bit in the XBR2 register. This word is called in initialization. The default is to apply weak pullups. They get in the way of pin sharing between the LCD and the buttons in this application.

On line 47 we have the word ' init-lcd' which is modeled closely after the BASIC program. Once again we mix forth and assembler to make the word simpler and more efficient. Note the forth word ' ms' which delays for a given number of Milliseconds. It is based on ' us' above, and counts cycles in a loop.

The code word ' pins@' is defined on line 69. In assembler it reads the pins port into the accumulator A, then pushes A onto the data stack for high level forth to deal with. The assembler macros ' Apush' and ' 0push' are predefined in amrForth and make it simpler to get data on and off the data stack in assembler words.

The high level forth word ' button' on line 74 gets into an endless begin,again loop. It reads the button port with ' pins@' , then checks each data bit until it finds a button pressed, or loops back to try again. If a button is found, the pin data is dropped from the stack, the ascii code for that button is placed on the stack, and the loop is ended with ' exit' . Actually ' exit' is an alias for ' RET' in assembler or ' return' in BASIC.

Two more code words follow on lines 83 and 88, called ' reading' and ' writing' . They set the data port pins and dirs to allow reading the buttons, then restore the pins to allow writing the LCD. They are used in the next word, ' read-button' .

On line 93, ' read-button' is a high level forth word that is fairly self explanatory. ' Reading' of course allows the button to be read. ' Button' waits for a button to be pressed. ' Emit' sends the button' s ascii code out the serial port. ' Cr' sends a carriage return out the serial port. ' 10 ms' waits for about 10 milliseconds. ' Writing' makes it possible to write to the LCD again.

The next few words are key handlers. The first is ' clear-lcd' on line 95. It puts the LCD into instruction mode, sends it the ' 1' instruction, goes back to data mode, then waits 100 milliseconds for the display to clear.

' Back' on line 97 similarly goes to instruction mode, send a ' $10' command, then goes back to data mode. No wait is needed.

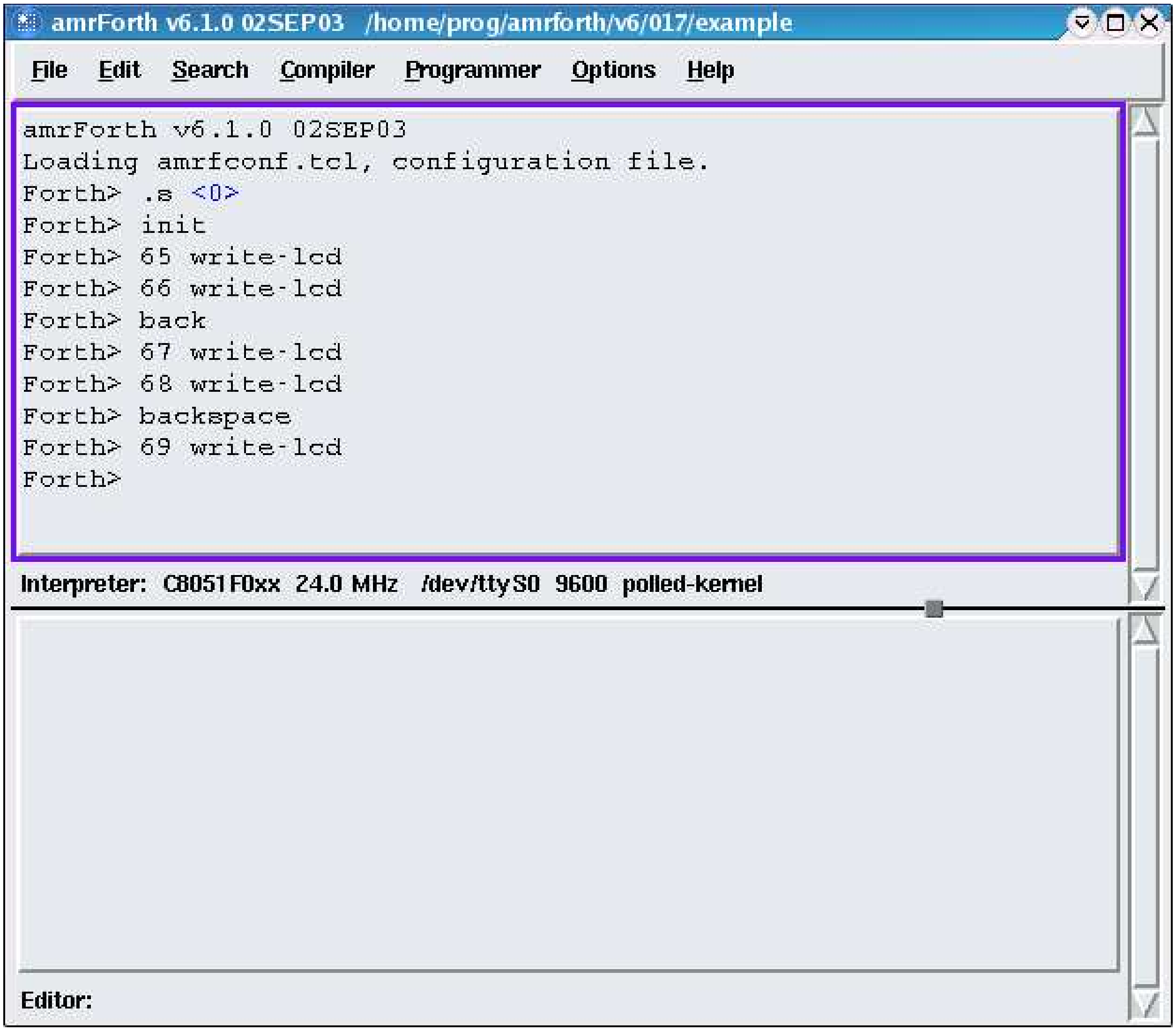
' Backspace' on line 99 uses ' back' , writes a space over the previous character, then uses ' back' again.

