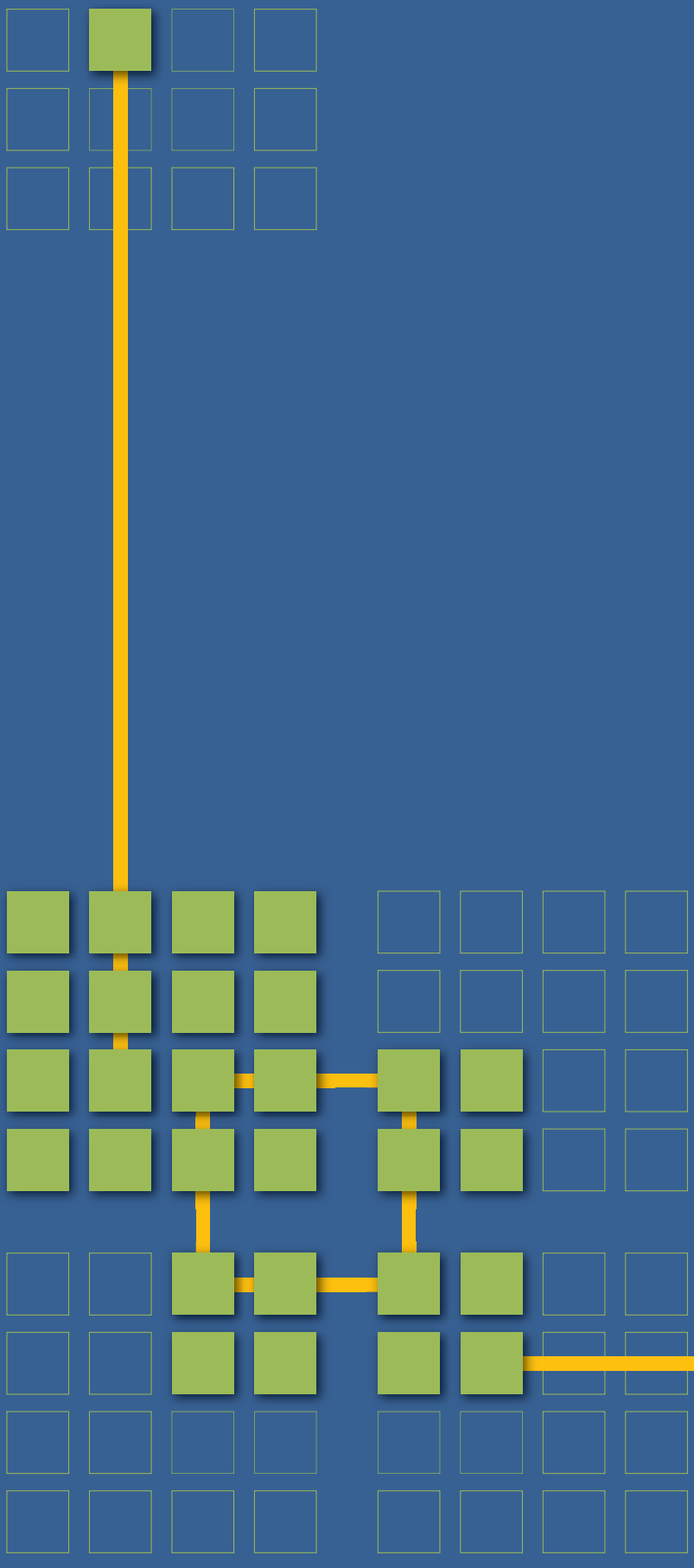
intellaSys
inventive to the core



VentureForth® Programmers Guide

A guide for programming the SEAforth® family of processors

Elizabeth D. Rather
and the Technical Staff of IntellaSys®

IntellaSys
20400 Stevens Creek Blvd, Fifth Floor
Cupertino CA 95014 USA
408.850.3270 voice
408.850.3280 fax
http://www.IntellaSys.net

Table of Contents

List of Figures

List of Tables

Code Listings

Chapter 1 Introduction to SEAforth Programming

1.1 Welcome

Send feedback

You are about to embark on a new adventure. The SEAforth family of multicore processors represents an entirely new technology. It is so new, in fact, that those of you who are designing SEAforth into new devices are, yourselves, pioneers, along with those of us at IntellaSys who are bringing this new technology into being.

The wonderful thing about pioneering new technology is that you have an opportunity to open new product opportunities and take a giant leap forward in your market. The downside is that, like all pioneers, you will not find open roads and smooth, well-marked paths.

The lack of "well-marked paths" is most visible in the arena of programming tools. While it is theoretically possible to run conventional, high-level languages on the SEAforth platform, such an approach forces a return to sequential threaded thinking, inefficient generated code, and the attachment of large external program memory arrays. This negates many of the unique advantages of this parallel computing platform.

From the perspective of common, sequential thinking, performance of the SEAforth chip might actually look quite unfavorable compared to other platforms, simply because most current hardware architectures offer advanced, complicated memory access features to help overcome shortcomings of inherently sequential tasking.

By treating SEAforth architecture in familiar and common ways, one would miss the ways in which it is actually a much more powerful concept. It is important to stop thinking of the SEAforth platform as a "nail" simply because hammers have been such familiar, beloved and even fancy tools for so long a time.

True leaps forward in processor performance require a fresh new thinking as well as the willingness to let go of familiar approaches and views. Only then will SEAforth architecture unleash its full power and

become the "rocket in a world of tractors" that it can be. Even the architects that built the SEAforth platform have only scratched the surface of what designs are possible.

Having said all this, we are actively working on techniques to marry sequential language elements with advanced parallel computing. The simple reason for this is that a sequential, external program can hold the deep state information and complex algorithms needed to implement such applications as TCP/IP stacks or other complex protocols, language compilers, event rich human interfaces or other complex application areas.

This manual describes the current state of programming tools for the SEAforth multicore processors. These tools are undergoing rapid development, even as this is being written. Therefore, you may expect relatively frequent updates as new software capabilities are developed, and in response to feedback from you, our fellow pioneers.

1.2 Definitions and Notational Conventions

Send feedback

Any new technology necessarily introduces some specialized terms, or uses common words with specialized meanings. Since the SEAforth chips are based in part on the Forth Programming Language, some of these terms are shared with Forth. However, some Forth concepts and words are used differently in connection with SEAforth technology. In addition, this book uses some special notation to represent the movement of data in the array of cores or nodes. This section describes these things.

SEAforth elements

A SEAforth multicore array consists of some number of individual cores, which share a common architecture and instruction set but differ somewhat in their capability for I/O and communication depending upon their position in the array. Individual cores are identified as "nodes," and numbered, with Node 00 being in the lower left corner of the array and the highest-numbered node in the upper right corner. There are also several "external nodes," which exist only in the host system for the purpose of constructing boot streams (as described in Section and Section 4.3.

Language elements

The VentureForth compiler parses source using spaces (or "whitespace characters") to separate tokens. A token may be a defined command or a number according to the current radix (decimal or hex); anything else will generate an error message. Command names may contain any combination of printable ASCII characters other than whitespace characters. Examples of commands include **`dup`**, **`+`**, and **`@p+`**. Although there is a fixed number of primitive commands in VentureForth, the programming process consists of defining new commands in the compiler, which compile target code and callable subroutines.

Stacks

A SEAforth core is fundamentally a stack machine. It features two stacks: a *data stack* (whose primary use is passing data between words), and a *return stack* (whose primary use is managing call nesting and returns). The return stack is also used for auxiliary purposes, however. Since the data stack is far more visible to the programmer than the return stack, in this document it is usually referred to as just "the stack."

Code listings

Code is displayed in **`this font`**. Lengthy sections of code are treated as specialized figures, and referenced in the Table of Listings in the front of this book.

Case sensitivity

VentureForth is case-insensitive. Common practice is to use lower-case. In documentation, upper case is sometimes used to refer to machine instructions as distinct from VentureForth commands.

Glossary entries

The commands described in this book are presented in "glossaries," including for each command whether it's a compiler command (designated by the letter "H" for *host* or a designation showing what target parts include the command (in its character set or ROM). The target designations include "A" for *all* target nodes, S24 for SEAforth S24 parts only, or S40 for SEAforth 40C18 parts only). The glossary entry also shows the command's stack effect (including the return stack effect, when it is affected), pronunciation as needed (since many commands use symbols), and a brief description of what it does.

Stack effects

One of the essential requirements for using each instruction is to know what, if anything, it expects on the data or return stacks, and what, if anything, it leaves. Glossary entries for each command will show stack effects in parentheses. Within the parentheses, there is a dash which separates the stack *before* execution of the command from the stack *after* it. The top two data stack items are in Registers T (top) and S (second); the top return stack item is in Register R. The data stack's commnets are the default. If the return stack is effected by a word **R:** is used to denote it.

For example, consider **push**, which removes an item from the data stack and pushes it onto the return stack. Its stack effect would be shown as:

```
( x – R:x )
```

Where several items are involved, they will be designated numerically. For example, the stack effect of **over** (which copies the second stack item to the top) would be:

```
( x1 x2 – x1 x2 x1 )
```

Data types

The Forth language, on which the SEAforth architecture is based, is a weakly-typed language. That is, although various kinds of data are recognized (addresses, integer numbers, characters, etc.), Forth compilers do not attempt to enforce type or overload operators. It is the programmer's responsibility to supply appropriate data to various commands. Within this book, where the type of an item on the stack is important, we will represent it by using the notation in Table 1.1.

Table 1.1 Data type notation

Data type	Notation
Single cell, no particular type	x
signed integer	n
positive integer	+n
unsigned integer	u
18-bit cell whose content is described in text	w
double-length integer (35 or 36 bits)	d_1 (msw) d_2 (lsw)
address, unspecified size	addr
18-bit address	a18
9-bit address	a9

Numbers

Most numbers in this document are decimal. Hex numbers are indicated by the prefix $. Thus:

- 10 equals decimal 10
- $10 equals decimal 16

Comments

Comments in code are of three forms:

- A left parenthesis **`(`** followed by at least one space and one or more characters and terminated by a right parenthesis **`)`**
- A backslash **`\`** followed by at least one space and one or more characters, terminated by the end of the line on which the backslash appears
- A multi-line block of comments is prefaced by **`0 [if]`** and terminated by **`[then]`** (conditional compilation directives, described in Section 3.4.5, are used to skip over the comment text).

All comments are processed by the compiler only. They compile no code for the target, and are used solely to help the programmer's understanding.

Send feedback

You may have noticed this message at the beginning of each section. These are links that will generate an email message to the maintainers of this book, who will be grateful for any comments you may have on that section.

1.3 Development Environment

Send feedback

The VentureForth programming toolset includes a PC connected in some fashion to hardware supporting a SEAforth processor, as shown in Figure 1.1. Evaluation boards are available from IntellaSys, which support an S24 or S40C18 chip and a USB connection to the PC. Equivalent custom hardware may also be used. Boot ROMs in SEAforth nodes support program downloading using flash memory via an SPI port or synchronous or asynchronous serial ports using mechanisms described in Section 4.3.

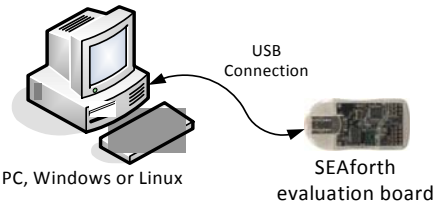


Figure 1.1 VentureForth programming environment

The PC acts as a *host* for the VentureForth compiler, simulator, and other tools in support of a SEAforth *target*. Versions of the VentureForth development system are available for Windows and Linux.

Chapter 2 Using VentureForth

This chapter describes the procedures needed for writing and testing programs on the SEAforth family of multi-core processors.

The VentureForth toolset includes a compiler, simulator and debugging package, together with utilities for downloading object programs into a SEAforth core and testing them.

For details of the hardware-level of the chip itself, please refer to the Data Sheet for the appropriate part.

2.1 Source Management

Send feedback

The VentureForth SEAforth installer creates three directories:

- **`VentureForth`** is the highest-level directory. Within this directory are three lower-level directories:
- **`docs`** contains documentation, including your license agreement, **`readme`**, your Data Sheet, and this book.
- **`projects`** contains application folders, including sample applications.
- **`vf`** contains the VentureForth compiler and simulator source code, as well as the sources for the pre-existing code in the ROM for the particular chips you're using. The sources for the ROM code are used to generate a symbol table so that you can reference these routines. The ROM library routines for the S24 parts are documented in Appendix A, while the ROM library routines for the S40C18 parts are documented in Appendix B.

The **`projects`** directory contains a folder for each sample application. We recommend that you create a new folder in the **`projects`** directory for each new application, usually by copying an existing project folder to use as a template.

Within a project folder, there will be files with three possible extensions:

- **`.f`** indicates a file that will be processed by the compiler, to add compiler features such as macros.
- **`.vf`** indicates code that will run on the chip.
- **`.vfp`** indicates a project file that launches the application.

We will focus on a sample project called blinktest.

2.1.1 Setting Up a Project File

Send feedback

Each project file shipped with VentureForth contain two sections: one that launches an application under Linux, and one that launches the same application under Windows. They do not conflict with one another; each OS will ignore the parts that don't apply. You may delete the section you aren't using, if you wish.

The easiest way to set up your own project file is simply to copy an existing project directory, and give it the name of your project (e.g. **`blinktest`**). Then you may customize it by following these steps:

1. Put the file or files for your application in the directory. This application has a main file called **`blinktest.vf`** and other files that incorporate different sub-applications run by different nodes.
2. Edit the **`project.vfp`** file, using your regular programming editor. The Windows portion of the project file for **`blinktest`** is shown in Listing 2.1. The **`INCLUDE=`** statement identifies the main load file for your application. This file should include all setup steps, define the programs for all nodes, construct the load stream, and perform any other desired steps such as setting up the simulator. If there are other files, they should also be loaded by this file. The **`TITLE=`** statement allows you to specify the text that will appear on the title bar of the window when VentureForth is running this project.

Listing 2.1 Windows portion of the **`blinktest`** project file

```
[STARTUP]
INCLUDE=blinktest.vf
TITLE=Chip Test
```

The Linux portion consists of a brief shell script at the beginning of this file. For **`blinktest`**, it looks like Listing 2.2. If you are using Linux, you must edit this to reference your project.

Listing 2.2 Linux portion of the **`blinktest`** project file

```
#!/usr/bin/env vfl
.( Blink Test )
INCLUDE blinktest.vf
\\
```

2.1.2 Loading Application Files

Send feedback

Your project will be compiled by a *master file*, which is identified in its project file. This file, in turn, may load additional files to complete your application.

The first responsibility of the master file is to generate entry points for whatever code is available in ROM for use by your application. This is done by a command of the form:

```
v.VF +include" <version>/romconfig.f"
```

...where *version* represents the particular SEAforth chip that you are using. For example, the an S40C18 evaluation board uses **`c7Fr01`**. This file will be found in your **`vf`** directory, whose path is correctly specified by **`v.VF`**.

The command:

```
include <filename>
```

will cause the file *filename* to be processed by the compiler.

The word **`include`** will assume that files are in the same directory as the master file. If you have a large number of files (e.g., programs for many nodes) you may wish to organize them in sub-directories, which may be reflected in your **`include`** statement.

It is good practice to load all your files out of the master file. This makes it easier to manage your application, because you can see in one place exactly which files are used and in what order.

Glossary

`include <filename>` (—) H *include*
Directs the compiler to process *filename* (which may include path specifications).

`v.VF` (— addr len) H *v-dot-vf*
Pushes onto the host's stack the string parameters (address and length) for the path to files in the **`vf`** directory that contain ROM code for each supported version of the SEAforth chip.

`+include" <filename>"` (addr len —) H *plus-include*
Directs the compiler to process *filename* (a string terminated by a **`")`** in the path given by the string parameters on the stack.

2.1.3 Launching a VentureForth Application

Send feedback

A VentureForth application will have a **`.vfp`** (project) file constructed according to instructions in Section 2.1.1. Launch it as follows:

- Under Windows, double-click on its icon.
- Under Linux, type **`./<filename>`** on the command line.

These steps will launch the VentureForth compiler and compile its application master file and (if requested) download it to the chip.

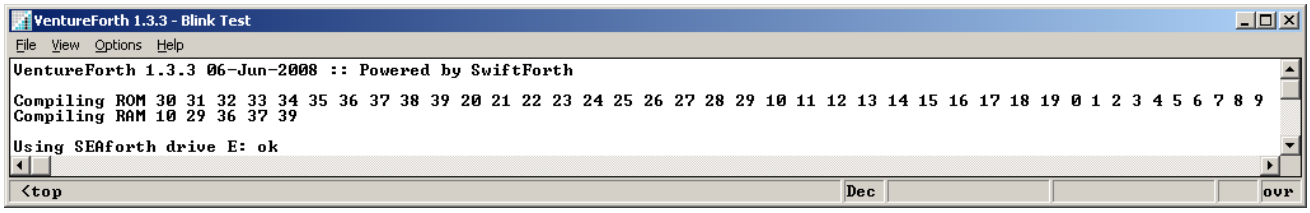


Figure 2.1 Sample application loaded in VentureForth under Windows

Figure 2.1 shows the result of launching the **`blinktest.vf`** sample application provided with your SEAforth evaluation board under the Windows OS. The version discussed here is for the SEAforth S40C18.

