DALLAS SEMICONDUCTOR MAXIM

Microcontroller ENGINEERING REVIEW

Volume 1

The fast microcontroller: a decade of growth and innovation

System designers know that the microcontroller is the heart of any embedded system—that's where the action takes place. For over 18 years, Dallas Semiconductor has been redefining the ubiquitous 8-bit microcontroller. Perhaps the biggest improvements in the last 10 years have been made in the speed of instruction execution. Our 1 clock-per-machine cycle processors reached the ultimate performance goal—1 clock-per-machine cycle currently at 33 million instructions per second (MIPS). Using this core, our secure, networked, and mixed-signal 8051-microcontroller families continue to set the standard for feature integration and innovation.

. . . (the 8051) is one of the most popular 8-bit microcontroller architectures in the world.

Why base a family of innovative microcontrollers on the venerable 8051 instruction set? Because quite simply it is one of the most popular 8-bit microcontroller architectures in the world. The instruction set is simple to understand, making it a favorite of embedded system designers. Many of the instructions directly address I/O pins, allowing quick manipulation (bit-banging) of external peripherals. A tremendous variety of on-chip peripherals is available in an almost limitless number of combinations. In addition, development tools for the 8051-microcontroller family are widely available, so it is easy and inexpensive to start developing an application.

Safe and secure

In 1987 Dallas Semiconductor introduced the DS5000T, an independently developed microcontroller based on the 8051 instruction and feature set. To offer new features and benefits, our engineers based the design on NV SRAM technology rather than EPROM. Leveraging its leadership in low-power technology, the memory partitioning and battery-backup circuitry was integrated directly onto the microcontroller die. The chief advantage of this system was speed. Writing to most nonvolatile memories is slow, but NV SRAM can read or write in a single cycle at high speed. This makes it ideal for high-speed, nonvolatile data-logging applications where data must be captured in real-time. When combined with an external SRAM and battery, the result is a complete microcontroller system with up to 64kB of nonvolatile program and data memory.

NV SRAM technology enables both data and program memory to be in-system reprogrammed, byte-by-byte, on the fly. In a standard microcontroller system, program memory needs either to be physically removed from the system (EPROM) or block erased, prohibiting memory access during the erasure (flash). NV SRAM-based microcontrollers can be quickly and easily programmed from a PC or device programmer by its serial port. A ROM-resident bootstrap loader downloads program and data directly to the microcontroller, allowing fast debugging or field upgrades.

The distinctive advantages of NV SRAM provide a new perspective for firmware security. Because the bootstrap loader completely controls the loading of the program into NV SRAM, we encrypted the address and data bus with a 40-bit or 80-bit encryption key. Any program or data loaded into the microcontroller is automatically encrypted before it is stored in the SRAM. This encryption thwarts hackers from stealing the program or data in the microcontroller. During the execution of an instruction, the microcontroller fetches an encrypted op code, decrypts and executes it in a single machine cycle. The use of NV SRAM allows read/write access at full speed, with no delay for instruction decoding.

Table of Contents

The DS5240/50 are the only microcontrollers capable of providing the highest level of security while executing every instruction at full speed.

These security features culminated in the DS5240/50, the newest of what are now known as the secure microcontrollers, used in payment systems worldwide. Introduced in 2002, these tamper-reactive microcontrollers incorporate a 4 clock-per-machine cycle core as well as enhanced triple-DES encryption of their program memory. No other microcontrollers provide this level of security while executing every instruction at full speed. Security is further enhanced by the addition of intrusion-detection inputs and on-chip tamper sensors that automatically erase the memory as a tamper response. An integral microprobe shield prevents die tampering. Again, NV SRAM is best for high-security applications. Its high-speed write timing allows the microprocessor to erase confidential or sensitive data faster than any other type of memory.

More speed, less power

Although the 8051 processing core remained static from its conception in the late 1970s through the 1980s, embedded systems did not. System designers continued to improve and upgrade their 8051-based applications by adding new software features and peripherals. This "feature creep" pushed the limits of available 8051 performance. Unfortunately, improvements to the 8051 core failed to keep pace, and it appeared that system designers would have to switch to another processor and perform a costly redesign to upgrade their systems.

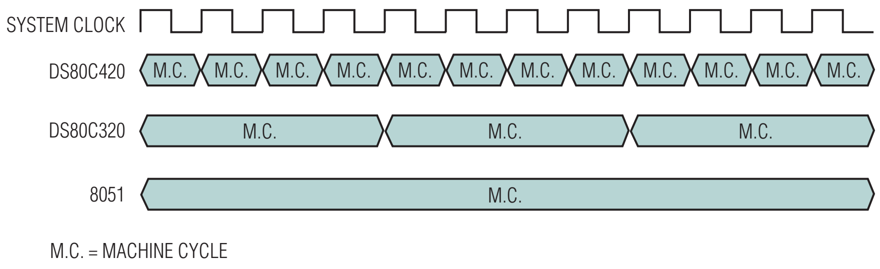


Figure 1. *Reducing the number of clocks per machine cycle allows 3x the performance with the same instruction set (12 vs. 4 vs. 1 clock-per-machine cycle).*

The performance bottleneck was the 1970s vintage processing core of the 8051 microcontroller. Although external crystal speeds approached 40MHz, the traditional 8051 still required 12 clocks to execute a single machine cycle. Each instruction required from 1 to 4 machine cycles, meaning an instruction could take as little as 12 or as many as 48 oscillator clocks. Throughput was therefore limited to just over 3 MIPS, even while executing something as basic as a string of 1-cycle NOP instructions (**Figure 1**).

