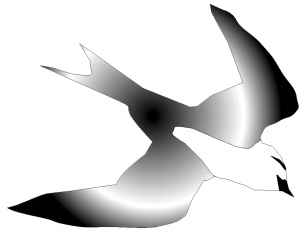

Getting Results Faster...

SwiftX 68K

for Motorola 68K-family Targets

FORTH, Inc.
111 N. Sepulveda Boulevard, Suite 300
Manhattan Beach, California USA 90266–6847
310.372.8493 800.55.FORTH FAX: 310.318.7130
forthsales@forth.com www.forth.com

First edition, January 1998
Printed 6/22/98

FORTH, Inc.
111 N. Sepulveda Boulevard, Suite 300
Manhattan Beach, California USA 90266–6847
310.372.8493 800.55.FORTH FAX: 310.318.7130
forthsales@forth.com www.forth.com

CONTENTS

List of Figures

List of Tables

Welcome!

Important Information in This Book

This book is designed to accompany all SwiftX 68K Family systems. It includes important information to help you connect the accompanying target board to your host PC. Failure to read and follow the instructions and recommendations in this book can cause you to experience frustration and, possibly, a damaged board. Since we want your experience with SwiftX to be a happy one, we urge you to make it easy on yourself. Read before plugging!

Scope of This Book

This book covers hardware-specific information about the setup and use of the target board supplied with your SwiftX system, and discusses hardware-related details of SwiftX development. It contains one appendix for each board-level product supported by SwiftX for the 68K Family; refer to the section for the board you will be using.

This book does not contain general user instructions about SwiftX and the Forth programming language, nor does it provide comprehensive information about the 68K Family. Refer to Motorola's documentation for information about the 68K Family in general, and your processor in particular; to the *SwiftX Reference Manual* to learn about the SwiftX Cross-Development System; and to the *Forth Programmer's Handbook* to learn about Forth.

Audience

This manual is intended for engineers developing software for processors in embedded systems. It assumes general familiarity with board-level issues such as power supplies and connectors.

How To Proceed

Begin with "Getting Started" on page 1. It will guide you through the process of connecting the target board supplied with this system and installing the software.

After you have installed and tested SwiftX on the PC and on the target board, all SwiftX functions will be available to you. That is a good time to refer to Section 1 of your *SwiftX Reference Manual* to learn how to operate your SwiftX system, including the demo application.

Support

A new SwiftX purchase includes a Support Contract. While this contract is in effect, you may obtain technical assistance for this product from:

FORTH, Inc.
111 N. Sepulveda Boulevard, Suite 300
Manhattan Beach, California USA 90266-6847
310-372-8493 or 800-55-FORTH
forthhelp@forth.com

1. GETTING STARTED

This section provides a "road map" to help you get off to a good start with your SwiftX development system.

Three major steps are required to install SwiftX and begin using it:

1. *Connect the target board* provided with this system to your PC. To do so, follow the instructions given in the appendix for your board (see Table 1 below).

Table 1: Boards documented in this manual

Board	Connection instructions	Page
New Micros NMIX-0332	Appendix A: NMIX-0332 Board Instructions	65
Vesta SBC332	Appendix B: Vesta SBC332 Board Instructions	73

We strongly advise you to set up and use the test board supplied with this system until you are familiar with SwiftX, even if your application's actual target hardware will be different. Once you understand the interactive development environment, you will find it much easier to compile and install the software on your specialized hardware.

2. *Install the software* by following the instructions in the *SwiftX Reference Manual*, Section 1.3. The installation procedure creates a SwiftX program group on Windows' Start > Programs menu, from which you may launch the main program by selecting the icon for your target board. The other icons in this program group provide access to documentation files.

   If you need to uninstall SwiftX, use the "Add/Remove Programs" utility under Control Panels.

3. *Run the demo application* included with the system, following the instructions in the *SwiftX Reference Manual*, Section 1.4.3. For any board-specific issues involved with running the demo, such as using a different serial port, refer to the appendix in this book that corresponds to your board.

   General procedures for developing and testing software with SwiftX are provided in the *SwiftX Reference Manual*, Section 1.4.1.

2. THE 68K FAMILY ASSEMBLER

In this section, we assume you understand the hardware and functional characteristics of the M68000 family of processors (as described in Motorola manuals for the various members of this family).

Where **BOLDFACE** type is used here, it distinguishes Forth words (such as register names) from Motorola names. Sometimes, these names are the same; the names **SFC** and **DFC** can be used as a Forth word and as Motorola's name. Where boldface is *not* used, the name refers to Motorola's usage or to hardware issues which are not particular to Forth.

This section supplements, but does not replace, the relevant Motorola manuals. Although we attempt, where possible, to be compatible with Motorola's mnemonics, postfix notation and Forth's stack have been used to simplify the assembler's operation. Departures from Motorola usage are noted; nonetheless, use Motorola's manuals for a detailed description of specific instructions.

The CPU32, the instruction-processing module of some members of the M68300 family of embedded controllers, is based on the industry-standard MC68000 core processor, with many features of the MC68010 and MC68020.

2.1 CODE DEFINITIONS

Code definitions normally have the following syntax:

```
CODE <name>   <assembler instructions>  RTS  END-CODE
```

For example:

```
CODE DEPTH ( -- n) \ Return the current stack depth
   S0 U) D0 MOV    \ Fetch the bottom stack address
   S D0 SUB        \ Subtract the current stack pointer
   2 # D0 ASR      \ Convert to number of cells
   D0 S -) MOV     \ Push value onto stack
   RTS   END-CODE
```

All code definitions must be terminated by the command **END-CODE**. As an alternative to the normal **RTS**, the phrase:

```
' STOP BRA
```

returns through the SwiftOS multitasking executive, leaving the current task idle and passing control to the next task. This phrase may be used before **END-CODE** to terminate the action by idling the current task (whereas the normal behavior would be to execute the next word).

You may name a code fragment using the form:

LABEL <name>

This creates a definition that returns the address of the next code space location, in effect naming it. You may use such a location as the destination of a branch, for example. The code fragments used as exception handlers are constructed in this way, and the named locations are then passed to **EXCEPTION**, which connects the code address to a specified exception vector.

If **LABEL** is used to name a standalone code fragment, the fragment must be terminated by **END-CODE**.

The critical distinction between **LABEL** and **CODE** is that, if you reference a **CODE** definition inside a colon definition, the cross-compiler will assemble a call to it. If you invoke it interpretively while connected to a target, it will be executed. In contrast, reference to a **LABEL** under any circumstance will return the *address* of the labelled code.

Within code definitions, the words defined in the following sections may be used to construct machine instructions.

Glossary

CODE <name> *(—)*
Start a new assembler definition *name*. If the definition is referenced inside a target colon definition, it will be called.

LABEL <name> *(—)*
Start an assembler code fragment name. If the definition is referenced, either inside a definition or interpretively, the address of its code will be returned on the stack.

END-CODE *(—)*
Terminate an assembler sequence started by **CODE** or **LABEL**.

References Exception handling, Section 2.5
SwiftOS multitasking executive, *SwiftX Reference Manual*, Section 4.

2.2 Addressing Modes

Instructions specify operand location in one of three ways:

- *Register Specification*—the number of the register is a register field of the instruction.
- *Effective Address*—the address is developed using modes such as indirect, immediate, indexed, etc.
- *Implicit Reference*—the definition of certain instructions implies the use of specific registers.

2.2.1 Register Notation and Usage

In the Forth assembler, registers are defined and identified as follows:

- <Dn> Specifies one of the data registers **D0–D7**.
- <An> Specifies one of the address registers **A0**–**A7**.

Three of the address registers are special Forth system pointers, and have been given standard names to indicate their functions, as described in Table 2.

Table 2: Special register assignments

Register	Name	Description
A5		Reserved for future use
A5	**U**	Address of the *user* area of the currently running task.
A6	**S**	Data *stack* pointer (address of the top item).
A7	**R**	*Return* stack pointer (address of the top item).

Register **U** must be saved and restored, if used. Registers **S** and **R** must be treated as pointers to the 32-bit-wide stacks.

The register names **A5**–**A7** are valid, but use of the Forth names is preferred, to make clear to readers of the code that these registers have special functions.

Registers **D0**–**D7** and **A0**–**A3** are scratch and may be used freely.

2.2.2 Opcode and Addressing Mode Specification

The Forth assembler for the M68K retains, to the extent it is feasible, compatibility with the names of Motorola's opcodes. However, the syntax employed is more in line with that of other Forth assemblers.

Table 3: Motorola and Forth assembler notation examples

Motorola mnemonic	SwiftX assembler example	Description
ADD	`D0 D1 ADD`	Add **D0** to **D1** — result in **D1**.
ADDA	`D0 A1 ADD`	Add **D0** to **A1** — result in **A1**.
ADDI	`1024 # D0 ADD`	Add 1024 to **D0**.
ADDQ	`1 #Q D0 ADD`	Add 1 to **D0** immediate quick.

Motorola's assembler assigns a separate mnemonic name for each addressing mode for many of the opcodes. For example, **ADD** may be one of four variants.

Forth uses one **ADD** instruction modified by its parameters, resulting in a far more compact assembler. Some examples are given in Table 3.

2.2.3 Operand Size Specification

The operation sizes of the assembler are 32-bit (*cell*), 16-bit (*half-cell*, or *word*), and 8-bit (*byte*). The default is cell. This may be overridden by use of the instruction prefixes **B.** and **W.**, where applicable. **W.** specifies that the following instruction is to be assembled as a half-cell (i.e., a word) instruction, and **B.** specifies byte operation. For example:

```
D0 B. NEG   D0 W. NEG   D0 NEG
```

assemble instructions to perform a two's complement operation on the low-order byte, low-order word, and all 32 bits of data register 0, respectively.

Operation size is also affected by the use of an immediate operator. **#B**, **#W**, and **#** imply an operation size, and do not require the use of **B.** or **W.** before the instruction. For example:

```
1 #B D0 ADD   1 #W D0 ADD   1 # D0 ADD
```

assemble the instructions to add 1 to data register 0, with the operation size as byte, word, and cell, respectively.

2.2.4 Effective Address Specification

The Forth assembler uses the following symbols to specify the various effective addressing modes:

Data Register Direct

The operand is in the data register specified by the effective address register field.

Notation: `<Dn>`
where *Dn* may be **D0**–**D7**.

Address Register Direct

The operand is in the specified address register.

Notation: <An>
where *An* may be **A0**–**A7**, or the named registers **U**, **S**, **R**.

Address Register Indirect

The address of the operand is in the specified address register.

Notation: <An> **)**
where *An* may be **A0**–**A7**, or the named registers **U**, **S**, **R**.

Address Register Indirect with Post-increment

The address of the operand is in the address register specified in the register field. After the operand address is used, it is incremented by one, two, or four, depending upon whether the size of the operand is byte, word, or cell. If the address register is **R** and the operand size is byte, the address is incremented by two to maintain even address alignment of the hardware stack.

Notation: <An> **)+**
where *An* may be **A0**–**A7**, or the named registers **U**, **S**, **R**.

Address Register Indirect with Pre-decrement

As above, except the address is decremented before it is used.

Notation: <An> **-)**
where *An* may be **A0**–**A7**, or the named registers **U**, **S**, **R**.

Address Register Indirect with Displacement

The address of the operand is the sum of the contents of the address register and the sign-extended 16-bit integer in the extension word.

Notation: <offset> <An**)**>
where *offset* is a sign-extended, 16-bit integer, and *An***)** may be **A0)**, etc., or **U)**, **S)**, and **R)**.

U) is a special case of the above. Arguments to **U)** must be the name of a **USER**

variable. For example, to access the user variable **BASE**, you might use:

```
BASE U) D0 MOV
```

Address Register Indirect with Index

The address of the operand is the sum of the contents of the address register, the sign-extended displacement integer in the low-order 8 bits of the extension word, and the contents of the index register.

Notation: `<disp> <index> +X <An>)`
or `<disp> <index> +XL <An>)`
where *disp* is an 8-bit, sign-extended integer; *index* may be any data or address register; and *An***)** may be **A0)**, etc., or **U)**, **S)**, and **R)**. **+X** specifies that the low-order word of the index register is used, and **+XL** specifies that all 32 bits of the index register are to be used in the address computation.

For instance, if **D0** contains the value 12 and **A0** contains the address of an array in memory, the instruction:

```
4 D0 +XL A0) D1 MOV
```
will load register **D1** with the value located at the 16th byte (12+4) of the array.

Absolute Addressing

The address of the operand is contained in the extension word(s). The assembler assembles a word or a cell address, appropriately.

Notation: `<addr> AB`
where *addr* may be any absolute address. **AB** may not be used with **BRA**, **BSR**, or **DBRA**.

Program Counter with Displacement

The address is the sum of the program counter and the sign-extended, 16-bit integer in the extension word. The value of the program counter is the address of the word following the opcode.

