J1a SwapForth Reference

James Bowman
jamesb@excamera.com
Excamera Labs
Pescadero
California USA

ANS Forth Compliance Label

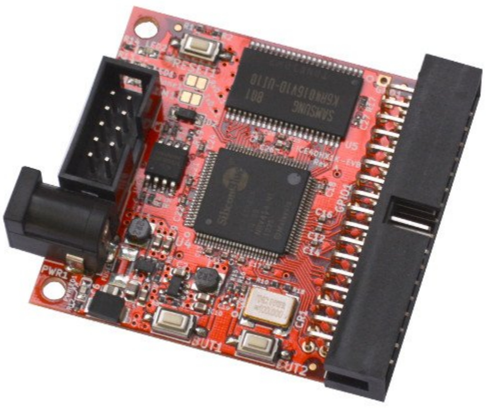
J1a SwapForth is an ANS Forth System

Providing names from the **Core Extensions** word set
Providing names from the **Double-Number** word set
Providing names from the **Facility** word set
Providing names from the **Facility Extensions** word set
Providing names from the **String** word set
Providing names from the **Programming-Tools** word set
Providing names from the **Programming-Tools Extensions** word set

Contents

Chapter 1

Getting started



J1a SwapForth is a 16-bit version of SwapForth, intended as an interactive Forth system using very little logic and RAM. The system currently fits on a Lattice iCE40HX-1k FPGA. The J1a and peripherals use 1200 logic elements. SwapForth uses 4.7 Kbytes of RAM, leaving about 3.3 Kbytes for the application. . This version is adapted to run on an Olimex ICE40 board and is compiled using the all open source icestorm toolchain. Access to the on-board LEDs (2x), BUTTONs (2x), SRAM (512kbyte), SPI FLASH (2MByte) and GPIO (16-bit) is implemented.

After installing the icestorm, git, and the Olimex Arduino or RPi programmer tools you can install and run it on an Olimex ICE40 board like this

```
git clone https://github.com/pbrier/swapforth.git
cd swapforth/j1a
iceprogduino olimex-ice40/j1a.bin
python shell.py -h /dev/ttyUSB0
```

(where `/dev/ttyUSB0` is the appropriate port your serial adaptor was assigned). You should see something like

```
Contacting... established
Loaded 208 words
>
```

And you can now try the usual Forth things, e.g.

```
1 2 + .
3   ok
```

There is a fairly complete core ANS-compatible Forth system running on the board, including a compiler.

1.1 Some demos

You can control the two on-board LEDs

```
-1 leds
  ok

0 leds
  ok
```

and to make them blink

```
: blink
  32 0 do
    i leds
    100 ms
  loop
;
blink
```

There is an Easter date calculator

```
    new
    #include ../demos/easter.fs
```

Now you can do

```
    >2015 .easter
    2015 April 5   ok
```

Or even

```
>: 20easters
+  2035 2015 do
+    cr i .easter
+  loop
+;
 ok
>20easters

2015 April 5
2016 March 27
2017 April 16
2018 April 1
2019 April 21
2020 April 12
2021 April 4
2022 April 17
2023 April 9
2024 March 31
2025 April 20
2026 April 5
2027 March 28
2028 April 16
2029 April 1
2030 April 21
2031 April 13
2032 March 28
2033 April 17
2034 April 9   ok
```

1.2 Building from scratch

After installing the icestorm tools, run

```
~/Documents/ice/swapforth/j1a $ make clean
~/Documents/ice/swapforth/j1a $ make j1a
```

This will produce `j1a.bin` - but it only contains the very bare-bones system; the rest of SwapForth still needs to be compiled. To do this, load `j1a.bin` and start the shell (assuming your board's serial appears at `ttyUSB0`):

```
$ iceprogduino olimex-ice40/j1a.bin
$ python shell.py -h /dev/ttyUSB0 -p ../common/
Contacting... established
Loaded 143 words
-
```

Then compile the rest of SwapForth and write the finished executable with these commands:

```
#include swapforth.fs
#flash build/nuc.hex
#bye
```