In 1991 we set out to reengineer the 8051 microcontroller for performance. The engineering team started by analyzing the traditional 8051 design. The original 12 clock-per-machine cycle architecture was terribly wasteful; most instructions were forced to execute dummy cycles. Engineers rebuilt the CPU from the ground up so that it only required 4 clocks-per-machine cycle instead of 12. A second internal data bus eliminated architectural bottlenecks that might hamper performance. High-powered I/O drivers increased switching speed during external memory operations. All the internal peripherals such as timers and serial ports ran at the faster clock speeds. But every step of the way, there was one absolute necessity—the instruction set had to remain op-code compatible with the 8051 instruction set.

The result? A new 8051-based microcontroller delivered triple the efficiency of the original 8051 core, with the majority of instructions operating three times faster for the same oscillator frequency. In addition to the increased efficiency of the core, the maximum external oscillator frequency of most devices increased to 33MHz or 40MHz. System designers previously held back by the older, much slower 8051 were able to upgrade their system to a maximum speed of 10 MIPS without software changes.

In addition to the speed improvement, the core redesign yielded another benefit: reduced power consumption. The laws of physics decree that power consumed by a digital circuit is proportional to the number of transistors switched and the switching rate (frequency). Because the new core used fewer oscillator clocks per machine cycle, it consumed significantly less power per instruction per second than a traditional 8051.

Power management modes temporarily reduced the power consumption of the microcontroller through the use of a software-configurable internal clock divider. By reducing the machine cycle rate from 4 clocks-per-machine cycle to 64 or 256 clocks-per-machine cycle, power consumption was further reduced. A switchback feature let the device return to divide-by-4 mode upon receipt of an external interrupt or detection of a serial-port start bit. This allowed the device to remain in a low-power state but quickly resumed full-speed operation when needed. **Figure 2** shows the relative power consumption in different modes.

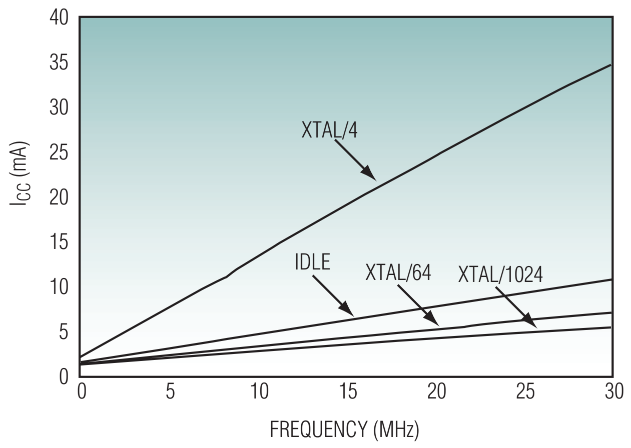


Figure 2. During periods of reduced activity, our power management modes consume less current than Idle mode but still allow CPU operation.

33 MIPS and beyond

In 1997 Dallas Semiconductor began designing a core for ultimate performance. Applications based on the 8051 were continuing to evolve, and customers clamored for even more performance. The engineering team set their sights on the performance peak: a microcontroller that executed the 8051 instruction set but used just 1 clock-per-machine cycle. Using a highly parallel architecture and a new fabrication process, a pin-for-pin, drop-in replacement 8051 was designed.

The result is the new DS89C420/430/440/450, ultra-high-performance 1 clock-per-machine cycle microcontrollers capable of executing up to 33 MIPS (**Figure 3**). These devices break previous performance barriers, providing 16-bit microcontroller performance with an 8-bit price. A variety of bus addressing modes allow users to fine-tune processor operation to the needs of the specific application design. But most importantly, they remain 100% 8051 instruction-set compatible and still execute existing 8051 applications faster than any other 8051-based microcontroller.

In addition to the lightning-fast core, the DS89C420/30/40/50 incorporate up to 64kB of in-system programmable flash memory. The ROM-based bootstrap loader allows modification of the microcontroller code before, during, or after final assembly, offering maximum flexibility. Unlike other microcontrollers that use proprietary or nonstandard interfaces, the DS89C420 bootstrap loader is accessed by its serial port from a standard PC, using any terminal emulator software.