Notation: `<addr> PC)`

where *addr* is any absolute address.

The address is assembled as its 16-bit, signed offset from the program counter's value when the instruction being assembled executes. In Forth, the address is often provided by using **LABEL** to name a location.

Program Counter Relative with Index

The address is the sum of the program counter; the sign-extended, 8-bit displacement integer; and the contents of the index register. See Address Register Indirect with Index (above).

Immediate Data

May be byte, word, or cell.

Notation: `<n> #B` *or* `<n> #W` *or* `<n> #`
where *n* is the immediate data, a literal value.

Quick Immediate Mode

The quick mode is a special case of immediate addressing. Three instructions may use this mode: **ADD**, **SUB**, and **MOV**. The immediate data is stored in a field in the opcode.

Notation: `<n> #Q`
For **ADD** and **SUB**, *n* is an unsigned integer from 1–8. For **MOV**, *n* is a signed, 8-bit integer (values -128 to 127).

2.3 Instruction Set Details

This section contains detailed information about each M68K instruction as it is implemented in the Forth assembler.

Because of the nature of the microcode in the M68K, illegal address modes for one instruction may produce legal opcodes for completely different instructions. It is, therefore, necessary to be reasonably conversant with the M68K assembler before attempting to write **CODE** definitions. Although the Forth assembler checks for most types of errors, in deference to the virtual impossibility of 100%

error checking in a processor of this complexity, it is primarily the programmer's responsibility to ensure that instructions are assembled correctly.

ADD — add binary

ADD the source operand to the destination operand, and store the result in the destination location. For example:

D0 D1 ADD
Add the cell in data register 0 to the cell in data register 1, result in **D1**.

2 #Q PTR U) ADD
Add 2 (quick mode) to the cell in the **USER** variable **PTR**.

4 D0 +X A0) A1 W. ADD
Add the word whose address is the sum of the contents of address register 0 plus the low-order word of data register 0 plus the 8-bit offset 4, to the low-order word of address register 1. The source is sign-extended, and the operation is performed using all 32 bits of **A1**.

AND — logical and

AND the source operand to the destination operand, and store the result in the destination location. The contents of an address register may not be used as an operand. For example:

7F #W D0 AND
Logically AND the contents of the low-order word of data register 0 with the immediate value $7F_H$. Only the low-order word of **D0** is affected.

D0 FF8001 AB B. AND
Logically AND the contents of the low-order byte of data register 0 to the byte at the **AB**solute address $FF8001_H$.

ASL, ASR — arithmetic shift

Arithmetically shift the bits of the operand in the direction specified. The carry bit and the extend bit receive the last bit shifted out of the operand. The shift count for shifting a register may be specified in two ways:

Immediate: The shift count is specified in the instruction (range 1–8).

Register: The shift count is contained in a data register specified in the instruction.

The contents of memory may be shifted one bit only, and the operation size is restricted to a word. For example:

S) ASL
Shift the 16-bit word which is on top of the parameter stack one bit to the left.

2 # D0 W. ASL
Shift the contents of the low-order word in **D0** two bits to the left.

The only allowable immediate operator with shift instructions is **#**. The operation size prefixes **W.** and **B.** must be used to alter the default operation size.

BCHG — test a bit and change

A bit in the destination operand is tested and the state of that bit is reflected in the **Z** condition code. After the bit is tested, its state is changed. If a data register is the destination, the bit numbering is modulo 32. If the destination is a memory location, the bit numbering is modulo 8. The bit number may be specified in one of two ways:

Immediate: The bit number is specified in an extension word.
Register: The bit number is contained in a data register specified in the instruction.

For example:

2 # -2 A0) BCHG
Test and change bit two of the byte whose address is two less than the address contained in A0.

D0 D1 BCHG
The bit whose number is contained in data register 0 is tested and changed in data register 1.

The only allowable immediate operator for bit instructions is **#**.
B. and **W.** specify operation size.

BCLR — test a bit and clear

A bit in the destination operand is tested, and the state of that bit is given in the **Z** condition code. After the bit is tested, it is cleared. If a data register is the destination, the bit numbering is modulo 32; if the destination is a memory location, the bit numbering is modulo 8. The bit number may be specified in one of two ways:

Immediate:	The bit number is specified in an extension word.
Register:	The bit number is contained in a data register specified in the instruction.

For example:

```
2 # -2 A0) BCLR
```
Test and clear bit number two of the byte whose address is two less than the address contained in A0.

```
D0 D1 BCLR
```
The bit whose number is contained in data register 0 is tested and cleared in data register 1.

The only allowable immediate operator for bit instructions is **#**. **B.** and **W.** are used to specify operation size.

BGND — enter background mode (CPU32 only)

If background mode is enabled, the processor will suspend instruction execution and enter background mode. The FREEZE output is asserted. Upon exiting from background mode, execution continues with the instruction pointed to by the current program counter.

If background mode is not enabled, an illegal instruction exception is generated.

BRA — branch always

Program execution continues at location PC+displacement. *Displacement* is a two's complement integer (8-, 16-, or 32-bit, as necessary) which counts the relative distance in bytes. The value in *PC* is two plus the current instruction location. For example:

`' PAUSE BRA`
Branch always to the address of **PAUSE**.

The address modifier **AB** is not required with **BRA**, as the instruction assumes that the top stack item is the destination address for the branch.

BSET — test a bit and set

A bit in the destination operand is tested, and the state of that bit is reflected in the **Z** condition code. After the bit is tested, it is set. If a data register is the destination, the bit numbering is modulo 32. If the destination is a memory location, the bit numbering is modulo 8. The bit number may be specified in one of two ways:

Immediate:	The bit number is specified in an extension word.
Register:	The bit number is contained in a data register specified in the instruction.

For example:

`2 # -2 A0) BSET`
Test and set bit number two of the byte whose address is two less than the address contained in A0.

`D0 D1 BSET`
The bit whose number is contained in data register 0 is tested and set in data register 1.

The only allowable immediate operator for bit instructions is **#**.
B. and **W.** specify operation size.

BSR — branch to subroutine

The address of the instruction immediately following the **BSR** instruction is pushed onto the return stack. Program execution then continues at location PC+displacement. See **BRA**.

BTST — test a bit

A bit in the destination operand is tested, and the state of that bit is reflected in the **Z** condition code. If a data register is the destination, the bit numbering is

modulo 32. If the destination is a memory location, the bit numbering is modulo 8. The bit number may be specified in one of two ways:

Immediate: The bit number is specified in an extension word.
Register: The bit number is contained in a data register specified in the instruction.

For example:

2 # -2 A0) BTST
Test bit number two of the byte whose address is two less than the address contained in A0.

D0 D1 BTST
The bit whose number is contained in data register 0 is tested in data register 1.

The only allowable immediate operator for bit instructions is **#**. The **B.** and **W.** commands are used to specify operation size.

CLR — clear an operand

Clear destination to all zero bits. Size may be byte, word, or cell. For example:

CTR U) CLR
Clear the cell in the **USER** variable **CTR**.

D0 W. CLR
Clear the low-order word of data register 0.

CMP — compare

Subtract the source operand from the destination operand, and set the condition codes according to the result; the destination operand is unchanged. The operation may be byte, word, or cell. If the destination is an address register, the operation size is restricted to word or cell. For example:

D0 D1 W. CMP
Compare the low-order word in **D0** to the low-order word in **D1**.

D0 A0 W. CMP
Compare the low-order word in **D0** with the low-order word in **A0**. **D0** will be sign-extended, and the compare will use all 32 bits of **A0**.

A0)+ A1)+ CMP
This is a special case of the **CMP** instruction. The operands are always addressed with the post-increment addressing mode, using the specified address registers. The size of the operation may be byte, word, or cell.

COM — (NOT) logical complement

Take the one's complement of the destination operand, and store the result in the destination location. The size of the operation may be specified to be byte, word, or cell. The Forth name **COM** avoids conflict with the logical operator called **NOT**. For example:

A0) W. COM
Perform a one's complement on the 16-bit word whose address is the contents of address register 0.

DBCC, DBRA — test condition, decrement and branch

DBCC is a looping primitive of three parameters: a condition, a data register, and a displacement. When **DBCC** is executed, the specified condition code is tested. If the condition is met, execution falls through to the next instruction. If the condition is not met, the low-order word of the specified data register is decremented by one. If the data register reaches -1, program execution falls through to the next instruction; otherwise, a branch is taken to the specified address. Usage of **DBRA** is the same as for **DBCC**, except the branch is unconditional. For example:

<addr> **D0 0= DBCC**
Branch to location *addr* until either the Z bit is set or **D0** reaches -1.

<addr> **D0 DBRA**
A special case of **DBCC** which tests no condition and may be used as a simple "loop until count is exhausted."

See Section 2.4 for a further discussion of assembler structures that may be used with **DBRA** and **DBCC**, and condition codes that may be used with **DBCC**.

DIVS — signed divide

Divide the signed destination operand by the signed source operand, and store the signed result in the destination. The destination operand can be either 32-bit or 64-bit, and the source operand can be either 16-bit or 32-bit. The instruction uses one of four forms, of which the last three are only available on the CPU32 and 68020 and above:

- Divide a 32-bit quantity by a 16-bit quantity, to yield a 16-bit quotient and a 16-bit remainder. The 16-bit quotient is in the low-order word, and the 16-bit remainder is in the high-order word. The sign of the remainder is always the same as the dividend, unless the remainder is equal to zero.
- Divide a 32-bit quantity by a 32-bit quantity, to yield a 32-bit quotient; the remainder is discarded.
- Divide a 64-bit quantity by a 32-bit quantity, to yield a 32-bit quotient and a 32-bit remainder.
- Divide a 32-bit quantity by a 32-bit quantity, to yield a 32-bit quotient and a 32-bit remainder.

Two special cases may arise:

- Division by zero causes a trap.
- If an overflow occurs, a condition bit will be set and the operands will not be affected.

For example:

```
512 #W D0 DIVS
```
Divide the contents of **D0** by 512.

DIVU — unsigned divide

Divide the unsigned destination operand by the unsigned source operand, and store the unsigned result in the destination. The destination operand can be either 32-bit or 64-bit, and the source operand can be either 16-bit or 32-bit. The instruction uses one of four forms, described under **DIVS**, above.

For example:

D2 D0 D1 DIVU
Divide a 64-bit number in registers **D0** and **D1** by a 32-bit number in register **D2**. The high-order cell of the 64-bit number is in **D0**.

EOR — exclusive OR

Exclusive-OR the source operand to the destination operand, and store the result in the destination. The size of the operation may be specified to be byte, word, or cell. The source operand must be a data register or immediate data. For example:

D0 FF8000 AB B. EOR
Exclusive-OR the low-order byte of data register 0 with the byte at memory location $FF8000_H$.

0AA # D0 W. EOR
Exclusive-OR the low-order word of data register 0 with the immediate value $0AA_H$.

EXG — exchange registers

Exchange the contents of two registers. The operation size is always cell. When exchanging data and address registers, the data register must appear first. For example:

D0 A1 EXG
Exchange contents of **D0** and **A1**.

EXT, EXTB — sign extend

Extend the sign bit of a data register from a byte to a word, or from a word to a cell, depending on the operation size selected. If the operation size is word, bit 7 of the destination data register is copied to bits 8–15 of that register. If the operation size is cell, bit 15 is copied to bits 16–31. For example:

D0 W. EXT
Sign extend the low-order byte of **D0** to a word.

D0 EXT
Sign extend the low-order word of **D0** to a cell.

D0 EXTB
Sign extend the low-order byte of **D0** to a cell.

JMP — jump

Program execution continues at the address specified by the instruction. The operation is unsized. For example:

WAIT AB JMP
Jump to the absolute address named **WAIT**.

A0) JMP
Jump to the address in address register 0.

JSR — jump to subroutine

The address of the instruction immediately following the **JSR** is pushed onto the return stack. Program execution then continues at the address contained in the instruction. See **JMP**.

LEA — load effective address

Load the effective address into the specified address register. For example:

-2 D0 +XL A0) A0 LEA
The address computed as the sum of -2 and the cell in data register 0 and the address in address register 0 is loaded into address register 0.

LINK — link and allocate

Push the specified address register on the stack, then add the immediate operand to the stack pointer. The immediate operand is specified by:

<n> **H**

following the link instruction. The address register is loaded with the address of the old contents of the register. For word operation, the immediate operand is the signed-extended word. For cell operation, the immediate operand is a 32-bit number. This instruction is normally used to allocate stack space for local variables in a procedure. The immediate operand is normally negative; and the stack grows by the size specified, plus four bytes for the saved con-

tents of the address register.