Here we see that the ROM images for all nodes on a 40-node part have been compiled; this step constructs the symbol tables for routines in each node's ROM so that the next step, compiling code to run in RAM, will be able to reference them. The next line generates code for some of the "outside" nodes (capable of doing I/O). The last line establishes a connection with a FORTHdrive, downloads the code, and starts it running.

The process of developing and testing code such as this example is covered in the next several sections.

2.2 Compiling Code for a SEAforth Node

Send feedback

In this example, the master file **`blinktest.vf`** (shown in Listing 2.3) specifies what code will run in each node, and in what order the code should be delivered to the nodes.

.

Listing 2.3 `blinktest` Sample application listing

```
\ Pass through many nodes and ports, toggling I/O
\ as much as possible; make node 29 toggle the LED

\ This example file wiggles many of the output pins
\ available on the S40C18 testboard, including three
\ digital pins and the three DAC pins.

v.VF +include" c7Fr01/romconfig.f"

33 {node  0 org  'r--- =p  node}
10 {node  0 org  here =p  include toggle17-1.vf  node}
20 {node  0 org  here =p  include toggle17-1.vf  node}
29 {node  0 org  here =p  include blinkLED.vf    node}
36 {node  0 org  here =p  include sawtooth.vf    node}
37 {node  0 org  here =p  include sawtooth.vf    node}
39 {node  0 org  here =p  include sawtooth.vf    node}

\ Build a stream that reaches each of the programmed nodes
19 29 39 38 37 36 35 34 33 32 31 30 20 10 14 nodePath

reset     \ reset prepares the system to run the code in the Simulator
          \ enter  simulate  or  sim   to start the simulator
cr
[x]  find-drive  /USBdrive  [x]>USBdrive  close-drive
```

At the beginning, the line:

```
v.VF +include" c7Fr01/romconfig.f"
```

compiles the ROM code for the target part (on an S40C18 evaluation board). This generates symbol table information for all the entry points in the ROMs. The path to this file is relative to the location of this file in the directory structure. The program for each node is specified separately in five steps.

1. Specify the node, for example, **`10 {node`**
2. Specify the address at which compilation starts, for example, **`0 org`**
3. Specify the entry point address for the code, for example, **`here =p`** (This can be done anytime before leaving the node).
4. Provide the node's source code. The source can be inline in the file or **`include`**d from another file. In this example we used three files in addition to the master file.
5. Invoke **`node}`** at the end of each node

Each node's program goes into a buffer, which will be sent to the chip when all nodes are complete, using a process described in more detail in

Section . The code is collected into a stream which will be loaded into the chip via a root node and thence through a predefined path; the command that does this is **nodepath**. Once the stream is built, the chip is then reset, the I/O connection to the part is found and initialized, and the stream is delivered. This process, known as a *stream loader*, delivers each packet of code to the node for which it was compiled.

The last line is specific to the specific target board; it transmits the stream designated by **[x]** (which was constructed by **nodepath**) to the device.

The actual program that each node will be asked to run in this example is defined in one of several separate files: **toggle17-1.vf**, **sawtooth.vf**, and **blinkLED.vf**. These are loaded as necessary for each node to be exercised..

Listing 2.4 Example of a program to blink an LED

```
$8000 1 >stk                \ Initial pulse width to T
: blinkLed ( - )
  $3FE00 # a!               \ Mask to control pulse width
  begin
     620 # for              \ Repeat 621 times
         30003 # !b         \ Drive pins at bits 17 and 1 high
         dup for
            . . . unext     \ Delay
         20002 # !b         \ Drive pin at bit 1 low
         dup not for
            . . . unext     \ Delay
      next
    $10000 # . + a@ and     \ Increments pulse width & masks with A
    | again                 \ Repeat infinitely
```

The program **blinkLed** in Listing 2.4 is designed to be the entire behavior of a node connected to an LED (Node 29 in Listing 2.3). It is constructed as an infinite loop (see Section 3.3.5); once its designated node starts running this, it will blink until it is reset. This is typical of node programs. However, if you want a behavior to be performed only once or a specified number of times, you must provide the node with something to do once it is finished. An empty **begin ... again** loop is a possible strategy; putting the node to sleep is an alternative that is more conservative of power. How you put a node to sleep depends on where it is in the array; see Table 4.2 for the respective commands

2.3 Downloading Your Program To The Array

Send feedback

The only way to get an application into a SEAforth chip is to send it to one of the nodes that is able to load an application through external I/O pins. Each of the SEAforth nodes that has a boot driver understands a protocol for booting an application. In this section, we will discuss the basic principles of booting, illustrated by our sample program. The process is discussed in more detail in Section 4.3.

2.3.1 Streams

Send feedback

A *stream* is the mechanism for delivering a collection of images (containing program and/or data) to nodes. The compiler builds a memory image for each node that has code or data compiled into it. Subsequently, the stream syntax is used to build a port-executable delivery wrapper around the images so that the images will be delivered to nodes by cooperative execution of all nodes along a path.

The path through which the stream will pass can be specified as a continuous snake-like path, or as a series of branching streams. Some applications can be delivered using a simple non-branching path. The syntax for that could look like this:

```
19 18 17 16 15 14 13 12 11 10 20 30 31 32 3 34 35 36 37 38 39 29 22 nodePath
```

In this example the stream will enter through the synchronous serial port on Node 19, and go through all 22 nodes in the order given. The last number passed to **`nodePath`** is the count of nodes used in this stream (22).

Many applications will need to have the nodes begin execution in a synchronized way, such that no node begins interacting with its neighbors before they have had their code delivered and are ready to interact. This can be done by calling the word **`19Stream`** which will build a stream suitable for sending to all nodes through node 19 (the node used on the SEAforth evaluation boards to load using synchronous serial communication). This stream will use branched delivery to fill all nodes that need code or data, and a wake-up synchronization strategy called *dominos*. This can be completely customized, using a process described further in Section 4.3. Similar pre-defined commands to use alternative initial nodes are available for the S40C18 parts. These are described in Section 2.3.2.

Glossary

nodePath (n_1 n_2 ... n_n n —) H *node-path*
Constructs the stream path through which the program for each node will be delivered, for *n* nodes beginning with n_1.

2.3.2 Stream Delivery

Send feedback

Once a stream has been built, three procedures are provided to deliver it:

- via a synchronous port (on Nodes 10 or 19 on the S40C18 parts),
- via an asynchronous port (Node 33 on the S40C18), or
- from an SPI-controlled flash device (Node 32 on the S40C18).

The evaluation board uses Node 19, which is connected to the USB interface to the PC. The command **`19stream`** provides a generic stream load to all nodes on both the S24 and S40C18 parts. The S40C18 parts also offer generic stream loaders for the other supported external ports listed above. These are described in the glossary below.

Alternative custom methods that may be used in applications are described in Section 4.3 and Section 5.2.

Glossary

`10stream` (—) S40 10-stream
Loads all nodes via the synchronous port on Node 10.

`19stream` (—) A 19-stream
Loads all nodes via the synchronous port on Node 19.

`32stream` (—) S40 32-stream
Loads all nodes via the SPI port on Node 32.

`33stream` (—) S40 33-stream
Loads all nodes via the synchronous port on Node 33.

2.4 Using the VentureForth Simulator

Send feedback

When code is compiled, the object for each node is placed into a virtual space representing the ROM and RAM content of the node. From there we can gather up the compiled object and send it to a real chip, using the *stream loader* which is described later. However, during simulation we

can skip the download process, and have the simulator just pretend that all the code is already in the SEAforth chip.

The command **reset** tells the simulator to run as though it is executing the code that is already in the SEAforth chip's RAM. Alternatively, we can attach a testbed which simulates an external connection, and the command **power** will tell the simulator to simulate the actual download process prior to running the application code.

The command **simulate** starts the simulator. The word **simulate** has a shorter synonym, **sim**. When **simulate** is invoked, a two-dimensional display appears, showing the status of all nodes. Pressing the space bar causes the simulation to proceed, at a rate of one step per press, by default. You may request more detailed representations of up to four nodes, by using the command **watch4**.

Listing 2.5 Example of simulator initialization

```
power  ( Initialize the simulator to include the boot process)
19 -1 20 05 watch4       \ Detailed info for nodes 19, 20. and 5
1224 setstep  -1 setmax  \ Quickly display every 1224th step
sim                      \ Type space to single-step, return to exit.
```

The simulator commands may be used from the command line or included in the file that loads your application. Most of the demo applications provided with the SEAforth evaluation system automatically set up simulator displays. An example is shown in Listing 2.5.

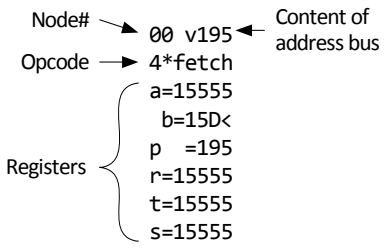


Figure 2.2 Information presented for each node in the simulator

The display for each node in the simulator is organized in a grid mimicking the ordering of nodes on the actual chip. You'll notice that some nodes are red and some black; the red nodes are the ones that are active. These correspond to the nodes for which programs were compiled in Figure 2.1, plus Node 19, which is handling the host communications. When you first launch the simulation, you'll see that all of them are attempting to do a fetch, and the registers all have the same initial values. The value $15D in the B register is the IOCS (I/O Control and Status) location; B is set to this by **reset**, and many chip functions make use of it.

```
node 05
adr=.  0 pc=  1 iw=366BC
slot=2 opc=1D inst=push
  s     t     r      a   b
15555 15555 15554   195 15D
15555       157FE
15555       3FFFE
15555       15554
    0       15554
15400      ^15554
15555       15554
15555       15554
15555^4     15554 1A4D854
```

Figure 2.3 Detailed node information produced by the **watch<n>** words

The little carets (such as < in the B register line) indicate that a node is reading or writing to or from a neighbor. They are intended to look like arrows, so you can see that a node wants to, for example, read from its left neighbor. The indentation of the B register display makes room for left/right arrows. The top arrow is represented by a "v" on the top line, and the bottom arrow (^) is between the rows. Node 00 can receive input from its neighbor on the right, Node 01, or from Node 06. Since it's in the lower left corner of the array, however, it has no neighbors on the left and bottom sides. Often black (inactive) nodes have arrows on all four sides (unless they're on an edge), indicating that they can receive commands from any neighbor, although in some circumstances they await commands only from certain neighbors.

Figure 2.3 shows the detailed display of one of the nodes selected by **watch1**, **watch2**, or **watch4**. Here you can see not only the state of all registers, but the entire stacks as well. You can also see the current instruction word being processed, the current slot, current opcode, and its translation as an instruction (**push**, in this case).

Note: since the simulator runs in the host PC, the glossary shows the PC stack, not a SEAforth stack.

Glossary

reset (—) H *reset*
Select the mode that simulates code in the target chip.

power (—) H *power*
Select the mode that simulates a download before running the application.

simulate (or **sim**) (—) H *sim*
Start or resume simulation. While the simulator is running, you may single-step it by pressing the spacebar, or stop it by pressing any other key.

simq (—) H *sim-q*
Start or resume simulation and run for the number of steps previously specified by **setmax** or **upto**. "Q" stands for "quiet," meaning that the display will not refresh.

setmax (n —) H *set-max*
Set the simulation to run for *n* steps or until stopped by a non-space keypress, whichever comes first. A value of -1 means run continuously.

setstep (n —) H *set-step*
Set the simulation to run for *n* steps without refreshing the display between steps.

upto (n —) H *up-to*

Set the simulation to run without display until step *n* has occurred, and then the simulator will exit. Invoke **sim** again to begin simulating. The simulator will use the values supplied to **setmax** and **setsteps**.

fixate (n —) H *fixate*
Focus the simulation on Node *n*, by stepping continuously until a key is pressed, but updating the display only when Node *n* is awake.

active (n —) H *active*
Step quietly (without display) until Node *n* becomes active, then continues with **simulate**.

watch1 (n —) H *watch-one*
Add a detailed display of Node *n* to the simulator display.

watch2 (n_1 n_2 —) H *watch-two*
Add a detailed display of Nodes n_1 and n_2 to the simulator display.

watch4 (n_1 n_2 n_3 n_4 —) H *watch-four*
Add a detailed display of up to four nodes to the simulator display. The four will be in the order n_1=top left, n_2=bottom left, n_3=top right, n_4=bottom right. A value of -1 will cause that position to be skipped.

.adrs (addr len —) H *dot-address*
Display a combination memory dump and disassembly for the current node, starting at *addr* and extending for *len* words.

dump (addr len —) H *dump*
Display a more comprehensive memory dump and disassembly for the current node, starting at *addr* and extending for *len* words. This is particularly effective for dumping a stream in the "external node" **[x]** (see Section 4.4). You can see which node particular sections of code were compiled for.

dumpRAM (—) H *dump-RAM*
Runs **.adrs** for the full RAM on every node in the chip, providing a complete listing of the RAM code compiled for this project.

dumpROM (—) H *dump-ROM*

Runs **.adrs** for the full ROM on every node in the chip, providing a complete listing of the ROM code compiled for this project.

Chapter 3 VentureForth Commands

In this chapter we'll discuss the VentureForth commands that the compiler will render into machine code. In many cases, there is a one-to-one correspondence between VentureForth words and SEAforth machine opcodes. However, there are also a number of compiler directives whose purpose is to manage program flow-of-control structures, comments, and other compiler-level functions, as well as pre-defined macros that provide useful capabilities.

Some commands that have specific purposes are documented in their respective sections; this chapter covers the general-purpose functions.

3.1 Primitive Commands

Send feedback

This section covers the commands that correspond directly to machine instructions. Here we will treat them as language elements.

Each of the commands in this section has a one-to-one correspondence to the opcode it compiles. Each compiles its opcode in the next open slot of the instruction word that the compiler is currently constructing. The data sheet specifies the runtime characteristics of each of these opcodes.

3.1.1 Stack Operations

Send feedback

The commands described in this section affect only the stack Registers T, S, and R, plus the consequent adjustments to the data and return stacks as a result of items being pushed or popped.

Stack management on the circular stacks in the SEAforth parts requires special strategies. Its bounded stacks allow two unconventional usages:

- "Circular re-use" means *reading* beyond the theoretical end of the stack. Because the stacks are implemented as arrays, reading a

value (**pop**, **drop**) just moves the pointer to the previous location modulo 8 (the size of the circular stack), leaving the data in place. When read repeatedly, the pointer eventually returns to that position, where the same data will be read again. This saves reloading literals or calculated values from memory when they are used repetitively.

- "Programming with abandon" is simply leaving data on the stack, until it's overwritten when the stack pointer circles around. After writing to a location, the pointer moves to the next location modulo the stack size. When a stack is written repeatedly, the effect is to overwrite previously written data which has been "abandoned." This saves crucial opcode space (and time) compared to explicitly moving the stack pointer back (**drop**, **pop**).

In both situations it's important to distinguish between intentionally and inadvertently causing the behavior: one is a feature, the other is a bug!

Glossary

dup (x — x x) A *dup*
Pushes a copy of the top stack item onto the stack.

drop (x_1 x_2 — x_1) A *drop*
Discards the top data stack item.

over (x_1 x_2 — x_1 x_2 x_1) A *over*
Pushes a copy of the second stack item onto the stack.

push (x — R:x) A *push*
Pops the top item from the data stack, pushing it onto the return stack.

pop (R:x — x) A *pop*
Pops the top item from the return stack, pushing it onto the data stack.

3.1.2 Register Operations

Send feedback

The words described in this section are used to manage the A and B registers, which contain 18-bit or 9-bit numbers, respectively.

Register A can be written and read, and is used for addressing memory and ports as well as for temporary storage.

Register B can be written but not read. It is set to each node's I/O Control and Status location (IOCS, $15D) at reset/power-on, and is

generally left there unless two or more addresses are needed. It is mainly used to accessi either IOCS or the address of its neighbor node.

NOTE: don't confuse the register commands **a@** and **a!** (which read and write the A register itself) with the memory commands **@a** and **!a** (which read and write the *contents of the memory or port address* that is currently in A). Similarly, don't confuse **b!**, which puts something in the B register, with **!b** which stores something into the location whose address is in B. See Section .

Glossary

a@ (— x) A *a-fetch*
Fetches the value in Register A and pushes it onto the stack. Register A is unchanged.

a! (x —) A *a-store*
Pops the top stack item and stores it in Register A.

b! (addr —) A *b-store*
Pops the data stack and stores the low-order 9 bits of the top stack item in Register B.

3.1.3 Addressing Operations

Send feedback

Each SEAforth core has 64 18-bit words of RAM and 64 18-bit words of ROM. Addresses $00 to $3F refer to RAM, while addresses $80 to $BF refer to ROM. When addressing either RAM or ROM, bit 6 is not decoded. As a result, addresses $40 to $7F map to the addresses $00 to $3F in RAM, and $C0 to $FF map to $80 to $BF in ROM.

The words described in this section provide access to local memory and ports. They use either the A or B register (which contain 18-bit or 9-bit numbers, respectively) or Register P.

NOTE: don't confuse the commands **a@** and **a!** (which read and write the A register itself) with **@a** and **!a** (which read and write the *contents of the memory or port address* that is currently in A).

