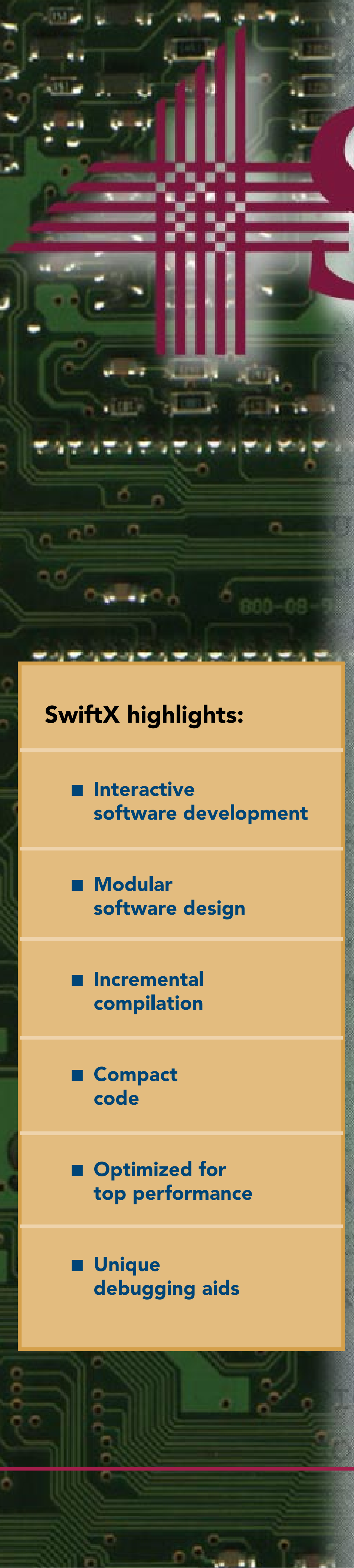


SwiftX™

Getting Results Faster

SwiftX highlights:

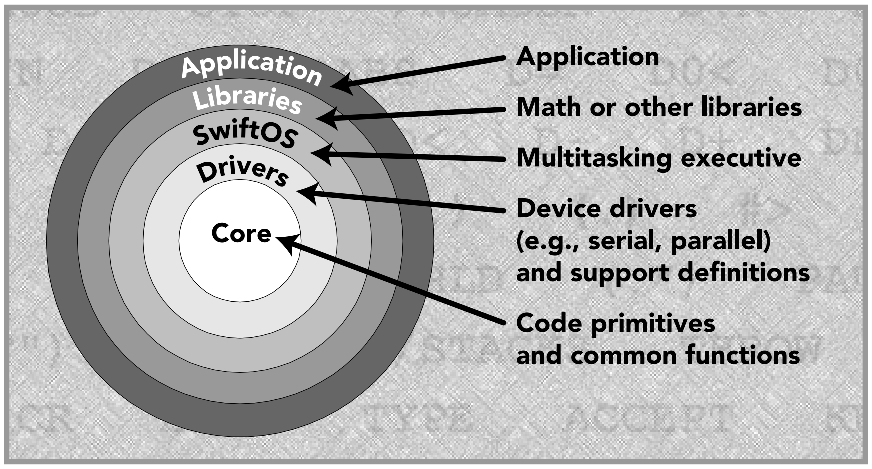
- Interactive software development
- Modular software design
- Incremental compilation
- Compact code
- Optimized for top performance
- Unique debugging aids

SwiftX — Integrated Cross-Development Software

The SwiftX cross-development system uses the power of Windows to give you a fast, easy-to-use way to write and test embedded system software. Based on ANS Forth, it's the latest in a long line of products that, for more than 25 years, have helped programmers to *get results faster* using the Forth programming language.

Forth is the only language *designed from first principles for embedded control.* From NASA space shuttle instrumentation to FedEx handheld package-tracking devices, and from point-of-sale terminals to medical devices, home appliances, test instruments, communications, and many other application areas — Forth has shortened development times while producing small, fast, reliable programs. SwiftX is appropriate *anywhere there is a need for real-time data acquisition, analysis, and control.*

Let us help you to learn how to get results faster in your own projects, with SwiftX…


Application
Libraries
SwiftOS
Drivers
Core
Application
Math or other libraries
Multitasking executive
Device drivers (e.g., serial, parallel) and support definitions
Code primitives and common functions

SwiftX IDE Features

SwiftX's easy-to-use Integrated Development Environment (IDE) provides a common user interface across all target systems. It supports efficient, interactive development without external debuggers, emulators, or other expensive, cumbersome tools.