LPSTOP — low power stop

When this instruction is executed, the processor is forced into a low-power standby mode. The processor remains in this mode until a user-specified or higher-level interrupt or reset occurs. Takes a value on the stack, which is the immediate value to be stored into the status register. For example:

2700 LPSTOP
Move the immediate value 2700 into the status register, and enter the low-power standby mode.

LSL, LSR — logical shift

Logically shift the bits of the operand in the direction specified. The carry bit and the extend bit receive the last bit shifted out of the operand. The shift count for shifting a register may be specified in two ways:

Immediate:	The bit number is specified in an extension word.
Register:	The bit number is contained in a data register specified in the instruction.

The contents of memory may be shifted one bit only, and the operation size is restricted to a word. For example:

S) LSL
Shift the 16-bit word which is on top of the parameter stack one bit to the left.

2 # D0 W. LSR
Shift the contents of the low-order word in **D0** two bits to the right.

The only allowable immediate operator with shift instructions is **#**. The operation size prefixes **W.** and **B.** must be used to alter the default operation size.

MMOV — move multiple registers

Allows a selected group of registers to be moved to, or from, sequential memory locations. The ... **//** ... **///** specifies how many of the registers are to be moved. In the case of a word move to a register list, the values are sign

extended. See Motorola's documentation for a more complete discussion of this instruction. For example:

`RL S -) MMOV D0 D3 // A0 A4 ///`
Move a register list onto the parameter stack. The list consists of data registers **`D0`**–**`D3`** and address registers **`A0`**–**`A4`**.

`S )+ RL MMOV D0 D3 // A0 A4 ///`
The inverse of the previous instruction.

The register list specification may consist of any number of contiguous groups of registers. A single **`//`** separates groups, and **`///`** ends the instruction. A single register is a valid group, therefore:

```
RL S -) MMOV D0 // A0 A1 ///
```

is a valid form of the instruction.

`MOV` — move data from source to destination

Move the contents of the source operand to the destination operand. The data is examined as it is moved, and the condition codes are set accordingly. The size of the operation may be set to byte, word, or cell for all but moves to address registers, which may not be byte. For example:

`S )+ D0 MOV`
Pop the top 32-bit stack item into register **`D0`**.

`S ) S -) MOV`
Duplicate the top stack item. This is the definition of **`DUP`**.

`2 W) A0 W. MOV`
Move the 16-bit word whose address is two plus the contents of register **`W`** into register **`A0`**. Note that the result in **`A0`** will be sign-extended to 32 bits.

`A0 )+ A1 )+ B. MOV`
Move the byte pointed to by **`A0`** to the address pointed to by **`A1`**, and increment both **`A0`** and **`A1`**.

MOVEC — move control register

Move the contents of the specified control register to the specified general register, or vice versa. This is always a 32-bit transfer, even though the control register may be implemented with fewer bits. Unimplemented bits are read as zeroes. This is a privileged instruction. For example:

A0 VBR MOVEC
Move the contents of address register 0 to the Vector Base Register.

MTSR — move to status register

Move the contents of the source operand to the status register. The source operand is a word, and all bits of the status register are affected. This is a privileged instruction. For example:

700 #W MTSR
Move the immediate data into the status register. Immediate arguments must be specified as **#W**.

>USP, <USP — move to/from user stack pointer

Transfer the contents of the user stack pointer to/from the specified address register. This is a privileged instruction.

MULS — signed multiply

Multiply the signed destination operand (16-bit or 32-bit) by the signed source operand (16-bit or 32-bit), and store the signed result (32-bit or 64-bit) in the destination.

The instruction uses one of three forms, of which the last two are available only on the CPU32 and the 68020 and higher:

- Multiply two signed 16-bit operands, yielding a 32-bit signed result. All 32 bits of the product are saved in the destination data register.
- Multiply two signed 32-bit operands, yielding a 32-bit signed result. The least-significant 32 bits of the product are saved in the destination data register.
- Multiply two signed 32-bit operands, yielding a 64-bit signed result. All 64 bits of the product are saved in a pair of destination data registers.

Example:

```
S ) D0 D1 MULS
```
Multiply the top item on the stack by the number in **D1**, and return the result in registers **D0** and **D1**. The high-order cell is in **D0**, and the low-order cell is in **D1**. The operation is performed using signed arithmetic.

MULU — unsigned multiply

Multiply the unsigned destination operand (16-bit or 32-bit) by the unsigned source operand (16-bit or 32-bit), and store the unsigned result (32-bit or 64-bit) in the destination.

The instruction uses one of three forms, described under **MULS** above.

Example:

```
S ) D0 D1 MULU
```
Multiply the top item on the stack by the number in **D1**, and return the result in registers **D0** and **D1**. The high-order cell is in **D0**, and the low-order cell is in **D1**. The operation is performed using unsigned arithmetic.

NEG — negate

Subtract the destination operand from zero. The result is stored in the destination location. The size of the operation may be specified to be byte, word, or cell. For example:

```
S ) NEG
```
Perform a two's complement on the top 32-bit stack item. The definition of **NEGATE**.

NOP — no operation

No operation occurs.

OR — logical inclusive OR

Inclusive-OR the source operand to the destination operand, and store the result in the destination. The operation size may be specified to be byte, word, or cell. The contents of an address register may not be used as an operand.

For example:

D0 FF8000 AB B. OR
Inclusive-OR the contents of the low-order byte of data register 0 into the byte at location $FF8000_H$.

1 #W D0 OR
Inclusive-OR a 1 into the low-order word of **D0**.

PEA — push effective address

Compute an effective address, and push the cell result onto the return stack. For example:

2 S) PEA
Push contents of **S** + 2 onto the return stack (an improbable usage in normal Forth programming).

PMOV — (MOVEP) move peripheral data

Transfer data between a data register and alternate bytes of memory, starting at the specified location and incrementing by two for each byte transferred. The memory address is specified using the address register indirect with displacement mode. If the address is even, all the transfers are made on the high-order half of the data bus; if the address is odd, all transfers are made on the low-order half. On an 8- or 32-bit bus, the instruction still accesses every other byte. For example:

D0 0 A0) PMOV
Move the 4 bytes in data register 0 to the address in address register 0, incrementing the address by two for each byte transfer.

RESET — reset external devices

Assert the reset line, causing all external devices to be reset. The processor state is unaffected. This is a privileged instruction.

ROL, ROR — rotate without extend

Rotate the bits of the operand in the direction specified. The extend bit is not included in the rotation. The shift count for the rotation of a register may be

specified in two different ways:

Immediate: The shift count is specified in the instruction (shift range 1–8).
Register: The shift count is contained in a data register specified in the instruction.

The size of the operation may be specified to be byte, word, or cell. The content of memory may be rotated one bit, and the operand size is restricted to cell. For example:

`A0 ) ROR`
Rotate right the 32-bit cell pointed to by address register 0.

`2 # D0 W. ROL`
Rotate left, twice, the low-order word of data register 0.

The only allowable immediate operator with shift instructions is **`#`**. The operation size prefixes **`B.`** and **`W.`** must be used to alter the default operation size.

`ROXL`, `ROXR` — rotate with extend

Rotate the bits of the operand in the direction specified. The shift count for the rotation of a register may be specified in two different ways:

Immediate: The shift count is specified in the instruction (shift range 1–8).
Register: The shift count is contained in a data register specified in the instruction.

The size of the operation may be specified to be byte, word, or cell. The content of memory may be rotated only one bit, and the operand size is restricted to cell. For example:

`A0 ) ROXR`
Rotate right the 32-bit cell pointed to by address register 0.

`2 # D0 W. ROXL`
Rotate left, twice, the low-order word of data register 0.

The only allowable immediate operator with shift instructions is **`#`**. The operation size prefixes **`B.`** and **`W.`** must be used to alter the default operation size.

RTE — return from exception

The status register and program counter are pulled from the return stack. The previous status register and program counter are lost. All bits in the status register are affected. This is a privileged instruction.

RTR — return and restore condition codes

The condition codes and program counter are popped from the return stack. The previous condition codes and program counter are lost. The supervisor portion of the status register is unaffected.

RTS — return from subroutine

The program counter is pulled from the return stack. The previous program counter is lost.

SUB — subtract binary

Subtract the source operand from the destination operand, and store the result in the destination location. The size of the operation may be byte, word, or cell. For example:

```
D0 D1 SUB
```
Subtract data register 0 from data register 1, result in **D1**.

```
2 #Q S0 U) SUB
```
Subtract 2 (quick mode) from the cell in the **USER** variable **S0**.

```
0 D0 +X A0) A1 W. SUB
```
Subtract the word whose address is the sum of the contents of address register 0 plus the low-order word of data register 0 plus the 8-bit offset, from the low-order word of address register 1. The source is sign extended, and the operation is performed using all 32 bits of **A1**.

STOP — load status register and stop

Move a 16-bit immediate operand into the status register, and suspend execution. An interrupt is required for the processor to continue execution. Takes a value on the stack, which is the immediate value to be stored into the status register (see **LPSTOP**).

SWP — swap register halves

Swap 16-bit halves of a data register. (Name changed to avoid conflict with **SWAP**.)

TBLS — table lookup and interpolate (CPU32 only)

TBLS returns a rounded byte, word, or cell signed result; **TBLSN** returns an unrounded byte, word, or cell signed result; **TBLU** returns a rounded byte, word, or cell unsigned result; and **TBLUN** returns an unrounded byte, word, or cell unsigned result. These instructions have two modes of operation: table-lookup-and-interpolate mode and data-register-interpolate mode.

The syntax for table-lookup-and-interpolate mode is:

```
<ea> Dx TBLxx
```

where *ea* is the effective address of the lookup table containing a linearized representation of the dependent variable Y as a function of X. The low-order word of the data register *Dx* contains the independent variable X. This variable consists of an 8-bit integer part and an 8-bit fractional part. The integer part is scaled by the operand size, and is used as an offset to the table. The selected entry in the table is subtracted from the next consecutive entry. A fractional portion of this difference is taken by multiplying by the interpolation fraction. The adjusted difference is then added to the selected table entry. The result is returned in the destination data register, *Dx*.

The syntax for data-register-interpolate mode is:

```
Dym Dyn Dx TBLxx
```

where registers *Dym* and *Dyn* contain the two table entries. Only the fractional portion of the *Dx* register is used in the interpolation; the integer portion is ignored. See Section 7 of the *Motorola 68000 Programmer's Reference Manual* for the use of the table instructions.

TRAP — trap and begin exception processing

Initiate exception processing. The operand specifies one of 16 possible software exception vectors.

TST — test an operand

Compare the operand with zero. No results are saved; however, the condition codes are set accordingly. The size of the operation may be byte, word, or cell. For example:

CTR U) TST
Test the **USER** variable **CTR** for equality to zero.

FF8001 AB B. TST
Test the byte at location $FF8001_H$ for equality to zero.

2.4 Assembler Structures

In conventional assembly language programming, program structures (loops and conditionals) are handled with explicit branches to labeled locations. This is contrary to principles of structured programming, and is less readable and maintainable than high-level language structures.

Forth assemblers in general, and SwiftX in particular, address this problem by providing a set of program-flow macros, listed in the glossary at the end of this section (instructions **DBRA** and **DBCC** are described in Section 2.3). These macros provide for loops and conditional transfers in a structured manner, and work like their high-level counterparts. However, whereas high-level Forth structure words such as **IF**, **WHILE**, and **UNTIL** test the top of the stack, their assembler counterparts test the processor condition codes.

The program structures supported in this assembler are:

```
BEGIN  <code to be repeated> Dx DBRA
BEGIN  <code to be repeated> Dx <cc> DBCC
BEGIN  <code to be repeated> AGAIN
BEGIN  <code to be repeated> <cc> UNTIL
BEGIN  <code>  <cc> WHILE  <more code>  REPEAT
<cc> IF  <true case code>  ELSE  <false case code>  THEN
```

In the sequences above, *cc* represents condition codes, listed in a glossary below. The combination of the condition code and the structure word (**UNTIL**, **WHILE**, **IF**) assembles a conditional branch instruction *Bcc*, where *cc* is the con-

dition code. The word **NOT** following a condition code inverts its sense. The other components of the structures—**BEGIN**, **REPEAT**, **ELSE** and **THEN**—enable the assembler to provide an appropriate destination address for the branch.

All conditional branches use the results of the previous operation which affected the necessary condition bits. Thus:

```
S )+ D0 MOV   0= NOT IF
```

executes the true branch of the **IF** structure if the top stack item (popped into **D0** by the **MOV** instruction, which set the condition bits) is non-zero.

Table 4 shows the instructions generated by a SwiftX conditional phrase. These examples use **IF**. **WHILE** and **UNTIL** generate the same instructions, but satisfy the branch destinations slightly differently. See the glossary below for details. Refer to your processor manual for details on the condition bits.

Note that the standard Forth syntax for sequences such as **0< IF** implies *no branch in the true case*. Therefore, the combination of the condition code and branch instruction assembled by **IF**, etc., branch on the *opposite* condition (i.e., ≥0 in this case).

