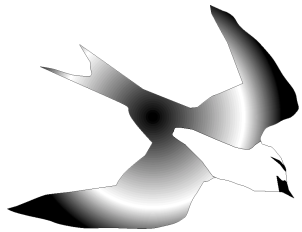

Getting Results Faster...

SwiftX 69R000

for UTMC UT69R000 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 6/28/00

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 UT69R000 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 UT69R000; 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 UT69R000. Refer to UTMC's documentation for information about the UT69R000, 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
EVB	Appendix A: UTMC Evaluation Board Configuration and Use	29

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 and to an uninstall utility. (The uninstall option is provided here in Windows 3.1 only; if you are running Windows 95, you can use the Remove button on the Start > Settings > Taskbar... > Start Menu Programs dialog.)

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 UT69R000 ASSEMBLER

The UT69R000 is a monolithic CMOS, 16-bit microcontroller. Operating at two clock cycles per instruction, this processor provides high throughput rates for real-time systems. The UT69R000 is based on a Harvard architecture with separate non-multiplexed instruction and data ports, each 16 bits wide. The register set includes 20 general-purpose 16-bit registers the designer can use in pairs for 32-bit operations, and several special-purpose registers. A built-in UART provides a serial communication port. The UT69R000 employs a load/store architecture. This means that the only references to data memory come from load or store instructions. All other instructions operate with only register and immediate data operands.

Throughout this book, we assume you understand the hardware and functional characteristics of the UT69R000 microcontroller as described in the UTMC *UT69R000 Data Sheet* and *Assembly Language Manual*. We also assume you are familiar with the basic principles of Forth.

This section supplements, but does not replace, the manufacturer's manuals. Departures from the manufacturer's usage are noted here; nonetheless, you should use the manufacturer's manuals for a detailed description of the instruction set and addressing modes.

Where **`boldface`** type is used here, it distinguishes Forth words (such as register names) from UTMC names. Usually these are the same; for example, the name **`PUSH`** can be used as a Forth word and as a UTMC mnemonic. Where boldface is *not* used, the name refers to the manufacturer's usage or to hardware issues that are not particular to SwiftX or Forth.

2.1 SwiftX Assembler Principles

Assembly routines are used to implement the Forth kernel, to perform direct I/O operations when desired, and to optimize the performance of interrupt handlers and other time-critical functions.

The SwiftX UT69R000 cross-compiler provides an assembler for the UTMC UT69R000 microcontroller. The mnemonics for the UT69R000 opcodes have been defined as Forth words which, when executed, assemble the corresponding opcode at the next location in code space.

Most instructions use the manufacturer's mnemonics, but postfix notation and Forth's data stack are used to specify operands. Words that specify registers, addressing modes, memory references, literal values, etc. *precede* the mnemonic.

Some of the manufacturer's mnemonics have been replaced by different names, plus options to describe operations. The principal use of this strategy is in connection with conditional jumps. SwiftX constructs conditional jumps by using a condition code specifier followed by **IF**, **UNTIL**, or **WHILE**. Table 4 on page 11 summarizes the relationship between SwiftX condition codes used with one of these words and the corresponding UTMC mnemonic.

References Assemblers in Forth, *Forth Programmer's Handbook*, Sections 1.3 and 4.0

2.2 Code Definitions

Code definitions normally have the following syntax:

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

For example:

```
CODE DEPTH ( -- n)      \ Return the current stack depth
   S R0 MOV   U R1 MOV   S0 U) R1 ADD   R1 R1 LR
   R0 R1 SUB   TPUSH   R1 T MOV
RET END-CODE
```

All code definitions must be terminated by the command **END-CODE**.

As an alternative to the normal **RET**, whose behavior is to execute the next word, the phrase:

```
WAIT X JC
```

may be used before **END-CODE** to terminate a routine by an unconditional jump through the SwiftOS multitasking executive, leaving the current task idle and passing control to the next task.