When you are developing and testing your code, the SwiftX IDE supports an active link to your target device. If target RAM is available, SwiftX can automatically download compiled programs so they are *ready for interactive testing on the target immediately*.

Cross-compiler

SwiftX's cross-compiler creates a customized, ROMable version of your code for embedded applications. Features of the SwiftX cross-compiler include:

- Optimizing compiler produces *fast, compact target code.*
- *The compile-download-test cycle is extremely short*. Compiling an entire kernel takes only a few seconds; even substantial applications can be compiled in under a minute.
- *High-level Forth code is portable across all SwiftX targets;* only assembler functions are CPU-dependent.
- You have *complete control over kernel configuration and contents.* Include or omit features as needed to yield a compact kernel.
- SwiftX links to familiar Windows programmers' editors, so with a single command you can see the source for any compiled word, see the location where a compiling error occurred, or see cross-reference information.

Assembler and Disassembler

SwiftX includes an integrated *macro assembler* for writing any procedure in machine code, if necessary. High-level and assembler definitions can be intermingled in the same source file, and assembler code uses the same simple mechanisms as high-level definitions for interfacing with the application. You can develop applications almost entirely in high-level, then—after revising your algorithms and debugging the logic—quickly convert the most time-critical routines to machine code.

Configurable, Multitasking Kernel

The SwiftX kernel is supplied *in source form*, and is fully configurable—*you can choose which features to include* to support your applications. Simple programs can take only a few hundred bytes, and use very little RAM.

One company modified the SwiftX kernel to put their handheld device into "sleep" mode after only one cycle through the multitasker idle loop, thus using 70% less power than an off-the-shelf "real-time" kernel that was supplied only in binary and was not modifiable.

Sample kernel sizes

	16-bit CPU		32-bit CPU	
	Program	Data	Program	Data
Core*	1628	468	2902	852
Double-precision arithmetic	513	0	578	0
Multitasker	436	0	454	0
Drivers**	1794	174	1386	290
Clock and calendar	986	6	722	0
Initialization	225	118	328	154
Total	**5582**	**766**	**6370**	**1296**

* representative set of functions
** serial I/O, XTL on the 16-bit CPU

SwiftX systems include SwiftOS, an extremely fast multitasking executive. Features of the SwiftX kernel include:

- Time-critical interrupt service routines link directly to interrupt vectors *with no OS overhead,* to ensure fast, deterministic response.
- Non-preemptive task-servicing algorithm utilizes time spent waiting on I/O to *maximize service to all tasks.*
- C*omplete context switch in only a few machine instructions.*
- *Semaphores and global variables* support intertask synchronization.

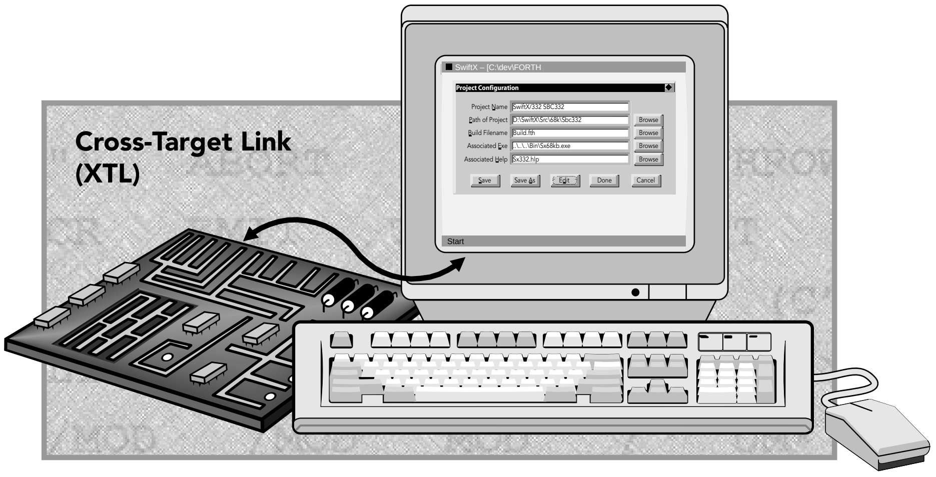
Debugging Tools

SwiftX supports fully interactive debugging using its “Cross-Target Link” (XTL) to communicate with the target system. The actual nature of the link depends on the target hardware. Most targets use a serial line connected to an on-board serial port; the Motorola 68332 and 68HC16 use Motorola’s “Background Debugging Mode” (BDM), and communicate via a parallel port.