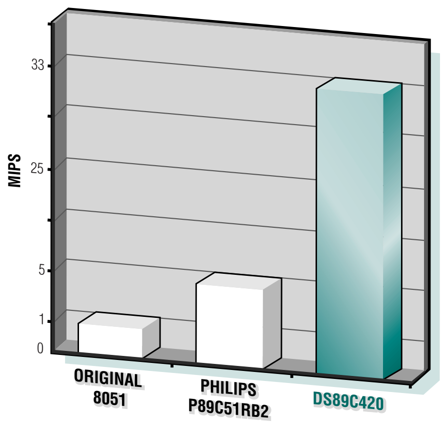


Figure 3. The DS89C420 outstrips the competition by clocking 33 MIPS.

Fast execution times beg for bigger program sizes

The advantage of speed is lost if programmers cannot have sufficient memory address space for their expressions. The traditional 8051 used a 16-bit memory bus, restricting the memory range to 64kB. For some applications this limited memory range was sufficient. But as applications increased in code size and complexity, we realized applications needed a solution that maintained as much 8051 compatibility as possible.

Every step of the way there was one absolute necessity—the instruction set had to remain op-code compatible with the 8051 instruction set.

Some designers found it possible to expand the addressing range by using bank-switching techniques. I/O lines double as address lines, expanding memory at the sacrifice of peripheral I/O. But this has two major shortcomings. Firstly, code must be segmented into 64kB or smaller chunks, which is a time-consuming task that must be redone each time the code is modified. Secondly, software routines must be written to manually switch the I/O lines to their appropriate state each time the code transitions between segments. The software overhead associated with these efforts degrades performance.

A better solution would implement a device with a larger address bus that addresses more memory. One such device, the DS80C400, has a 24-bit address bus that directly addresses 16MB of program memory and 16MB of data memory. This is done without requiring any new op codes in the 8051 instruction set. Two modes are provided. The first is a paged addressing mode, which incorporates advanced automatic bank switching, greatly speeding expanding memory access while remaining binary compliant with traditional 8051 compilers. The second contiguous mode

For more information about maximizing performance while minimizing power consumption, refer to the white paper "Teaching Old Dogs New Tricks: Improving the Power Efficiency of 8051-Based Designs" (www.maxim-ic.com/Design/8051power.pdf).

allows transparent addressing of the entire 16MB memory map, and requires a compiler that provides the extra operand required for the 24-bit addresses. The larger address space allows faster access to larger programs, opening new possibilities such as large libraries of math functions, lookup tables, or even the Java™ virtual machine, supported by networked microcontrollers including the DS80C390 and DS80C400 that execute the Tiny Internet Interfaces (TINI®) runtime environment (See "Exploring the TINI Platform" on page 6).

Data pointers double performance

Far-reaching improvements to all facets of the chip were necessary to avoid creating performance bottlenecks. A most important improvement involved accessing MOVX data memory. Manipulation of data memory on the original 8051 was a cumbersome affair. Accessing a single byte of MOVX memory required multiple cycles to load the single 16-bit data pointer before reading or writing the target address.

... the DS89C420 remains 100% 8051 instruction-set compatible so it still executes existing 8051 applications faster than any other 8051-based microcontroller.

The inefficiencies multiplied if software needed to perform a block copy operation, which involved moving data from one MOVX memory location to another. The single data-pointer limitation forced it to double as both the source and destination address in a block copy operation. The operation on a traditional 8051 microcontroller has been a complicated, multistep procedure:

1) Load the source address into the data pointer.

2) Increment or modify the data pointer to the next datum.

3) Fetch the data from MOVX memory into the accumulator.

4) Save the modified source address to a storage register.

5) Load the destination address into the data pointer.

6) Increment or modify the data pointer to the next datum.

7) Write the data from the accumulator to MOVX memory.

8) Save the modified destination address to a storage register.

The larger address space allows faster access to larger programs such as the Java virtual machine, supported by the DS80C390 and network microcontrollers.

One quickly notes that almost half the steps in the above procedure are dedicated to juggling the source and destination addresses in and out of the single data pointer, which impedes overall performance. The solution adds a second data pointer, creating dedicated registers for the source and destination. With the second data pointer, much of the data manipulation can be handled in hardware, reducing software overhead. The dual data pointers are individually addressable, and a dedicated data pointer select bit indicates which data pointer is the active data pointer during MOVX instructions. The same block copy operation performed with dual data pointers takes many fewer steps:

Perform initialization only once:

1) Initialize the source address into the first data pointer.

2) Initialize the destination address into the second data pointer.

Main loop:

1) Fetch the data into the accumulator.

2) Increment or modify the first data pointer to the next source datum.

3) Switch data pointer selector to second data pointer.

4) Write the data from the accumulator to MOVX memory.

5) Increment or modify the data pointer to the next datum.