You may name a code fragment or subroutine using the form:

LABEL <name> <assembler instructions> **END-CODE**

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 or call, for example. The code fragments used as interrupt handlers are constructed in this way, and the named locations are then passed to **INTERRUPT**, which connects the code address to a specified interrupt vector.

The critical distinction between **LABEL** and **CODE** is:

- 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.
- 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; if the definition is referenced interpretively while connected to a target, it will be executed.

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.

WAIT *(— addr)*

Return the address of the multitasker entry point that deactivates the current task. Used as a code ending (instead of **RET**) at the end of code that initiates an I/O operation, where the interrupt that signals completion of the I/O will be used to wake up the task.

END-CODE *(—)*

Terminate an assembler sequence started by **CODE** or **LABEL**.

References Interrupt handling, Section 3.3
SwiftOS multitasking executive, *SwiftX Reference Manual*, Section 5

2.3 REGISTERS

UT69R000 registers are shown in Figure 1. The 20 data registers **R0** through **R19** hold word (cell) data; even/odd register pairs form 32-bit long-word registers, arranged as shown in Figure 1. The Accumulator **ACC** is used for several purposes: remainder for divides, product for multiplies, extended shift register, and as a pointer for instruction memory loads and stores.

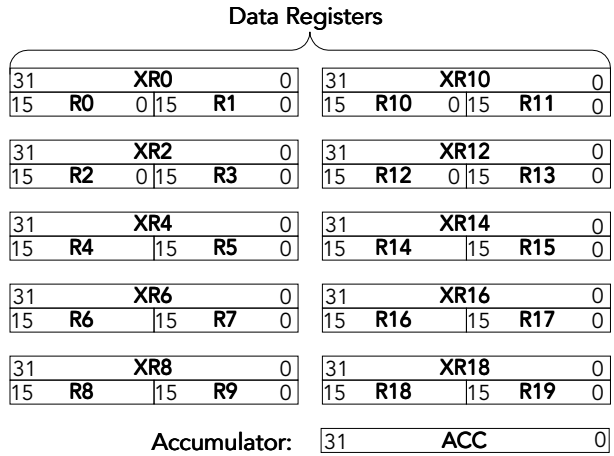

Figure 1. UT69R000 Registers

Special purpose registers are described in UTMC documentation. The registers used with **INR** and **OTR** instructions are given names as in the descriptions for those instructions. For example, you might use:

```
RCVR R0 INR
```

to read from the on-board UART's receiver register into **R0**.

Certain data registers are used specially in SwiftX, and are therefore given names indicating their function, as shown in Table 2. These may not be used except for their intended purpose without saving and restoring. Other registers are available as scratch registers, and do not need to be saved or restored.

Table 2: Register designations in SwiftX

Register	SwiftX Name	Description
R19	R	Subroutine return register (return stack pointer)
R18	R'	Extension for upper half of R
XR18	XR	Double R
R17	I	Interrupt return register
R16	I'	Extension for upper half of I
XR16	XI	Double I
R15	T	Top item on data stack
R14	T'	Extension for upper half of T
XR14	XT	Double T
R13	S	Forth data stack pointer
R12	U	User pointer

References Register usage in SwiftX implementation, Section 3.1.1

2.4 Addressing Modes

The notation for specifying addressing modes in SwiftX differs from common assembler notation, in that the mode specifiers are operands that precede the mnemonics. In this UT69R000 assembler, instruction mnemonics are words

that actually assemble opcodes using parameters left on the stack by the mode operands. Note that the syntax is <operand(s)> <opcode> and the order of operands is always <source> <destination>. SwiftX assembler modes and operands are separated by spaces.

All UT69R000 instructions require a source operand and a destination operand, either explicit or implied. Depending upon the instruction, the source operand may be:

- a general data register (**R0**–**R19**, **XR0**–**XR18**)
- a special purpose register (**ACC**, **SP**, etc.)
- a literal (two's complement five-bit value)
- an immediate/absolute (16-bit immediate value or address, stored as the second word of the instruction)

The destination operand must be a data register (**R**<n>, **XR**<n>) for all instructions except the **BR** and **JC** instructions.