The ' handler' on line 101 decides how to handle a given character. Depending on the character it will write the character to the LCD, or clear the LCD, or perform a backspace, or read a button, or do nothing.

The word ' init' on line 108 eliminates weak pullups, then initializes the LCD.

The main routine, ' go' , on line 110, initializes the system with ' init' , gets into an endless loop of waiting for a key from the serial port, then does something with it in ' handler' .

Try testing some of the handler words interactively as follows:



```
amrForth v6.1.0 02SEP03
Loading amrfconf.tcl, configuration file.
Forth> .s <0>
Forth> init
Forth> 65 write-lcd
Forth> 66 write-lcd
Forth> back
Forth> 67 write-lcd
Forth> 68 write-lcd
Forth> backspace
Forth> 69 write-lcd
Forth>
```

Interpreter: C8051F0xx 24.0 MHz /dev/ttyS0 9600 polled-kernel

After this string of commands you should the word ' ACE' on your LCD display. Of course 65=A, 66=B, 67=C, 68=D, 69=E. If you finally type ' clear-lcd' the ' ACE' will disappear.

```
\ ---------- The Programs ---------- /

\ Program #1
\ See the Parallax website, www.parallax.com,
\ for the executable version of this program.
\ --- The original demo program.
```

```
' PROGRAM: Terminal.bas
' The Stamp serves as a user-interface terminal. It accepts text via
' RS-232 from a' host, and provides a way for the user to respond to
' queries via four pushbuttons.

Symbol S_in  = 6          ' Serial data input pin
Symbol S_out = 7          ' Serial data output pin
Symbol E     = 5          ' Enable pin, 1 = enabled
Symbol RS    = 4          ' Register select pin, 0 = instruction
Symbol  keys = b0         ' Variable holding # of key pressed.
Symbol char  = b3         ' Character sent to LCD.
Symbol Sw_0  = pin0       ' User input switches
Symbol Sw_1  = pin1       ' multiplexed w/LCD data lines.
Symbol Sw_2  = pin2
Symbol Sw_3  = pin3


' Set up the Stamp's I/O lines and initialize the LCD.
begin:       let pins = 0              ' Clear the output lines
             let dirs = %01111111      ' One input, 7 outputs.
             pause 200                 ' Wait 200 ms for LCD to reset.

' Initialize the LCD in accordance with Hitachi's instructions for 4-bit
' interface.
i_LCD:       let pins = %00000011      ' Set to 8-bit operation.
             pulsout E,1               ' Send data three times
             pause 10                  ' to initialize LCD.
             pulsout E,1
             pause 10
             pulsout E,1
             pause 10
             let pins = %00000010      ' Set to 4-bit operation.
             pulsout E,1               ' Send above data three times.
             pulsout E,1
             pulsout E,1
             let char = 14             ' Set up LCD in accordance with
             gosub wr_LCD              ' Hitachi instruction manual.
             let char = 6              ' Turn on cursor and enable
             gosub wr_LCD              ' left-to-right printing.
             let char = 1              ' Clear the display.
             gosub wr_LCD
             high RS                    ' Prepare to send characters.

' Main program loop: receive data, check for backspace,
' and display data on LCD.
             main:        serin S_in,N2400,char      ' Main terminal loop.
             goto bksp
out:         gosub wr_LCD
             goto main

' Write the ASCII character in b3 to LCD.
wr_LCD:      let pins = pins & %00010000
             let b2 = char/16          ' Put high nibble of b3 into b2.
             let pins = pins | b2      ' OR the contents of b2 into pins.
             pulsout E,1               ' Blip enable pin.
             let b2 = char & %00001111 ' Put low nibble of b3 into b2.
             let pins = pins & %00010000 ' Clear 4-bit data bus.
             let pins = pins | b2      ' OR the contents of b2 into pins.
             pulsout E,1               ' Blip enable.
             return

' Backspace, rub out character by printing a blank.
bksp:        if char > 13 then out     ' Not a bksp or cr? Output character.
             if char = 3 then clear    ' Ctl-C clears LCD screen.
             if char = 13 then cret    ' Carriage return.
             if char <> 8 then main    ' Reject other non-printables.
             gosub back
             let char = 32             ' Send a blank to display
             gosub wr_LCD
             gosub back                ' Back up to counter LCD's auto' increment.
             goto main                 ' Get ready for another transmission.
back:        low RS                    ' Change to instruction register.
             let char = 16             ' Move cursor left.
             gosub wr_LCD              ' Write instruction to LCD.
             high RS                   ' Put RS back in character mode.
             return

clear:       low RS                    ' Change to instruction register.
             let b3 = 1                ' Clear the display.
             gosub wr_LCD              ' Write instruction to LCD.
             high RS                   ' Put RS back in character mode.
             goto main

' If a carriage return is received, wait for switch input from the user.
' The host program (on the other computer) should cooperate by waiting for
' a reply before sending more data.

cret:        let dirs = %01110000      ' Change LCD data lines to input.
loop:        let keys = 0
             if Sw_0 = 1 then xmit     ' Add one for each skipped key.
             let keys = keys + 1
             if Sw_1 = 1 then xmit
             let keys = keys + 1
             if Sw_2 = 1 then xmit
             let keys = keys + 1
             if Sw_3 = 1 then xmit
             goto loop

xmit:        serout S_out,N2400,(#keys,10,13)
             let dirs = %01111111       ' Restore I/O pins to original state.
             goto main