These constructs provide a level of logical control that is unusual in assembler-level code. Although they may be intermeshed, care is necessary in stack management, since **REPEAT**, **UNTIL**, **AGAIN**, **ELSE**, and **THEN** always work on the addresses on the stack.

Table 4: Instructions generated by SwiftX conditional structure words

Phrase	Instruction assembled	Description
0< IF	BPL	Branch if the N bit is not set.
0< NOT IF	BMI	Branch if the N bit is set.
0= IF	BNE	Branch if the Z bit is not set.
0= NOT IF	BEQ	Branch if the Z bit is set.
0> IF	BLE	Greater-than-zero; branch if the Z bit is set or if N and V differ from each other.
0> NOT IF	BGT	Less-than-or-equal; branch if the Z bit is clear and N and V are both set or both clear.

Table 4: Instructions generated by SwiftX conditional structure words

Phrase	Instruction assembled	Description
S< IF	BGE	Signed less-than; branch if the N and V bits are both set or both clear.
S< NOT IF	BLE	Signed greater-or-equal; branch if the Z bit is set, or if the N and V bits differ from each other.
CS IF	BCC	Branch if the carry bit is clear.
CS NOT IF	BCS	Branch if the carry bit is set.
VS IF	BVC	Branch if the V bit is clear.
VS NOT IF	BVS	Branch if the V bit is set.
U> IF	BLS	Unsigned greater-than; branch if either the carry or zero bits are set.
U> NOT IF	BHI	Unsigned less-than-or-equal; branch if both the carry and zero bits are clear.
NEVER IF	BRA	Unconditional branch (equivalent to **AGAIN**).

In the glossaries below, the stack notation *cc* refers to a condition code. Available condition codes are listed in the glossary that begins on page 31.

Glossary **Branch Macros**

BEGIN *(— addr)*

Leave the current address *addr* on the stack. Doesn't assemble anything.

AGAIN *(addr —)*

Assemble an unconditional branch to *addr*.

UNTIL *(addr cc —)*

Assemble a conditional branch to *addr*. **UNTIL** must be preceded by one of the condition codes given below.

WHILE *($addr_1$ cc — $addr_2$ $addr_1$)*

Assemble a conditional branch whose destination address is left empty, and leave the address of the branch *addr* on the stack. A condition code (see below)

must precede **WHILE**.

REPEAT *($addr_2$ $addr_1$ —)*

Set the destination address of the branch that is at *$addr_1$* (presumably having been left by **WHILE**) to point to the next location in code space, which is outside the loop. Assemble an unconditional branch to the location *$addr_2$* (presumably left by a preceding **BEGIN**).

IF *(cc — addr)*

Assemble a conditional branch whose destination address is not given, and leave the address of the branch on the stack. A condition code (see below) must precede **IF**.

ELSE *($addr_1$ — $addr_2$)*

Set the destination address *$addr_1$* of the preceding **IF** to the next word, and assemble an unconditional branch (with unspecified destination) whose address *$addr_2$* is left on the stack.

THEN *(addr —)*

Set the destination address of a branch at *addr* (presumably left by **IF** or **ELSE**) to point to the next location in code space. Doesn't assemble anything.

Glossary

Condition Codes

0< *(— cc)*

Return the condition code that—used with **DBCC**, **IF**, **WHILE**, or **UNTIL**—will generate a branch on positive (N bit not set).

0= *(— cc)*

Return the condition code that—used with **DBCC**, **IF**, **WHILE**, or **UNTIL**—will generate a branch on non-zero (N bit set).

0> *(— cc)*

Return the condition code that—used with **DBCC**, **IF**, **WHILE**, or **UNTIL**—will generate a branch on zero or negative (Z bit set or N and V differ).

S< *(— cc)*

Return the condition code that—used with **DBCC**, **IF**, **WHILE**, or **UNTIL**—will generate a branch on greater-than-or-equal (N and V bits are the same).

CS *(— cc)*

Return the condition code that—used with **DBCC**, **IF**, **WHILE**, or **UNTIL**—will generate a branch on carry clear.

VS *(— cc)*

Return the condition code that—used with **DBCC**, **IF**, **WHILE**, or **UNTIL**—will generate a branch on overflow (V) clear.

U> *(— cc)*

Return the condition code that—used with **DBCC**, **IF**, **WHILE**, or **UNTIL**—will generate a branch on unsigned less-than-or-equal (either carry or Z set).

NEVER *(— cc)*

Return the condition code that—used with **DBCC**, **IF**, **WHILE**, or **UNTIL**—will generate an unconditional branch.

NOT *(cc_1 — cc_2)*

Invert the condition code cc_1 to give cc_2.

2.5 EXCEPTION HANDLING

The procedure for defining an exception handler in SwiftX involves two steps: defining the actual exception-handling code, and attaching that code to a processor exception or trap number.

The handler itself is written in code. The usual form begins with **LABEL** <name> (**CODE** is not needed, as such routines are not invoked as subroutines), and ends with an **RTE** (Return from Exception) instruction and **END-CODE**.

To attach the code to the handler, use the word **EXCEPTION**, which takes address of the trap handler and an exception number, and links them such that when the exception occurs, it will be vectored directly to the code. No overhead is imposed by SwiftX, and no task needs to be directly involved in exception handling. If a task is performing additional high-level processing (for example, calibrating data acquired by interrupt code), the convention in SwiftX is that the handler code performs only the most time-critical processing, and notifies a task of the event by modifying a variable or by setting the task to wake up. Further information on task control may be found in the *SwiftX Reference Manual*, Section 4.

An example of a simple exception handler is the one provided for the Periodic Interrupt Timer in **\Swiftx\Src\68K\PIT332.fth**. It looks like this:

```
LABEL <TIMER>          \ Periodic timer interrupt handler
   TC1 # MSECS CELL+ AB ADD
   CS IF   1 #Q MSECS AB ADD  THEN
   TC0 [IF]  TC0 # MSECS AB ADD [THEN]\ Accumulate milliseconds
   TC3 # SECS CELL+ AB ADD
   CS IF   1 #Q SECS AB ADD  THEN
   TC2 [IF]  TC2 # SECS AB ADD  [THEN]\ Accumulate seconds
   RTE   END-CODE

<TIMER> $42 EXCEPTION
```

This code simply increments two 64-bit variables used as counters, **MSECS** and **SECS**. Two double-cell time constants calculated from the clock rate are involved, **TC0**/**TC1** for milliseconds, and **TC2**/**TC3** for seconds. The conditional compilation lines involving **TC0** and **TC2** are simply omitting the high-order parts of these time constants if they are zero.

The word **EXCEPTION"** is an enhanced version of **EXCEPTION** used for diagnostic purposes. It takes the address of the exception handler to branch to and the exception number, and is followed by a string terminated by a **"**. The string address is passed in **A0** to the exception handler when an exception occurs.

Background tasks cannot handle output messages; therefore, it's important that this type of exception handler never be used for exceptions that may be generated by a background task. The exceptions in the following lists are intended for use when you are debugging code in an interactive session.

The following diagnostic 68K exceptions are predefined in the file **\Swiftx\Src\68K\Trap68k.fth**:

```
<BERR>  2 EXCEPTION" Bus Error"
<BERR>  3 EXCEPTION" Address Error"
<TRAP>  4 EXCEPTION" Illegal Inst"
<TRAP>  5 EXCEPTION" Zero Divide"
<TRAP>  6 EXCEPTION" CHK Inst"
<TRAP>  7 EXCEPTION" TRAPV Inst"
<TRAP>  8 EXCEPTION" Priv Violation"
<TRAP>  9 EXCEPTION" Trace"
```

```
<TRAP> 10 EXCEPTION" A-line Emulator"
<TRAP> 11 EXCEPTION" F-line Emulator"
```

Alternatively, for CPU32-family processors such as the 68332, the file **\Swiftx\Src\68K\Trap32.fth** provides these diagnostic handlers:

```
<BERR>  2 EXCEPTION" Bus Error"
<BERR>  3 EXCEPTION" Address Error"
<INST>  4 EXCEPTION" Illegal Inst"
<TRAP>  5 EXCEPTION" Zero Divide"
<TRAP>  6 EXCEPTION" CHK Inst"
<TRAP>  7 EXCEPTION" TRAPV Inst"
<TRAP>  8 EXCEPTION" Priv Violation"
<TRAP>  9 EXCEPTION" Trace"
<TRAP> 10 EXCEPTION" A-line Emulator"
<TRAP> 11 EXCEPTION" F-line Emulator"
```

Note that **<BERR>** is used for bus errors, **<INST>** for illegal instructions in the CPU32 set, and **<TRAP>** for all other traps. In fact, these routines (found in the files listed above) share some code, and also provide examples of the generic exception-processor word **.EXCEPTION**, which displays the error messages.

As part of the power-up initialization in the target, the word **/EXCEPTIONS** must be called to initialize all the exception vectors. This is done by the word **START**, in **\Swiftx\Src\68K**<platform>**\Start.fth**.

Glossary

EXCEPTION *(addr n —)*

Store address *addr* into exception vector *n*. Two versions are supplied: the **INTERPRETER** version is used to set the code image vector, while the **TARGET** version sets the vector at run time in the target (a rare occurrence).

/EXCEPTIONS *(—)*

Move the address of the exceptions table (built by various calls to **EXCEPTION** and **EXCEPTION"**) to the CPU Vector Base Register (VBR), and enable interrupts. This word must be executed as part of the power-up initialization, after all exceptions have been constructed.

.EXCEPTION *($addr_1$ $addr_2$ $addr_3$ —)*

Display an exception error message, where $addr_1$ is the Program Counter value when the exception occurred, $addr_2$ is the access address (or 0), and $addr_3$ is the address of a counted string containing an exception error message. This is the shared end-processing for the diagnostic exception handlers managed by **EXCEPTION"**.

EXCEPTION" `<message>` *(addr n —)*

Attach an exception trap handler, whose code is at *addr,* to processor exception *n*.

References Conditional compilation, *SwiftX Reference Manual*, Section 4.1.2
Background tasks, *SwiftX Reference Manual*, Section 4.6

3. Platform-specific Implementation Issues

This section covers specific implementation issues involving the 68K family processors in general, and the 68332 in particular. Board-specific issues are covered in their respective appendices.

3.1 Implementation Strategy

A variety of options are available to a programmer implementing a Forth kernel on a particular processor. In SwiftX, we attempt to optimize, as much as possible, both for execution speed and object compactness. This section describes the implementation choices made in this system.

3.1.1 Execution Model

The execution model is a subroutine-threaded scheme, with in-line code substitution for simple primitives. The M68K's subroutine stack is used by target primitives as Forth's return stack, and may contain return addresses.

If you depend on the return stack to be identical with the subroutine stack, your code will not be portable to systems which separate these stacks. Do not use the return stack except under the specific rules given in Section 3.1.2.

You may easily see the optimization strategies by decompiling some simple definitions. For example, the source definition for **/STRING** is:

```
: /STRING ( c-addr1 len1 u -- c-addr2 len2)
   OVER MIN >R  SWAP R@ +  SWAP R> - ;
```

but if you decompile it, you get:

```
SEE /STRING
9B6   4 A6) A6 -) MOV
9BA   ' MIN BSR
9BE   A6 )+ A7 -) MOV
9C0   ' SWAP BSR
9C4   A7 ) A6 -) MOV
9C6   A6 )+ D0 MOV
9C8   D0 A6 ) ADD
9CA   ' SWAP BSR
9CE   A7 )+ A6 -) MOV
9D0   A6 )+ D0 MOV
9D2   D0 A6 ) SUB
9D4   RTS  ok(T)
```

This example clearly shows the combination of in-line code and subroutine calls in this implementation.

When the last thing in a definition is a reference to another high-level word, the compiler automatically substitutes a **BRA** for the **BSR**, to save the subroutine return. Consider **TUCK**, defined as:

```
: TUCK ( x1 x2 -- x2 x1 x2)   SWAP OVER ;
```

If you type **SEE TUCK**, you will get:

```
04E4   ' SWAP BSR
04E6   4 A6) A6 -) MOV
04EA   RTS  ok(T)
```

However, if you make a new definition:

```
: -TUCK   OVER SWAP ;  ok(T)
```

and type **SEE** -**TUCK**, you get:

```
163C   4 A6) A6 -) MOV
1640   ' SWAP BRA  ok(T)
```

In both cases, the **MOV** instruction is the code substituted for **OVER**, but **SWAP** is called with a **BRA** in the second case, as it is the last thing in the definition, and *its* **RTS** will handle the return, saving the need for one in **-TUCK**.

More extensive optimization is provided in a few words, such as **+**. In this case, the compiler can detect whether the last argument to **+** is a short literal, a large literal, or a value left on the stack from a previous operation, and will produce optimized code for each case.

If you are an experienced Forth programmer and would like to study these implementation strategies, we encourage you to look at the file **Core.f**.