Note that in the case of **OTR** and **STR** instructions, the "source" operand specifies the destination of the operation. For example:

```
R0 R1 STR
```

writes the contents of **R1** to the location in data memory whose address is in **R0**.

Table 3 lists the addressing modes of the UT69R000 and provides examples that show the difference between UTMC and SwiftX assembler notation.

Table 3: Addressing modes

Mode	UTMC	SwiftX
Data Register Direct	`ADD R0,R1`	`R1 R0 ADD`
Literal	`ADD R0,1`	`1 R0 ADD`
Immediate Long	`ADD R0,100`	`100 R0 ADD`
Data Register Indirect	`LR R0,R1`	`R1 R0 LR`
Stack Pointer Indirect	`PUSH R0`	`R0 PUSH`
Absolute	`LR R0,1234`	`1234 R0 LR`

The source operand passed to the bit shift mnemonics (**SAR**, **SCR**, **SLR**) specifies or contains the direction and amount of the shift (negative is a right shift, 0 or greater is a left shift). For further details, see the description of **SAR** in the *UT69R000 Assembly Language Manual*.

The source operand passed to bit operation mnemonics (**RBR**, **SBR**, **TBR**) specifies a bit position, with 31 specifying the *least* significant bit (LSB). For further details, see the description of **RBR** in the *UT69R000 Assembly Language Manual*.

2.5 DIRECT TRANSFERS

In Forth, most control transfers are performed using structures (such as those described in Section 2.6) and code endings (described in Section 2.2). Good Forth programming style involves many short, self-contained definitions (either code or high level), without the unstructured branching and long code sequences that are characteristic of conventional assembly language. The Forth approach is also consistent with principles of structured programming, which favor small, simple modules with one entry point, one exit point, and simple internal structures. However, direct transfers are useful at times, particularly when compactness of the compiled code overrides all other criteria.

The **BR** instruction is not implemented as a mnemonic. Instead, the **JC** mnemonic will automatically assemble a **BR** if the branch range can be represented as a five-bit literal. The destination for **JC** is specified as an absolute address, which the assembler will convert to an offset. All condition code mnemonics specified in UTMC documentation (e.g., *UT69R000 Assembly Language Manual*, Table 3) are implemented for use with **JC**. More "Forth-like" condition specifiers **0<**, etc., are also available for use with the structures described in Section 2.6.

These are deferred branches; the instruction following a **JC** must be a single-word opcode, and is always executed. The macro **NOP** (**R0 R0 MOV**) is available to fill the position after a deferred branch in the case that no useful instruction can be devised to fill the slot.

To create a named label for a target location in the host dictionary, use the form:

```
LABEL <name>
```

described in Section 2.2. Invoking *name* returns the address identified by the

label, which may be used as a destination for a **JC**.

2.6 ASSEMBLER STRUCTURES

In conventional assembly language programming, control 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. 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> 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, which are listed in a glossary beginning on page 13. The combination of a condition code and a structure word (**UNTIL**, **WHILE**, **IF**) assembles a conditional branch instruction, **BR** or **JC** depending upon the range of the branch. 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:

```
T R0 CMP   0< IF   NOP
```

executes the true branch of the **IF** structure if the value in **R0** is less than the top stack item in **T** (**R17**), that is, if (**T** - **R0**) is negative. Note the use of the **NOP** following the **IF**, which is required since the instruction following a branch (which will be assembled for the **IF**) is always executed.

In high-level Forth words, control structures must be complete within a single definition. In **CODE**, this is relaxed. However, control structures that span routines are not recommended—they make the source code harder to understand and harder to modify.

Table 4 shows the conditions generated by a SwiftX condition specifier. These may be used with **IF**, **WHILE** and **UNTIL**. See the glossary below for details. Refer to your processor manual for details about 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). If **NOT** is used following a condition, it inverts the condition; otherwise, it assembles a **NOT** (one's complement) instruction.