```

```
\ Program #2
\ See ~/amrforth/v6/017/basic/example.bas
\ or   \amrforth\v6\017\basic\example.bas
\ for the current executable version of this program.
\ --- Our initial BASIC version.
```

```
BASIC
' example.bas  An example application for amrBASIC.
' LCD and Keypad for an RS232 terminal.
' R/W is tied high, we always write, never read the LCD.

Symbol En = 2              ' Enable pin, 1 = enabled.
Symbol RS = 3              ' Register Select pin, 0 = instruction.
Symbol keys = b0           ' Number of key pressed.
' Symbol buttons = b1      ' Last read of pins.
Symbol nibble = b2         ' Partial character.
Symbol char = b3           ' Character sent to LCD.

Symbol Sw_0 = pin4         ' User input switches.
Symbol Sw_1 = pin5         ' Multiplexed with LCD data lines.
Symbol Sw_2 = pin6
Symbol Sw_3 = pin7

' Set up the i/o's and initialize the LCD.
begin:       let pins = %00000011                    ' Clear all but rs232 pins.
             let dirs = dirs | %11111100             ' 6 output pins.
             pause 200                               ' Wait 200 ms for LCD to reset.

' Initialize the LCD for a 4 bit interface.
             let pins = %00110000       ' Set LCD to command mode,
             pulseout En 10                          ' and send the 0 command
             pause 100                  ' three times with a suitable
             pulseout En 10                          ' delay in between to initialize
             pause 100                  ' the LCD.
             pulseout En 10                          ' Pulse the Enable line for
             pause 100                  ' 100 us, then wait 100 ms.
             let pins = %00100000       ' 4 bits command,
             pulseout En 10                          ' written to LCD.
             pause 100                  ' Give LCD time to digest.
             let char = $28                          ' 40 chars, 2 lines, 5x7 font
             gosub wr_LCD               ' command written to LCD.
             pause 100                  ' Give LCD time to digest.
             let char = $0e                          ' Underline cursor command,
             gosub wr_LCD               ' is written to LCD.
             pause 10                   ' Give LCD time to digest.
             let char = $01                          ' Clear display command,
             gosub wr_LCD               ' written to LCD.
             pause 100                  ' Give LCD time to clear.
             let char = $02                          ' Set output mode command,
             gosub wr_LCD               ' written to LCD.
             pause 10                   ' Give it time.
             high RS                                 ' Set LCD to character mode.

' Main program loop:  receive data, check for backspace, and display
' data on LCD.
main:        serin char   ' Main loop waits for a serial character,
             goto bksp    ' checks for control characters first,
out:         gosub wr_LCD ' writes data char if not a control char,
             goto main    ' loops indefinitely.

' Write the ASCII character in b3 (char) to LCD.
wr_LCD:      let pins = pins & %00001011             ' Clear 4 bit data bus.
             let nibble = char & %11110000           ' High nibble of char.
             let pins = pins | nibble   ' Or contents of nibble onto pins.
             PULSEOUT En 10                          ' Blip enable pin.
             let nibble = char * 16                  ' Low nibble of char.
             let pins = pins & %00001011             ' Clear 4 bit data bus.
             let pins = pins | nibble   ' Or contents of nibble onto pins.
             PULSEOUT En 10                          ' Blip enable pin.
             return                                  ' wr_LCD is a subroutine.

' Backspace, rub out character by printing a blank.
```

```
bksp:        if char > 31 then out     ' Output if not a control character.
             if char = 3 then clear    ' Ctrl-C clears LCD screen.
             if char = 13 then cret    ' Carriage return, wait for button.
             if char <> 8 then main    ' Reject other control characters.
             gosub back                ' Move cursor back once.
             let char = 32             ' Send a blank to the display.
             gosub wr_LCD
             gosub back                ' Move cursor back once more.
             goto main                 ' Not a subroutine, go to main.

back:        low RS                    ' Change to instruction register.
             let char = $10            ' Move cursor left.
             gosub wr_LCD              ' Write instruction to LCD.
             high RS                   ' Back to character mode.
             return                    ' back is a subroutine.

clear:       low RS                    ' Change to instruction register.
             let char = 1              ' Clear the display.
             gosub wr_LCD              ' Write instruction to LCD.
             high RS                   ' Back to character mode.
             pause 100                 ' Wait for display.
             goto main                 ' clear is not a subroutine.

cret:
             let dirs = %00001100      ' Change LCD data lines to inputs.
             let pins = pins | %11111000 ' Set pins for reading.


loop:        let keys = $30            ' $30 = ascii 0.
             if Sw_0 = 1 then xmit     ' Send '0' if switch 0 pressed.
             let keys = keys + 1       ' $31 = ascii 1.
             if Sw_1 = 1 then xmit     ' Send '1' if switch 1 pressed.
             let keys = keys + 1       ' $32 = ascii 2.
             if Sw_2 = 1 then xmit     ' Send '2' if switch 2 pressed.
             let keys = keys + 1       ' $33 = ascii 3.
             if Sw_3 = 1 then xmit     ' Send '3' if switch 3 pressed.
             goto loop                 ' Read switches until one pressed.

xmit:        serout keys 13 10         ' Send key, carriage return and
             pause 10                  ' linefeed, then wait 10 ms.
             let pins = pins & %00001011 ' Clear data pins, make
             let dirs = %11111100      ' outputs, set LCD to data mode.
             goto main                 ' xmit is not a subroutine.

RUN begin    ' Start the BASIC program at the label 'begin'.

END                       ' End of BASIC compiling.