Figure 4 shows how a 1000-byte block copy routine on a 33MHz DS89C420 takes 33% less execution time when dual data pointers eliminate the overhead associated with juggling a single data pointer. Some members of the high-speed and ultra-high-speed microcontroller families also have additional optional data pointer enhancements. The auto increment/decrement feature (denoted as AID in Figure 4) automatically increments or decrements the data pointer following a MOVX-related instruction, eliminating the need for the INC DPTR instruction. The auto-toggle feature (denoted as TSL in Figure 4) automatically toggles the active data pointer following a MOVX-related instruction, eliminating the instruction that switches between data pointers. Figure 4 shows the relative execution times when all these features are considered together. Note that with all features enabled, the DS89C420 performs a 1000-byte block copy routine 103% faster than the original 8051 microprocessor.

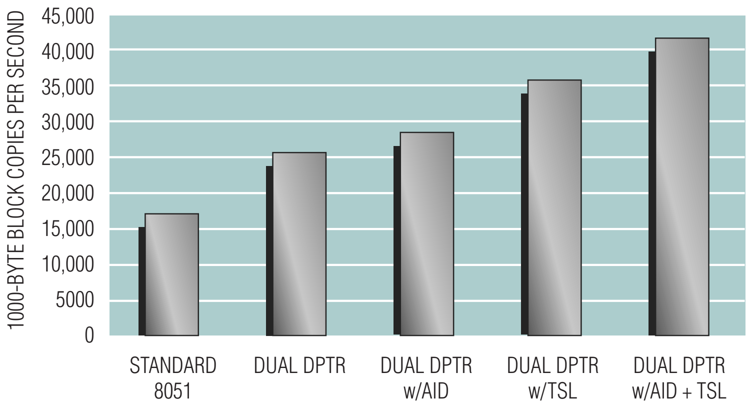


Figure 4. Dual data pointer enhancements improve the speed of block copy operations.

Looking ahead

As applications demand more and more speed, Dallas Semiconductor works harder to exceed previous performance designs. Whether it is faster stack accesses, expanded addressing, or just raw processing speed, our microcontroller designs continue to meet the needs of embedded system designers.

But competitive designs demand more than just speed. More sophisticated applications require larger programs, so we are expanding our line of drop-in 8051 microcontrollers to include 64kB of flash memory. Our new-product pipeline has peripherals in design to increase the capabilities of their embedded systems while simultaneously reducing board space. The networked microcontrollers have advanced features including CAN, Ethernet, and 1-Wire® net connectivity for multitier networking. Secure microcontrollers have hardware-based math accelerators for public-key cryptography and support rapid zeroization of the keys as a tamper response. Mixed-signal microcontrollers perform real-world signal processing necessary to make better end equipment.

The road map in **Figure 5** demonstrates our commitment to performance. We are continually refining the microcontroller core and fabrication processes to optimize performance. Our original 8051 was clocked at 12MHz; that microcontroller would have to be clocked at 1.2GHz to achieve the performance of what is soon available. Whether it is high-performance processors or development-tool support, Dallas Semiconductor will meet the needs of embedded systems designers for years to come.

Java is a trademark of Sun Microsystems.
I-Wire and TINI are registered trademarks of Dallas Semiconductor.

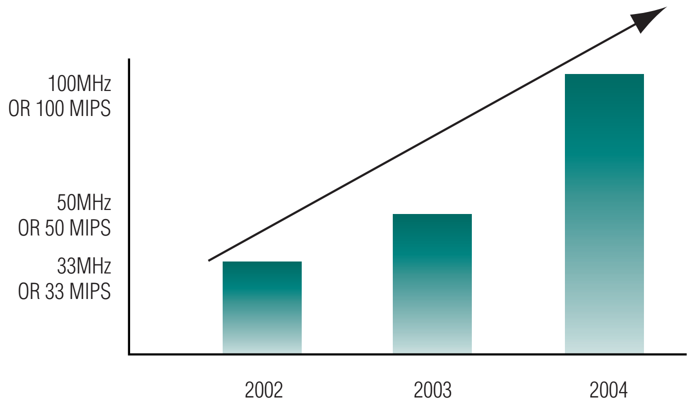
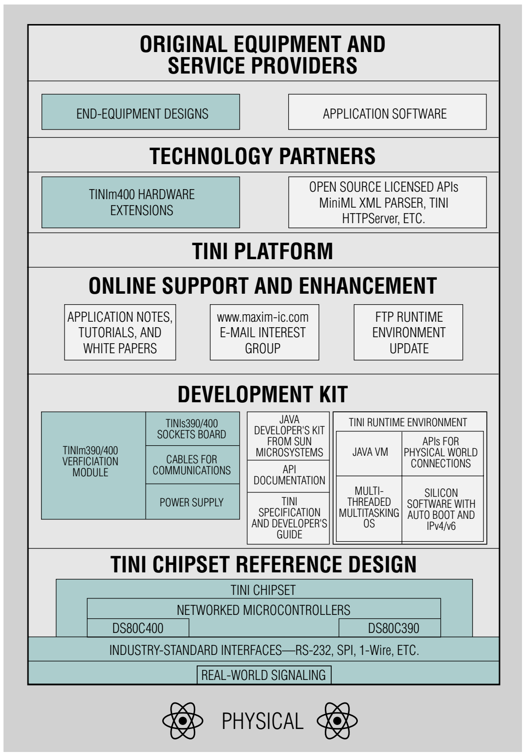


Figure 5. Performance of our 1-clock core is expected to break the 100 MIPS barrier in the near future.