Glossary

@a (— x) A *fetch-a*
Fetches the *contents* of the location whose address is in Register A and pushes it onto the stack. Register A is unchanged.

!a (x —) A *store-a*
Pops the top stack item into the memory location whose address is in Register A. Register A is unchanged.

@a+ (— x) A *fetch-a-plus*
Fetches the contents of the memory location whose address is in Register A and pushes it onto the stack. Register A is incremented by one when pointed to a RAM or ROM memory address, but not when Register A points to a port.

This command is used to access consecutive memory locations. The incrementing behavior of this instruction is such that it will automatically wrap at the address limits of ROM or RAM. That is, if the address in A is $07F (RAM) an increment will wrap to 000, and the address $0FF will wrap to $080.

!a+ (x —) A *store-a-plus*
Pops the top stack item into the memory location whose address is in Register A. Register A is incremented by one when pointed to a RAM or ROM memory address, but not when Register A points to a port.

This command is used to fill consecutive memory locations. The incrementing behavior of this instruction is such that it will automatically wrap at the address limits of ROM or RAM. That is, if the address in A is $07F (RAM) an increment will wrap to 000, and the address $0FF will wrap to $080.

@b (— x) A *fetch-b*
Fetches the *contents* of the location whose address is in Register B and pushes it onto the stack. Register B is unchanged.

!b (x —) A *store-b*
Pops the top stack item and stores it in the location whose address is in Register B. Register B is unchanged.

@p+ (— x) A *fetch-p-plus*
Fetches the contents of the location pointed to by Register P and pushes it onto the stack. Register P is incremented by one when pointed to a RAM or ROM memory address, but not when it points to a port.

This command may be used to access consecutive memory locations. The incrementing behavior of this instruction is such that it will automatically wrap at the address limits of ROM or RAM. That is, if the address in the Register P is $07F (RAM) an increment will wrap to 000, and the address $0FF (ROM) will wrap to $080.

!p+ (x —) A *store-p-plus*
Stores the top stack item in the location pointed to by Register P. Register P is incremented by one when pointed to a RAM or ROM memory address, but not when it points to a port.

This command may be used to fill consecutive memory locations. The incrementing behavior of this instruction is such that it will automatically wrap at the address limits of ROM or RAM. That is, if the address in Register P is $07F (RAM) an increment will wrap to 000, and the address $0FF (ROM) will wrap to $080.

3.1.4 Logical Operations

Send feedback

The commands in this group perform Boolean and complement operations.

Glossary

not (n_1 — n_2) A *not*
Performs a bitwise inversion on n_1 to give n_2.

and (n_1 n_2 — n_3) A *and*
Performs a logical AND of n_1 and n_2 to produce n_3.

xor (n_1 n_2 — n_3) A *xor*
Performs an exclusive OR of n_1 and n_2 to produce n_3.

3.1.5 Arithmetic Operations

Send feedback

There are very few single-instruction arithmetic commands in VentureForth. Instead, there are subroutines available in most nodes to do some computations. See Section A.2 or Section B.1 for details.

Please note that the **+** and **+*** instructions take two cycles. As a result, they must be preceded by either a **nop** (**.**, see Section 3.4.4) or one of the instructions flagged as "helps **plus**." See the data sheet for details.

Glossary

+ (n_1 n_2 — n_3) A *plus*
Adds n_1 and n_2 to produce n_3. In standard mode, the carry bit is not affected, but in extended mode it is; see Section 3.1.6.

+* (n_1 n_2 — n_1 n_3) A *plus-star*
Computes a partial product, given a multiplicand in the high bits (possibly all 18 bits) of S and a multiplier in the low bits (possibly all 18 bits) of A. The low word of the result is in A and the high word is in T. **+*** is a building block for multiply and related operations. It should be used only in standard mode. If it is used in EA mode, the carry must be cleared first.

The S24 doesn't use the A register. Instead, multiplicand is in the low bits of T and the result is in T alone.

2* (n_1 — n_2) A *two-star*
Performs an arithmetic left shift on n_1 to give n_2 (equivalent to multiplying by 2).

2/ (n_1 — n_2) A *two-slash*
Performs an arithmetic (signed) right shift on n_1 to give n_2 (equivalent to dividing by 2).

Arithmetic macros and subroutines are discussed further in Section A.2 for the S24 parts, and Section B.1 for S40C18 parts.

3.1.6 Extended Mode Arithmetic (S40C18 only)

The SEAforth 40C18 parts extend the instruction set to facilitate arithmetic on large numbers. Such arithmetic is required by, for example, cryptographic algorithms that involve modulo multiplication and exponentiation. Only **+** and **+*** are affected by extended mode.

The major change that extended mode brings to **+** is the use of the carry bit. In standard mode, the carry bit is ignored (treated as zero). In extended mode, however, **+** adds the top two stack items *and* the carry bit, which is latched out of the high-order bit of the T register.

Extended mode is selected by setting bit 9 ($100) in Register P. When that bit is not set, **+** and **+*** do not affect the carry bit or Register A.

To add large numbers in extended mode, first clear the carry bit, then add pairs of 18 bit words of the numbers, starting with the low order word, so that the carry propagates upward through the sum (an effect known as *ripple carry*). To subtract, on the S24 you may add the one's complement (produced by using **not**) of one of the numbers to the other in in a similar fashion, except that in this case the initial carry is set to 1,

thus producing the two's complement of one of the arguments. On the S40C18 you may **negate** and add; see Section B.1.

To clear the carry latch, you may use the phrase:

```
dup xor dup . +
```

To set it, you may use:

```
dup xor not dup . +
```

If you have throwaway values on the stack (e.g., left from a previous operation) you may do this more concisely: adding two *positive* numbers clears the carry bit, while adding two *negative* numbers sets it.

The extended mode **+*** is used as a building block to multiply two numbers (in S and A), and works by computing a series of partial products, which are added as they are generated. It maintains a double-length (36-bit) value in Registers T and A (which, for convenience, we'll refer to as T.A), with the least-significant word in A. Note: T should be cleared before the first **+*** in a set.

If the least-significant bit of A is 0, then the 37-bit (sign extended) value n T.A is simply shifted right one bit. However, if the least-significant bit of A is 1, the content of S is first added to T before the whole 37-bit value in T.A is shifted. In either case, the LSB of T is shifted into the MSB of A.

The rules for ripple carry and the potential need for a nop or other delay which applies to the plus opcode also applies here (see the S40C18 Data Sheet for details). Preceding a **+*** with a **nop** will always provide sufficient time for the ALU to settle, but other approaches are also discussed in the data sheet.

The result of executing 18 **+*** instructions will be a signed 18x18 multiply producing a 36-bit product of S and A; however, it is possible, through careful treatment of data, to perform shorter and faster multiplys.

3.2 Definitions, Calls and Returns

Send feedback

The Forth language, upon which the SEAforth architecture is based, is extremely modular. Subroutines (often called "words") are extremely small, often only a few instructions in length. The modularity of Forth facilitates extreme code density as a consequence of being able to re-use even very short code segments. This offers advantages not only in overall program size, but also program reliability, because it's extremely easy to validate short, simple code segments.

Use of the data stack for parameter passing is an important element in making the relationship between these tiny routines simple and fast. There are no "calling sequences" such as one finds in C and other high-level languages: Forth words simply expect their parameters on the data stack and leave their results there.

The basic structure of a definition in VentureForth is:

```
: <name> ( stack-on-entry - stack on exit ) <commands> ;
```

Subsequent reference to *name* in other definitions will cause VentureForth to compile a **call** to it. Note that the comment containing stack information has no technical consequence — it is merely a comment, but an essential one for program maintainability.

For example:

```
: double ( n – n*2 )   dup . + ;
: 2+2 ( – n )   2 # double -;
```

In the definition **2+2**, VentureForth will compile a literal 2 followed by a tail-recursion jump to **double*** and a return to whatever word called **2+2**.

SEAforth nodes differ from conventional microcontrollers in that their local memory is extremely limited. As a result, programmers are encouraged to take advantage of the fact that all the symbol-table information for a definition resides only in the host compiler; only executable code exists in the target. Therefore, rather than have **2+2** above actually call **double**, one might program this sequence thus:

```
: 2+2 ( – n )   2 #
: double ( n – n*2 )   dup . + ;
```

Here the literal 2 is pushed on the stack, and the code sequence

```
dup  . + ;
```

(which follows immediately in target memory) is executed. **2+2** "falls through" into **double**, and the **return** compiled by **;** at the end of **double** returns to the caller of **2+2** *or* the caller of **double** (if it was called explicitly rather than via **2+2**). The definition of **2+2** is faster due to the absence of the jump around **double** in the first version.

Here's a similar example of two entry points into shared code:

```
: abs ( n - n )   -if
: neg ( n - n )   not 1 # . + then ;
```

Here **abs** will perform **neg** if its argument is negative.

* **double** could be defined more efficiently using **2***, but this version is more instructive.

The various words used to manage calls and returns in VentureForth follow.

Glossary

: <name> (—) A *colon*
Generates a label for the *name* that immediately follows, and begins compiling code associated with that label. Subsequent use of the label in a definition will compile a **call** to it. Labels reside only in the host compiler, and do not affect the target image, which contains only code.

-; (—) A *minus-semi-colon*
Converts a **call** that immediately precedes it to a jump (that is, nothing is pushed on the return stack and there will be no return). In the code example given above, if **2+2** is defined as:

```
: 2+2 ( — n )   2 # double -;
```

...then there will simply be a jump to **double**, and the return at the end of **double** will go to the location following the call to **2+2**.

; (R:addr —) A *semi-colon*
Generates a return to the address in the R register, popping the return stack. The compiler will automatically convert a **call** to a **jump** and omit the return if a **call** immediately precedes **;** (tail recursion).

call (— R:addr) A *call*
During compile time, **call** pops address off of host stack and compiles a call to that address.

Generates a call to the specified address, pushing the address in the P register onto the Return Stack. This is useful when you wish to call something other than a named entry point, such as a port address. For example:

```
'rd-u call
```

...executes a multiport call to the Right, Down, and Up ports.

;: ($R:addr_1$ — $R:addr_2$) A *semi-colon-colon*
Used for making co-routines. The **co-routine** opcode calls the address on the return stack. It is named because of its usefulness in going back-and-forth between two cooperating routines. In this view it returns to its caller while saving its own location for a later return.

An example of this in use is shown in Listing 3.1.

Listing 3.1 Example of co-routines

```
: poll ( - )
   @b $200 # and if     \ Is write request set?
      '---u # b!        \ "Up" neighbor port to B
      @b push ;:        \ call in B to R; execute it.
   'iocs # b!  then     \ Restore IOCS address to B
drop ;                  \ Discard 'if's' argument & return
```

The purpose of this code is to respond to a write request from the neighbor node in the Up direction (see Section 4.1), which will be indicated by bit $200 in the IOCS (whose address is in B). If the bit is set, we will respond by storing the Up port's address in B. We know that this port will contain a **`call`**, because that is how inter-node conversations are initiated, so we can read this from the port and push it onto the return stack. The actual **`;:`** command simulates executing the call in such a way that the return will go to the next instruction (**`'iocs # b!`**). The last instructions reset B to its normal contents (IOCS), and discard the result of the **`and`** given to **`if`** (since VentureForth's **`if`** does not discard its parameter).

3.3 Program Structures

Send feedback

The Forth language has always used structured techniques to manage loops and conditionals, rather than arbitrary branches. This section covers the words used to manage various forms of loops and conditionals in VentureForth. These are conceptually similar to the structures in Standard Forth, but there are some critical differences. Programmers accustomed to programming in Standard Forth should be alert to this fact!

As in Standard Forth, all program structures must be inside a definition. Unlike Standard Forth, however, it is possible to branch into what appears to be a different word. This is addressed in Section 3.2. In other words, the destination of a branch doesn't necessarily have to fall within the same definition as its origin. The important exception to this is that structures built with **`for`**, **`next`**, and **`unext`** (Section 3.3.4) use the return stack for the loop counter; therefore, you may not do anything that will modify R within the range of a **`for...next`** or **`for...unext`** structure such that the loop counter becomes unavailable. That is, you may do a **`push`**

and subsequent **pop** if both are within the loop, but an unbalanced **push** or **pop** will affect the availability of the loop counter.

Being faced with small program resources and tight timing constraints encourages iterative code optimization. Because of the natural isolation of code in each node, the code optimization proceeds toward more and more specialized and less general functions. Pieces of code may shift between caller and called in ways that may look capricious to the casual observer but which may represent a significant savings over first cut coding. It is therefore especially important for VentureForth programmers to stop and document relationships and dependencies that will not be obvious to the unwary reader.

3.3.1 A Simple Branch

Send feedback

The word **jump** compiles an unconditional branch to an address specified by the host stack, when executed it will transfer control to the ompiled address. It is used by a number of the words in the next sections, such as **repeat**, **again**, and **else**, all of which need to compile an unconditional branch.

The P register can be used to read, write or fetch instructions from either memory or a neighbor port. The only way to load a value into P is with a branching instruction, of which **jump** is the simplest and most common. The most common direct use of **jump** is in connection with port execution, described in Section 4.3.1.

Glossary

jump (—) A *jump*

At compile time, **jump** takes an addredd on the host's stack and compiles a jump to it. At run time, **jump** branches the the address.

3.3.2 Conditionals

Send feedback

The basic conditional structure in VentureForth is:

```
... if <code-for-T-nonzero> else <code-for-T=0> then ...
```

The word **if** will test the value in T. If it is non-zero, the code between **if** and **else** will be executed, and then control passes to the word following

then. If T is zero, the code following at **else** will be executed, instead. Figure 3.1 shows the logical flow in this structure.

NOTE: the data stack is not popped by VentureForth's **if** *as it is in Standard Forth! T is not altered!*

The **else** clause may be omitted in its entirety. That is, you may have a conditional of the form:

```
... if <code-for-T-nonzero> then ...
```

In this case, if the value in T is zero, the code immediately following **if** will be skipped, and control will pass to the code following **then**.

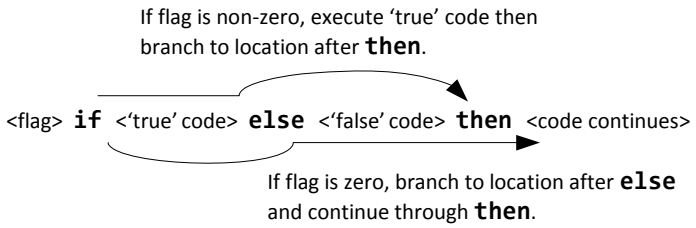


Figure 3.1 Diagram of an **if ... else ... then** structure

In addition to **if**, there is also a word **-if** that behaves similarly, except that it executes the code following if T is negative, rather than in the non-zero case, thus:

```
... -if <code-for-T-negative> else <code-for-T-positive> then ...
```

As with **if**, the **else** clause may be omitted.

Glossary

if (n — n) A *if*
Tests the value in T. If it is non-zero, continues executing the code immediately following. If T is zero, branches to the point immediately following **else** (if there is one) or **then** (if there is no **else** clause).

-if (n — n) A *minus-if*
Tests the sign bit of the value in T. If it is set (T is negative), continues executing the "true" code immediately following. If the sign bit of T is zero (T is positive), branches to the point immediately following **else** (if there is one) or **then** (if there is no **else** clause).

else (—) A *else*
Begins the optional "false" part of a conditional structure.

then (—) A *then*
Terminates a conditional structure by resolving a forward branch compiled by **if**, **-if**, **else**, or **meanwhile** (described in Section 3.3.3).

3.3.3 Indefinite loops

Send feedback

An "indefinite loop" is one for which it is not known programmatically how many times it will repeat. Typically, an indefinite loop is expected to repeat either *while a condition exists* or *until an event occurs*. VentureForth provides structures optimized for both conditions.

The structure:

```
... begin <code-always-executed> while <conditional-code> repeat ...
```

...will continue to execute the conditional code so long as the argument to **while** (in T) is non-zero. If **while** sees a non-zero, the conditional code will execute and **repeat** will branch unconditionally back to the point immediately following **begin**. Figure 3.2 shows a diagram of this structure.

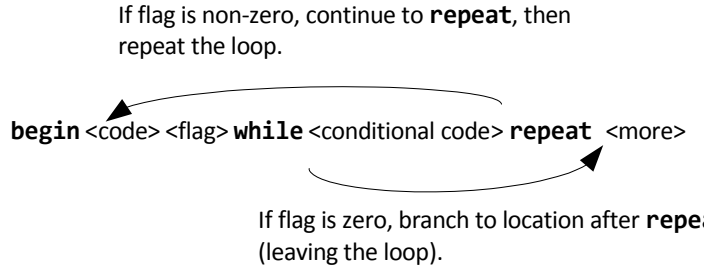


Figure 3.2 Diagram of a **begin ... while ... repeat** structure

NOTE: The data stack is not popped by VentureForth's **while** *as it is in Standard Forth! T is not altered!*

There do not have to be any commands between **begin** and **while**. If there are commands there, they will always be executed at least once. It is quite possible for the conditional code to be executed no times, however, if the **while** test fails the first time.

As in the case of **-if**, there is also a **-while**. It tests the high bit of T, and will continue while T is negative.