Now run `make -C olimex-ice40` again - this compiles an FPGA image with the complete code base built-in, which has the full set of words defined:

```
$ iceprogduino olimex-ice40/j1a.bin
$ python shell.py -h /dev/ttyUSB0 -p ../common/
Contacting... established
Loaded 207 words
```

Chapter 2

Available Words

2.1 ANS Core Words

J1a SwapForth implements most of the core ANS 94 Forth standard. Implemented words are:

`! # #> #s ' (* */ */mod + +! +loop , - . ." / /mod 0< 0= 1+ 1- 2! 2* 2/ 2@ 2drop 2dup 2over 2swap : ; < <# = > >body >in >number >r ?dup @ abort abort" abs accept align aligned allot and base begin bl c! c, c@ cell+ cells char char+ chars constant count cr create decimal depth do does> drop dup else emit environment? evaluate execute exit fill find fm/mod here hold i if immediate invert j key leave literal loop lshift m* max min mod move negate or over postpone quit r> r@ recurse repeat rot rshift s" s>d sign sm/rem source space spaces state swap then type u. u< um* um/mod unloop until variable while word xor [['] [char]]`

The core word **`environment?`** is not implemented. J1a SwapForth also implements the following standard words:

`.( .r .s /string 0<> 0> :noname <> ?do again ahead case cmove cmove> compile, d+ d. d.r d0= d2* dabs dnegate dump endcase endof erase false hex key? m+ marker ms nip of pad parse refill restore-input save-input sliteral throw true tuck u.r u> unused within words [compile] \`

Double numbers are supported using the standard `.` suffix. The Forth 200x number prefixes are supported: `$` for hex, `#` for decimal, `%` for binary, and `'c'` for character literals. `parse-name` is also implemented.

2.2 Additional Words

The following words are not standard. Some are traditional Forth words, others are specific to the J1a SwapForth implementation.

`.x`

`( n -- )`

display n as a 4-digit hex number

`-rot`

`( x1 x2 x3 -- x3 x1 x2 )`

rotate the top three stack entries

`bounds`

`( start cnt -- start+cnt start )`

prepare to loop on a range

`forth`

`( -- a )`

variable: most recent dictionary entry

`io!`

`( x a -- )`

store `x` to IO port `a`

`io@`

`( a -- x )`

fetch from IO port `a`

`leds`

`( x -- )`

write `x` to the onboard LEDs (bit0=D1, bit1=D2

`new`

`( -- )`

restore code and data pointers to the power-up state

`s,`

`( a u -- )`

add the `u`-character string `a` to the data space

`tth`

`( -- a )`

variable: tethered mode

---

buttons

`( -- a )`

Read current button state (bit0=BUTTON1, bit1=BUTTON2)

---

gpio!

`( a -- )`

Write GPIO value D0..D15 (set gpio_dir bits first if you want signal output)

---

gpio@

`( -- a )`

Read GPIO value D0..D15

---

gpio_dir

`( a -- )`

Set GPIO direction. 16bit direction (1=OUTPUT, 0=INPUT)

---

sramw

`( d a -- )`

Write data ’d’ to SRAM address ’a’

---

sramr

`( a -- d )`

Read data from SRAM address ’a’

Chapter 3

Using SwapForth

3.1 Raw UART access

At boot, SwapForth listens for a command on the UART. Connection parameters are 115200 8N1, and any terminal program should be able to connect. Note that RTS can be used as a reset signal (if connected), so you should make sure that it is set OFF by the terminal program. You can switch it during your session using `CTRL-T CTRL-R`:

```
$ miniterm.py --rts=0 --eol LF /dev/ttyUSB0 115200
--- Miniterm on /dev/ttyUSB5: 115200,8,N,1 ---
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
--- forcing RTS inactive
  ok
  ok
  ok
```

3.2 The SwapForth shell

The SwapForth shell is a Python program that runs on the host PC. It has a number of advantages over raw UART access:

- command-line editing
- command history

- word completion on TAB
- local file `include`
- `^C` for interrupt

3.2.1 Invocation

The shell is a Python program. To run it, go to the appropriate directory and type:

```
python shell.py -h /dev/ttyUSB0
```