Exploring the TINI platform

The TINI platform consists of a microcontroller-based chipset and supporting firmware, both created by Dallas Semiconductor. This platform, along with hardware and software development kits, allows rapid prototyping and deployment of IP network-enabled, real-world measurement and control systems using the industry-standard Java programming language (**Figure 1**).



ORIGINAL EQUIPMENT AND SERVICE PROVIDERS
END-EQUIPMENT DESIGNS
APPLICATION SOFTWARE
TECHNOLOGY PARTNERS
TINIm400 HARDWARE EXTENSIONS
OPEN SOURCE LICENSED APIs MiniML XML PARSER, TINI HTTPServer, ETC.
TINI PLATFORM
ONLINE SUPPORT AND ENHANCEMENT
APPLICATION NOTES, TUTORIALS, AND WHITE PAPERS
www.maxim-ic.com E-MAIL INTEREST GROUP
FTP RUNTIME ENVIRONMENT UPDATE
DEVELOPMENT KIT
TINIm390/400 VERFICIATION MODULE
TINIs390/400 SOCKETS BOARD
CABLES FOR COMMUNICATIONS
POWER SUPPLY
JAVA DEVELOPER'S KIT FROM SUN MICROSYSTEMS
API DOCUMENTATION
TINI SPECIFICATION AND DEVELOPER'S GUIDE
TINI RUNTIME ENVIRONMENT
JAVA VM
APIs FOR PHYSICAL WORLD CONNECTIONS
MULTI-THREADED MULTITASKING OS
SILICON SOFTWARE WITH AUTO BOOT AND IPv4/v6
TINI CHIPSET REFERENCE DESIGN
TINI CHIPSET
NETWORKED MICROCONTROLLERS
DS80C400
DS80C390
INDUSTRY-STANDARD INTERFACES—RS-232, SPI, 1-Wire, ETC.
REAL-WORLD SIGNALING
PHYSICAL

Figure 1. The TINI platform allows rapid prototyping and deployment of IP network-enabled real-world measurements.

Getting started with TINI

The fastest way to begin developing an embedded application is to use a pre-built, proven reference design for the hardware portion of the system. The TINI Verification Module (TVM) was developed for this purpose; it also serves as a reference design for the DS80C400 network microcontroller that forms the center of the TINI chipset contained on the TVM. Dallas Semiconductor provides complete schematics and a parts list for the TVM so that all or part of the design can be reproduced to meet the requirements of a specific project. In many cases, only a subset of the complete TINI chipset is needed for a targeted, end-equipment solution. The TINI Verification Module allows software development to begin using a pretested design while the development of a more optimized hardware design proceeds in parallel, reducing overall time-to-market.

The TINIm400-144-02 is a TVM implemented on a 144-pin SO DIMM, a form factor made popular by notebook PC DRAM. Together with the accompanying TINIs400 sockets board, it forms a comprehensive development system that includes the following features:

- DS80C400 processor running at 29.5MHz
- 1MB battery-backed SRAM and 1MB flash ROM
- 10/100 base-T Ethernet connection
- Two 1-Wire ports (one for internal on-board use and one for external connections)
- Two RS-232 serial ports, including full-flow control lines on one port; and a CAN and SPI™ port

Besides the TINIm400 module and TINIs400 sockets board, the only hardware needed is a power supply (8V to 20V AC/DC) and appropriate cables for connections to the sockets board such as Cat 5 for Ethernet, 9-pin sub D for serial, and RJ11 for 1-Wire. All software required to develop and run Java applications on the TINI platform is available as a free download from www.maxim-ic.com and www.sun.com.

From the ground up

Consider this application example—a remote agricultural station needs to monitor temperature, rainfall, and humidity levels and adjust an irrigation system based on measured weather patterns. A personal computer could perform this task, but its uptime might not be sufficient for an unattended application of this type. As a sub-PC system, TINI is much less expensive, more compact, requires far less power, and is easier to maintain. In addition, TINI supports many low-level communications interfaces that PCs generally do not.

With the wide range of industry-standard interfaces supported by TINI, a broad range of sensors and actuators can be used to collect the weather data and control the irrigation system. If a device uses an interface not directly supported by TINI, custom I/O libraries can be used to map the device onto the TINI memory bus with appropriate support circuitry.

The TINI OS is multitasking and multithreaded, so the agricultural station software can simultaneously communicate with multiple devices while it processes data in the background (**Figure 2**).

All software required to develop and run Java applications on the TINI platform is available as a free download from www.maxim-ic.com and www.sun.com.