The structure:

```
... begin <repeating-code> until ...
```

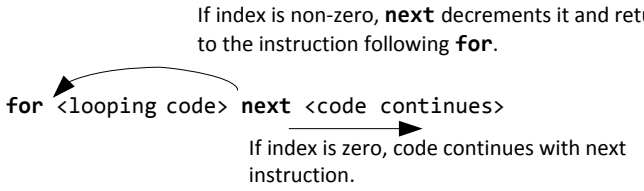


Figure 3.3 Diagram of a **`for ... next`** structure

...will continue to execute until the argument to **`until`** (in T) is non-zero. So long as the argument is zero, the repeating code will be executed. When **`until`** sees a non-zero, execution will continue at the next instruction following **`until`**.

NOTE: The data stack is not popped by VentureForth's **`until`** *as it is in Standard Forth! T is not altered!*

The variant **`-until`** is similar to **`until`**, except that it will exit the loop when T is *negative*, rather than when it is non-zero.

The structure:

```
... meanwhile <skipped-first-time> then <repeating-code> until ...
```

...will initially branch forward to the point following **`then`**, and execute the repeating code. So long as **`until`** sees a non-zero value in T, it will branch back to the code immediately following **`meanwhile`**. This structure may also be terminated with **`-until`**, in which case it will loop until T is negative. In most cases this structure will be faster than a **`while`** loop because there is only one branch executed each iteration inside the loop, but be careful to observe that the condition for the loop exit for a **`while`** loop is opposite from the condition for **`until`**, and this will also affect the overall loop efficiency.

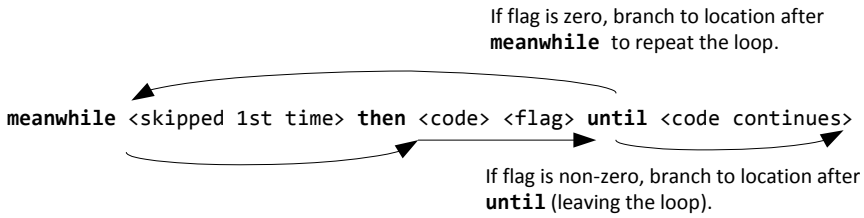


Figure 3.4 Diagram of a **`meanwhile ... then ... until`** structure

Following is a summary of the words used in indefinite loops.

Glossary

begin (—) A *begin*
Marks an address to which a backward branch may be compiled. Used to begin loop structures.

while (n — n) A *while*
Branches forward to an address marked by **repeat** if the value in T is zero. Falls through and continues the loops if the value in T is non-zero.

-while (n — n) A *minus-while*
Branches forward to an address marked by **repeat** if the value in T is positive (its sign bit is zero). Falls through and continues the loop if the value in T is negative (sign bit is one).

until (n — n) A *until*
Branches backward to an address marked by **begin** if the value in T is zero. Falls through, exiting the loop, if T is non-zero.

-until (n — n) A *minus-until*
Branches backward to an address marked by **begin** if the value in T is positive (its sign bit is zero). Falls through, exiting the loop when the value in T is negative (its sign bit is one).

repeat (—) A *repeat*
Unconditionally branches back to an address provided by **begin** and marks the start of the code reached by while loop exit.

meanwhile (—) A *meanwhile*
Unconditionally branches forward to an address provided by **then**, and also marks the point to which a subsequent **until** or **-until** may branch.

3.3.4 Finite loops

Send feedback

A finite loop is used when you have a known number of times for which you wish your code to be repeated.

The structure:

```
... <n> for <repeating-code> next ...
```

will execute the repeating code *n+1* times. The word **for** will transfer *n* from the data stack to the return stack. **next** will test it; while it is non-zero, it will decrement it by one and branch back to the location following **for**. When the value is zero, it will discard it and exit the loop.

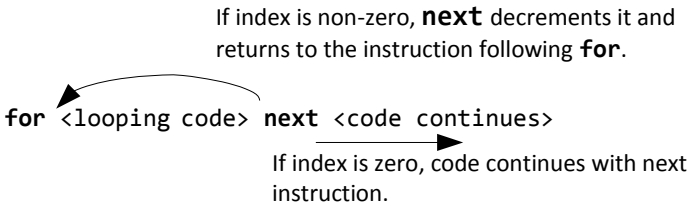


Figure 3.5 Diagram of a **for ... next** structure

An important variant of **next** is **unext** (*micro-next*). It supports loops that reside entirely within a single instruction word. It manages R like **next**, but instead of branching to a memory location, it branches back to slot 0 of the current instruction word. The body of the loop, therefore, can contain up to three opcodes. When the **unext** loop exits, it continues with the opcode in the following slot. **unext** is especially useful in that doesn't have to fetch an instruction until the loop finishes.

It is possible to start a finite loop using **begin**, but you must previously have placed your counter in R using **push**.

Following is a summary of the words used in finite loops.

Glossary

for (n — R:n) A *for*
Begins a finite loop by pushing the *n* onto the return stack, and marks an address to which a backward branch may be compiled. Used to begin finite loop structures which will repeat *n+1* times.

next (R:n — R:n-1 *if non-zero* | *if zero*) A *next*
Tests the counter in R. If it is non-zero, it decrements it and branches back to the point marked by **for** or **begin**. If the counter is zero, it is popped from the return stack and execution continues with the instruction following **next**.

unext (R:n — R:n-1 *if non-zero* | *if zero*) A *micro-next*
Tests the counter in R. If it is non-zero, it decrements it and branches back to slot 0 in the current instruction word. If the counter is zero, it is popped from the return stack and execution continues with the opcode following **unext**. Unext uses address on host stack to verify that structure is in single instruction word. **unext** aborts if the structure contains more than an instruction word.

3.3.5 Infinite loops

Send feedback

An infinite loop is one which has no built-in countdown or terminating condition. It is commonly used for the highest-level behavior of a process or node, representing its entire behavior while power is on.

The infinite loop structure is:

```
... begin <repeated-code> again ...
```

The code inside the loop will repeat until the node is reset.

Glossary

again (—) A *again*
Unconditionally branches back to a location marked by **begin**.

3.4 Compiler Directives

Send feedback

All compilers need a set of commands that control the process of generating code, but do not themselves generate target code. Such commands are commonly called "compiler directives." VentureForth is no exception. These words are discussed in this section.

The behavior of the VentureForth compiler, like other Forth compilers, is simply to parse text (usually from a source file) and execute the words as they are encountered. Some of these words are "defining words," which will cause new definitions to be created; some add code to definitions currently under construction; some create program structures, as described in Section 3.3; others perform other useful functions such as obtaining addresses of words, nodes, ports, or other objects and doing useful things with them.

The words described in this section are all executed on the *host* computer for the purpose of helping build a program which will be installed and executed there at some later time on a SEAforth node. As a result, stack parameters are for the host's stack, and do not affect the target's stack or registers, helping build a program which will be installed and executed on a SEAforth node at some later time.

3.4.1 Comments

Send feedback

Well-commented code is essential for code sharing within a programming team, as well as for publishing or sharing code and long-term code maintenance.

Glossary

((—) H *left-paren*
Begins a comment (text that will be ignored by the compiler) terminated by **)** (right-paren). **(** is a word, and therefore must be followed by a space delimiter. **)** is just a terminating delimiter, however, and does not need to be set off by spaces.

**** (—) H *back-slash*
Begins a comment that extends to the end of the current line. Like **(**, **** must be followed by a space.

3.4.2 Address Management

Send feedback

The words in this section are used to specify the node as well as addresses to which code will be compiled and where execution may begin. These words, like others in this section, control behavior of the compiler, not the target, and therefore only affect the host's stack.

Glossary

equ <name> (n —) H *e-q-u*
Defines a constant in the compiler. Use of *name* pushes *n* onto the stack.

{node (n —) H *bracket-node*
Starts compiling code for node *n*.

node} (—) H *node-bracket*
Terminates compilation for the current node.

org (addr —) H *org*
Starts compiling code for address *addr* in the current node.

3.4.3 Node Initialization

Send feedback

The words in this section are used to set up the initial state of a node, including its stacks and registers. These may be used with all nodes, *except* that the root node (the original node managing the stream loading process) may not initialize its A or B registers or its stacks, because they are used in the booting process.

Glossary

=p (addr —) H *equal-p*
Starts executing code for the current node at *addr*.

=a (x —) H *equal-a*
Initializes Register A to *x* at startup.

=b (addr —) H *equal-b*
Initializes Register B to *x* at startup.

>stk (x_1 x_2 ... x_n n —) H *to-stack*
Initializes the target's data stack to contain the *n* values listed. The first value, x_n, will be in T.

>rtn (x_1 x_2 ... x_n n —) H *to-return*
Initializes the target's return stack to contain the *n* values listed. The first value, x_n, will be in R.

3.4.4 Slot Management

Send feedback

Each word can contain up to four instructions. However, certain instructions must be in certain positions (slots). Also, some instructions, such as **+**, have timing requirements that can be satisfied by putting one or more NOPs in a slot. Refer to the Data Sheet for details.

Glossary

. (—) H *no-op*
Skips a slot in an instruction word. Often used before **+** and **+***, and may be required at other times. Consult the documentation for the specific SEAforth parts you are using for further information.

nop (—) H *no-op*
Synonym for **.** (mostly used in descriptive text; **.** is the preferred usage in program source).

| (—) H *bar*

Guarantees that the next open slot is slot 0. If the next open slot is not 0 it fills the rest of the current instruction word with **.** (**nop**).

?| (—) H *question-bar*

Guarantees that the next open slot is slot 0 or slot 1. If the next open slot is not 0 or 1, fills the rest of the current instruction word with **.** (**nop**).

3.4.5 Conditional Compilation

Send feedback

There are times when it is convenient to configure compilation of code within a file. That is, you may have a file of definitions of which some are required in all nodes, but some only in certain nodes. Since code space is a scarce resource, it is good to be able to have switches that specify what node you're compiling for so that you may selectively use or skip certain features. The commands for doing this resemble the

```
if ... else ... then
```

clause in general VentureForth (see Section 3.3.2).

A common use of this feature is to provide for a block of commentary that may extend over multiple lines and include parentheses, as shown in in Listing 3.2.

Listing 3.2 Use of conditional compilation for block comments

```
0 [if]                    \ Note, this is always 'false'
      ...                 \ Extended comments go here (as needed)
[then]                    \ Comment block ends
```

As is the case with the **if ... else ... then** structure, the **[else]** clause may be omitted, but every **[if]** *must* be terminated by a **[then]**. And it's perfectly legitimate to nest **[if] ... [else] ... [then]** structures.

These commands are part of the host computer's Forth system; therefore, stack behavior reflects the host's data stack, not the SEAforth processor's circular stack.

Glossary

[if] (t —) H *bracket-if*

If *t* is non-zero, the text following will be processed until the next **[else]** or **[then]** is reached. Note that *t* will be removed from the data stack.

[else] (—) H *bracket-else*
If the *t* before the most recent **[if]** is zero, the source following **[else]** will be processed until the next **[then]** is reached.

[then] (—) H *bracket-then*
This word marks the point at which interpretation of source will resume.

3.4.6 Programming Tools

The words described in this section are used at compile time to provide access to the code under construction.

There are two methods of compiling literals, though one is really a higher-level implementation of the other.

```
@p+ 42 ,
```

...will push the number 42 onto the stack when this code is executed. The phrase:

```
42 #
```

...will do the same thing. Which is preferable depends on the circumstances. The command **,** (comma) will not affect the slot counter. For example, the sequence in Listing 3.3 will leave 0 1 2 3 4 5 on the stack, and the compiled result will look like Figure 3.6. However, the code when executed will push the desired numbers 0 -1 2 and -4 onto the stack (with -4 in T).

Listing 3.3 Example of consecutive literals

```
0 org
here @p+
here 0 , @p+
here -1 , @p+
here 2 , @p+
here -4 ,
here
```

The reason is that, although **,** (comma) advances **here**, it does not advance the slot counter in the word being compiled. Therefore, the instructions were compactly compiled in Word 0, but when they are executed they will advance Register P appropriately, picking up the consecutive literal values.

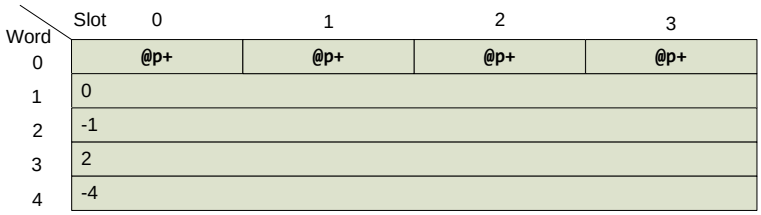


Figure 3.6 Result of compiling the code in Listing 3.3.

The circumstances leading to a choice of **,** over **#** are best illustrated by an example. In this example the goal is to place on the stack the literal number 1 beneath a word which is the encoded form of the instructions **dup dup xor** (which, if executed, is a handy shorthand way to zero T). One way to do this would be to write

```
1 #  @p+  | dup dup xor
```

and that would be fine, but if you want to put the instruction word **dup dup xor** on the stack with the literal 1 on top of that, saying:

```
@p+ 1 #   | dup dup xor
```

would have them in reverse order. **@p+** in slot 0, **@p+** in slot 1 with 1 in the next word, and then **dup dup xor** in the next word. The work around would be:

```
@p+ @p+  | dup dup xor  |  1 ,
```

Glossary

' <name> (— addr) H *tick*
Returns on the host's stack the target address of *name* (assuming it has been previously defined in the target).

's <name> (n — addr) H *<n>'s*
Returns Node *n*'s address of *name* (assuming it has been previously defined in that node).

, (n —) H *comma*
Compiles the number *n* into the next available word. This command is often used in conjunction with **@p+** to make a literal. For example, the sequence:

```
@p+ 35 ,
```

...will compile the **@p+** instruction followed by the number 35. When this code is executed, the **@p+** will fetch the 35 and push it onto the data stack, while incrementing the P register past it.

here (— addr) H *here*
Pushes the address of the next available target location onto the host's stack at compile time. For example, the sequence:

```
here 42 ,
```

...will compile the number 42 into the dictionary, and push its address onto the host's stack at compile time. You may subsequently compile a reference to this address.

(n —) H *number-sign*
Compiles *n* into the dictionary as a literal (**@p+** followed by *n*). This means when the code in which it appears is executed, it will be pushed onto the stack. The compiled result of **35 #** is equivalent to the sequence:

```
@p+ 35 ,
```

Chapter 4 Inter-node Communication

Each SEAforth chip is a community of nodes. Versions have been produced with 24 nodes (arranged 6x4) and 40 nodes (arranged 10x4); other variants will doubtless be announced in time. Certain nodes on the outside of the array have special I/O capabilities, and thus can communicate with external devices; interior nodes can only communicate with their immediate neighbors.

4.1 Basic Node Geography

Send feedback

Figure 4.1 shows the designated functional types of nodes in a generic S40C18 part (some specific parts may differ). The specific capabilities of the outer nodes are described in more detail in the Data Sheet for each individual part.

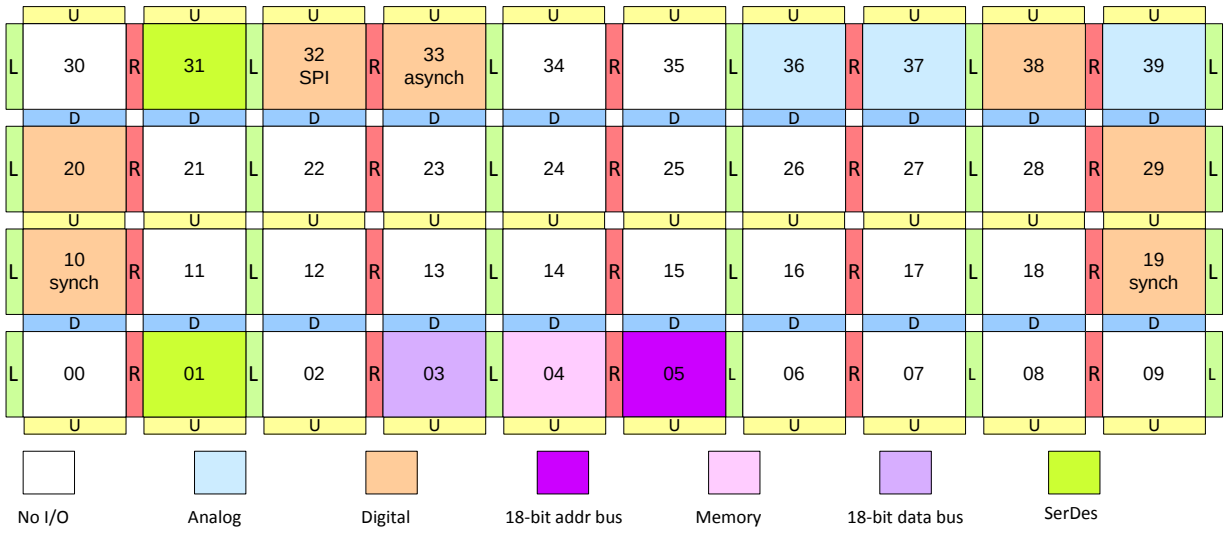


Figure 4.1 Block diagram of S40C18, showing I/O and direction ports

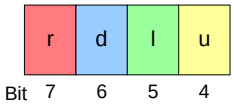


Figure 4.2 Direction bits

Each individual node can communicate to its immediate neighbors using ports that are described using directional terminology: up, down, left, and right. However, two adjacent nodes both communicate to each other using the *same* direction. That is, if Nodes 00 and 01 communicate with each other, *both* will use their "right" ports.