```

```
\ Program #3
\ See ~/amrforth/v6/017/basic/modular.bas
\ or   \amrforth\v6\017\basic\modular.bas
\ for the current executable version of this program.
\ Modular BASIC version.
```

```
BASIC
' example.bas  An example application for amrBASIC.
' LCD and Keypad for an RS232 terminal.
' R/W is tied high, we always write, never read the LCD.
' This version is done in a modular style for easier debugging.

Symbol En = 2             ' Enable pin, 1 = enabled.
Symbol RS = 3             ' Register Select pin, 0 = instruction.
Symbol keys = b0          ' Number of key pressed.
' Symbol buttons = b1     ' Last read of pins.
Symbol nibble = b2        ' Partial character.
Symbol char = b3          ' Character sent to LCD.

Symbol Sw_0 = pin4        ' User input switches.
Symbol Sw_1 = pin5        ' Multiplexed with LCD data lines.
Symbol Sw_2 = pin6
Symbol Sw_3 = pin7

' Set up the i/o's and initialize the LCD.
init:        let pins = %00000011      ' Clear all but rs232 pins.
             let dirs = dirs | %11111100 ' 6 output pins.
             pause 200                 ' Wait 200 ms for LCD to reset.
' Initialize the LCD for a 4 bit interface.
             let pins = %00110000      ' Set LCD to command mode and
             pulseout En 10            ' send the init command
             pause 100                 ' three times with a suitable
```

```
              pulseout En 10            ' delay in between (100 ms)
              pause 100                 ' to initialize the LCD.
              pulseout En 10            ' Pulse the Enable line for
              pause 100                 ' 100us, then wait 100 ms.
              let pins = %00100000      ' 4 bits command,
              pulseout En 10            ' written to LCD.
              pause 100                 ' Wait for LCD to digest data.
              let char = $28            ' 40 chars, 2 lines, 5x7 font
              gosub wr_LCD              ' command written to LCD.
              pause 100                 ' Give LCD time to digest.
              let char = $0e            ' Underline cursor command,
              gosub wr_LCD              ' written to LCD.
              pause 10                  ' Give LCD time to digest.
              let char = $01            ' Clear display command,
              gosub wr_LCD              ' written to LCD.
              pause 100                 ' Let LCD digest the command.
              let char = $02            ' Set output mode command
              gosub wr_LCD              ' written to LCD.
              pause 10                  ' Give LCD time to digest it.
              high RS                   ' Set LCD back to character mode.
              return                    ' init is a subroutine.

' Write the ASCII character in char (b3) to LCD.
wr_LCD:       let pins = pins & %00001011            ' Clear 4 bit data bus.
              let nibble = char & %11110000          ' High nibble of char.
              let pins = pins | nibble  ' Or contents of nibble onto pins.
              PULSEOUT En 10            ' Blip enable pin.
              let nibble = char * 16    ' Low nibble of char.
              let pins = pins & %00001011            ' Clear 4 bit data bus.
              let pins = pins | nibble  ' Or contents of nibble onto pins.
              PULSEOUT En 10            ' Blip enable pin.
              return                    ' wr_LCD is a subroutine.

back:         low RS                    ' Change to instruction mode.
              let char = $10            ' Move cursor left instruction.
              gosub wr_LCD              ' Write instruction to LCD.
              high RS                   ' Back to character mode.
              return                    ' back is a subroutine.

clear:        low RS                    ' Change to instruction mode.
              let char = 1              ' Load clear display instruction.
              gosub wr_LCD              ' Write instruction to LCD.
              high RS                   ' Back to character mode.
              pause 100                 ' Wait for display.
              return                    ' clear is a subroutine.

cret:
              let dirs = %00001100      ' Change LCD data lines to inputs.
              let pins = pins | %11111000            ' Set pins for reading.
                                        ' cret is a subroutine which
                                        ' eventually returns via xmit.
loop:         let keys = $30            ' $30 = ascii 0.
              if Sw_0 = 1 then xmit     ' Send 0 if switch 0 pressed.
              let keys = keys + 1       ' $31 = ascii 1.
              if Sw_1 = 1 then xmit     ' Send 1 if switch 1 pressed.
              let keys = keys + 1       ' $32 = ascii 2.
              if Sw_2 = 1 then xmit     ' Send 2 if switch 2 pressed.
              let keys = keys + 1       ' $33 = ascii 3.
              if Sw_3 = 1 then xmit     ' Send 3 if switch 3 pressed.
              goto loop                 ' Read switches until one pressed.
                                        ' Loop eventually jumps to xmit.
xmit:         serout keys 13 10         ' Send key, carriage return and
              pause 10                  ' linefeed, then wait 10 ms.
              let pins = pins & %00001011            ' Clear data pins, make
              let dirs = %11111100      ' outputs, set LCD to data mode.
              return                    ' xmit is a subroutine.

handler:
              if char > 31 then wr_LCD  ' Output if not a control character.
              if char = 3 then clear    ' Ctrl-C clears LCD screen.
              if char = 13 then cret    ' Carriage return, wait for button.
              if char <> 8 then leave   ' Reject other control characters.
              gosub back                ' Move cursor back once.
              let char = 32             ' Load a blank into char then
              gosub wr_LCD              ' write it to the LCD.
              gosub back                ' Move cursor back once more.
leave:        return                    ' Get ready for another transmission.
                                        ' handler is a subroutine.
begin:        gosub init                ' BASIC program entry point.
' Main program loop:  receive data, check for backspace,
' and display data on LCD.
main:         serin char   ' Wait for a serial character.
              gosub handler             ' Handle that character.
              goto main    ' Go back for more characters indefinitely.
```

```

RUN begin    ' Start the BASIC program at the label 'begin'.

END          ' End of BASIC compiling.