3.1.2 Data Format and Memory Access

The M68K is a 32-bit processor. This implementation assumes a flat memory model (i.e., no paged memory management). The high byte of a 32-bit cell is the lowest address—i.e., this is a *big-endian* machine.

Some CPUs in this family do not support unaligned memory accesses; on all versions SwiftX attempts to maintain address alignment according to the rules in ANS Forth. If you manually construct an un-aligned dictionary (e.g., by an odd argument to **ALLOT**) you may re-align it using **ALIGN**. If you have an odd address on the stack (e.g., by adding an offset that may be odd) you can convert it to the next aligned address using **ALIGNED**.

The Forth virtual machine has two stacks with 32-bit items. Stacks grow downward from high RAM. The return stack is the CPU's subroutine stack, and functions analogously to the traditional Forth return stack (i.e., carries return addresses for nested calls). A program may use the return stack for temporary storage during the execution of a definition, subject to the following restrictions:

- A program shall not access values on the return stack (using **R@**, **R>**, **2R@**, or **2R>**) that it did not place there using **>R** or **2>R**;
- A program shall not access from within a **DO** loop values placed on the return stack before the loop was entered;
- All values placed on the return stack within a **DO** loop shall be removed before **I**, **J**, **LOOP**, **+LOOP**, or **LEAVE** is executed;
- All values placed on the return stack within a definition shall be removed before the definition is terminated or before **EXIT** is executed.

3.1.3 Support for 16-bit or 32-bit Multiply/Divide

Some processors (e.g., those using the 68000 core and the 68060) in the 68K family support only 16-bit multiply/divide operations, while others (e.g., the CPU32 core, the 68020 and 68030) can perform 32-bit multiply/divide operations. SwiftX provides support for both versions. The file **..\68K\Math16** provides low-level Forth multiply and divide operations based on the 16-bit architecture, while **..\68K\Math32** provides the same operators based on a 32-bit architecture. Both versions provide an identical external interface, consistent with the description of the operators *****, **UM***, **M***, **/**, **UM/MOD**, ***/MOD**, ***/**, **/MOD**, and **MOD** in ANS Forth and the *Forth Programmer's Handbook*, where a single-cell integer *n* or *u* is 32-bits wide and a double-cell integer *d* is 64 bits wide. The **Math32** versions will be significantly faster, of course.

To select the version appropriate for your target, edit the main load file **..\68K**<board>**\Kernel.f** to **INCLUDE MATH16** or **INCLUDE MATH32**, as appropriate.

3.1.4 SwiftOS Multitasker Implementation

The M68K CPU family supports a very efficient SwiftOS implementation, with five instructions required to de-activate a task and six to activate one. The subroutine-threaded implementation means there is no **I** register (see *SwiftX Reference Manual*, Section 4.3) to be saved and restored, only the return and data stacks.

As on most systems, a task's **STATUS** user variable contains space for one instruction (two bytes, in this case) followed by the address of the next task in the round robin. There are two instructions that can be used here; each is defined as a **CONSTANT** and is listed in Table 5.

Table 5: SwiftOS user status instructions

Name	Value (hex)	Instruction	Description
WAKE	4E90	**A0) JSR**	Call to wake-up code pointed to by **A0**.
SLEEP	4EF9	**AB JMP**	Jump to next task (address in next cell).

These instructions are stored in a task's **STATUS** to control task behavior. For example, **PAUSE** sets it to **WAKE** and deactivates the task; it will wake up after exactly one circuit through the round robin. You may also store **WAKE** in the **STATUS** of a task in an interrupt routine to set it to wake up. When a task is being started, its **STATUS** is set to **SLEEP** as part of the startup process. Because the return stack is also the CPU's subroutine stack, it is also used to pass information during a task swap. That is, the **JSR** to the wake-up code in the awakening task's **STATUS** passes the task address on this stack.

If you wish to review the simple code for this SwiftOS, you will find it in the file **Swiftx\Src\68K\Tasker.f**.

3.2 68332 Hardware Configuration

The Motorola 68332 has some unique implementation issues. In particular, this processor includes a number of high-level hardware features in a System Integration Module, or SIM, that may be configured in a number of different ways. This section discusses these issues.

3.2.1 Chip-select Management

SwiftX provides a set of mnemonic commands used to specify the chip-select configuration of a 68332 processor. These words provide values that are OR'ed together, as appropriate, to build a table in PROM that is moved to the SIM programmable chip-select circuits. The table has 12 rows, one for each chip-select, in the order CSBOOT, CS0, … CS10.

The table used to control a particular platform is given in the file **Start.f** for that target; for example,

\Swiftx\Src\68K\SBC332\Start.f

In general, set up the registers in each row, from left to right, as defined in the *Motorola System Integration Module User's Manual*:

```
MODE, BYTE, R/W, STRB, DSACK, SPACE, IPL, AVEC
```

Figure 1 shows the default configuration of the chip-select table.

```
\ VESTA SBC332 CHIP SELECT TABLE

CREATE CHIPS ( CSPAR0) 3FFF W,  ( CSPAR1) 3FF W,

   Async  Both   Read   AS  9 WS  S/U  7 IPL       'PROM 128K   \ CSBOOT
   Async  Upper  R/W    AS  0 WS  S/U  7 IPL          0 256K   \ CS0
   Async  Lower  R/W    AS  0 WS  S/U  7 IPL          0 256K   \ CS1
   N/C                                                         \ CS2
   N/C                                                         \ CS3
   N/C                                                         \ CS4
   N/C                                                         \ CS5
   N/C                                                         \ CS6
   N/C                                                         \ CS7
   N/C                                                         \ CS8
   N/C                                                         \ CS9
   N/C                                                         \ CS10

HERE CHIPS - EQU |CHIPS|   IDATA
```

Figure 1. Example of a chip-select table

Each of the twelve chip-select registers controls a particular block of address space, whose base address and size are given in the rightmost entries in the table. For example, in Figure 1, the CSBOOT chip-select controls a block that is 128K in size, starting at a base address given by the constant **'PROM**.

Mode

The mode option controls port synchronization. This option is specified by one of the following words:

Asynch Synch

If **Asynch** is specified, the DSACK field must be used to specify synchronization, with **AS** or **DS**.

Byte

This option controls port-access specifiers (lower or upper byte, or both), as specified by one of the following words:

```
Lower  Upper  Both
```

Read/Write

This option controls read/write access to the block, as specified by one of the following words:

```
Read  Write  R/W
```

STRB

This option controls address strobe and data strobe timing, as specified by one of the following words:

```
AS  DS
```

DSACK

This option controls the Data Strobe Acknowledge, by specifying a number of wait states to be inserted to optimize the bus speed in a particular application, as specified by:

<n> **WS**

where *n* is the desired number of wait states (0–13)

SPACE

This option controls address space checking, as specified by one of the following words:

```
Cpu  Usr  Supv  S/U
```

IPL

This option specifies interrupt priority-level checking:

```
<n> IPL
```

where *n* is the desired interrupt priority level (0–7)

AVEC

This option controls the Autovector Enable. The default is disabled; to enable it, use the word:

```
Avec
```

Block-size Specifiers

Each of the block-size specifiers expects an option register value, which is constructed by the preceding words, and a base address on top of the stack.

```
2K  8K  16K  64K  128K  256K  512K  1M
```

The block-size specifier makes the actual table entry, based on the previous option specifiers.

Always start a chip-select table sequence with **Sync** or **Async**. Alternatively, the word **Unused** specifies that the chip-select is unused, and compiles zeroes for both base and option registers.

The following example compiles a base/option pair to read an async, 16-bit-wide, 64K memory device whose base address is 10000_H:

```
Async  Both  Read  AS  1 WS  S/U  0 IPL   10000 64K
```

An example of a complete table is given in Figure 1. The phrases following the definition **LABEL CHIPS** specify the chip-select pin-assignment registers. The phrase at the end sets the size of the table, whose starting address is given by **CHIPS**.

3.2.2 Additional SIM Configuration

Additional SIM registers may be defined in the file **..\68K\<board>\Start.f** as follows:

```
FFFFFA00 EQU SIMCR    \ Configuration
FFFFFA04 EQU SYNCR    \ Clock Synthesizer Control
FFFFFA20 EQU SYPCR    \ System Protection Control
FFFFFA44 EQU CSPAR0   \ Chip Select Pin Assignment
```

/CPU (defined in the same file) sets up the SIM control registers named above, and outputs the **CHIPS** table to the processor via the Background Debug Mode (BDM) interface, in order to allow a ROM-less target. The default values are shown in Table 6.

Table 6: Default settings for SIM registers

Register	Value	Description
SIMCR	00CF	Watchdog and counters disabled.
SYNCR	7F00	16.777216 MHz clock from 32.768 Hz crystal.
SYPCR	0004	Watchdog off, bus monitor enabled, 64-clock bus timeout.

References Background Debug Mode, Section 3.5

3.3 68332 TIMERS

The 68332 timer uses a 32.768 kHz crystal, and runs at 1024 Hz (1024 interrupts per second). The timer is set up for interrupt vector number 42_H with interrupt request level 1. See Section 4.1.7 of the *MC68332 System Integration Module (SIM) User's Manual* for details.

COUNTER returns the current millisecond count. It (and all other timing commands) adjusts the interrupt count by the ratio 1000/1024 to return true milliseconds. **TIMER**, always used after **COUNTER**, obtains a second millisecond count and subtracts the value left on the stack by **COUNTER**, thereby displaying the elapsed time (in milliseconds) since **COUNTER**. **COUNTER** and **TIMER** may

be used to time processes or the execution of commands. The usage is:

```
COUNTER <process or command to be timed> TIMER
```

3.4 68332 Serial Channel

The queued serial module (QSM) is a major part of the MC68332 microcontroller. It provides high-speed communication to other devices. The QSM contains two serial ports, divided into two sub-modules: the serial communications interface (SCI) and the queued serial peripheral interface (QSPI).

The 68332 SwiftX system provides support for the SCI. The SCI is a full-duplex, universal asynchronous receiver transmitter (UART) interface. It is fully compatible with the SCI systems found in other Motorola MCUs, such as those of the M68HC11 and M68HC05 families.

All data received by the SCI are placed in a 256-byte queue. The baud rate for the SCI can range from 64 baud to 524 Kbaud with a 16.768 MHz system clock.

The SCI is initialized by **/SERIAL** with a default baud rate of 9600. The word **BAUD** may be used to change the baud rate after **/SERIAL** is executed.

The QSPI is not currently supported by the SwiftX/332 system. Refer to Section 5 of the *MC68332 SIM User's Manual* for further information.

3.5 Use of the BDM for XTL Communications

The CPU32 element of the 68332 provides a special mode of operation called Background Debug Mode (BDM), which is unique in that the XTL primitives are implemented in CPU microcode and are accessed through a dedicated serial interface that shares pins with other development features. This provides crash-proof debugging of the target system, while leaving the on-chip, asynchronous serial port free for application use. The definitive documentation for the BDM may be found in Motorola's *CPU32 Reference Manual*, in the section on "Development Support." The following sections discuss the BDM-specific features of this SwiftX implementation.

3.5.1 BDM Operation

When the CPU32 is operating in BDM, normal instruction execution is suspended. The CPU32 accepts commands from a high-speed, full-duplex, synchronous serial interface to provide communication between the CPU and the host development system.

When enabled, the BDM can be initiated by externally generated breakpoints, internal peripheral breakpoints, the background (**BGND**) instruction, and catastrophic exception conditions.

The target system's BDM interface pins are connected to a parallel printer port on the host computer. Through the parallel port's control and status registers, the host's XTL interface can read and write the following target CPU lines:

- RESET
- BKPT/DSCLK
- IPIPE/DSO
- IFETCH/DSI
- FREEZE

The host asserts BKPT and monitors the FREEZE line to determine when the CPU32 enters BDM. While the CPU32 is in BDM, the host toggles DSCLK while shifting data to DSI and from DSO.

This implementation of the XTL uses the following BDM commands:

- Read/write the A/D register
- Read/write system register
- Read/write memory location
- Fill memory block
- Resume execution

The XTL interface provides the following debug functions:

- Display and modify target RAM
- Display target CPU registers
- Download code to target RAM

- Interactively test target Forth words

The **RESET-BDM** command must be issued before BDM can be used. **RESET-BDM** enables BDM by forcing a CPU reset with BKPT asserted. BDM remains enabled until the CPU is reset without BKPT asserted (such as by power-up or by pressing the reset button). Note the use of **RESET-BDM** in **...\68K\Start.f**.

The target-host XTL protocol uses register D7 to pass a function code to the host when the target goes into background mode by executing the **BGND** instruction. The supported function codes are listed in Table 8 on page 50. When a command response (codes 253–255) from the target signals reset, or successful or unsuccessful completion of a command, the target's current **BASE** is in register D6.