A mnemonic may be used to refer to a chip's direction ports: **rdlu** (Right, Down, Left, Up). Each port is represented by one bit in a nibble, as shown in Figure 4.2. A set of constants has been defined in the compiler to provide the appropriate bit patterns for every possible combination. Their names are of the form: `'----` where each of the dashes represents one of the direction bits. For example, `'rd--` is "right and down," while `'r--u` is "right and up," `'r---` is "right only," and `'-dlu` is "down, left, up."

A summary of the direction port specification constants, showing the resulting port addresses, is given in Table 4.1. Note that bits 4 and 6 are active low, so that, for example, 0000 is down and up (bits 4 and 6 active).

Table 4.1 Direction port selection constants

Name	R (bit 7)	D (bit 6)	L (bit 5)	U (bit 4)	Port	Result
`'----`	0	1	0	1	$155	No port selected
`'---u`	0	1	0	0	$145	Up
`'--l-`	0	1	1	1	$175	Left
`'--lu`	0	1	1	0	$165	Left, Up
`'-d--`	0	0	0	1	$115	Down
`'-d-u`	0	0	0	0	$105	Down, Up
`'-dl-`	0	0	1	1	$135	Down, Left
`'-dlu`	0	0	1	0	$125	Down, Left, Up
`'r---`	1	1	0	1	$1D5	Right
`'r--u`	1	1	0	0	$1C5	Right, Up
`'r-l-`	1	1	1	1	$1F5	Right, Left
`'r-lu`	1	1	1	0	$1E5	Right, Left, Up
`'rd--`	1	0	0	1	$195	Right, Down
`'rd-u`	1	0	0	0	$185	Right, Down ,Up

Table 4.1 Direction port selection constants *(continued)*

Name	R (bit 7)	D (bit 6)	L (bit 5)	U (bit 4)	Port	Result
`'rdl-`	1	0	1	1	$1B5	Right, Down, Left
`'rdlu`	1	0	1	0	$1A5	Right, Down, Left, Up

Any node can communicate with its immediate neighbors through the appropriate (RDLU) port. If a node wishes to read from or write to its neighbor, it puts an appropriate address in Register A or B to read or write, or in P to execute.

As a simple example, here is how nodes in various parts of the array would put themselves to sleep pending commands from elsewhere.

Table 4.2 Commands to enter "sleep" mode

Node positions	"Sleep" command
Corners of array	`'rd-- jump`
Sides of array	`'rd-u jump`
Interior of array	`'rdlu jump`
Top or bottom of array	`'rdl- jump`

4.2 Port Execution

SEAforth supports a powerful inter-node communications strategy known as *port execution*. This strategy takes advantage of the fact that a port is one cell (18 bits) wide, and can contain up to four instructions. It works this way:

1. Node A decides it wants to send instructions to Node B, so it writes an instruction word to the port it shares with Node B; Node A sleeps until the data goes across the port and is accepted by Node B.
2. Node B must decide on its own that it wants to execute from Node A, which it does by performing a jump to the port it shares with A.

If Node B jumps to the port before A does its write, then Node B will sleep until the instructions arrive.

3. When the condition exists that Node A has performed the instruction write, and Node B has performed the jump, then Node B will execute the instruction word.

A port address is just a location in address space. You can read or write it or jump to it, just as you would a memory location. However, the "contents" of that location are not passive data or machine instructions, as you'd find in memory, but are dependent on the action of the neighbor on the other side of the port.

Jumping to a port means relinquishing complete control to the neighbor on the other side of the port. A port jump is the universal communications protocol, because the protocol itself becomes specified by what passes through the port.

When Register P is pointing to a port, the usual auto-increment is suppressed, so that subsequent instruction fetches will use the same port address. Additionally, instructions that would normally increment the Register P (such as **`@p+`**) will have the increment operation suppressed. While in this state, a node executes everything that is sent to the port it is fetching from. This state can be exited by sending a branch instruction in the stream, such as a **`jump`**, a **`call`** or a return. This means that a node can be made to execute code that occupies *none* of its RAM or ROM.

4.3 Building Program Loading Streams

Send feedback

A process has been developed for sending a stream of compiled object code to various nodes of a SEAforth processor by using the processor's port execution facility. The stream will enter through an I/O node, and then be sent through ports to other nodes. Using this facility we can send programs to the RAM of any node or combination of nodes, and also initialize the stacks and registers of nodes so that the program we send to the RAM does not have to contain initialization code to perform these operations.

An example of a stream is given in the sample application Listing 2.3. This information is primarily intended as a tutorial to describe possible application strategies.

4.3.1 Definition of Terms

Following are some terms that are used primarily in describing the stream loading function.

I/O Node: A node connected to external pins that can perform I/O functions such as serial I/O and SPI. These are always on the outside of the chip. Consult your chip's data sheet for details.

Root Node: The I/O node into which the stream is inserted.

Stream Path: The order in which the stream passes through nodes. The first node in the Stream Path is the *root node*.

Port Execution: A SEAforth node can point Register P to the address of a port by executing a branch to that address. When this register is pointed at a port, the next instruction fetch will cause the node to sleep pending the arrival of data on the port. When the data arrives, it will be placed into IW (the Instruction Word), and executed just as if it came from RAM or ROM.

Multiport execution: The addresses of ports are encoded in such a way that one address can contain bits that specify as many as four ports. A multiport address is an address in which more than one port address bit is active. Multiport execution occurs when the a node is performing port execution and the address in Register P is a multiport address. It is required that only one neighbor node send code to a node that is performing multiport execution. The purpose of multiport execution is to allow a node to accept work from any direction, but only one direction at a time.

Port pump: a process in which a node executes a loop that reads data from one port and/or sends data to another port. There are several kinds of port pumps that may differ in their form and purpose. If normal branching or **`next`** commands are used, then the pump must reside in RAM or ROM. If **`unext`** is used for the loop, and especially if the loop instruction is executed from within a port, then no assistance from RAM or ROM is required. This is the primary usage of the term "port pump." The port executing a port pump has the useful property that Register P can be used to address at least one (and possibly both) of the directions. If Register P is used for both directions it is called a *multiport address port pump*. This pump uses the same address for the read address and the write address, and so is a more efficient use of node resources. However, it requires careful coordination so that the input direction is active during the reads and the output direction is active during the writes.

Domino awakening: A method of starting all the nodes after their initialization by sending a wake up signal that gets passed from node to node. After nodes are initialized by the stream loader, they are put to sleep until the signal awakens them, preventing program code from interfering with the loading and initialization of other nodes.

Domino path: The order in which nodes are awakened. This is not necessarily the same as the stream path. However, as it passes through a given node, the domino path must include that port that was the entry port for the stream path for that node.

Pinball: the word that is sent from node to node, following the domino path, to cause the various nodes to awaken. By convention, this is a **`RETURN`** instruction.

Current node: the node that consumes a stream via instruction words or stores it more permanently into RAM or into a stack or an address register within that node. When the stream is in motion (and before the pinball is released) there is always one and only one current node. While setting up for a pump, or initializing registers, or configuring the domino path, a single node is current. If a node is running a two-port pump using **`unext`**, it is no longer considered a current node.

4.3.2 Starting the Stream

Every node in the Stream Path begins in one of two states: either waiting at a multiport fetch or executing a multiport branch. In both cases the multiport address will include the port through which the stream will enter. This is the normal reset condition of the chip.

The stream is first delivered to one of the nodes with I/O capability. This will be the root node for this stream. The I/O nodes expect to be passed three words of information:

1. Execution Address
2. Load Address
3. Count

In the case of the port loader, the load address will be the address of the port that connects the root node to the next node in the stream path, unless the code is for the boot node. For example, if we use an S40C18 device that has an SPI node at Node 32, then the DOWN port of Node 32 (address $115) will connect to Node 22. In this example, Node 32 will pass the stream to its DOWN port, so the stream will begin execution in Node 22. Refer to to see the relationship between Nodes 32 and 22.

4.3.3 Nesting

Send feedback

Continuing with this example, the content of the stream as it enters Node 22 will be instructions that will cause Node 22 to send most of the stream on to the next node in the stream path. Bearing in mind that the node we are entering will be doing either a multiport fetch or a multiport jump, we must wake it in a way that works for both cases. Therefore the first action of a nest is to send two executable words in rapid succession:

- a **`call`** to the port we are using to enter the node, and
- four instances of **`.`** (**`nop`**).

The effect of the **`call`** must be considered from the point of view of the multiport fetch or the multiport jump. If the node is waiting in a fetch, then the **`call`** will wake the node, but the **`call`** instruction itself will be treated as data and dropped, because the node drops the data that awakens it. On wake up, the node will notice which direction the next data comes from and make a **`call`** to that port, thus yielding the same result: a **`call`** to the port that is sending the stream.

If the node is performing a multiport jump instead of waiting in a fetch, then the **`call`** will be executed, and the node will have its Register P pointed at the port.

The **`call`** to the port through which we are entering may appear redundant at first. However, it serves two purposes:

- It makes sure that while the stream is entering the node only the port we want to use is reading (turning off the effect of a multiport jump).
- Also, the **`call`** will cause the address the node was executing to be placed on the return stack.

Therefore, if R is not changed during initialization, this node will go back to its multiport jump when the stream loading process is done. If the node was waiting, it will return to that state at the end of stream loading if we do not initialize R to point to application code.

In our example, after the **`call`** has focused the attention of Node 22 to its Up port, Node 22 will be told to fetch a literal value using the P register as a pointer, thus allowing the next word in the stream to be data. This data item will appear on Node 22's data stack. Node 22 will then be told to use the **`a!`** instruction to place this value in the A register. This process can be used to set Node 22's A register to point to the next node in the stream path, so that a loop using **`@p+ !a+`** will read data from the

upstream (source) side of the stream path, and send the stream to the downstream side. By carefully calculating the lengths of the stream data segments we can arrange it so that each node will execute commands long enough to enter a port pump, and then send data downstream until all the downstream ports have been fed. Finally, more commands will arrive to be executed, and these commands will initialize the node's RAM and registers.

4.3.4 The Domino Awakening

Once all of the programs have been delivered to nodes and the registers have been initialized, we want each node to begin performing its appointed task. However, the performance of that task is likely to involve using ports to communicate with it neighbors. Therefore we do not want the node to begin until we are sure that all of the nodes have been given their respective tasks, and are also waking up to start the application. Therefore there are two requirements:

1. Put each node to sleep after initializing it.
2. Wake up all the nodes at (relatively) the same time, without interfering with the initialization we have performed for those nodes.

The domino awakening process is designed so that a given node can wake up more than one neighbor node, allowing a rapid spread of the wake-up signal.

To meet these requirements, we put nodes to sleep as we finish initializing them by executing a **`call`** to a multiport address. This address must include the address of each port to which we want to send the pinball, and also the address of the port from which the node was initialized. Then we send a word that does a fetch on that multiport address. This will cause the node to sleep pending the arrival of data on one of the specified ports. We are not going to send any more data to that node until we want the node to wake up. When the pinball eventually arrives, the instruction word that does the fetch will do a subsequent store to the next node or nodes we want to wake up. Because this instruction word sleeps until the wake-up data arrives, then passes the wake-up data to the next node, then enters the current node's application, we call this process the *domino awakening*.

4.3.5 Summary of Steps

Send feedback

A domino is a sequence of two instruction words. The first word causes the node to focus its attention on the domino path (i.e., jump to a multiport address that consists of all the ports in the domino path with respect to this node).

The second word contains one of the following sequences:

1. **`@p+ !p+`** (normal domino)
2. **`@p+ !p+ ;`** (next-to-last domino)
3. **`@p+ drop ;`** (last domino)

The **`@p+`** will cause the node to wait for a pinball to come to it on the domino path. The appearance of the pinball will satisfy the read caused by the **`@p+`** against the multiport jump's destination address, and the remainder of the domino will be executed (usually **`!p+`**).

The **`!p+`** will cause the pinball to be sent to all the ports included in the domino path for the affected node, thus a multiport write will occur. This write will send the pinball to those nodes that are "downstream" in the domino path, thereby waking them.

The multiport write will also send the pinball back to the node that awakened the current node. Since that node will still have its Register P focused on the domino path, the pinball will be executed. Since the pinball is a return instruction, the node that receives the reflected pinball will execute the return instruction and go to the address specified in the R register. This address will either be the address specified as the start address, or if no start address has been specified, it will be the address of what the node was doing when the stream first arrived, i.e., a multiport fetch or a multiport branch. It is important to note that the acceptance of the reflected pinball causes the write to that port to be completed. If we did not use the pinball as the return command, then the node sending the pinball would have an unsatisfied write pending in the upstream direction of the domino.

In the case of the final node in a domino chain, there is no node to which the pinball must be sent, while there is often a direction to which the pinball must not be sent. Therefore, there is no **`!p+`** in this node's domino instruction. Instead, the last domino (specified by the word **`edomino`** in the program) will be:

```
. @p+ drop ;
```

Note two differences: the pinball is dropped because it is not needed any more, and there is a **;** at the end. This **;** exists because there is no downstream node to reflect the pinball back for the purpose of sending the end node to its code.

There is one more special case. The second to the last domino in the chain (the penultimate domino) will *not* receive a reflected pinball, because the last domino does not reflect it with a **!p+**. Therefore the penultimate domino (specified by the word **pdomino** in the program) will be

```
. @p+ !p+ ;
```

Note that the normal domino word (**. . @p+ !p+**) begins with two NOPs. This is so that after the pinball is sent on using **!p+** the node that sent the pinball downstream will immediately be looking for a new instruction and therefore it will see the reflected pinball coming to it via the multiport write that the downstream node performs. If the sending node does not pay attention to its ports immediately, the reflected pinball may not be seen, because the write performed by the downstream node will be satisfied by the node or nodes downstream from it.

4.3.6 Domino Awakening Example

Send feedback

Following is an example showing specific steps required to awaken a series of nodes:

1. Use VentureForth to compile code to simulated nodes.
2. Specify initial data stack contents, return stack contents, A and B register contents, and runtime start address, as shown in Listing 4.1(within the **{node ... node}** for a particular node).

Listing 4.1 Example of the setup for a domino awakening

```
8 org  here =p
1 $a3 $a4 $a5 $a6 $a7 $a8 7 >rtn
$1000 $2000 2 >stk
'r--- =a
'r--- =b
```

3. Use the simulator (Section) to test the code. The simulator will initialize registers and stacks as specified above.
4. Specify a load order for a stream.

The stream compiler will create a stream suitable for loading through port execution.

Listing 4.2 Example of a stream (for an S40C18)

```
10 >root
20 >node 21 >node 11 >node 12 >node 13 >node 14 >node 15 >node 16 >node
<init <init <init <init <init <init <init <init <init <init
```

The stream compiler will:

- examine the RAM content of each node, and include in the stream instructions to load only those segments of RAM that have had code or data compiled into them.
- include instructions to initialize the stacks as specified.
- include instructions to initialize the A and B registers as specified.
- include instructions to initialize R so that the node will begin executing at the specified address.
- include the instruction for the node to start its program or wait for a domino to awaken it.

4.4 Boot Examples

Send feedback

The example program **`3boot.vf`** demonstrates three different ways to boot code into an S40C18 evaluation board:

- Asynchronous serial boot code at Node 33
- Synchronous serial boot code at Node 19
- SPI boot code at Node 32

Each of the other nodes has the same code compiled for it, which simply runs an endless loop keeping the node awake and noticeably *red* in the simulator.

VentureForth has buffer space for RAM and ROM for each node in the chip. It also has four buffers for "external nodes." These buffers, which reside in the host's memory, are where bootstreams are compiled. The current external buffer can be selected by the word `<n>` **`:xnode`**, where `<n>` is 0, 1, 2, or 3 to denote which buffer will become active.

4.4.1 An Asynchronous Serial Stream

Send feedback

The construction of an asynchronous serial stream for Node 33 (shown in Listing 4.3) begins:

```
0 :xnode 33 >root
```

which means that the following bootstream will be compiled into external buffer 0. The root node, the one with the asynchronous boot code in ROM, will be Node 33.

Listing 4.3 Example of an asynchronous boot stream

```
\ Compile a boot stream with Node 33 as the point of entry.
0 :xnode 33 >root
     34 35 25 15 05  5 >branch
         4 <branch      \ Initialize Nodes 34, 35, 25, 15, and 05.
     24 14 04  3 >branch
         4 <branch      \ Initialize Nodes 24, 14 and 04.
     23 13 03  3 >branch
         4 <branch      \ Initialize Nodes 23, 13 and 03.
900000 4 test-Serial    \ Connect the testbed, port 4 baud 900k.
```

The next line says:

```
34 35 25 15 05  5 >branch
```

This specifies a branch of three nodes in the stream starting at the root, Node 33, going to 34, 35, 25, 15, and 05. The following line says:

```
4 <branch
```

which causes the streamloader to initialize the four nodes in that branch while coming back to the root, Node 33.