Table 4: Conditions generated by SwiftX conditional structure words

Phrase	Condition code used	Description
0<	**GE**	Branch if greater than or equal to zero.
0< NOT	**LT**	Branch if less than zero.
0=	**NE**	Branch if not equal to zero.
0= NOT	**EQ**	Branch if equal to zero.
0>	**LE**	Greater-than-zero; branch if less than or equal to zero.
0> NOT	**LT**	Less-than-or-equal; branch if less than zero.
OV	**NV**	Branch if the overflow bit is not set.
OV NOT	**V**	Branch if the overflow bit is set.
CS	**NC**	Branch if the carry bit is not set.
CS NOT	**C**	Branch if the carry bit is set.
NEVER	**X**	Unconditional branch.

The conditions **OV** and **CS** may be combined with each other or other conditions.

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, because **REPEAT**, **UNTIL**, **AGAIN**, **ELSE**, and **THEN** always use the addresses on the stack.

In the glossary entries below, the stack notation *cc* refers to a condition code. Available condition codes are listed beginning on page 13.

Glossary **Branch Macros**

BEGIN *(— addr)*

Leave the current code 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 (see 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.

Condition Specifiers

0< *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on greater-than or equal.

0= *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on non-zero.

0> *(— cc)*

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

CS *(— cc)*

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

NEVER *(— cc)*

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

NOT *(cc_1 — cc_2)*

If cc_1 is a valid condition code, invert it to give cc_2; otherwise, assemble a **NOT** (one's complement) instruction. For example:

```
0= NOT IF          inverts 0= to assemble EQ BR or EQ JC
R0 R1 NOT          assembles a NOT instruction
```

2.7 Assembly Language Macros

The important thing to remember when considering assembler macros is that the various elements in SwiftX assembler instructions (register names, mnemonics, etc.) are Forth words that are executed to create machine language

instructions. Given that this is the case, if you include such words in a colon definition, they will be executed when that definition is executed, and will construct machine language instructions at that time, i.e., expanding the macro. Therefore:

An assembly language macro in SwiftX is a colon definition whose contents include assembler commands.

The only complication lies in the fact that SwiftX assembler commands are not normally accessible in colon definitions (see "scoping," *SwiftX Reference Manual*, Section 3.6.2). This is necessary because there are assembler versions **IF**, **WHILE**, and other words that have very different meanings in high-level Forth. When you use **CODE** or **LABEL** to start a code definition, those words automatically select the assembler search order, and **END-CODE** restores the previous search order. However, to make macros, you will need to manipulate search orders more directly.

The relevant commands for manipulating vocabularies for assembler macros are given in the Glossary at the end of this section. Here are a few simple examples.

Example 1: SPUSH

```
ASSEMBLER
: SPUSH ( r -- )  \ Push r onto the data stack
    1 S SUB       \ Add a cell on top of the data stack
    S SWAP STR ;  \ Store the content of r in that cell
```

This is a simple macro that is commonly used in the SwiftX kernel. Others are given in the glossary at the end of this section.

This mechanism is also used to provide the code for the words that are expanded in place, as shown in the next example.

Example 2: Direct code compilation

```
COMPILER
: DUP   [+ASSEMBLER]  T SPUSH  [PREVIOUS] ;

TARGET
: DUP ( x -- x x)   DUP ;
```

The SwiftX cross-compiler's version of **DUP** is executed whenever it encounters a **DUP** in a **TARGET** colon definition. It will expand the **SPUSH** macro in place, assembling the instructions:

```
1 S SUB
S T STR
```

In the compiler's definition of **DUP**, it was necessary to add the **ASSEMBLER** scope temporarily to the search order to get access to **T** and **SPUSH**.

The second definition of **DUP** provides an executable version that you can type during debugging, and whose execution token is returned by **'** and **[']**. It is an essential feature of SwiftX that all target words are available for interactive execution from the command window during a debugging session. Providing these additional definitions adds a tiny amount of space in the target, but the entire strategy provides significant performance improvement. It is cost-effective whenever the overhead of a **CALL** and **RET** combination is large compared to the actual content of the word itself (e.g., up to two or three instructions). Many small and frequently-used Forth words are defined in this way.