References XTL logic, *SwiftX Reference Manual*, Section 4.9.2

3.5.2 BDM XTL Protocol

Use of the BDM significantly changes the Cross-Target Link (XTL) protocol from the serial protocol described in the *SwiftX Reference Manual*.

Whereas a serial protocol implies a table of commands encoded as bytes that are interpreted on the target, use of the BDM by the host to communicate with the target requires no cooperation whatever in the target, and hence no response code. Instead, the host merely asserts commands via the BDM, as shown in Table 7. To give parameters to the target, the host stores them in the target's memory using the WRITE function.

When the Debug facility in SwiftX is launched, a target system is compiled and downloaded using the BDM's FILL function. The target is then activated via its "power-up" sequence, whereupon it transmits its start-up greeting to the host for display. Thereafter, the target is awaiting commands from the host, which it will execute. During the course of its execution, the target may request services (e.g., keyboard input or display functions) from the host. When the target needs host services, or when it has finished processing, it will set up its registers as described in Table 8 and FREEZE.

3.5.2.1 Host-to-target Communication

The host communicates to the target using the defined BDM commands summarized in Table 7. Further details of the BDM protocol may be obtained from the *CPU32 Reference Manual*.

Table 7: Host-to-target commands

Command	Mnemonic	Description
Read A/D register	RARED/RDREG	Read the selected address or data register and return the results via the serial interface.
Write A/D register	WARED/WDREG	The data operand is written to the specified address or data register.
Read system register	RSREG	The specified system control register is read. All registers that can be read in supervisor mode can be read in BDM.
Write system register	WSREG	The operand data is written into the specified system control register.
Read memory location	READ	Read the data at the specified memory location. The source function code register (SFC) specifies the address space.
Write memory location	WRITE	Write the operand data to the specified memory location. The destination function code register (DFC) specifies the address space.
Dump memory block	DUMP	Used in conjunction with READ to dump large blocks of memory.
Fill memory block	FILL	Used in conjunction with WRITE to fill large blocks of memory. Used by SwiftX for the **DOWNLOAD** function.
Resume execution	GO	Resume execution at the current PC. Used by SwiftX when it has completed a requested target function, or to start target execution.
No operation	NOP	A null command. Used by SwiftX as a filler.

The data stack is passed to the target with each host command, and returned with its response. The stack is passed bottom item first. When sent from the

host to the target, the top stack item is the execution address for the function. The target execute the function using **CATCH**. When returned by the target to the host, the top stack item is the function's **THROW** code (0 indicates successful completion).

3.5.2.2 Target-to-host Communication

The target-host XTL protocol uses register D7 to pass a function code from the target to the host when target goes into background mode by executing the **BGND** instruction. Table 8 lists the possible function codes from the target, with other parameters, if any, returned in registers as indicated. Any other value returned in D7 indicates abnormal entry to background mode.

Table 8: Target-to-host responses

Fn	Description	Parameters from target
255	Announce reset	
254	Ack (command completed successfully)	D6 = **BASE**
253	Nak (command aborted)	
252	**EMIT** (display character)	D0 = *char*
251	**TYPE** (display string)	D0 = *length*, D1 = *addr*
250	**CR** (new-line function)	
249	**PAGE** (clear-screen function)	
248	**AT-XY** (cursor position)	D0 = *row*, D1 = *col*
247	**KEY** (input key and send it back)	A6 = cell address of *char*, A5 = *task*
246	**KEY?** (test for keypress, send status back)	A6 = cell address of *flag*, A5 = *task*
245	Display address	D0 = *addr*
244	**ACCEPT** (input string, return actual length)	D0 = *length*, D1 = *addr*, A0 = actual *length* A5 = *task*

These response codes enable the host to provide virtual terminal services to

the target.

If the program under test never returns control to the host, pressing the Escape key will display the message ESCAPE and abort out of the wait loop. This is useful when testing a routine whose behavior is an infinite loop, or when a program behaves unexpectedly.

3.5.3 Register Display and Address Illumination

The presence of the BDM gives this system the ability to examine registers and to handle breakpoint functions, which are not supportable in a serial XTL. The command **R.** (see glossary below) displays the registers; the following is an example of the display it generates:

```
R. CPU RUNNING
 PC= 0000C250   SR= 2000      VBR= 00000000
USP= FE37F3F9  SSP= 0000C24C  SFC= 5         DFC= 5
 D0= 00000026   D1= 00000F63   D2= FFFFFFFF   D3= FFFFFFFF
 D4= FFFFFFFF   D5= FFFFFFFF   D6= 0000000A   D7= 000000FE
 A0= 00000D98   A1= 0000C250   A2= FFFFFFFF   A3= FFFFFFFF
 A4= FFFFFFFF   A5= 0000C250   A6= 0000C0FC   A7= 0000C24C
```

If the CPU was running, it is placed in background mode while this *snapshot* of the register set is displayed.

Register display can be useful if a program crashes during testing. It may be possible through successive **R.** displays to determine what the target is doing. The data and return stack pointers (registers **I** and **W)** and the PC offer primary clues. In a multitasking application, the user pointer (register **U**) is also of interest.

Of course, for this information to be of real use, you would want to know in what definition an address such as the contents of the return stack falls. The word **.'** provides *address illumination*: given an address, it will show what definition the address is in. For example, you could type:

```
0A44 .'
```

and get:

```
DIGIT +06  ok(T)
```

showing that the address is six bytes into the definition **DIGIT**.

Glossary

R. *(—)*

Display target CPU registers and CPU mode—running or FREEZE mode (BDM).

.' *(addr —)*

Display the name of the target definition before *addr*, and *addr*'s offset within that definition.

3.5.4 Breakpoint Support

The BDM makes it possible to support true breakpoints for debugging. Systems without a BDM or equivalent may do software breakpoints, but the very process of causing and processing the breakpoint can corrupt some of the registers or other processor information that might be critical to the debugging effort.

To use **BREAKPOINT**, simply place it at some point in a target definition. **BREAKPOINT** can be placed at any level of nesting in a target definition. When it is executed, it returns control to the host just as if the original word being executed had completed normally. At this point, you may interact with the target to learn whatever is of interest at the breakpoint—taking great caution not to break anything that would prevent completion of the code following the **BREAKPOINT**. For example, you may examine the stack non-destructively by typing **.S**, you may examine the contents of **VARIABLE**s or other data structures, and you may even change the stack or stored data values.

RESUME will return control after the **BREAKPOINT** and allow completion of the target definition. Multiple instances of **BREAKPOINT** may be executed in a series but must not be nested, because the **BREAKPOINT** and **RESUME** sequence is not reentrant.

Glossary

BREAKPOINT *(—)*

Cease executing, saving the execution environment, and return control to the host.

RESUME *(—)*

Resume execution following a **BREAKPOINT**, using the saved execution environment.

References **.S** and other debugging words, *SwiftX Reference Manual*, Section 2.4

4. WRITING I/O DRIVERS

The purpose of this section is to describe approaches to writing drivers for your SwiftX system. The general approach is not significantly different from writing drivers in assembly language or C: you must study the documentation for the device in question, determine how to control the device, decide how you want to use the device for your application, and then write the code.

However, a few suggestions may help you take advantage of SwiftX's interactive character and Forth's ease of interfacing to various devices. We will discuss these in this chapter, with examples from some common devices.

We must assume you have some experience writing drivers in other languages or for other hardware.

4.1 GENERAL GUIDELINES

Here we offer some general guidelines that will make writing and testing drivers easier.

1. **Name your device registers**, usually by defining them with **`CONSTANT`** or **`EQU`**. This will help make your code more readable. It will also help "parameterize" your driver: for example, if you have several devices that are similar except for their hardware addresses, you can write the common control code and pass it a port or register address to indicate a specific device, efficiently reusing the common code.

   Special registers associated with other devices may be named at the beginning of the file containing the driver. For example, the file containing code for the 68332 Periodic Interval Timer (PIT) contains:

```
$FFFFFA22 CONSTANT PICR  \ Periodic interrupt control register
$FFFFFA24 CONSTANT PITR  \ Periodic interrupt timing register
```

2. **Test the device** before writing a lot of code for it. It may not work; it may not be connected properly; it may not work exactly like the documentation says it should. It's best to discover these things before you've written a lot of code based on incorrect information, or have gotten frustrated because your code isn't behaving as you believe it should!

   If you've named your registers and have your target board connected, you can use the XTL to test your device. Memory-mapped registers can be read or written using **C@**, **C!**, **@**, **!**, etc. (depending on the width of the register), and the **.** ("dot") command can be used to display the results. (Usually you want the numeric base set to **HEX** when doing this!) For example, to look at the Port A data register, you could type:

   ```
   PORTA C@ .
   ```

   Try reading and writing registers; send some commands and see if you get the results you expect. In this way, you can explore the device until you really understand it and have verified that it is at least minimally functional.

3. **Design your basic strategy for the device.** For example, if it's an input device, will you need a buffer, or are you only reading single, occasional values? Will you be using it in a multitasked application? If so, will more than one task be using this device? In a multitasked environment, it's often advisable to use interrupt-driven drivers so I/O can proceed while the task awaiting it is asleep, and other tasks can run. An interrupt (or expiration of a count of values read, etc.) can wake the task. If the device is used by just a single task, you can build in the identity of the task to be awakened; if multiple tasks will be using it, you can use a facility variable to control access to the device and to identify which task to awaken. See the section on the SwiftOS multitasker in the *SwiftX Reference Manual* for a discussion of these features.
4. **Keep your interrupt handlers simple!** If you're using interrupts, the recommended strategy is to do only the most time-critical functions (e.g., reading an incoming value and storing it in a buffer or temporary location) and then wake the task responsible for the device. High-level processing can be done by the task after it wakes up.
5. **Respect the SwiftOS multitasking convention** that a task must relinquish the CPU when performing I/O, to allow other tasks to run. This means you

should **PAUSE** in the I/O routine, or **STOP** after setting up streamed I/O that will take place in interrupt code.

4.2 Example: System Clock

SwiftX uses the timer overflow (TOF) interrupt to provide basic clock services. This provides a good example of a simple interrupt routine. The complete source may be found in **Swiftx\Src\68K\Pit.f**.

We would like to be able to set a time of day, and have it updated automatically. Our first design decision is the units to store: milliseconds, seconds, or just a count of clock ticks. Storing clock ticks provides the simplest interrupt code, but requires the routine that delivers the current time of day to convert clock ticks to time units. This is the best approach, as it minimizes the low-level code. Returning time of day is never as time-critical as servicing frequent clock ticks!

This feature has no multitasking impact. No task owns the clock, nor does any task directly interface to it. Instead, the clock interrupt code just runs along, incrementing its counter, and any task that wants the time can read it by calling the word **@NOW** (or one of the higher-level words that calls it).

Our counter will be two cells, one containing a millisecond counter and the other a fractional component. With a millisecond tick rate, it will roll over every 49 days. This means it can be used to time intervals of up to 49 days, in addition to returning the time of day. The interrupt routine has to pick up the low-order cell and increment it; if it overflows, the high-order part must be incremented. The interrupt routine looks like this:

```
2VARIABLE TICKS \ Holds the 32-bit TOF tick interrupt count.

LABEL <TIMER>   \ Periodic timer interrupt handler
   TC1 # MSECS CELL+ AB ADD
   CS IF   1 #Q MSECS AB ADD   THEN
   TC0 [IF]  TC0 # MSECS AB ADD  [THEN] \ Accumulate milliseconds
   1 #Q TICKS CELL+ AB ADD              \ Accumulate date/time
   CS IF   1 #Q TICKS AB ADD   THEN
   RTE   END-CODE

\ Attach <TIMER> to the timer overflow exception:
   <TIMER> $42 EXCEPTION
```

4.3 EXAMPLE: SERIAL I/O

Serial I/O provides a somewhat more complex example. The Forth language provides a standard Application Programming Interface (API) for serial I/O, with commands for single-character input and output (**KEY** and **EMIT**, respectively), as well as for stream input and output (**ACCEPT** and **TYPE**, respectively).

In SwiftOS, we assume that a terminal task may have a serial port attached to it. The serial I/O commands are vectored in such a way that a definition containing **TYPE**, for example, will output its string to the port attached to the task, executing the definition using that task's vectored version of **TYPE**. Thus, you can write a definition that produces some kind of display and, if a task attached to a CRT executes it, the output will go on the screen; but if a task controlling a printer executes it, the text will be printed.

Our mission in writing a serial driver is to provide the *device-layer versions* of these standard routines. As shipped, SwiftX includes a serial driver that communicates to a "dumb terminal" (or terminal emulator) directly connected to the SCI serial port on a 68332. Its driver may be found in **Swiftx\Src\68K\Sciterm.f**.