3.2.2 Command reference

`#bye` - quit SwapForth shell

`#flash` - copy the target state to a local file

`#include` - send local source file

`#noverbose` - turn off include echo

`#time` - measure execution time

3.3 Tethered Mode

J1b SwapForth supports *tethered mode*, which makes the UART protocol easier to use for host programs. The SwapForth shell uses tethered mode. To enter tethered mode, write one to the variable **`tth`** :

```
1 tth !
```

In tethered mode, **`accept`** transmits byte value 30 (hex `1e`, ASCII code RS). This allows the listening program to know that the target machine is ready to accept a line of input. In addition, **`accept`** does not echo characters as they are typed.

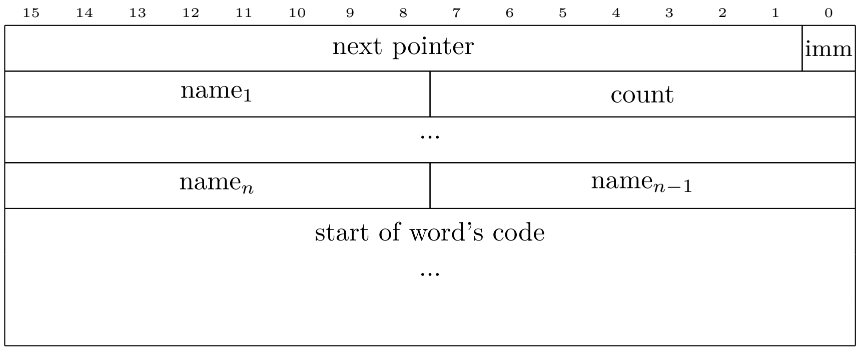
Chapter 4

Memory

4.1 Memory map

The J1a SwapForth implementation uses 8Kbytes of RAM for code and data. The standard Forth words access this RAM. Cells are 16-bits, and must be aligned to a 16-bit boundary.

4.2 Dictionary Layout



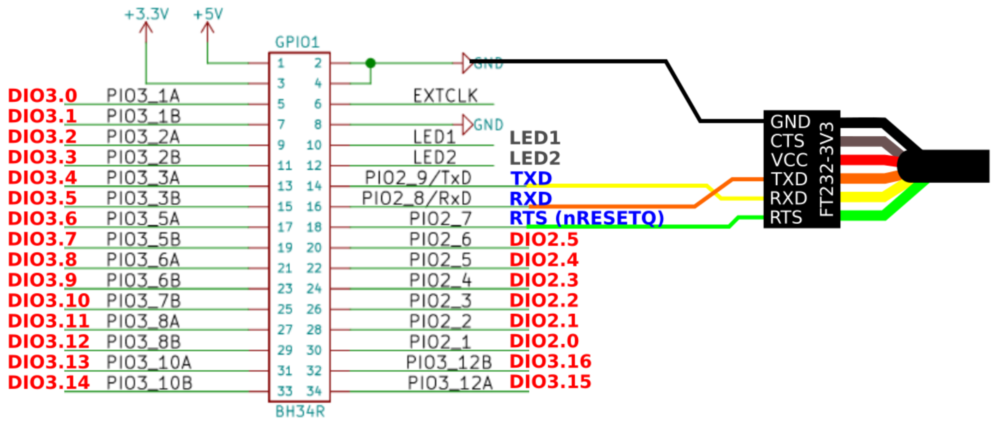
The SwapForth dictionary is a linked list; the variable `forth` holds the start of this list. Each dictionary entry has the following fields:

- **next pointer** - address of the next dictionary entry, or zero for the last dictionary entry
- **imm** - immediate bit, set if the word is immediate
- **count** - length of the name, in characters, 1-31
- **name**$_1$ **- name**$_n$ - characters in name. If the length of the name is even, then a padding byte is appended

Chapter 5

Olimex ICE40 Hardware interface

This shows the connections to the serials interface via the GPIO1 connector and the mapping of the GPIO pins. Note: all signals are 3V3!