```

```
\ Program #4
\ See ~/amrforth/v6/017/example\keylcd.fs
\ or   \amrforth\v6\017\example\keylcd.fs
\ for the current executable version of this program.
\ --- The Forth version.
```

```
\ keylcd.fs  Driving a serial port, an LCD, and 4 buttons.
include basic.fs  \ For the 'pulseout' word.
in-meta

\ ----- Assembler macros ----- /

a: .E    2 .P0 ;a         \ Assembler macros name the E (enable)
a: .RS   3 .P0 ;a         \ and RS (Register Select) pins.
a: pins  P0 ;a                          \ Call port 0 'pins'.
a: dirs  PRT0CF ;a        \ Call PRT0CF 'dirs', as in BASIC.

\ ----- Initializing the LCD ----- /

code instruction   (  - )  .RS clr   next c;         \ Clear RS pin.
code data  (  - )  .RS setb  next c;                 \ Set RS pin.
code us  ( c - )
SP inc  Apop  begin
6 # R7 mov  begin  R7 -zero until  nop
ACC -zero until  next c;
: ms  ( n - ) for  250 us 250 us 250 us 250 us  next ;

: pulseout-E  (  - )
   [ asm  .E cpl ] 100 us [ asm  .E cpl ] ;

\ Based on working BASIC code.
: write-lcd  ( c - )
   [ in-assembler         \ Inline assembler language.
   %00001011 # pins anl ] \ Instruction mode, clear data bus.
   dup                    \ Copy the character (in forth).
   [ in-assembler         \ Inline assembler again.
   SP inc  Apop                         \ Pop data stack into A register.
   %11110000 # A anl      \ Clear control bits.
   A pins orl ]                         \ Write data nibble to data bus.
   pulseout-E             \ Call pulseout-E (in forth).
   16 *                   \ Shift character 4 bits left.
   [ in-assembler         \ Inline assembler again.
   SP inc  Apop                         \ Pop data stack into A register.
   %00001011 # pins anl   \ Clear data pins.
   A pins orl ]                         \ Write nibble to data pins.
   pulseout-E                           \ Clock data bus (in forth).
   ;                      \ Return from subroutine.

code no-pullups  (  - )
   $80 # XBR2 orl  \ Disable weak pullups.
   next c;

\ Also based on working BASIC code.
: init-lcd  (  - )        \ Run right after power on reset.
   [ in-assembler         \ Inline assembler.
   %00000011 # pins mov   \ Set command mode, 0 command.
   %11111100 # dirs orl ] \ Set pins as outputs.
   200 ms    \ In high level forth, wait 200 ms for LCD reset.
   [ in-assembler         \ Inline assembler.
   %00110000 # pins mov ] \ Init instruction
   pulseout-E 100 ms      \ clocked into LCD 3 times
   pulseout-E 100 ms      \ with 100 ms delay in between.
   pulseout-E 100 ms      \ High level forth.
   [ in-assembler         \ Inline assembler.
   %00100000 # pins mov ] \ 4 bit mode instruction
   pulseout-E 100 ms      \ clocked in LCD (in forth).
   $28 write-lcd 100 ms   \ 40 chars, 2 lines, 5x7 font.
   $0e write-lcd 10 ms    \ Underline cursor.
   $01 write-lcd 100 ms   \ Clear display.
   $02 write-lcd 10 ms    \ Put LCD into output mode.
   data                   \ Return to character mode.
   ;                      \ Return from subroutine.

\ ----- Reading the Keypad ----- /