There are two basic approaches to implementing serial drivers in Forth, which differ depending on whether the primitive layer is single-character I/O or streamed I/O. In the first case, the primitives support **KEY** and **EMIT**; **ACCEPT** then consists of **KEY** inside a loop, and **TYPE** consists of **EMIT** inside a loop. In the second case, **KEY** and **EMIT** call **ACCEPT** and **TYPE**, respectively, with a count of 1. Both approaches are valid: single-character I/O is simpler to implement, but streamed I/O is optimal for systems on which many tasks may be performing serial I/O concurrently. The driver discussed here uses single-character I/O.

The first basic decision to make is whether the I/O will be interupt driven or polled. A polled driver checks the device status to see whether an event has occurred (e.g., a character has been received or is ready for output), whereas an interrupt-driven approach relies on an interrupt to signal that an event has occurred. Interrupt-driven drivers tend to have less overhead, but polled drivers are easier to implement and test.

In addition, the nature of the device has some bearing. For example, incoming keystrokes from a keyboard or pad are relatively infrequent (occurring at

human, rather than computer, speeds); if polling were used, the routine would check many, many times before the next character arrives, thus creating needless overhead. On the other hand, when sending a string of characters, the next character is ready as soon as the last one is gone (which will be quickly). For such situations, we would use polled output and interrupt-driven input.

In the sections that follow, we'll show how we would write and test the device-layer serial I/O words, and then show how to connect them to the high-level words in the serial API.

4.3.1 Polled Serial Output

To output a single character, all you have to do is verify that the port is ready, and then write the character to the port.

We can test the port easily, by writing a very simple word to output a single character:

```
CODE SPIT ( char -- )   S )+ D0 MOV   D0 SCDR AB W. MOV
    RTS  END-CODE
```

Such a word could be edited into a file, but it may be just as easy to type it at the keyboard, if your XTL is active. You can try it immediately:

```
65 SPIT
```

This should send an A character to whatever is connected to SCI 1 (e.g., a terminal emulator). If this works, we then must take care of the fact that, if we're calling it repeatedly in a loop (for streaming output), the port may not always be ready. That can be handled this way:

```
CODE SPITS ( char -- )
   BEGIN   0 # SCSR AB BTST \ Check for port ready
      0= NOT UNTIL          \ Repeat till ready
   S )+ D0 MOV              \ Get character
   D0 SCDR AB W. MOV        \ Output character
   RTS   END-CODE
```

You can test this by putting it in a loop:

```
: GO   ( addr n -- )      \ Output n chars from addr
   0 DO  DUP C@ SPIT  1+ LOOP  DROP ;

PAD 50 65 FILL            (puts 50 A characters at PAD)
PAD 50 GO                 (should output 50 A characters)
```

All that remains is to fulfill the requirement that I/O words should relinquish control of the CPU for the multitasker. Since the time when other tasks potentially could run is in the polling loop, while we are waiting for input, that is where we should give other tasks the opportunity to run. The final primitive word, then, becomes:

```
CODE (SCI-EMIT) ( char -- )    \ Output chars
   BEGIN  ' PAUSE CALL       \ Pause for other tasks
   0 # SCSR AB BTST          \ Check for port ready
      0= NOT UNTIL           \ Repeat till ready
   S )+ D0 MOV               \ Get character
   D0 SCDR AB W. MOV         \ Output character
   RTS   END-CODE
```

The low-level word for **TYPE** is simply the single-character **EMIT** behavior, **(SCI-EMIT)** in this case, in a loop:

```
: (SCI-TYPE) ( addr u -- )  \ Output u chars from addr
   0 ?DO                    \ Repeat for u chars
      COUNT (SCI-EMIT)      \ Output next char
   LOOP DROP ;              \ Done; discard addr
```

Note that the use of **COUNT** here takes advantage of the literal behavior of the word: it takes an address, and returns a byte from that address plus the address of the next byte. Although **COUNT** is designed to return the length and address of a counted string (whose length is in its first byte), it is also perfect for running through a string, as in this case.

References
Principles of serial I/O in Forth, *Forth Programmer's Handbook*, Section 3.8
COUNT, *Forth Programmer's Handbook*, Section 2.3.5.2
PAUSE and multitasker requirements, *SwiftX Reference Manual*, Section 4

4.3.2 Interrupt-driven Queued Serial Input

Input is somewhat more complex than output. Rather than reproduce the entire code here, we will ask that you refer to the file **`Swiftx\Src\68K\Sci332.f`** as you read these notes.

We usually need to buffer incoming characters. We certainly don't want to miss a character because we were busy when it appeared on the interface. So the input side of this driver will have a 256-byte buffer, plus four pointers used to manage the process. The buffer is called **`SCI-RQ`** (SCI Receive Queue). The pointers are defined as offsets into the buffer; the actual data begins at **`SCI-RQ`** + 4. The layout is given in Table 9.

Table 9: Input queue pointers

Offset name	Value	Description
`RIN`	0	Offset for next received byte
`ROUT`	2	Offset to next byte to remove
`RTASK`	4	Task to be awakened
`RDATA`	8	Start of actual data

As is common in SwiftOS, we separate interrupt-level processing from task-level processing, doing only the bare minimum at interrupt time. In this case, the interrupt code fetches the character from the input port and puts it in the next input location, indicated by **`RIN`**, and awakens the task responsible for the port. This is done by **`<SCI>`** (just as the convention of names in parentheses is used to indicate the low-level components of high-level functions, the convention of names in angled brackets is used to indicate interrupt routines).

The phrase:

```
<SCI> 40 EXCEPTION
```

attaches the code to the exception vector assigned to the SCI.

A dummy task called **`NOTASK`** is provided; it is only a variable, not a real task, because it only serves as a place for the interrupt code to store the "wakeup" value if an interrupt occurs when no task is asleep awaiting character input

from this port.

The balance of the processing of incoming characters is done by task-level code. There are three code primitives, described in the glossary at the end of this section.

Although **(SCI-KEY?)** is sufficient as a primitive for **KEY?**, both **(SCI-AWAIT)** and **(SCI-READ)** are required for **KEY**. This is because **KEY** must wait until a key is received, and in the SwiftOS multitasking environment the waiting must be done in an inactive state, so other tasks can run. The definition of **(SCI-KEY)**, then, is:

```
: (SCI-KEY) ( -- char)  (SCI-AWAIT) PAUSE (SCI-READ) ;
```

The task will be suspended in **(SCI-AWAIT)** until a key is available, at which time it will **PAUSE** (ensuring that there is at least one **PAUSE**, even if a key was already in the queue) and then read the key.

Glossary

(SCI-KEY?) *(— flag)*
Returns a flag that is true if a character has been received. The primitive for **KEY?**.

(SCI-AWAIT) *(—)*
Sets **RTASK** to the task executing this word, and checks the queue. If there are no characters, suspends the task. If there's at least one character, sets **RTASK** to **NOTASK** and returns.

(SCI-READ) *(— char)*
Returns the next character from the queue.

4.3.3 Port and Task Initialization

All that remains is to provide port initialization, and to attach these device-layer functions to an actual SwiftOS task.

The word **/SCI** in the file **Swiftx\Src\68K\Sciterm.f** initializes the port (the naming convention `/name` is often used for words that initialize a port,

device, or function *name*). As usual with initialization routines, it is intended to be called in the high-level **START** routine in the file **Swiftx\Src\68K**<board>**\Start.f**. **/SCI** initializes the buffer pointers and sets a default baud rate and port control bits.

SCI-TERMINAL in **Swiftx\Src\68K\Sciterm.f** is intended to be executed by a terminal task to set its vectored serial I/O functions to the device-layer versions for the SCI. As an example of how this is done, look at the definition of the terminal task set up to run the demo application at the end of **Swiftx\Src\68K**<board>**\Debug.f**:

```
256 TERMINAL CONSOLE          \ Define a task

: DEMO   CONSOLE ACTIVATE     \ Start task
   DUMB SCI1-TERMINAL         \ Initialize task vectors
   BEGIN  CALCULATE  AGAIN ;\ Run CALCULATE indefinitely
```

Here, the command **DEMO** starts the task **CONSOLE** executing the balance of the definition following **ACTIVATE**; it performs the initialization steps **DUMB** and **SCI-TERMINAL**, and then enters an infinite loop performing the **CALCULATE** demo routine.

APPENDIX A: NMIX-0332 BOARD INSTRUCTIONS

This section provides information pertaining to the NMIX-0332, which is supported by SwiftX for the M68K family. It includes instructions for setting up your board and connecting it to your host PC, as well as additional information specific to the implementation of the SwiftX kernel and SwiftOS on this board.

A.1 BOARD DESCRIPTION

The NMIX-0332 is a fully configured development system for the Motorola 68332 microcontroller, designed for prototyping and development.

Features include:

- M68332 CPU (up to 20 MHz)
- 64K RAM (two sockets, each configurable for up to 1 Mb RAM or 2 Mb PROM)
- Programmable wait states for external memory
- One RS232 channel
- High-speed parallel host-target communications using BDM
- AC or DC power
- All bus signals brought out to headers
- Prototyping area

A.2 BOARD CONNECTIONS

We strongly advise you to set up and use the test board supplied with this system until you are familiar with SwiftX, even if your actual target hardware is

different. Once you understand the interactive development environment, you will find it much easier to compile and install the software on your specialized hardware.

Installing SwiftX on the NMIX-0332 requires only a small amount of effort, but proceed with care, because a mistake could damage the board.

Caution—Do not put the board on a conducting surface when power is applied to it.

This product as shipped by FORTH, Inc. is supplied with an NMIX-0332 board, a 5V power supply, and a communications cable. This cable is deliberately short, because it is a parallel cable. If you extend it, the system may not run properly. The board is shipped configured and tested for use with SwiftX. To complete the assembly and connections, follow these steps.

1. Plug the power-supply cable into the NMIX-0332 (the connector is labeled J10, located in the upper-left corner of the board).
2. Attach the serial cable to J6 on the NMIX-0332 (with pin 1 located on the side closest to the edge of the board and nearest to J12).
3. Connect the communications cable between the Background Debug Mode (BDM) interface port (pins 1–10 of J12) on the NMIX-0332 and a parallel port on your PC. Any parallel port will work; in the booting process, SwiftX will determine which one you used. Be sure that pin 1 of the DIP connector (marked by a small triangle and using the brown wire) is connected to pin 1 of J12.
4. Plug the power supply into 110VAC, thus powering up the NMIX-0332. You may now run your SwiftX system, as described in the following sections.

If you purchase the NMIX-0332 directly from the manufacturer rather than from FORTH, Inc., it may not ship with a BDM cable or with the BDM header on the board at J12. If you must build a cable, the following chart shows the correct configuration:

BDM	10-pin header	DB-25	Port bit
	1 brown		
-berr	2 red	9 D7	pd7
	3 orange	25 GND	
-bkpt	4 yellow	3 D1	pd1

BDM	10-pin header	DB-25	Port bit
	5 green	18 GND	
freeze	6 blue	10 -ack	ps6
-reset	7 purple	5 D3	pd3
dsi	8 gray	2 D0	pd0
	9 white		
-dso	10 black	11 BUSY	ps7

The BDM header should occupy the first ten pins of J12. The next chart describes how the BDM cable connects to this header:

/DS 1	2 /BERR
GND 3	4 /BKPT
GND 5	6 FREEZE
/RST 7	8 IPIPE1
+5 9	10 IPIPE0
SIZ0 11	12 /TSTME
SIZ1 13	14 RMC
DSACK0 15	16 /AVEC
DSACK1 17	18 PC2
AS 19	20 CS4
CS1 21	22 NC
AS64 23	24 CLKOUT
HALT 25	26 MODCLK

A.3 Development Procedures

On the NMIX-0332, the SwiftX kernel resides in RAM, downloaded and managed using the CPU32's Background Debugging Mode (BDM). As a result, procedures for preparing and installing new kernels may differ from those described in the generic SwiftX manual and from those for other 68K family processors.

A.3.1 Starting a Debugging Session

Launch SwiftX as described in Section 1.3.1 of the *SwiftX Reference Manual*. You may change any options you wish and, when you are ready, you may compile your kernel by selecting Project > Debug from the menu. This completely compiles the kernel and downloads it into the target using the BDM.

If the board is not connected properly, you will get the message, `No Target`. This means the kernel was properly compiled, but the host failed to establish XTL communication with the target. Check your connections and select Project > Debug again.

If the connection was successfully established, the target will display the system ID and its creation date. At this point, you may directly execute words on the target, examine its memory or I/O ports, and define additional words which will be downloaded automatically for testing. To try it, type:

```
2 6 + .
```

The numbers you typed will be transmitted to the target, followed by the commands **+** and **.** (the command "dot," which types a number). These commands will be executed on the board, which will add the numbers and display the sum. The sum will be displayed on your screen, because your PC is providing the target's keyboard and display functions.

Whether or not you are connected to a target, you may use **LOCATE**, **DUMP**, and many other debugging commands, described in Section 2.4 of the *SwiftX Reference Manual*.