The following lines construct two more branches, ending with the phrase:

```
4 <branch
```

This tells the compiler to initialize the last in the branch, this time including the root node. The root node is initialized last because it also triggers the domino that causes each node in the stream to begin executing as nearly simultaneously as possible. That's the end of the stream compilation for this boot node.

Since this is run in the simulator, a testbed is required to simulate the actions of the outside asynchronous device that's providing the data. Here's the line that initializes the asynchronous serial testbed:

```
900000 4 test-Serial   \ Connect the testbed, port 4 baud 900k
```

The first number is the baud rate and 900000 is the fastest that the SEAforth chips can keep up with. The second number is the internal port number that the simulator is to use. Ports are numbered 1, 2, 3, and 4 for Right, Down, Left, and Up: **rdlu**. In this case number 4, the Up port, is being used by the simulator.

Port numbers 1 and 2, Right and Down, are never used for this purpose, as they always have a neighbor and are never on the edge of the chip.

4.4.2 A Synchronous Serial Stream

The synchronous stream loader uses Node 19 as the root, and External Node 01. This part of the example defines a path similar to the asynchronous serial path, as shown in Listing 4.4.

A synchronous serial testbed is required to simulate the actions of the outside synchronous serial master that controls the clock for this bootload. Here's the line that initializes the testbed:

```
4 36 360  test-Sync  \ Connect the testbed.
```

Listing 4.4 Example of a synchronous boot stream

```
\ Compile a boot stream with Node 19 as the point of entry.
1 :xnode 19 >root
     29 39 38 37 36  5 >branch
        4 <branch      \ Initialize nodes 39, 38, 37, and 36.
     28 27 26  3 >branch
        4 <branch      \ Initialize nodes 28, 27, 26, and 29.
     18 17 16 3 >branch
        3 >branch      \ Initialize nodes 18, 17, and 16.
     09 08 07 06 4 >branch
        5 <branch      \ Initialize nodes 6-9 plus the root 19.
4 36 360 test-Sync     \ Connect the testbed.
```

The first number is the internal port number, where 1, 2, 3, and 4 are Right, Down, Left, and Up. Note the order, **rdlu** (see Section 4.1). In this case number 4, the Up port, is the port of entry. The second is the number of simulation clocks between input bits. If you go too fast, the boot node

may be able to keep up, but when the nodes in warm are awoken they take extra time and if they are too close to the root the delay can back up to the root and cause it to stop long enough to miss a clock, hence two bits are dropped. The last number is how many clocks to hold off before sending in the first bit.

4.4.3 A Flash Memory Stream Using SPI

Send feedback

For the preceding two testbeds you can have more than one of either, so long as each one is given its own external node space via **`:xnode`**. But the SPI testbed uses slightly different technique. The SPI uses a one-time buffer and pointers in an area that is set up by **`StreamFlash`**, which copies and converts the external node data. Because the SPI node is a protocol master, it sets the timing values and the testbed must respond. Therefore no other input is required by the test-SPI setup word.

Listing 4.5 Example of an SPI/flash memory stream

```
\ Compile a boot stream with Node 05 as the point of entry.
2 :xnode 32 >root
     31 30 20 10 0  5 >branch
         4 <branch   \ Initialize Nodes 00, 10, 20, 30, and 31.
     21 11 01  3 >branch
         2 <branch   \ Initialize Nodes 01 and 11.
     22 12 02  3 >branch
         5 <branch   \ Initialize Nodes 02, 12, 22, 21, and 32.

StreamFlash  test-SPI \ Connect the testbed.
\ patch count for faster clocking in simulation
32 {node  0. !ok. 2!  ' spi-boot 3 - org
   here 1 scrub  0 ,  node}
```

Note that the testbeds are not general simulations of external hardware but are designed knowing what the ROM will require and are only smart enough to keep the ROM happy when starting clean from reset.

4.4.4 Summary of Stream Loader Commands

Send feedback

The most commonly-used commands for constructing and managing boot streams are given in the glossary below. Please note the highest-level word **`nodePath`** is described in Section 2.2.

Glossary

[x] (—) H *bracket-x*
Make the current external node the current node. See **:xnode** below.

[x'] (—) H *bracket-x-prime*
The external node next in numeric sequence after **[x]** becomes the current node. See **:xnode** below.

:xnode (n —) H *colon-x-node*
Select a buffer on the host PC to which the stream will be compiled and make it the current node by executing **[x]**. In the compiler, it is an extra node, and as such has many of the properties of nodes such as a separate location counter. It has much more capacity than the 64 words of RAM in the internal nodes. Used if more that one stream is to be compiled for this project. Otherwise **0 :xnode** is used by default.

>root (n —) H *to-root*
Specify Node *n* as the port of entry for a boot stream being compiled. This must be a node that actually contains boot code in its ROM, and for which you have a physical connection in hardware.

>branch (n_1 n_2 ... n_n n —) H *forward-branch*
A list of connected nodes, followed by the number of nodes in the branch. Used to make a branching path instead of a snakelike single path.

<branch (n —) H *return-branch*
Return *n* nodes back on the branch just specified. The nodes in this path will be initialized at this point in the stream loader. If the root node is the last node initialized by this command, then the stream is finished.

>node (n —) H *to-node*
Connect a port pump to this node, and add it to the current path. This is a lower-level component of **>branch**.

<node (—) H *from-node*
Backs up one node in the stream, but does not initialize the current node, which is unchanged.

<init (—) H *from-init*
Initializes the current node before executing **<node** to back up one place. This is a lower-level component of **<branch**.

Appendix A S24 ROM and Library Functions

The SEAforth evaluation kits include not only the compiler and sample applications, but also user-accessible code in the ROMs of the various nodes and some library routines provided in source. These differ in the different versions of the SEAforth architecture. Those available in the S24 are documented in this chapter.

A.1 S24 Overview

Send feedback

The SEAforth S24 was the first production part in the SEAforth family. Figure A.1 shows its configuration.

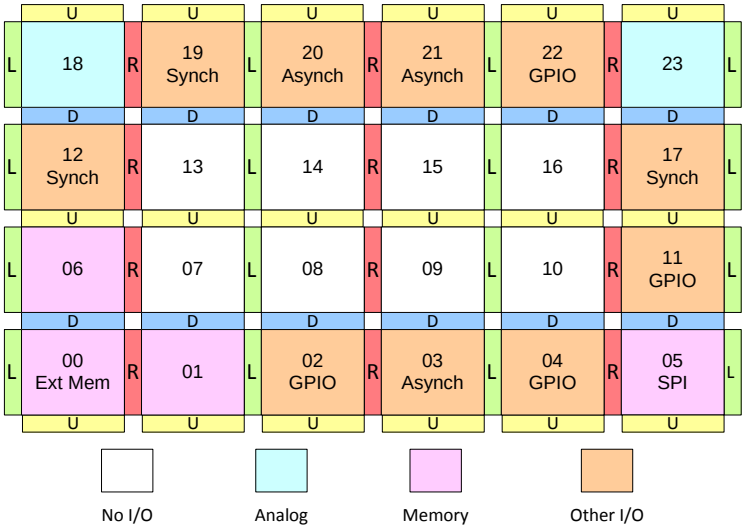


Figure A.1 Map of S24 nodes, showing I/O capabilities

The S24 parts do not support Extended Mode arithmetic and have a somewhat limited set of ROM and Library functions.

A.2 Arithmetic Functions

Send feedback

Generic arithmetic primitives for the SEAforth processors are described in Section 3.1.5. This section describes some higher-level functions available in most nodes of the S24.

A.2.1 Multiply and MAC

Send feedback

The ROM in most nodes contains a signed/unsigned fixed-point multiplication routine. It has several entry points, described in the glossary.

Glossary

su* (n u — n' n'*u) S24 *s-u-star*
Multiplies the signed number in S by the unsigned fixed-point multiplier (where 1.0 is represented by $40000) in T. Since $u < 1.0$, the absolute value of a product is always less than the absolute value of n. The binary point of the product is the same as in the multiplicand n.

Only the 16 most significant bits of n are used in calculation. Two least-significant bits of multiplicand n are just truncated. Therefore, to achieve better accuracy, it's advisable to shift the multiplicand two bits left before calling **su***, and then shift the product $n*u$ in T two bits right.

mac18 (n_1 n_2 u' u' — n_1 n_2+n_1*u') S24 *mac-18*
Performs a "multiply/accumulate" (MAC) calculation in which n_1 is a signed multiplicand, n_2 is the accumulate component, and u' is a normalized, unsigned fixed-point multiplier (as in **su*** above), whose two least-significant bits are zero.

There are several variants of the MAC function, which differ in the number of bits in the multiplier (e.g. 18 in **mac18**). The smaller the number, the faster the calculation. The following entry points are available:

```
mac2 mac4 mac6 mac8 mac10 mac12 mac14 mac16
```

The accumulater parameter n_2 is signed fixed-point, with the same scale as u'.

The multiplier u' must be normalized, that is, its two least significant bits must be 0. This could be achieved by masking them out with **$3FFFC and** or with **2/ 2/ 2* 2*** or with **2* 2***

A.2.2 Alternative Entry Points on Synchronous Serial Nodes

Send feedback

The synchronous serial nodes 12, 17, 19 do not have enough space in ROM to for the fast (but bulky) **su*** implementation. Therefore, these nodes have a smaller but somewhat slower version of **su*** with no MAC entry points.

A.3 SPI I/O Support

Send feedback

The SPI boot routine for the S24 is located in node 05. This node does not contain the code sequence used in the other boot nodes to ensure that their inputs start out in weak pull-down mode and are stable before being tested. Serial Data In is on pin 17, Data Out is pin 3, Clock is on pin 1 and Select is on 5. Following power-up, these pins will be driving a weak pull-down. If connected to an SPI slave device, these pins will remain at zero.

An SPI-controlled memory device is a slave device that receives an eight-bit read command and a 24-bit address parameter, both sent high bits first. During execution of the read command, the memory device emits bytes, one bit at a time, high bits first. This output is completely synchronous and can be paused and resumed, or in fact stopped at any time and between any pair of bits. This means it can be treated as a bit serial memory of arbitrary depth. The only real nod to bytes within the part is that the address parameter refers to byte locations and command information is organized as bytes.

The implementation of stream structure in an SPI memory device is simply a contiguous packing of 18-bit words, high bits first. Because the SPI memory is a simple slave device, the SPI boot routine must control the signal lines and produce its own read command and start address and pass them to the memory.

The SPI interface serves as a protocol master. The memory device to which it interfaces has a limited but well-defined set of functions that will nevertheless differ between certain devices. Because of variation in SPI memory designs, there is no standard specification for writing such

memories. Even if there were room in ROM for a write function, it would not be prudent to put one there. Some applications may need to provide a write function via RAM loaded routines. Insofar as some portions of the ROM driver may serve a useful function for this purpose, they will be included in this API specification.

This driver uses a set of constants, shown in Table A.1, to specify wave form components. The names are chosen to represent three traces in a vertical wave form. The three columns, left to right are Select, Data-out and Clock and correspond to pins 5, 3, and 1 in that order. The character **+** is used in the name to represent a high output and **-** to represent a low out. Using this nomenclature, it is possible to visualize the generated waveform by simply listing the literal names in a column in the order they are invoked in the driver.

Table A.1 Constants used to specify wave form inputs

Name	Pin 5 Select	Pin 3 Data-out	Pin 1 Clock
---	0	0	0
--+	0	0	1
+--	1	0	0
+-+	1	0	1

The lowest level output word in this specification is the wave form composer **half**, which stores the wave-form constant in the IOCS and waits for a specified delay period (one half bit time).

The **ibit** routine is the lowest level input function. It samples the current value of pin 17 and shifts this value into the low order bit of the top stack item. After 18 such inputs the top item will have been completely replaced, and the high order bit will be the first bit sampled. The second stack item is neither changed nor examined. It appears in the stack diagram because it is expected that this function will be used together with output words where this item is reserved for the delay count. The top stack item is actually returned as a ones-complemented value. It is common practice when designing code interfaces for this processor, to shift responsibility for some function elements between subroutine and caller to improve overall size and performance. If the instruction after a **call** is another **call** or a branch, it can easily absorb an extra opcode in its slot 0, whereas the called function may not be able to absorb this opcode without much greater cost. In the present case it would cost two extra words to return a non-inverted value from both exit points, whereas the **not** in front of **next** is completely free.

There are three words that are used to create specific types of bit-wide output wave forms, all of which take the same two stack arguments. The value in S is the delay count which is used but preserved, and T contains an accumulator (which will be preserved) that may hold either an input or output in the process of being read or written. The word **`select`** produces the chip select sequence that is used to start or stop each command sequence. The word **`rbit`** generates the input clock, at the end of which a sample can be taken. It also is used to generate an output zero bit, and is used together with the main body of **`obit`** to generate either a zero or one output based upon the high bit of the top stack item. The output accumulator is sampled but not shifted.

The word **`8obits`** outputs a byte from the high bits of the top stack item and leaves it shifted left by eight. The delay count in the second stack item determines one half the bit delay. The word **`18ibits`** takes only the half-bit delay count as an input and returns above it a full 18-bit value received, high bits first.

The highest level word in the driver API is **`spi-boot`**. It takes a half-bit delay value on the stack beneath the first of two command values; a second command value is passed in on top of the return stack. A return address is not needed or used because interpretation of the input stream will determine the exit address. The command values are each a pair of bytes, packed into the high 16 bits of each 18-bit item. When this function is entered from the ROM cold start, the high byte of the first parameter will be the standard read command $03. The next byte together with the the two bytes from the return stack compose a 24-bit read starting address. This address is set to zero by the cold start. Once the read command is issued any number of contiguous bits may be read from the memory.

Program control then falls into the **`spi-exec`** function that reads and executes a standard sub-stream three-word header. This function also represents the sub-stream concatenation address that is used as the execution address for all but the last sub-stream. On entry and exit to this function, the delay value is preserved beneath a don't-care value. The mode of the memory device must still be the execution of a continuous read command. Note that the byte alignment at the beginning and end of each sub-stream is not material. Each one begins immediately upon the next bit after the previous one ends.

The word **`spi-copy`** will transfer a sub-stream given that the three arguments that describe it have already been read, and the memory device is currently in read mode positioned at the starting bit of the substream. The execution address must be on top of the return stack. The transfer address must be in the A register and the transfer count is on top

of the stack above the half-bit delay count. Exit will be to the execution address with trash on top of the delay count.

Glossary

spi-boot (n cmd_1 R: cmd_2 — n x) A *s-p-i-boot*
Boots a stream from the SPI. See the text above for a description of the two command words.

spi-exec (n x —n x) A *s-p-i-exec*
Reads and executes a standard sub-stream three-word header. This function also represents the sub-stream concatenation address that is used as the execution address for all but the last sub-stream. The value *n* is the baud counter.

spi-copy (x n x —n) A *s-p-i-copy*
Transfers a sub-stream, given that the three arguments that describe it have already been read and the memory device is currently in read mode positioned at the starting bit of the sub-stream. The transfer address is in A. The baud counter is returned.

18ibits (n —n w) A *18-eye-bits*
Reads a full 18-bit word and returns it in T. The value *n* is the baud counter.

8obits (n w —n w') A *eight-oh-bits*
Outputs a byte from the high bits of *w* and leaves it shifted left by eight.

obit (n w —n w) A *oh-bit*
Generates either a zero or one output bit based upon the high bit of *w*.

rbit (n w —n w) A *r-bit*
Generates the input clock, at the end of which a sample can be taken. It also generates an output zero bit, and is used together with the main body of **obit** to generate either a zero or one output based on the high bit of the top stack item. The output accumulator is sampled but not shifted.

select (n w —n w) A *select*
Produces the chip select sequence used to start or stop each command sequence.

half (n_1 w n_2 — n_1 w) A *half*
Constructs a wave form given n_2, which is one of the constants listed in Table A.1, which is stored in the IOCS. The word *w* is preserved but not used. The parameter n_1 is a time constant representing one half bit cell. This delay is measured by a three-**nop unext** loop whose nominal units are around 6.5ns.

A.4 S24 Asynchronous I/O

Send feedback

The asynchronous serial routines to support bootstrap or data input are located in nodes 04, 20, and 21. All of these nodes include anti-latch-up power-up/reset sequences to insure that their inputs start up in weak pull-down mode and are stable long enough before being tested to ensure that the test for boot responsibility will be accurate. All input is half duplex and makes dedicated use of a single pin connected to bit 17 internally. The node's bit 1 pins are reserved for an output function not implemented in ROM. The output pins are left in weak pull-down state at startup.

The ROM code does not support single byte input. Instead, a special three-byte format is used, such that each byte triad contributes 18 data bits and begins with an extra-long start bit as well as a long baud measurement bit in the first byte. The asynchronous protocol is described in more detail in Section 5.2.2.

Most of thelabels defined by the serial driver are not intended as external API points because of the custom nature of the protocol supported. The primary API is via the word **`ser-exec`**, which takes a dummy input (i.e., its content is irrelevant) and returns the baud rate counter on the stack. It reads a standard boot stream three-word header packet and then loads the described sub-stream and branches to its entry point (with the baud counter on top of the stack). When used to read part of a boot stream that is a concatenation of several sub-streams **`ser-exec`** is used as the concatenation address contained in all but the last sub-stream.