Once a data path has been established between the sensors/actuators and TINI, incoming data must be recorded and analyzed by software. TINI's ability to run Java code frees developers from the need to be familiar with the internal details of the DS80C400 processor. However, time-critical sections of code can be customized if necessary, using the Java native-method mechanism to include highly optimized assembly code.

The TINI runtime environment contains a full Java VM and API (applications programming interface) that includes a subset of the Java 1.1 API as well as additional functionality unique to TINI, such as device I/O routines for the specialized communications protocols. Java's robust networking API and its emphasis on security and memory management make it an ideal choice for the TINI environment. The Java support that TINI provides allows applications to be developed using one of the many integrated development environments (IDEs) available for Java. Applications can also be developed on another platform such as a PC and ported to TINI when they are complete.

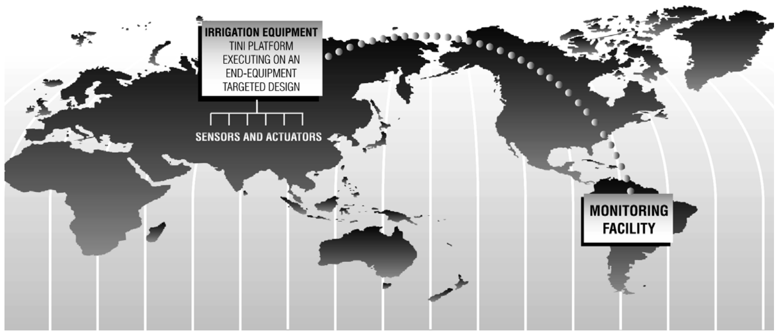


Figure 2. By using the TINI platform, equipment can be monitored and controlled over wireless or wired networks.

If the Java support in TINI is not needed, it can be removed without sacrificing all the functionality that TINI provides. The core of the TINI OS is contained in the ROM of the DS80C400 and includes a full IPv4/IPv6 network stack as well as automatic network boot capability using TFTP. These capabilities can be used without the Java VM, and applications running in this manner can be written either directly in assembly language or compiled from C.

From local to global

The agricultural station could be considered complete at this point if a local, closed-loop control system was the goal. However, without wider network capabilities, any data collected by the station must be retrieved manually. If the software needs to be updated, that must be done manually as well. In addition, there would be no way to verify that the station is running properly without actually traveling to its location and checking.

Almost all systems can benefit from some level of networking, even if it is only used for maintenance. The standards-based networking of TINI makes adding this capability straightforward. Once the network connection is in place, applications can be tested and updated remotely and multiple TINI installations can be managed from a single location.

TINI is flexible enough to adapt to different networking requirements. Connecting TINI to an Ethernet network is the simplest route and provides the highest speed, but Ethernet is not always available. The agricultural station can be in an isolated location with limited connections to the outside world. In this case, TINI's dialup PPP networking capability requires only a modem and standard phone line, cellular phone,[1] or equivalent system to connect to the Internet.

Once the network connection has been established, the range of possible uses is vast. TINI includes support for standard Internet protocols such as TCP/IPv4/v6, DNS, DHCP, HTTP, and FTP. The agricultural station could host its own web page or provide an FTP interface to download collected data with minimal coding required. If a specialized protocol is required, TINI's complete implementation of the java.net API allows creation of any type of network interface desired.

The default system shell that is included with the TINI runtime environment provides additional flexibility during application development. This shell features a Unix-like environment with a password-protected network login for multiple users over Telnet. It also includes FTP capabilities

[1]For an example, refer to idenphones.motorola.com/iden/developer/news_dallas.jsp.

As a sub-PC system, the TINI chipset is much less expensive, more compact, requires far less power, and is easier to maintain. . . (it) supports many low-level communications interfaces that PCs generally do not.

so Java applications may be uploaded to the TINI file system and then tested and debugged from a Telnet session.

Building beyond the platform

TINI ends at the boundary of the Java runtime environment, where real-world application development begins. To accelerate the design of their own products and services, first-time developers might want to evaluate the hardware and software offered by more experienced developers.

Some of the tools and libraries produced by TINI technology partners include the following:

- TiniAnt (tiniant.sourceforge.net/): Extension to Java Ant that simplifies building applications for TINI
- MinML and MinML-RPC (www.wilson.co.uk/xml/minmlrpc.htm): XML parser and XML-RPC remote procedure call library optimized to run on TINI
- TiniHttpServer (www.smartsc.com/tini/TiniHttpServer/): Full-featured web server designed specifically for TINI
- An X10 Library (www.jpeterson.com/rnd/): Allows control of X10 home automation devices from TINI
- TINI Rapture (sourceforge.net/projects/tinirapt/): Cron-style daemon used to start applications on TINI automatically
- Java IrDA Lite (sourceforge.net/projects/jir/): IrDA Lite implementation that runs on TINI

The TINI runtime environment contains a full Java VM and API that includes a subset of the Java 1.1 API as well as additional standards-based functionality.