A.3.2 Using SwiftX With Other Hardware

If you are using a target system other than the NMIX-0332 for your final application, you may need to make some changes to the parameters of your kernel. The configuration procedure described in the *SwiftX Reference Manual*, Section 4.2, will help you to do this.

If your host will use the BDM on the target, you do not need to make any procedural changes. However, if your target will execute out of PROM, you will

need to burn a PROM. The Project > Build menu item compiles a target image and writes it in a file suitable for burning into a PROM.

To prepare a target image for PROM:

1. Select the Project > Build menu item—this generates a new **`Target.s19`** object file. Exit from SwiftX.
2. Use a PROM programmer utility of your choice to burn a new PROM using this file.
3. Turn off power to the board, and change the PROM. Apply power to the board.
4. Launch SwiftX and continue as described in Section A.3.1.

A.3.3 Running the Demo Application

As shipped, the interactive testing program **`Debug.f`** (loaded by the Project > Debug menu item) is configured to run a demo application described in the *SwiftX Reference Manual*.

The demo program may use either the host keyboard and screen via the XTL (the default configuration), or you may connect the NMIX-0332's serial port to a COM port on your host and set up a separate task in the target to talk to a standard terminal emulator utility in the target, as described in Section 4.8.

To run the demo program, select Project > Debug to bring up the target program. Type **`CALCULATE`** to start the application if you want it to use the XTL. The demo is automatically started on COM2 when Debug is launched.

Thereafter, follow the instructions in the *SwiftX Reference Manual*.

A.4 Board-level Implementation Issues

This sections describes features of the SwiftX implementation that are specific to the NMIX-0332.

A.4.1 System Hardware Configuration

The NMIX-0332 board provides 64K of RAM, and provision for a PROM at 80000_H. SwiftX as delivered runs in RAM. Its default memory configuration is shown in Figure 2.

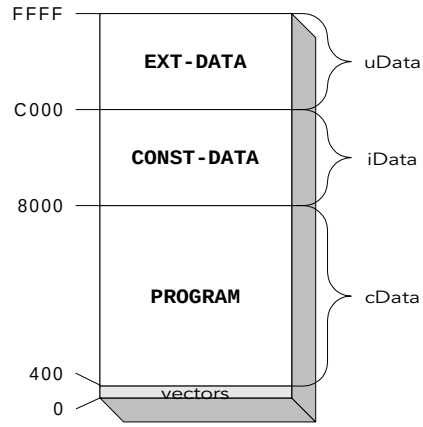

Figure 2. RAM memory allocation in the NMIX-0332

As shipped, the SwiftX system uses about 6.4K bytes of code space and 1.2K bytes of uData. Memory configuration is set in the **`..\NMIX0332\Config.f`** file. You may adjust this if necessary, to suit your needs.

A.4.2 Serial Port Support

As discussed in Section 3.4, this version of SwiftX supports a serial terminal on the serial communications interface (SCI) which is part of the 68332's queued serial module. Its driver may be found in **`..\68K\Sci332.f`**, which includes **`EMIT`**, **`TYPE`**, **`KEY?`**, **`KEY`**, and **`ACCEPT`** behaviors for a task associated with this port. Rules for use:

1. Define a task for the serial port, as described in Section 4.7.1 of the *SwiftX Reference Manual*.

2. In your one-time initialization, **CONSTRUCT** the task and call **/SCI** to initialize the port. Change the definition of **(BAUD)** for the desired initial baud rate.
3. In the task's start-up word, assign the **SCI-TERMINAL** vectors and a terminal type (such as the "dumb terminal" in **SwiftX\Src\Dumb.f**).

   If you write your code using the standard Forth versions of words like **EMIT**, **TYPE**, **KEY?**, **KEY**, and **ACCEPT**, they will be executed using the versions provided for the task executing them. That is, a word that does a **TYPE** will output using the host's SwiftX command window if it is executed by the XTL task, and will use the SCI output words if it is executed by a task configured for that port using the procedure outlined above.

   An example is provided for running a multitasking version of the demo program included with SwiftX:

```
256 TERMINAL CONSOLE          \ Define a task

: DEMO   CONSOLE ACTIVATE     \ Start task
   DUMB SCI-TERMINAL          \ Initialize task vectors
   BEGIN  CALCULATE  AGAIN ;\ Run CALCULATE indefinitely
```

   This defines a terminal task named **CONSOLE**. **DEMO** causes **CONSOLE** to initialize itself with the **DUMB** terminal type (providing versions of the vectored words **CR**, **TAB**, etc., for a simple ANSI terminal), sets the SCI driver routines for **TYPE**, **KEY**, etc., and then leaves it in an infinite loop running the conical pile calculator demo program.

APPENDIX B: VESTA SBC332 BOARD INSTRUCTIONS

This section provides information pertaining to the Vesta SBC332, which is supported by SwiftX for the M68K family. It includes instructions for setting up your board and connecting it to your host PC, as well as additional information specific to the implementation of the SwiftX kernel and SwiftOS on this board.

B.1 BOARD DESCRIPTION

The Vesta SBC332 is a fully configured development system for the Motorola MC68332 microcontroller. The system is supplied with a parallel BDM cable, printed hardware manual, and power supply.

B.2 BOARD CONNECTIONS

We strongly advise you to set up and use the test board supplied with this system until you are familiar with SwiftX, even if your actual target hardware is different. Once you understand the interactive development environment, you will find it much easier to compile and install the software on your specialized hardware.

Installing SwiftX on this test board requires only a small amount of effort, but proceed with care, because a mistake could damage the board.

Caution—Do not put the board on a conducting surface when power is applied to it.

This product is supplied with a Vesta SBC332 board, a 5V power supply, and a communications cable. This cable is deliberately short, because it is a parallel

cable. If you extend it, the system may not run properly. The board is shipped configured and tested for use with SwiftX. To complete the assembly and connections, follow these steps.

1. The power-supply connecting cable is terminated in a two-pin connector. Carefully note the + marking on this connector, and plug the connector into J1 on the SBC332 (in the upper-left corner of the board) with the correct polarity.

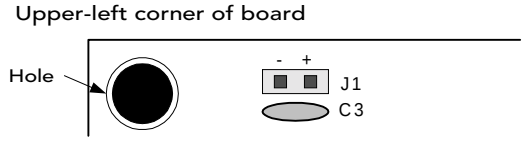

Figure 3. Power Connector J1 on the SBC332

2. Connect the communications cable between a parallel port on your PC and the background mode (BDM) interface port (J8) on the SBC332. Any parallel port will work; in the booting process, SwiftX will determine which one you used. Be sure that pin 1 of the DIP connector (marked by a small triangle and using the brown wire) is connected to pin 1 of J8 (marked with an asterisk). The two bottom pins on the cable's 10-pin DIP connector are not used and do not plug into the 8-pin J8.
3. Plug the power supply into 110VAC, thus powering up the SBC332. You may now start running your SwiftX system, as described in the following sections.

B.3 Development Procedures

On the Vesta SBC332, the SwiftX kernel resides in RAM, downloaded and managed using the CPU32's Background Debugging Mode (BDM). As a result, procedures for preparing and installing new kernels may differ from those described in the generic SwiftX manual and from those for other 68K family processors.

B.3.1 Starting a Debugging Session

Launch SwiftX as described in Section 1.3.1 of the *SwiftX Reference Manual*. You may change any options you wish and, when you are ready, you may compile your kernel by selecting Project > Debug from the menu. This completely compiles the kernel and downloads it into the target using the BDM.

If the board is not connected properly, you will get the message, `No Target`. This means the kernel was properly compiled, but the host failed to establish XTL communication with the target. Check your connections and select Project > Debug again.

If the connection was successfully established, the target will display the system ID and its creation date. At this point, you may directly execute words on the target, examine its memory or I/O ports, and define additional words which will be downloaded automatically for testing. To try it, type:

```
2 6 + .
```

The numbers you typed will be transmitted to the target, followed by the commands **+** and **.** (the command "dot," which types a number). These commands will be executed on the board, which will add the numbers and display the sum. The sum will be displayed on your screen, because your PC is providing the target's keyboard and display functions.

Whether or not you are connected to a target, you may use **LOCATE**, **DUMP**, and many other debugging commands, described in Section 2.4 of the *SwiftX Reference Manual*.

B.3.2 Using SwiftX With Other Hardware

If you are using a target system other than the SBC332 for your final application, you may need to make some changes to the parameters of your kernel. The configuration procedure described in the *SwiftX Reference Manual*, Section 4.2, will help you to do this.

If your host will use the BDM on the target, you do not need to make any procedural changes. However, if your target will execute out of PROM, you will

need to burn a PROM. The Project > Build menu item compiles a target image and writes it in a file suitable for burning into a PROM.

To prepare a target image for PROM:

1. Select the Project > Build menu item—this generates a new **`Target.s19`** object file. Exit from SwiftX.
2. Use a PROM programmer utility of your choice to burn a new PROM using this file.
3. Turn off power to the board, and change the PROM. Apply power to the board.
4. Launch SwiftX and continue as described in Section B.3.1.

B.3.3 Running the Demo Application

As shipped, the interactive testing program **`Debug.f`** (loaded by the Project > Debug menu item) is configured to run a demo application described in the *SwiftX Reference Manual*.

The demo program may use either the host keyboard and screen via the XTL (the default configuration), or you may connect the SBC332's serial port to a COM port on your host and set up a separate task in the target to talk to a standard terminal emulator utility in the target, as described in Section 4.8.

To run the demo program, select Project > Debug to bring up the target program. Type **`CALCULATE`** to start the application if you want it to use the XTL. The demo is automatically started on COM2 when Debug is launched.

Thereafter, follow the instructions in the *SwiftX Reference Manual*.

B.4 Board-level Implementation Issues

This sections describes features of the SwiftX implementation that are specific to the Vesta SBC332.

B.4.1 System Hardware Configuration

The Vesta SBC332 board provides 64K of RAM, and provision for a PROM at 80000_H. SwiftX as delivered runs in RAM. Its default memory configuration is shown in Figure 4.

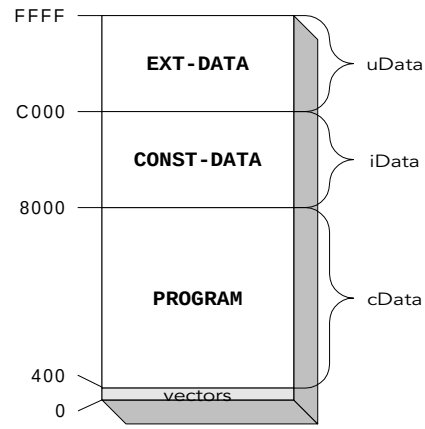

Figure 4. RAM memory allocation in the Vesta SBC332

As shipped, the SwiftX system uses about 6.4K bytes of code space and 1.2K bytes of uData. Memory configuration is set in the **`..\Sbc332\Config.f`** file. You may adjust this if necessary, to suit your needs.

B.4.2 Serial Port Support

As discussed in Section 3.4, this version of SwiftX supports a serial terminal on the serial communications interface (SCI) which is part of the 68332's queued serial module. Its driver may be found in **`..\68K\Sci332.f`**, which includes **`EMIT`**, **`TYPE`**, **`KEY?`**, **`KEY`**, and **`ACCEPT`** behaviors for a task associated with this port. Rules for use:

1. Define a task for the serial port, as described in Section 4.7.1 of the *SwiftX Reference Manual*.

2. In your one-time initialization, **CONSTRUCT** the task and call **/SCI** to initialize the port. Change the definition of **(BAUD)** for the desired initial baud rate.
3. In the task's start-up word, assign the **SCI-TERMINAL** vectors and a terminal type (such as the "dumb terminal" in **SwiftX\Src\Dumb.f**).

   If you write your code using the standard Forth versions of words like **EMIT**, **TYPE**, **KEY?**, **KEY**, and **ACCEPT**, they will be executed using the versions provided for the task executing them. That is, a word that does a **TYPE** will output using the host's SwiftX command window if it is executed by the XTL task, and will use the SCI output words if it is executed by a task configured for that port using the procedure outlined above.

   An example is provided for running a multitasking version of the demo program included with SwiftX:

```
256 TERMINAL CONSOLE          \ Define a task

: DEMO   CONSOLE ACTIVATE     \ Start task
   DUMB SCI-TERMINAL          \ Initialize task vectors
   BEGIN  CALCULATE  AGAIN ;\ Run CALCULATE indefinitely
```

   This defines a terminal task named **CONSOLE**. **DEMO** causes **CONSOLE** to initialize itself with the **DUMB** terminal type (providing versions of the vectored words **CR**, **TAB**, etc., for a simple ANSI terminal), sets the SCI driver routines for **TYPE**, **KEY**, etc., and then leaves it in an infinite loop running the conical pile calculator demo program.

GENERAL INDEX