The glossary below lists the most commonly used macros in this system.

Glossary

SPUSH *(r —)*

Push general register *r* onto the Forth data stack.

SPOP *(r —)*

Pop general register *r* from the data stack.

TPUSH *(—)*

Push **T** (top stack item, in **R17**) onto the data stack.

TPOP *(—)*

Pop **T** from the data stack.

RET *(—)*

Return to the address in **R** (**R19**).

IRET *(—)*

Return to the address in **IR** (**R13**, reserved for interrupt handlers).

U) *(addr — u)*

Given a user variable address *addr*, return its offset into the user area.

References Register names and usage in SwiftX, Section 2.3

Scoping and search orders (**ASSEMBLER**, **TARGET**, etc.), *SwiftX Reference Manual*, Section 3.6.2.

3. Implementation Issues

This section covers specific implementation issues involving various versions of the UT69R000 processor. For board-specific details, see the relevant appendix.

3.1 Implementation Strategy

A variety of options are available when implementing a Forth kernel on a particular processor. In SwiftX, we attempt to optimize, as much as possible, for both 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 UT69R000's CPU stack is used as Forth's return stack, and may contain return addresses. Code routines that make subroutine calls must preserve the contents of the subroutine linkage register **R** (**R19**) before making **CALL**s, usually by **PUSH**ing it. In colon definitions, the compiler handles this automatically, by saving it before a **CALL** or program structure and remembering to restore it if it was saved. Colon definitions that make no calls and contain no program structures do not unnecessarily save and restore **R**.

You may see examples of SwiftX UT69R000 optimization strategies by decompiling some simple definitions. For example, the source definition for **DABS** is:

```
: DABS ( d1 -- d2)   DUP 0< IF  DNEGATE  THEN ;
```

but if you decompile it, you get:

```
SEE DABS
0656   1 R15 SUB                          B1E1
0657   R15 R17 STR                        0A6F
0658   R19 PUSH                           0B77
0659   0< R19 CALL                        0F7F012F
065B   0 R17 CMP                          BE60
065C   R15 R17 LR                         066F
065D   0662 EQ JC
065F   1 R15 ADDU                         7D1F0003ADE1
0660   DNEGATE R19 CALL                   0F7F05F6
0662   R19 POP                            0777
0663   R19 R19 CALL                       0F7B ok
```

This example clearly shows the combination of in-line code and subroutine calls in this implementation. Note the **R19 PUSH** inserted by the compiler before the first **CALL** in this definition, and the corresponding **POP** at the end; this is the way subroutine nesting is handled on this processor.

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**, noticing particularly the sections marked **COMPILER**.

3.1.2 Data Format and Memory Access

Because the UT69R000 is a 16-bit cell-addressed processor with a "Harvard architecture" (separate code and data spaces), its directly addressable memory space is limited to 64K 16-bit cells for code and data. Operators for accessing this memory are discussed in Section 3.2.

The UT69R000 does not support individual addressing of 8-bit bytes, however, so this implementation uses a character size of 16 bits, the same as its cell size. This means that **C@** is the same as **@**, etc. In short, character operators are supported only for compatibility with byte-addressed implementations.

3.1.3 Stack Implementation and Rules of Use

The Forth virtual machine has two stacks with 16-bit items, located in RAM. Stacks grow downward in address space. The return stack is the CPU's subroutine stack, and it 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**.
- When within a **DO** loop, a program shall not access values that were 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.

3.1.4 SwiftOS Multitasker Implementation

The UT69R000 supports an efficient SwiftOS implementation, with eight instructions required to suspend a task and nine for it to resume. The subroutine-threaded implementation means there is no **I** register (see address interpreter, Section 5.3 of the *SwiftX Reference Manual*) to be saved and restored, only the return and data stack pointers and **T**.