Table 1. Example applications for the TINI platform

Serial-to-Ethernet Black Box	Humidity Monitor	Transaction Terminal
Home Monitoring/Automation	Barcode Printer	Parking Gate Controller
Temperature Monitor/Logger	Ticket Printer	Traffic Light Controller
Vending Machine Controller	Audio Paging Controller	Concrete Cure Monitor
Weather Station Monitor	Message Display Server	Lighting Controller
Web Camera Controller	19-Inch Rack Monitor	Virtual Software Modem
Remote Print Server	Server Room Monitor	Power Monitor
Networked MP3 Player	Smart Card Reader	Utility Meter
Door Lock Controller	Magnetic Card Reader	RFID Reader
Time/Attendance Terminal	Barcode Reader	Security Sensor Controller

For more information

For the latest TINI downloads and API documentation, visit www.maxim-ic.com/TINI. The 300+ page *TINI Specification and Developer's Guide* is also available online and contains many useful examples and explanations of the TINI platform.

Dallas Semiconductor maintains a mailing list for the TINI community. To subscribe, visit www.maxim-ic.com/TINI/lists.

SPI is a registered trademark of Motorola, Inc.

Asynchronous serial-to-Ethernet device servers

The sheer number of devices that use a serial port as a means for communicating with other electronic equipment is staggering. In fact, for many, a serial port provides the sole mechanism of communicating with the outside world. This includes thermostats, point-of-sale systems, remote monitors, barcode readers, receipt printers, RFID transceivers, blood-pressure meters, and many more in fields ranging from legacy test tools to the latest in building automation. Such devices have no direct means of participating in a larger computer network, yet new applications demand TCP/IP connectivity and Ethernet capabilities. Often, an expensive and time-consuming redesign is not an option.

. . . for many devices, a serial port provides the sole mechanism of communicating with the outside world.

This article explores an easy, economical way to move standalone serial devices to the Ethernet by retrofitting legacy systems built on the TINI platform using the DS80C390 or DS80C400 microcontrollers. Once a device is connected to the Ethernet, implementing TINI web services such as an HTTP server is straightforward.

RS-232 serial port

The asynchronous serial communication discussed in this article is based on the RS-232-C standard that dates back to the earliest days of recorded computer history. RS-232-C was published in 1969[1]. Most modern serial ports do not support all the signals defined in the standard. Moreover, the signals that are implemented are used in a fashion that is merely "fairly close" to that defined in the standard. We'll ignore the purely historical definitions, and concentrate on the way RS-232 is used today.

Space and mark

RS-232-C specifies a voltage level of +3V to +25V as "SPACE" (binary 0) and -3V to -25V as "MARK" (binary 1). The region between -3V and +3V is the "switching region." Many universal asynchronous receiver transmitters (UARTs) use the more modern (in relative terms) TTL voltage levels of 0V and +5V for 0 and 1. Special-purpose level translators, like the famous MAX-232, convert between TTL and RS-232 levels. Since the serial ports on the DS80C390/DS80C400 are TTL-level, no level translators are needed when interfacing to another TTL-level UART.

DCE and DTE

Data communications equipment (DCE) and data terminal equipment (DTE) are the two endpoints of a communications channel. The main difference is the serial connector pinout. A so-called null-modem can be used to convert between the two.

Table 1 shows the signals on a DB-9 DTE serial connector and the corresponding signals on another DTE when using a null-modem.

Flow control

Serial communication can be realized by sending on one pin (TD) and listening on another (RD). However, when two devices that communicate over RD/TD transmit at will, one might overrun the other, resulting in data loss. There are two ways flow control is commonly implemented:

- XON/XOFF (often loosely termed software flow control)
- RTS/CTS (often loosely termed hardware flow control)

[1]NASA has trouble deciphering computer tapes from this era, so the comparison is valid.

DTE PIN	SIGNAL NAME	NULL-MODEM
1	CD (Carrier Detect)	4 (DTR)
2	RD (Receive Data)	3 (TD)
3	TD (Transmit Data)	2 (RD)
4	DTR (Data Terminal Ready)	6 (DSR) and 1 (CD)
5	Common (Signal Ground)	5 (Common)
6	DSR (Data Set Ready)	4 (DTR)
7	RTS (Request To Send)	8 (CTS)
8	CTS (Clear To Send)	7 (RTS)
9	RI (Ring Indicator)	N/C

Table 1. *A null-modem can be used to connect the corresponding signals of two DB-9 DTE serial ports.*

The XON/XOFF flow-control scheme transmits in-band characters that cause the other side to pause (XOFF, 13h) and resume (XON, 11h) transmission. The XON and XOFF characters must be escaped in software by the sender and unwrapped by the receiver if they occur in a binary data stream.

RTS/CTS uses extra signaling lines. RTS (request to send) is asserted by the sender. The receiver responds with CTS (clear to send) when it is ready to receive data and clears CTS when its receive buffer is full.