Use of any other entry point to the serial driver implies that a custom protocol is being designed and must therefore be carefully co-ordinated with the outside agent that the SEAforth is communicating with.

The entry point **`ser-copy`** may be useful to read a sub-stream without a standard prefixed header. In this case the header information must be passed in by the caller. The data address is expected to be in the A register and the start address is just the return address in R at entry. The word count is passed in as the second of three values on the stack. The other two are ignored.

A single triad of bytes may be read using **`18ibits`**, which requires a dummy input value and returns a dummy value on top of the input word on top of the baud counter. Units smaller than three bytes can only be supported if the programmer takes responsibility for passing in an accurate baud counter. The interested reader is directed to the stack

comment for the entry point **byte** that is internal to **18ibits**. It may be found in the file **serial.vf** in the sub-directory appropriate to your chip version (e.g., **VentureForth/vf/c7Dr03/serial.vf**).

Glossary

ser-exec (x —n) A *ser-exec*
Reads a standard boot stream three-word header packet and then loads the described sub-stream and branches to its entry point (with the baud counter on top of the stack). The returned value *n* is the baud counter.

ser-copy (x n x —n) A *ser-copy*
Reads an *n*-wordsub-stream without a standard prefixed header. Data is stored starting at the address in A. The baud counter is returned.

18ibits (x n x —n) A *18-eye-bits*
Reads a single 18-bit word to the address in A, returning the baud counter.

A.5 S24 Synchronous I/O

Send feedback

The synchronous serial routines to support bootstrap or data input are located in nodes 12 17 and 19. All of these nodes include anti-latch-up power-up/reset sequences to ensure that their inputs start up in week pull-down mode and are stable long enough before being tested to insure that the test for boot responsibility will be accurate. The clock is on bit 17 and data pass on bit 1. The clock is always controlled externally but the data direction depends upon the protocol. The ROM routines only support the input mode. Details of the stream loader protocol are given in Section 5.2.3.

The unit of input or output, controlled by the external clock, are 18-bit words. The clock is bi-phase, in that data are transferred on each edge. Consequently, the clock will always return to zero. The start of input is signaled by the first low to high transition of the clock. No synchronization of flow occurs between words. There is a synchronization protocol that is specified for transitions between input and output. Stream data must be delivered within one clock period or risk losing two bit edges.

Only three of the internal labels defined by the serial driver are intended as external API points. The primary API is via the word **ser-exec**, which takes and returns a dummy value. It reads a standard boot stream three-word header packet, then loads the described sub-stream and branches

to its entry point (with the output dummy value in T). When used to read part of a boot stream that is a concatenation of several sub-streams, **ser-exec** is used as the concatenation address contained in all but the last sub-stream.

Use of any other entry point to the serial driver implies that a custom protocol is being designed and must therefore be carefully co-ordinated with the outside agent that the SEAforth is communicating with. The entry point **ser-copy** may be useful to read a sub-stream without a standard prefixed header. In this case, the header information must be passed in by the caller. The data address is expected to be in the A register, and the start address is just the return address in R at entry. The word count is passed on the stack on top of a trash value.

A single word may be read using **sget**, which takes no input value and returns the input word. Units smaller than a word are not supported by this specification.

Glossary

ser-exec (x —x) A *ser-exec*
Reads a standard boot stream three-word header packet and then loads the described sub-stream and branches to its entry point (with the baud counter on top of the stack).

ser-copy (x n —x) A *ser-copy*
Reads an *n*-word sub-stream without a standard prefixed header. Data is stored starting at the address in A. The baud counter is returned.

sget (— x) A *s-get*
Reads a single 18-bit word to the stack.

Appendix B S40C18 ROM and Library Functions

The SEAforth evaluation kits include not only the compiler and sample applications, but also user-accessible code in the ROMs of the various nodes and some library routines provided in source. These differ in the different versions of the SEAforth architecture. The S40C18 parts have significantly enhanced math and I/O functions, which are documented in this chapter.

B.1 Arithmetic Functions

Send feedback

Generic arithmetic primitives for the SEAforth processors are described in Section 3.1.5. This section describes some higher-level functions available in most nodes of the S40C18.

Note that in dealing with double-length numbers in this section common practice is to have the most-significant part in T and the least-significant part in A. If both parts are on the stack, the high-order part will be in the lower stack position (e.g. in S with the low-order part in T). *This is the opposite of the convention in Standard Forth*, but saves instructions in the SEAforth parts.

B.1.1 Arithmetic

Send feedback

The ROM in most nodes contains several extended-mode arithmetic operators, which are documented in this section.

Glossary

***** (n_1 n_2 — n_1 d_1, A:d_2) S40 *star*

Multiplies n_1 and n_2, giving a 35-bit product whose high-order word d_1 is in T and the low-order word d_2 is in Register A.

d_1 could be interpreted as the signed fixed result of a signed fixed

multiply, where n_1 or n_2 is a signed fixed-point fraction (where the value 1 is represented by $20000).

The low-order word of the double-length product is left in A. If you want it, you can get it by using **a@**. However, you should know the high bit of the value in A is inverted, so you need to exclusive-or A with $20000. The high word in T is shifted left one place, so it needs to be shifted right with **2/**.

negate (n — -n) S40 *negate*
Returns the 2's complement of a number. Library function.

u2/ (n — n') S40 *u-two-slash*
Performs an unsigned right shift one bit. Library function.

u/mod (d_1 d_2 +n — nr nq) S40 *u-slash-mod*

Divides a positive double integer by a positive single integer, where:

- d_1 is the high 17 bits of a 35-bit positive numerator
- d_2 is the low 18 bits of a 35-bit positive numerator
- *+n* is a 17-bit positive denominator.
- *nr* is the remainder
- *nq* is the quotient

u/mod clears the carry bit.

~u/mod (d_1 d_2 -n — nr nq) S40 *tilde-u-slash-mod*
This is a shorter synonym for **u/mod**, but you have to negate the denominator. It divides a positive double integer by a negated *positive* single integer, where:

- d_1 is the high 17 bits of 35 bits positive numerator
- d_2 is the low 18 bits of 35 bits positive numerator
- *-n* is a negated 17-bit positive denominator.
- *nr* is the remainder
- *nq* is the quotient

This word is useful where *-n* is a constant, as the negation takes time. **~u/mod** clears the carry bit.

-u/mod (d_1 d_2 -n — nr nq) S40 *minus-u-slash-mod*
Same as **~u/mod**, except it does *not* clear the carry bit.

B.1.2 Mathematics Library

The S40C18 ROM Bios includes a library of advanced math functions, which are described in this section.

interp (i m s —v) S40 *interp*

Provides a table linear interpolation function. The number of intervals or segments in the lookup table must be a power of 2, 2^n, and so the number of table entries is 2^n+1. L is the number of bits in the meaningful part of *i*. Then the bit length of each segment in the table is (L - n), and the segment parameter *s* is one less than this, or (L - n - 1). The value *m* is a bit mask, which could be calculated from *s* by:

```
1 + -1 SWAP LSHIFT INVERT
```

We assume that the size of interpolation table is fixed, so *s* and *m* are known at compile time and should be provided as explicit parameters rather than being calculated at run-time. The table itself is expected to start at location 0. The output *v* is given by the table value corresponding to the start of the closest segment less than *i* plus the linearly interpolated fractional distance into that segment.

Example:

```
0 org 1000 , 3000 , 4000 , 7000 , ( and 61 more values not shown)
2987 # $FFF # 11 # interp
```

In this example, *n* is 6 ($2^6 = 64$), with only the first few values of the table shown for simplicity. L is 18, so each segment is 12 bits long, or $F00. The input *i*, $2987, is split by *s* and *m* into two parts: 2 and $987. The 2 provides an index into the table, picking up the third table value (which is the starting value of the third segment), and the $987 calculates the fractional distance into the third segment. Thus the output *v* is:

t[2]+(t[3]-t[2])*$987/$F00 = 4000+(7000-4000)*$987/$F00

rotate (x y w — x' y') S40 *rotate*

Performs a fast approximation of rotating the 2D vector *x*,*y* by the small angle *w*. Instead of calculating (*x'*,*y'*) as:

x' = x * cos(w) - y * sin(w)
y' = y * cos(w) + x * sin(w)

it uses:

x' = x - y * w
y' = y + x'* w

The angle *w* is normalized such that PI = $20000. Library function.

triangle (x — y) S40 *triangle*
Computes a periodic triangle function (waveform) in *x* to give *y*. Could be used as part of a cosine implementation, and also as is. The input parameter *x* is an angle, normalized such that PI = $20000. The output *y* is $10000 (maximum) at *x* = 0, $-10000 (minimum) at *x* = $20000, and zero at *x* = $10000 and *x* = $30000.

taps: (y x c R: addr — y' x') S40 *taps*
Used *inside a colon definition* to construct Finite Impulse Response filters (FIR), Infinite Impulse Response filters (IIR), and parallel (multicore) FIRs. The FIR filter can be done in a single node, or it can be done in a pipeline of nodes, with more taps/coefficients in each node in the pipeline. Since **taps:** is entered via a **call**, the address *addr* of the table will be in Register R. When **taps:** finishes executing, it will return to the point from which the *definition containing it* (e.g., **fir-kernel**) was called.

Arguments to **taps:** are:

- *y* - partially convolved output sample
- *x* - input sample
- *y'* - output sample
- *x'* - oldest input sample from the "history."
- *addr* - address of filter coefficients array (signed fixed-point numbers, where 1 is represented by $20000), interleaved with the history array whose initial values are zeroes.
- c - size of filter coefficients array -1

Example 1, single core FIR:

```
: fir-kernel 4 # taps: a0 , 0 , a1 , 0 , a2 , 0 , a3 , 0 , a4 , 0 ,
: fir ( B:in -- out ) dup dup xor @b fir-kernel drop ;
```

Example 2, multicore FIR:

```
: long-fir-start dup dup xor @b fir-kernel !b !b ;
: long-fir-mid @b push @b pop fir-kernel !b !b ;
: long-fir-end @b push @b pop fir-kernel drop ;
```

Example 3, IIR:

```
: lp.15.2p 4 # taps: $4038 , 0 , $8070 , 0 , $4038 , here 0 ,
      $19D39 , 0 , $361E7 , 0 , ( here ) ,
: iir ( n - n') push dup dup xor pop lp.15.2p drop dup !a ;
```

This implements a low-pass, two pole Chebyshev filter with 0.5% ripple and a normalized cutoff frequency of 0.15. The non-zero coefficients in the **taps:** table are, in order, a0, a1, a2, b1, and b2. The additional address compiled by **taps:** provides the necessary recursion for an IIR filter.

B.2 General I/O Functions

Send feedback

Figure B.2 shows the I/O ports on an S40C18 chip. Words described in this section provide various I/O functions in the S40C18.

Glossary

bget (n — n x) S40 *b-get*

Returns one byte from an RS232 port using one start bit, eight data bits, and one stop bit. The eight data bits are packed into the unsigned eight least-significant bits of *x*. The parameter *n* is a delay in units of ~2.5ns, i.e., to receive the data at 31250 baud *n* should be 12800.

.

Figure B.2 Map of S40C18 nodes, showing I/O facilities

dac27 (m c p a w — m c p) S40 *dack-27*

Each use of **dac27** generates a triple pulse, with the following parts:

- 0 level amplitude, lasting *p-w*-1 time units
- *m* level amplitude, lasting *w* time units
- *a* level amplitude, lasting approximately 1 time unit.

p is the recurring pulse period, in units of 2.5 ns. For example, a *p* value of $208D results in a 48 KHz output rate.

Usually the *m* level amplitude is the maximum instantaneous value that the DAC can generate, which corresponds to an input to it of *m* = $1FF.

The digital value *u* to be converted is a 27-bit unsigned integer. Input values *w* and *a* are calculated from *u* as follows:

w = *u* / *m* (integer divide)

a = *u* mod *m*

Therefore *u* = *w*m* +*a*, and so graphically, the area under the *m* level subpulse is proportional to *w*m*, and the area under the *a* level subpulse (*a*1) represents the remainder.

In the case of *m* = $1FF, the 18 high order bits of *u* are *w*, and the nine low order bits are *a*.

To achieve the highest possible sample rate, p should be set to *u*(max)/*m* + 3 (the 0 and *m* subpulses are always at least one time unit long, and the *a* subpulse is slightly longer than one time unit.)

Preparing the *w* and *a* parameters for **dac27** involves the relatively slow **u/mod** operation, whose execution time depends on the numerator and denominator. Therefore, this should be done on another node and the results passed to the DAC node.

In cases where the resources to prepare the *w* and *a* parameters are not available, you should use **dac18**.

dac18 (m c p u — m c p) S40 *dack-18*

The *m* and *p* parameters are as described above for **dac27**, except that in **dac18**, *m* must be set to $100 (although the code assumes that *m* is a constant $100, it is a stack parameter to save memory).

The digital value to be converted, *u*, is an unsigned 18-bit integer.

Internally to **dac18**, the parameters *w* and *a*, described above in **dac27**, are calculated as:

$w = u/m$

$a = (u \bmod m) * c$

The coefficient c is a fixed point fraction $0 < c < 1.0$, where the input range is [0, $7F] with the value 1.0 represented by $80.

The reason for the c coefficient in **`dac18`** is that the width of the a subpulse is slightly longer than the unit widths of the 0 and m subpulses, and for best linearity of the DAC conversion, the a pulse amplitude should be adjusted. Parameter c is not needed in **`dac27`** because of the greater resolution in the input value u.

B.3 SPI I/O Support

Send feedback

The SPI boot routine for the S40C18 is located in node 32. Serial Data In is on pin 32-17, Data Out is pin 32-3, Clock is on pin 32-1 and Select is on 32-5. Following power-up, these pins will be driving a weak pull-down. If connected to an SPI slave device, these pins will remain at zero.

An SPI-controlled memory device is a slave device that receives an eight-bit read command and a 24-bit address parameter, both sent high bits first. During execution of the read command, the memory device emits bytes, one bit at a time, high bits first. This output is completely synchronous and can be paused and resumed, or in fact stopped at any time and between any pair of bits. This means it can be treated as a bit serial memory of arbitrary depth. The only real nod to bytes within the part is that the address parameter refers to byte locations and command information is organized as bytes.

The implementation of stream structure in an SPI memory device is simply a contiguous packing of 18-bit words, high bits first. Because the SPI memory is a simple slave device, the SPI boot routine must control the signal lines and produce its own read command and start address and pass them to the memory.

The SPI interface serves as a protocol master. The memory device to which it interfaces has a limited but well-defined set of functions that will nevertheless differ between certain devices. Because of variation in SPI memory designs, there is no standard specification for writing such memories. Even if there were room in ROM for a write function, it would not be prudent to put one there. Some applications may need to provide a write function via RAM loaded routines. Insofar as some portions of

the ROM driver may serve a useful function for this purpose, they will be included in this API specification.

This driver uses a set of constants, shown in Table B.2, to specify wave form components. The names are chosen to represent three traces in a vertical wave form. The three columns, left to right are Select, Data-out and Clock and correspond to pins 5, 3, and 1 in that order. The character **+** is used in the name to represent a high output and **-** to represent a low out. Using this nomenclature, it is possible to visualize the generated waveform by simply listing the literal names in a column in the order they are invoked in the driver.

Table B.2 Constants used to specify wave form inputs

Name	Pin 5 Select	Pin 3 Data-out	Pin 1 Clock
---	0	0	0
--+	0	0	1
+--	1	0	0
+-+	1	0	1

The lowest level output word in this specification is the wave form composer **`half`**, which stores the wave-form constant in the IOCS and waits for a specified delay period (one half bit time).

The **`ibit`** routine is the lowest level input function. It samples the current value of pin 17 and shifts this value into the low order bit of the top stack item. After 18 such inputs the top item will have been completely replaced, and the high order bit will be the first bit sampled. The second stack item is neither changed nor examined. It appears in the stack diagram because it is expected that this function will be used together with output words where this item is reserved for the delay count. The top stack item is actually returned as a ones-complemented value. It is common practice when designing code interfaces for this processor, to shift responsibility for some function elements between subroutine and caller to improve overall size and performance. If the instruction after a **`call`** is another **`call`** or a branch, it can easily absorb an extra opcode in its slot 0, whereas the called function may not be able to absorb this opcode without much greater cost. In the present case it would cost two extra words to return a non-inverted value from both exit points, whereas the **`not`** in front of **`next`** is completely free.

There are three words that are used to create specific types of bit-wide output wave forms, all of which take the same two stack arguments. The

value in S is the delay count which is used but preserved, and T contains an accumulator (which will be preserved) that may hold either an input or output in the process of being read or written. The word **`select`** produces the chip select sequence that is used to start or stop each command sequence. The word **`rbit`** generates the input clock, at the end of which a sample can be taken. It also is used to generate an output zero bit, and is used together with the main body of **`obit`** to generate either a zero or one output based upon the high bit of the top stack item. The output accumulator is sampled but not shifted.

The word **`8obits`** outputs a byte from the high bits of the top stack item and leaves it shifted left by eight. The delay count in the second stack item determines one half the bit delay. The word **`18ibits`** takes only the half-bit delay count as an input and returns above it a full 18-bit value received, high bits first.