The SwiftX XTL provides the following debug functions:

- Display and modify target RAM
- Display target CPU registers
- Download code to target RAM
- Interactive testing of target Forth words
- Optional data stack monitor window

On processors using the BDM, very little special software is required in the target to support interactivity. On other processors, the target XTL is implemented in a few hundred bytes in PROM on the target, along with a target kernel.



Project Name: SwiftX/332 SBC332
Path of Project: D:\SwiftX\Src\68k\Sbc332
Build Filename: Build.fth
Associated Exe: ..\..\..\Bin\Sx68kb.exe
Associated Help: Sx332.hlp

Source Libraries

SwiftX libraries are provided in source form, with standardized, portable APIs. SwiftX libraries include hundreds of routines, such as:

- integer and fixed-point fraction arithmetic and transcendental functions
- string handling
- clock and calendar
- number conversions
- drivers for serial ports, clock, and other devices
- …and many others, depending on the target CPU

Demo Target Board

SwiftX includes a simple board for the target processor family, including a representative kernel and a demo application. This is an ideal platform for learning about SwiftX and even for starting work on your application. If your final target hardware will be different, you may easily port the system to your board by adjusting the memory configuration, power-up initialization and, possibly, the XTL I/O driver.

Optional Features

On-board Interpreter

You can add an optional on-board interpreter—even a high-level Forth compiler—to your target system. This means you can attach a terminal (such as a laptop PC or PDA running a simple terminal emulator program) to your target in the field, and exercise any function in the target just as you can from the SwiftX IDE. You can even do simple programming, if you have configured your target with at least a few hundred bytes of program space in RAM, to test target applications or to perform field configuration. This is particularly useful for field engineers.

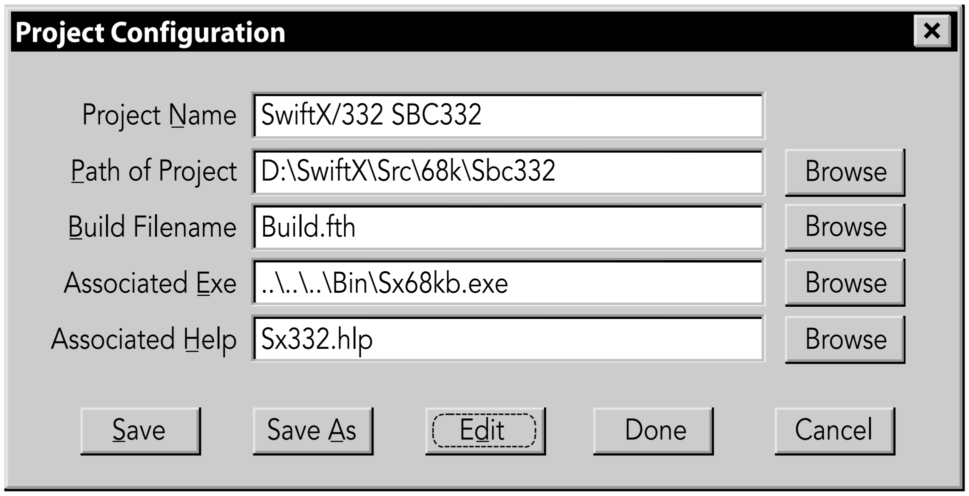
EPROM Emulators

Some targets (such as “small model” 8051s and targets with only on-chip ROM and RAM) don’t provide RAM for program storage during development—an EPROM emulator will enable you to avoid burning test PROMs. FORTH, Inc. supplies optional emulators for many target processors.

Developing Programs with SwiftX

SwiftX development is controlled through a *project file.* Two project files are provided with the system:

- **TRY** constructs and downloads a target image of the SwiftX kernel plus specified application functions, along with some debugging aids. It also activates the XTL, so the system is ready for interactive testing.
- **BUILD** constructs a target image as a file on the host PC, for separate downloading or for burning into PROM.