The single-cell user variable **STATUS** contains the address of either the **SLEEP** or **WAKE** code, and is followed by the **FOLLOWER** cell containing the address of the next task in the round-robin. **SLEEP** passes control to the **FOLLOWER** task, which will execute whichever routine is pointed to by *its* **STATUS** cell. **WAKE** returns control to the task, restoring its context to the state in which it left off when it called **STOP** or **PAUSE**.

If you wish to review the simple code for this SwiftOS multitasker, you will find it in the file **Swiftx\Src\UT69R000\Tasker.f**.

References SwiftOS task activation/deactivation, *SwiftX Reference Manual*, Section 5.2.1

3.1.5 XTL Implementation

The UT69R000 SwiftX system uses the built-in 9600-baud UART for the Cross-Target Link (XTL), whose control is described in Section 4.9 of the *SwiftX Reference Manual*.

3.2 Special Addressing Operators

The UT69R000 has several address spaces that are not accessed by the usual means. These include I/O addresses and code space outside the default 64K addressing range.

3.2.1 I/O Address Operators

SwiftX supplies fetch and store primitives for I/O ports. These words may be found in the file **`..\UT69R000\Extra.f`**, and are described in the glossary below. The notation *p-addr* in the stack comments indicates a port address.

Each of these "P" operators is analogous to the normal memory access operator, except it takes an *p-addr* instead of a linear address.

Glossary

`P@` *(p-addr — x)*

Fetch a cell (16-bits) from the I/O port address *p-addr*.

`P!` *(x p-addr —)*

Store a cell at I/O port address *p-addr*.

3.2.2 Extended Memory Access

SwiftX supplies simple primitives for access to program (code) and extended memory. These words may be found in the file **`..\UT69R000\Codemem.f`**, and are described in the glossary below. The notation *e-addr* in the stack comments indicates a 20-bit extended address, represented as a double-cell stack item.

Each of these "E" operators is analogous to the normal memory access operator, except it takes an *e-addr*. **`EDUMP`** is used primarily for diagnostic purposes.

Glossary

`@C` *(addr — x)*

Fetch a cell from program memory.

`!C` *(x addr —)*

Store a cell into program memory.

`E@C` *(e-addr — x)*

Fetch a cell (16-bits) from *e-addr* in extended memory.

`E!C` *(x e-addr —)*

Store a cell at *e-addr* in extended memory.

`EDUMP` *(e-addr n —)*

Dump *n* cells starting at *e-addr* in extended memory.

`EBUFFER:` `<name>` *(n —)*

Define an extended memory buffer *n* cells long in the currently selected uData section, which must reside in an extended data page. Execution of *name* will return a 20-bit address of the start of the buffer, in a double-cell stack item.

3.3 Interrupt Handling

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

The handler itself is written in code. The usual form begins with **LABEL** <name> and ends with an **IRET** (Return from Interrupt) and **END-CODE**.

To attach the code to the handler, use the word **INTERRUPT**, which takes an address for the handler code and an interrupt number, and links them such that when the interrupt occurs, it will be vectored directly to the code through a simple dispatcher described below. No task needs to be directly involved in interrupt 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 Section 5. of the *SwiftX Reference Manual*.

The UT69R000's interrupt vectors 0 through 14 are located in code space addresses 400_H–438_H. Since this is in PROM, SwiftX redirects these vectors to interrupt handler dispatch routines that call the actual interrupt handlers through a RAM table called **VRAM**. This makes it possible for you to add interrupts and test them without having to burn new PROMS. The RAM vector table is initialized to **IRET**s for all vectors that aren't supported in the default SwiftX kernel.

The dispatch routines preserve the return address and processor status before calling the handler at the corresponding location in **VRAM**. The handler must return to the dispatch routine using the **IRET** macro. Any registers used by the handler must be preserved, usually with **PUSH** and **POP** instructions. Interrupts are left disabled while a handler is executing. If you desire nested interrupt operation, the handler may explicitly enable interrupts, and nesting will then occur.