The highest level word in the driver API is **`spi-boot`**. It takes a half-bit delay value on the stack beneath the first of two command values; a second command value is passed in on top of the return stack. A return address is not needed or used because interpretation of the input stream will determine the exit address. The command values are each a pair of bytes, packed into the high 16 bits of each 18-bit item. When this function is entered from the ROM cold start, the high byte of the first parameter will be the standard read command $03. The next byte together with the the two bytes from the return stack compose a 24-bit read starting address. This address is set to zero by the cold start. Once the read command is issued any number of contiguous bits may be read from the memory.

Program control then falls into the **`spi-exec`** function that reads and executes a standard sub-stream three-word header. This function also represents the sub-stream concatenation address that is used as the execution address for all but the last sub-stream. On entry and exit to this function, the delay value is preserved beneath a don't-care value. The mode of the memory device must still be the execution of a continuous read command. Note that the byte alignment at the beginning and end of each sub-stream is not material. Each one begins immediately upon the next bit after the previous one ends.

The word **`spi-copy`** will transfer a sub-stream given that the three arguments that describe it have already been read, and the memory device is currently in read mode positioned at the starting bit of the substream. The execution address must be on top of the return stack. The transfer address must be in the A register and the transfer count is on top of the stack above the half-bit delay count. Exit will be to the execution address with trash on top of the delay count.

Glossary

spi-boot (n cmd_1 R: cmd_2 — n x) A *s-p-i-boot*
Boots a stream from the SPI. See the text above for a description of the two command words.

spi-exec (n x —n x) A *s-p-i-exec*
Reads and executes a standard sub-stream three-word header. This function also represents the sub-stream concatenation address that is used as the execution address for all but the last sub-stream. The value *n* is the baud counter.

spi-copy (x n x —n) A *s-p-i-copy*
Transfers a sub-stream, given that the three arguments that describe it have already been read and the memory device is currently in read mode positioned at the starting bit of the sub-stream. The transfer address is in A. The baud counter is returned.

18ibits (n —n w) A *18-eye-bits*
Reads a full 18-bit word and returns it in T. The value *n* is the baud counter.

8obits (n w —n w') A *eight-oh-bits*
Outputs a byte from the high bits of *w* and leaves it shifted left by eight.

obit (n w —n w) A *oh-bit*
Generates either a zero or one output bit based upon the high bit of *w*.

rbit (n w —n w) A *r-bit*
Generates the input clock, at the end of which a sample can be taken. It also is used to generate an output zero bit, and is used together with the main body of **obit** to generate either a zero or one output based upon the high bit of the top stack item. The output accumulator is sampled but not shifted.

select (n w —n w) A *select*
Produces the chip select sequence used to start or stop each command sequence.

half (n_1 w n_2 — n_1 w) A *half*
Constructs a wave form given n_2, which is one of the constants listed in Table B.2, which is stored in the IOCS. The word *w* is preserved but not used. The parameter n_1 is a time constant representing one half bit cell. This delay is measured by a two-**nop unext** loop whose nominal units are around 4.9ns.

B.4 S40C18 Asynchronous I/O

Send feedback

The asynchronous serial routines to support bootstrap or data input are located in Nodes 10 and 19. All input is half duplex and makes dedicated use of a single pin connected to bit 17 internally. The node's bit 1 pins are reserved for an output function not implemented in ROM. The output pins are left in weak pull-down state at startup.

The ROM code does not support single byte input. Instead, a special three-byte format is used, such that each byte triad contributes 18 data bits and begins with an extra-long start bit as well as a long baud measurement bit in the first byte. The asynchronous protocol is described in more detail in Section 5.2.2.

Most of the internal labels defined by the serial driver are not intended as external API points because of the custom nature of the protocol supported. The primary API is via the word **`ser-exec`**, which takes a dummy input (i.e., its content is irrelevant) and returns the baud rate counter on the stack. It reads a standard boot stream three-word header packet and then loads the described sub-stream and branches to its entry point (with the baud counter on top of the stack). When used to read part of a boot stream that is a concatenation of several sub-streams **`ser-exec`** is used as the concatenation address contained in all but the last sub-stream.

Use of any other entry point to the serial driver implies that a custom protocol is being designed and must therefore be carefully coordinated with the outside agent that the SEAforth is communicating with.

The entry point **`ser-copy`** may be useful to read a sub-stream without a standard prefixed header. In this case the header information must be passed in by the caller. The data address is expected to be in the A register and the start address is just the return address in R at entry. The word count is passed in as the second of three values on the stack. The other two are ignored.

A single triad of bytes may be read using **`18ibits`**, which requires a dummy input value and returns a dummy value on top of the input word on top of the baud counter. Units smaller than three bytes can only be supported if the programmer takes responsibility for passing in an accurate baud counter. The interested reader is directed to the stack comment for the entry point **`byte`** that is internal to **`18ibits`**. It may be found in the file **`serial.vf`** in the sub-directory appropriate to your chip version (e.g., **`VentureForth/vf/c7Fr01/serial.vf`**).

Glossary

ser-exec (x —n) A *ser-exec*
Reads a standard boot stream three-word header packet and then loads the described sub-stream and branches to its entry point (with the baud counter on top of the stack). The returned value *n* is the baud counter.

ser-copy (x n x —n) A *ser-copy*
Reads an *n*-word sub-stream without a standard prefixed header. Data is stored starting at the address in A. The baud counter is returned.

18ibits (x n x —n) A *18-eye-bits*
Reads a single 18-bit word to the address in A, returning the baud counter.

B.5 S40C18 Synchronous I/O

Send feedback

The synchronous serial routines to support bootstrap or data input are located in node 33. The clock is on bit 17 and data pass on bit 1. The clock is always controlled externally, but the data direction depends upon the protocol. The ROM routines only support the input mode.

The unit of input or output, controlled by the external clock, are 18-bit words. The clock is bi-phase, in that data are transferred on each edge. Consequently, the clock will always return to zero. The start of input is signaled by the first low to high transition of the clock. No synchronization of flow occurs between words. There is a synchronization protocol that is specified for transitions between input and output. Stream data must be delivered within one clock period or risk losing two bit edges.

Of the internal labels defined by the serial driver, only three are intended as external API points. The primary API is via the word **ser-exec**, which takes a dummy value and leaves the baud rate counter calculated for the last received word. It reads a standard boot stream three-word header packet, then loads the described sub-stream and branches to its entry point (with the baud counter on top the stack). When used to read part of a boot stream that is a concatenation of several sub-streams, **ser-exec** is used as the concatenation address contained in all but the last sub-stream.

Use of any other entry point to the serial driver implies that a custom protocol is being designed and must therefore be carefully co-ordinated with the outside agent that the SEAforth is communicating with. The

entry point **ser-copy** may be useful to read a sub-stream without a standard prefixed header. In this case, the header information must be passed in by the caller. The data address is expected to be in the A register, and the start address is just the return address in R at entry. The word count is passed on the stack on top of a trash value.

A single word may be read using **sget**, which takes no input value and returns the input word. Units smaller than a word are not supported by this specification.

Glossary

ser-exec (x —x) A *ser-exec*
Reads a standard boot stream three-word header packet and then loads the described sub-stream and branches to its entry point (with the baud counter on top of the stack).

ser-copy (x n —x) A *ser-copy*
Reads an *n*-word sub-stream without a standard prefixed header. Data is stored starting at the address in A. The baud counter is returned.

sget (— x) A *s-get*
Reads a single 18-bit word to the stack.

Appendix C: List of Commands

Send feedback

This section provides summary lists of all the commands described in this book, derived from the glossary entries. Each includes the name, stack usage, pronunciation, and page reference. If you are using an electronic version of this book, all the command names are "hot links" to the referenced page. Table C.1 lists all commands that execute on the host (e.g. compiler directives, simulator commands, and other development tools), while Table C.2 lists all target (SEAforth) commands. In the latter, the Location column identifies the SEAforth variants in which each command appears; "A" indicates All.

Table C.3 Host commands descrbed in this book

Command	Stack	Location	Pronunciation	Page
(	(—)	H	*left-paren*	*42*
.	(—)	H	*no-op*	*43*
,	(n —)	H	*comma*	*46*
:xnode	(n —)	H	*colon-x-node*	*63*
?|	(—)	H	*question-bar*	*44*
.adrs	(addr len —)	H	*dot-address*	*23*
' <name>	(— addr)	H	*tick*	*46*
's <name>	(n — addr)	H	*<n>'s*	*46*
[else]	(—)	H	*bracket-else*	*45*
[if]	(t —)	H	*bracket-if*	*44*
[then]	(—)	H	*bracket-then*	*45*
[x']	(—)	H	*bracket-x-prime*	*63*
[x]	(—)	H	*bracket-x*	*63*
{node	(n —)	H	*bracket-node*	*42*
****	(—)	H	*back-slash*	*42*
#	(n —)	H	*number-sign*	*47*

Table C.3 Host commands descrbed in this book

Command	Stack	Location	Pronunciation	Page
+include" <filename>"	(addr len —)	H	*plus-include*	*15*
<branch	(n —)	H	*return-branch*	*63*
<init	(—)	H	*from-init*	*63*
<node	(—)	H	*from-node*	*63*
=a	(x —)	H	*equal-a*	*43*
=b	(addr —)	H	*equal-b*	*43*
=p	(addr —)	H	*equal-p*	*43*
>branch	(n_1 n_2 ... n_n n —)	H	*forward-branch*	*63*
>node	(n —)	H	*to-node*	*63*
>root	(n —)	H	*to-root*	*63*
>rtn	(x_1 x_2 ... x_n n —)	H	*to-return*	*43*
>stk	(x_1 x_2 ... x_n n —)	H	*to-stack*	*43*
|	(—)	H	*bar*	*44*
active	(n —)	H	*active*	*23*
dump	(addr len —)	H	*dump*	*23*
dumpRAM	(—)	H	*dump-RAM*	*23*
dumpROM	(—)	H	*dump-ROM*	*24*
equ <name>	(n —)	H	*e-q-u*	*42*
fixate	(n —)	H	*fixate*	*23*
here	(— addr)	H	*here*	*47*
include <filename>	(—)	H	*include*	*15*
node}	(—)	H	*node-bracket*	*42*
nodePath	(n_1 n_2 ... n_n n —)	H	*node-path*	*20*
nop	(—)	H	*no-op*	*43*
org	(addr —)	H	*org*	*42*
power	(—)	H	*power*	*22*
reset	(—)	H	*reset*	*22*
setmax	(n —)	H	*set-max*	*23*

Table C.3 Host commands descrbed in this book

Command	Stack	Location	Pronunciation	Page
setstep	(n —)	H	*set-step*	*23*
simq	(—)	H	*sim-q*	*22*
simulate (or sim)	(—)	H	*sim*	*22*
upto	(n —)	H	*up-to*	*23*
v.VF	(— addr len)	H	*v-dot-vf*	*15*
watch1	(n —)	H	*watch-one*	*23*
watch2	(n_1 n_2 —)	H	*watch-two*	*23*
watch4	(n_1 n_2 n_3 n_4 —)	H	*watch-four*	*23*

Table C.4 Target commands described in this book

Command	Stack	Location	Pronunciation	Page
+	(n_1 n_2 — n_3)	A	*plus*	*29*
-;	(—)	A	*minus-semi-colon*	*33*
-if	(n — n)	A	*minus-if*	*36*
-u/mod	(d_1 d_2 -n — nr nq)	S40	*minus-u-slash-mod*	*76*
-until	(n — n)	A	*minus-until*	*39*
-while	(n — n)	A	*minus-while*	*39*
;	(R:addr —)	A	*semi-colon*	*33*
;:	(R:$addr_1$ — R:$addr_2$)	A	*semi-colon-colon*	*33*
: <name>	(—)	A	*colon*	*33*
!a	(x —)	A	*store-a*	*28*
!a+	(x —)	A	*store-a-plus*	*28*
!b	(x —)	A	*store-b*	*28*
!p+	(x —)	A	*store-p-plus*	*29*

Table C.4 Target commands described in this book

Command	Stack	Location	Pronunciation	Page
@a	(— x)	A	*fetch-a*	*27*
@a+	(— x)	A	*fetch-a-plus*	*28*
@b	(— x)	A	*fetch-b*	*28*
@p+	(— x)	A	*fetch-p-plus*	*28*
*****	(n_1 n_2 — n_1 d_1, A:d_2)	S40	*star*	*75*
+*	(n_1 n_2 — n_1 n_3)	A	*plus-star*	*30*
~u/mod	(d_1 d_2 -n — nr nq)	S40	*tilde-u-slash-mod*	*76*
10stream	(—)	S40	10-stream	20
18ibits	(n —n w)	A	*18-eye-bits*	*70*
18ibits	(x n x —n)	A	*18-eye-bits*	*72*
18ibits	(n —n w)	A	*18-eye-bits*	*84*
18ibits	(x n x —n)	A	*18-eye-bits*	*86*
19stream	(—)	A	19-stream	20
2*	(n_1 — n_2)	A	*two-star*	*30*
2/	(n_1 — n_2)	A	*two-slash*	*30*
32stream	(—)	S40	32-stream	20
33stream	(—)	S40	33-stream	20
8obits	(n w —n w')	A	*eight-oh-bits*	*70*
8obits	(n w —n w')	A	*eight-oh-bits*	*84*
a!	(x —)	A	*a-store*	*27*
a@	(— x)	A	*a-fetch*	*27*
again	(—)	A	*again*	*41*
and	(n_1 n_2 — n_3)	A	*and*	*29*
b!	(addr —)	A	*b-store*	*27*
begin	(—)	A	*begin*	*39*
bget	(n — n x)	S40	*b-get*	*79*
call	(— R:addr)	A	*call*	*33*
dac18	(m c p u — m c p)	S40	*dack-18*	*80*

Table C.4 Target commands described in this book

Command	Stack	Location	Pronunciation	Page
dac27	(m c p a w — m c p)	S40	*dack-27*	*80*
drop	(x_1 x_2 — x_1)	A	*drop*	*26*
dup	(x — x x)	A	*dup*	*26*
else	(—)	A	*else*	*36*
for	(n — R:n)	A	*for*	*40*
half	(n_1 w n_2 — n_1 w)	A	*half*	*70*
half	(n_1 w n_2 — n_1 w)	A	*half*	*84*
if	(n — n)	A	*if*	*36*
interp	(i m s —v)	S40	*interp*	*77*
jump	(—)	A	*jump*	*35*
mac18	(n_1 n_2 u' u' — n_1 n_2+n_1*u')	S24	*mac-18*	*66*
meanwhile	(—)	A	*meanwhile*	*39*
negate	(n — -n)	S40	*negate*	*76*
next	(R:n — R:n-1 *if non-zero* | *if zero*)	A	*next*	*40*
not	(n_1 — n_2)	A	*not*	*29*
obit	(n w —n w)	A	*oh-bit*	*70*
obit	(n w —n w)	A	*oh-bit*	*84*
over	(x_1 x_2 — x_1 x_2 x_1)	A	*over*	*26*
pop	(R:x — x)	A	*pop*	*26*
push	(x — R:x)	A	*push*	*26*
rbit	(n w —n w)	A	*r-bit*	*70*
rbit	(n w —n w)	A	*r-bit*	*84*
repeat	(—)	A	*repeat*	*39*
rotate	(x y w — x' y')	S40	*rotate*	*77*
select	(n w —n w)	A	*select*	*70*
select	(n w —n w)	A	*select*	*84*
ser-copy	(x n x —n)	A	*ser-copy*	*72*
ser-copy	(x n —x)	A	*ser-copy*	*73*

Table C.4 Target commands described in this book

Command	Stack	Location	Pronunciation	Page
ser-copy	(x n x —n)	A	*ser-copy*	*86*
ser-copy	(x n —x)	A	*ser-copy*	*87*
ser-exec	(x —n)	A	*ser-exec*	*72*
ser-exec	(x —x)	A	*ser-exec*	*73*
ser-exec	(x —n)	A	*ser-exec*	*86*
ser-exec	(x —x)	A	*ser-exec*	*87*
sget	(— x)	A	*s-get*	*73*
sget	(— x)	A	*s-get*	*87*
spi-boot	(n cmd_1 R: cmd_2 — n x)	A	*s-p-i-boot*	*70*
spi-boot	(n cmd_1 R: cmd_2 — n x)	A	*s-p-i-boot*	*84*
spi-copy	(x n x —n)	A	*s-p-i-copy*	*70*
spi-copy	(x n x —n)	A	*s-p-i-copy*	*84*
spi-exec	(n x —n x)	A	*s-p-i-exec*	*70*
spi-exec	(n x —n x)	A	*s-p-i-exec*	*84*
su*	(n u — n' n'*u)	S24	*s-u-star*	*66*
taps:	(y x c R: addr — y' x')	S40	*taps*	*78*
then	(—)	A	*then*	*37*
triangle	(x — y)	S40	*triangle*	*78*
u/mod	(d_1 d_2 +n — nr nq)	S40	*u-slash-mod*	*76*
u2/	(n — n')	S40	*u-two-slash*	*76*
unext	(R:n — R:n-1 *if non-zero* | *if zero*)	A	*micro-next*	*40*
until	(n — n)	A	*until*	*39*
while	(n — n)	A	*while*	*39*
xor	(n_1 n_2 — n_3)	A	*xor*	*29*

Index