Project Name: SwiftX/332 SBC332
Path of Project: D:\SwiftX\Src\68k\Sbc332
Build Filename: Build.fth
Associated Exe: ..\..\..\Bin\Sx68kb.exe
Associated Help: Sx332.hlp

After code is compiled and downloaded, you can execute target functions. You can provide parameters simply by entering them on the host; they are passed to the target automatically, and any resulting parameters are passed back to the host. *You are interacting with the target as though it had a terminal.*

The XTL lets you examine target memory (even hardware registers), decompile or disassemble target routines, and exercise any target function.

Getting Started

The demo target board supplied with the system enables you to start learning how to use all the features of SwiftX right away. The SwiftX package also includes a sample application—emulating a specialized handheld calculating device—that you can study, run, and even modify to become comfortable with SwiftX programming more quickly.

When your target hardware is ready, code you've been developing on the test board can be moved onto it quite easily, by taking three steps:

- Configure your memory map for the target device.
- Modify or add any device drivers that are peculiar to the new target.
- Adapt the startup code to properly initialize your target hardware and software.

Target Configuration

Target memory may be mapped for program memory (which may be ROM), initialized RAM, and uninitialized RAM. You can define multiple sections of each type, specifying the optimal location and size for your target hardware. You may, for example, place frequently accessed variables in on-chip RAM for maximum speed, while leaving larger arrays in external, or even paged, RAM.

I/O Drivers

It is very easy to develop custom I/O drivers in SwiftX. XTL communication with the target lets you examine and even modify I/O registers directly, to test the device's functionality. Typically, drivers are quite small, and are made up of several related simple definitions (e.g., one to read and one to write), which can also be tested interactively. Several examples (e.g., serial I/O, clock) are provided with your SwiftX system. To see how easily you can control digital devices such as motors and switches, look at the Washing Machine example on the following pages.

SwiftX was used to develop terminal firmware for Europay International's "Open Terminal Architecture" (OTA) used in smart-card point-of-sale terminals. Processor-independent *byte-codes*, or *tokens* (conceptually similar to Java implementations), are used by OTA for portable smart-card payment applications.

Interrupt Handlers

SwiftOS makes interrupt handling simple. The strategy is:

- Perform only the most time-critical actions at interrupt time.
- Notify the task responsible for the interrupting device that the interrupt has occurred.
- Defer complex logic to high-level routines executed by that task.

Notification may take the form of setting a flag, incrementing or decrementing a counter, or modifying the task's status so it will become active at the next opportunity in the multitasker cycle.

The basic form of an interrupt handler is:

```
LABEL <name> <code instructions>
<name> <dev#> INTERRUPT
```

where **LABEL** starts the definition of the routine, which is called *name*; the *code instructions* perform the necessary work of the routine; *dev#* is the device code or interrupt vector to which the routine will respond, and **INTERRUPT** is a special code-ending macro that assembles the appropriate return-from-interrupt instruction and attaches the address of *name* to the interrupt vector. When an interrupt occurs, it will be vectored directly to the code at *name* with *no overhead* imposed by SwiftOS.

Customizing Your Kernel

When your application is fully developed and tested, you may customize your kernel by removing any functions it doesn't need—just edit your project's main load file. In this way, you can reduce the size of your program to fit in even very small targets. Since all source is provided for your kernel, you have full control over what is included.

Burning PROMs

During development, it is convenient to be able to test your program in RAM. SwiftX's flexible memory configuration makes it easy to, for example, map more RAM during development than you may have in the final product, or to substitute RAM chips for PROMs during development. For some targets, it may be most convenient for you to use an EPROM emulator.

If you have limited program RAM, a good strategy might be to start with a simple SwiftX kernel in PROM. You can interactively test some application code in RAM and, as more of it becomes stable, add it to the kernel in PROM.

When your application has been tested, it's time to burn PROMs. SwiftX can write its object files in Intel Hex format, Motorola S-records, or binary files.

Programming with SwiftX — An Example

The method of developing a program in SwiftX is consistent with the recommended practices of top-down design and bottom-up coding and testing. However, Forth adds another element: extreme modularity. You don't write page after page of code and then try to figure out why it doesn't work; instead, you write a few brief definitions and exercise them, one by one.