```

```
code pins@  (  - n)       \ In assembly language.
   pins A mov             \ Read pins into A register.
   Apush  0push           \ Push pins onto data stack.
   next c;   \ Return from subroutine.

: button  (  - n)
   begin     pins@        \ Read button data onto data stack.
      dup %00010000 and if  drop $30 exit  then
      dup %00100000 and if  drop $31 exit  then
      dup %01000000 and if  drop $32 exit  then
      dup %10000000 and if  drop $33 exit  then
      drop   \ No match, try again.
   again ;   \ Indefinite loop, returns via one of the exits.

code reading  (  - )      \ In assembly language.
   %00001100 # dirs mov   \ Set all data pins to input.
   %11111000 # pins orl   \ Set data pins high for reading.
   next c;

code writing  (  - )
   %00001011 # pins mov   \ Set data pins low for writing.
   %11111100 # dirs orl   \ Make them all outputs.
   next c;

: read-button  (  - ) reading button emit cr 10 ms writing ;

: clear-lcd  (  - ) instruction 1 write-lcd data 100 ms ;

: back  (  - ) instruction $10 write-lcd data ;

: backspace  (  - ) back 32 write-lcd back ;

: handler  ( c - )
   dup 31 > if  write-lcd        exit  then
   dup  3 = if  drop clear-lcd   exit  then
   dup 13 = if  drop read-button exit  then
   dup  8 = if  drop backspace   exit  then
   drop ;

: init  (  - ) no-pullups init-lcd ;

: go  (  - ) init begin    key handler again ;
```