When a handler returns to the dispatch routine, the processor status flags are restored, the Pending Interrupt flag for that interrupt is reset, and execution resumes where it was when the interrupt occurred.

An example of a simple interrupt handler is the one provided for the Timer A interrupt in **\Swiftx\Src\UT69R000\TimerA.f**. It looks like this:

```
VARIABLE MSECS

LABEL <TICK>
   R0 PUSH   -100 R0 MOV   TA R0 OTR
   MSECS R0 LR   1 R0 ADD   MSECS R0 STR
   R0 POP   IRET   END-CODE

<TICK> 7 INTERRUPT
```

<TICK> is the Timer A interrupt handler, and 7 is the Timer A Interrupt Vector. This code accumulates a millisecond count in the counter **MSECS**.

Power-up initialization for all interrupts is done by the word **/INTERRUPTS**, which should be included in the word **START**, found in **\Swiftx\Src\UT69R000**<platform>**\Start.f**.

/INTERRUPTS must be called *before* any other initialization functions that enable specific interrupts.

Further details on the implementation and management of interrupt vectors may be found in **\Swiftx\Src\UT69R000\IntVec.f**

Glossary

INTERRUPT *(addr n —)*

Store address *addr* into the **VRAM** table entry for interrupt vector *n*. Two versions are supplied: the **INTERPRETER** version is used to set the code image vector, while the **TARGET** version sets the VRAM vector at run time in the target.

/INTERRUPTS *(—)*

resets all pending interrupts, clears all mask bits, and enables interrupts.

3.4 TIMERS

The UT69R000 has two on-board 16-bit interval timers, both of which are supported by SwiftX:

- Timer A, with 10 µS/bit resolution, is used to maintain a free-running interval timer whose units are milliseconds.
- Timer B, with 100 µS/bit resolution, is used to maintain a time-of-day clock and calendar.

Timer A is incremented every 10 µS and interrupts on overflow. The interrupt handler resets the counter to -100 so it overflows again in 1000 µS (1 millisecond). **COUNTER** returns the current 16-bit free-running counter of clock interrupts. **TIMER**, always used after **COUNTER**, obtains a second count, subtracts the value left on the stack by **COUNTER**, then displays 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
```

The Timer B interrupt is used to accumulate the current date and time. The Timer B interrupt handler increments the double-cell counter **SECONDS**. The timer is incremented every 100 µS and interrupts on overflow. The interrupt handler resets the counter to -10000 so it overflows again in 1,000,000 µS (1 second). **@NOW** and **!NOW** provide the interface for the generic SwiftX time and date library functions.

Glossary

@NOW *(— ud u)*

Checks for 24-hour rollover, incrementing **TODAY** as necessary, and returns the system time and date as *ud* seconds since midnight and system date *u* in MJD format.

!NOW *(ud u —)*

Sets the system time and date. For example:

```
12:30 HOURS 1/15/98 M/D/Y !NOW
```

References Clock, calendar, and timing libraries, *SwiftX Reference Manual*, Section 6.2

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 shall 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 as **`CONSTANT`**s or **`EQU`**s. 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.
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. You can simply read and write the ports, using **`P@`** and **`P!`** (Section 3.2), and use the **`.`** ("dot") command to display the results. (Usually you want the numeric base set to **`HEX`** when doing this!).

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 B interrupt to provide basic time-of-day services. This provides a good example of a simple interrupt routine. The complete source may be found in **`Swiftx\Src\UT69R000\TimerB.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! Fortunately, Timer B can be made to interrupt once per second, so we shall keep time in seconds since midnight.

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, or 32 bits. The timer is incremented every 100 µS and interrupts on overflow. The interrupt handler resets the counter to -10000 so it overflows again in 1,000,000 µS (1 second). 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 SECONDS \ Holds time-of-day in seconds since midnight.

LABEL <TOCK>      \ Timer B interrupt handler; increments SECONDS
   R0 PUSH   R1 PUSH                \ Save registers
   -10000 R0 MOV   TB R0 OTR        \ Reset timer counter
   SECONDS R0 LR   SECONDS 1+ R1 LR \ Fetch to XR0
   1 XR0 ADD                        \ Increment double value
   SECONDS R0 STR  SECONDS 1+ R1 STR\ Store incremented count
   R1 POP   R0 POP                  \ Restore registers
   IRET   END-CODE

<TOCK> 9 INTERRUPT
```