Suppose we are designing a washing machine. The highest-level definition might be:

```
: WASHER ( -- )   WASH  SPIN  RINSE  SPIN ;
```

The colon indicates that a new word is being defined; following it is the name of the new word, **WASHER**. The remainder are the words that comprise this definition. Finally, the definition is terminated by a semi-colon.

Typically, we design the highest-level routines first. This approach leads to *conceptually correct solutions with a minimum of effort*. But in Forth, words must be compiled before they can be referenced—so a listing begins with the most primitive definitions and ends with the highest-level words.

The facing page shows a complete listing of the washing machine program. Items in parentheses or following a backslash (****) are comments. The first few lines define a hardware port address (in this case, on a 68HC12) and six bit masks used to access individual bits on that port. Subsequent lines define application words that do the work.

The code in this example is nearly self-documenting; the comments identify groups of words and show the parameters being passed to certain words. When reading,

```
   : WASHER ( -- )   WASH  SPIN  RINSE  SPIN ;
```

it is obvious what **RINSE** does. To determine *how* it does it, you read:

```
   : RINSE ( -- )   FILL-TUB  AGITATE  DRAIN ;
```

When you wonder how **FILL-TUB** works, you find:

```
   : FILL-TUB ( -- )   FAUCETS ON  TILL-FULL  FAUCETS OFF ;
```

Reading further, one finds that **FAUCETS** is a mask specifying the bit of the port that controls the faucet, while **ON** is a word that turns on that bit.

Even from this simple example, it may be clear that Forth is not so much a language as a tool for building application-oriented command sets. The definition of **WASHER** is based not on low-level Forth words, but on words with names like **SPIN** and **RINSE** that make sense in the context of the application.

When developing this program, you would follow your top-down logic, as described above. But when the time comes to test it using SwiftX's Cross-Target Link (XTL), you see the real convenience of Forth's interactivity.

If your hardware is available, your first step would be to see if it works. Even without the code in the file, you could read and write the hardware registers by typing phrases such as:

```
   HEX PORT C@ .
```

The word **C@** fetches a character (byte) from the address specified by **PORT**. This would read port address 01 and display its current bit values. The similar phrase **PORT C!** (used in **ON** and **OFF**) stores a value in the address. You could also type:

```
   2 ON   2 OFF
```

to see if the clutch, controlled by the bit masked by 2, engages and disengages. If the hardware is unavailable, you might temporarily re-define **PORT** as a variable you can read and write, and so test the rest of the logic.

You can load your source file using the command **INCLUDE** `<filename>`, whereupon all functions defined in the specified file are available for testing. You can further exercise your I/O by typing phrases such as:

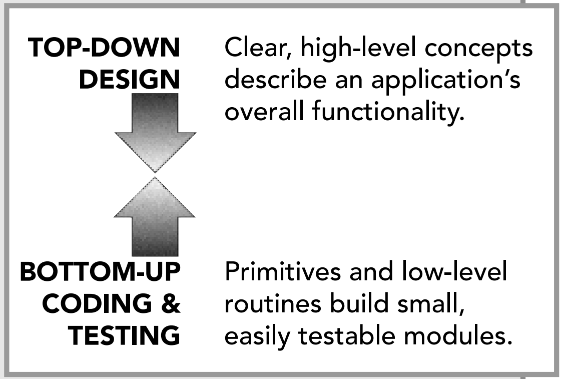
MOTOR ON *or* **MOTOR OFF**

to see what happens. Then you can exercise your low-level words, such as:

```
   DETERGENT ADD
```

and so on, until your highest-level words are tested.

As you work, you can use any additional programmer aids provided by SwiftX. You can easily change your code and re-load it. But your main ally is *the intrinsically interactive nature of Forth itself.*



TOP-DOWN DESIGN Clear, high-level concepts describe an application's overall functionality.

BOTTOM-UP CODING & TESTING Primitives and low-level routines build small, easily testable modules.

Parentheses and backslashes denote comments.

Listings begin with primitive functions.

A colon begins a new definition.