This J1a implementation for The Olimex ICE40 boards includes support for the following peripherals:

- 2x LEDs 2x Button
- GPIO connector
- UART
- SRAM (4 x 64k x 16bit accessable, 512kbyte total)
- SPI flash (2 MByte)

Access to peripherals is via the `io@` and `io!` words. Peripherals are port-mapped into a 16-bit IO address space. Most ports are either read-only or write-only. For read-only ports, writing to the port has no effect. For write-only ports, reading from the port gives zero.

As an example of direct port access, this reads the buttons and sets the LEDS while also reporting the value to the serial console

```
\ output buttons to leds and terminal
: .bleds
begin
  $2000 io@
  2 rshift 3 xor
  dup .x
  $0004 io!
again
;
```

```
\ display button io state as hex
: .buttons
begin
  $2000 io@ .x
again
;
```

```
\ Pulse each GPIO pin (D0=Fmax ... D15=Fmax/2^15)
: testio
$ffff 2 io!
begin
  65535 0 do
    i 1 io!
  loop
again
;
```

5.1 Port Map

5.1.1 $0001: GPIO data

The read-write port at address $0001 is for direct access to the GPIO1 connector. The port pins are assigned as follows (bottom row is closest to PCB):

Connection	Top row pins	Bottom row pins	Connection
5V	1	2	GND
3V3	3	4	GND
bit 0	5	6	100MHZ Clock
bit 1	7	8	GND
bit 2	9	10	LED1
bit 3	11	12	LED2
bit 4	13	14	TXD (out)
bit 5	15	16	RXD (in)
bit 6	17	18	RTS (in)
bit 7	19	20	x
bit 8	21	22	x
bit 9	23	24	x
bit 10	25	26	x
bit 11	27	28	x
bit 12	29	30	x
bit 13	31	32	x
bit 14	33	34	bit 15

Correspondingly the port bits are assigned to pins as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
34	33	31	29	27	25	23	21	19	17	15	13	11	9	7	5

Note that pin direction is controlled by the corresponding bit the port at address $0002.

5.1.2 $0002: GPIO direction

Each of the 16 bits controls the direction of the corresponding pin of the GPIO connector. 0 sets the pin to input, 1 means sets the pin to output. The bit-to-pin mapping is the same as for port $0001.

5.1.3 $0004: LEDs

The five on-board LEDS are controlled by write-only port at address $0004. Setting a bit to 1 lights the corresponding LED.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
														LED2	LED1

Built-in word **`leds`** writes to this port.

5.1.4 $0008: PIO output

Write-only port $0008 controls the flash (SPI) outputs.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
													flash SCK	flash MOSI	flash CS

5.1.5 $0010, $0020, $0040: SRAM

The 16bit x 256k SRAM chip is connected to IO ports. These ports can be used to Read and Write up to 512kByte of data (in 4 banks of 128 kbyte) The write port at address $0010 is setting the 16 bit SRAM R/W address The read-write port at address $0020 is for writing and reading the 16 bit SRAM data The write port at address $0040 is controlling the SRAM data direction and R/W bits, and to set the upper 2 address bits. The words **`sramw`** , **`sramr`** can be used to access these ports.

$0010 SRAM Address register:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Address A15:A0															

$0020 SRAM Data register:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data D15:D0															

$0020 SRAM Control register:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
											A17	A16	nOE	nWE	DIR

5.1.6 $0800: SB_WARMBOOT control

Write-only port $0800 is an interface to the SB_WARMBOOT module. When activated, the FPGA loads a new configuration from external flash. There can be up to four external configurations; configuration 0 is the base SwapForth, and configurations 1-3 are available for other uses. So to reload the base system:

```
4 $800 io!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
													BOOT	S1	S0

5.1.7 $1000: UART data

The read-write port at address $1000 is for UART transmission or reception. Writing to the port starts transmission of a byte, reading the port returns the incoming byte.

Standard words **key** , **key?** and **emit** can be used to access this port.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
								byte							

5.1.8 $2000: Buttons, SPI Flash and UART inputs

Read-only port $2000 contains the input signals from the Buttons, SPI flash, and UART.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
											Button BUT2	Button BUT1	flash MISO	UART key?	UART busy

Index