Many systems handle the midnight rollover in clock interrupt code. However, it's inefficient to check so frequently for something that happens so infrequently! In this example, we check in the word **@NOW** (see Section 3.4), which is the command for fetching the system date and time. Providing your application code always uses **@NOW** (or the higher-level words that call it) to fetch the time rather than reading **SECONDS** directly, you need never worry about midnight rollover, and the overall cost to the system is minimized.

APPENDIX A: UTMC EVALUATION BOARD CONFIGURATION AND USE

This section provides information pertaining to the UTMC Evaluation Board, which is supported by SwiftX for the UTMC UT69R000. 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 AND CONFIGURATION

The EVB is a minimal demonstration board provided by UTMC. It includes 32K words each ROM and RAM for program storage, 8K words data RAM. SwiftX configures this as shown in Figure 2.

A.2 DEVELOPMENT PROCEDURES

On the UTMC Evaluation Board, the SwiftX kernel resides in PROM, replacing the UTMC monitor. As a result, procedures for preparing and installing new kernels differ from those described in the generic SwiftX manual that may be RAM-based.

A.2.1 Starting a Development 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 compares it to the target's kernel in PROM.

If the source has changed, you will get the message, `Kernel Mismatch`, in which case you must generate a new code image and install it in the board's PROM.

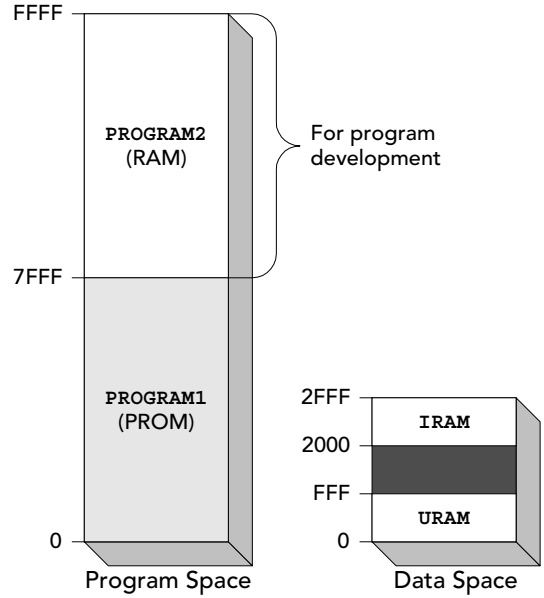

Figure 2. Memory configuration in UTMC EVB

If the board is not connected properly, you will get the message, `No XTL. Try again? (y/n).` 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 and the host version of the kernel matches the PROM's kernel, 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.2.2 Installing a New Kernel in PROM

Your UTMC Evaluation Board board is shipped with a SwiftX kernel installed in its on-board PROM. In order for SwiftX's Cross-Target Link (XTL) to work properly, the object generated by your kernel source must exactly match the installed kernel. Therefore, if you make any changes to your kernel, you must replace the kernel in PROM.

To install a new kernel:

1. Select the Project > Build menu item. This generates a new, ROMable **`Target.hex`** object file. At that point, exit SwiftX.
2. Launch the PROM burning utility of your choice, and burn **`Target.hex`** into a pair of PROMs. For each word in the **`.hex`** file, the first byte is the MSB, the second is the LSB.
3. Install the new PROM in your EVB.

A.2.3 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 uses the host keyboard and screen via the XTL.

To run the demo program, select Project > Debug to bring up the target program. Type **`CALCULATE`** to start the application.

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

GENERAL INDEX