Definitions can contain generic SwiftX words and any others you've defined…

…so, application-specific functions are defined in terms of previous definitions…

…until you reach the main application definition.

```
( Washing Machine Application )
\ Port assignments
  01 CONSTANT PORT
\ bit-mask    name       bit-mask     name
   1 CONSTANT MOTOR        8 CONSTANT FAUCETS
   2 CONSTANT CLUTCH      16 CONSTANT DETERGENT
   4 CONSTANT PUMP        32 CONSTANT LEVEL

\ Device control
: ON  ( mask -- )   PORT C@  OR  PORT C! ;
: OFF  ( mask -- )   INVERT  PORT C@  AND  PORT C! ;

\Timing functions
: SECONDS ( n -- )   0 ?DO  1000 MS  LOOP ;
: MINUTES ( n -- )   60 * SECONDS ;
: TILL-FULL ( -- )      \ Wait till level switch is on
      BEGIN  PORT C@  LEVEL AND  UNTIL ;

\ Washing machine functions
: ADD ( port -- )   DUP ON  10 SECONDS  OFF ;
: DRAIN ( -- )   PUMP ON  3 MINUTES  ;
: AGITATE ( -- )   MOTOR ON  10 MINUTES  MOTOR OFF ;
: SPIN ( -- )   CLUTCH ON  MOTOR ON
    5 MINUTES  MOTOR OFF  CLUTCH OFF  PUMP OFF ;
: FILL-TUB ( -- )   FAUCETS ON  TILL-FULL  FAUCETS OFF ;

\ Wash cycles
: WASH ( -- )   FILL-TUB  DETERGENT ADD  AGITATE  DRAIN ;
: RINSE ( -- )   FILL-TUB  AGITATE  DRAIN ;

\ Top-level control
: WASHER ( -- )   WASH  SPIN  RINSE  SPIN ;
```

The final program (with the XTL and unused functions removed from the kernel) is quite small, as you can see at right. The RAM figure includes stack space and system variables. Stack requirements vary, depending on the processor; for example, an 8051 would require less than the amounts shown here. With most microcontrollers, the entire program can reside in on-board ROM and RAM.

Compiled size of Washing Machine program

	ROM	**RAM**
Kernel	188	142
Washing machine program	297	0
TOTAL (in bytes)	**485**	**142**

Why Use SwiftX?

Programmers of embedded systems need a language that combines high-level operations with the ability to work easily with custom hardware. They need a compiler that adds minimal overhead in run time and memory. To complete a project on schedule, they need to test the software in the target environment, without time-consuming procedures for burning PROMs or downloading.

Here is how SwiftX addresses these needs:

Improves programmer productivity… SwiftX is highly interactive and includes a fully integrated development environment—so programmers are more productive and have *quicker project turnaround*.

Speeds testing of hardware & software… SwiftX speeds testing because you interact directly with the target hardware and software. You can even use SwiftX to debug your hardware, because SwiftX lets you read and write registers directly and interactively.

Minimizes hardware requirements… SwiftX code is compact—you can pack more functionality into any particular hardware than you ever imagined. The configurable SwiftX kernel and your applications will require far fewer resources than equivalent functionality obtained by using C or proprietary "real-time" kernels.

Runs fast… You benefit from very-low-overhead subroutine calls and from the real-time, multitasking features of SwiftOS. And the compact code minimizes paging and other high-overhead memory management strategies. SwiftX lets you incorporate both high-level and assembly definitions. You get *the performance you need*, while still conserving development time.

Simplifies software maintenance… SwiftX code is easy to maintain and modify thanks, in part, to its conciseness, its modularity, and its easy-to-use, on-line documentation. And its Forth foundation means that your application code is virtually an *application-specific language*—so you can develop your next similar project even more quickly, *even on another processor*.

Porting SwiftX to new CPUs

The structure and modularity of SwiftX's underlying design make it exceptionally easy to port, even to new CPU architectures—in only a few weeks. If you have a new or unusual target CPU, ask us for a quote.

About FORTH, Inc.

FORTH, Inc. was founded by the original developers of the Forth programming language, and is the oldest company providing Forth-based systems and services. Here are some of our additional offerings:

- Powerful Forth programming systems for Windows and MacOS
- EXPRESS "SoftLogic" for industrial-control applications
- polyFORTH for real-time PC applications
- Forth programming courses
- Custom software design and programming services, from drivers to complete applications

FORTH, Inc.
111 N. Sepulveda Blvd., Suite 300 • Manhattan Beach, CA 90266–6847
310.372.8493 • 800.55.FORTH • Fax 310.318.7130
www.forth.com • forthsales@forth.com