Some devices support flow control, some don't. Thus, the default setting is usually set to "no flow control," which should be overridden if a device is known to implement flow control.

Speed, data bits, stop bits, and parity

Other parameters that must be set correctly for successful communications are the transmission speed (bit rate), the number of data and stop bits, and the type of parity checking (if any). Most new devices use a setting of "8N1," which means 8 data bits, no parity, and 1 stop bit. However, legacy systems are known to use the full range of possibilities, so the correct setting might not be trivial.

TINI and networking

TINI is a technology platform developed by Dallas Semiconductor to allow rapid development on the DS80C390 and DS80C400 microcontrollers. Specifically, TINI encompasses a chipset definition, and an embedded operating system integrated with a highly optimized Java runtime environment. Using Java, programmers benefit from powerful features not commonly found in embedded development: multithreading, garbage collection, inheritance, virtualization, cross-platform capabilities, powerful networking support, and, last but not least, a multitude of free development tools. TINI users are usually shielded from assembly language coding. However, native language subroutines are supported and encouraged to optimize speed-critical paths or low-level hardware access. (The TINI operating system is written in native code, resulting in serial I/O throughput not significantly different from modern PCs.)

TINI encompasses a chipset definition, and an embedded operating system integrated with a highly optimized Java runtime environment.

In addition to full support of the *java.net* package, the TINI Java runtime also contains an implementation of the *javax.comm* subsystem. Since both TCP/IP and the serial ports are effortlessly accessible from Java, the TINI system easily lends itself to implementing serial-to-Ethernet bridges.

The TINIm390 verification module on an E10 socket, which is used in the following examples, is the hardware portion of the DS80C390 TINI development platform (the TINIm400 uses the DS80C400). In addition to SRAM, flash memory, Ethernet, CAN-bus, 1-Wire, etc., the system also has four serial ports. Two of the UARTs are internal to the DS80C390 (called *serial0* and *serial1*); two ports are external (using a 16550 build option). It is important to note that both serial connectors on the E10 socket are wired to *serial0* and just differ in DTE/DCE pin assignment.

The TINI environment is documented in great detail in *The TINI Specification and Developer's Guide* (Addison-Wesley, 2001). A free PDF copy can be downloaded from www.maxim-ic.com/TINIguide.

Examples

We'll start with two concrete applications and then present a short excerpt from a generic serial-to-Ethernet program that can be modified to suit almost any application. The examples are built using the TINIm390/400 verification modules.

The TINI verification module serves as a "black box" to connect multiple serial devices to the Ethernet. Depending on end-equipment requirements, the TINI can either pass the data straight through or parse, interpret, and modify the data stream (**Figure 1**).

Although you can run the examples from the slush developer's shell on the TINIm390/400, a more polished application would reside in flash, be self-starting in the event of a power loss, and use other TINI construction techniques to make the finished product virtually indestructible.

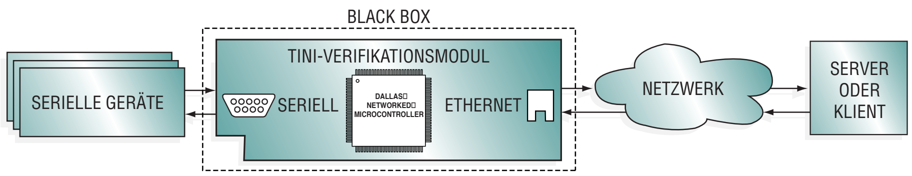


Figure 1. The TINI device server is used as a bridge between serial devices and Ethernet.

Some basic networking knowledge and programming experience are required to modify the examples. Working sample code is also downloadable from the Dallas ftp site (ftp://dalsemi.com).

Virtual modems

The first example, a "Virtual Modem,"[3] uses the TINIm390/400 to replace a physical modem and telephone line with TCP/IP connectivity. Assume a legacy device like a factory "machine status monitor" that uses a modem to dial into a central server several times a day to report machine status, load, and efficiency data. To eliminate the need for an ever-growing modem bank on the server side and to be able to use an existing LAN instead of phone lines to the equipment, one could

- rewrite the server software to be TCP/IP based, and
- use TINI virtual modems to replace the original modems at each machine.

The machine status monitors, however, do not have to be modified since the virtual modem behaves like a real modem as far as the end equipment is concerned!

Virtual modems can, of course, also be used in pairs instead of the configuration described above. When using two virtual modems, no server software needs to be changed at all and the TINI modules are a drop-in replacement for existing modems.

Behind the scenes, a virtual modem establishes a TCP connection whenever it receives the "ATD" modem dial command. An "ATH" disconnect command closes the TCP connection. The software also implements a number of other classic AT modem commands and is recognized as a true modem by Microsoft® Windows® networking, for example. In addition, a virtual modem listens on a TCP port and can answer incoming "calls" signaled by a "RING" to the end equipment.

In addition to full support of the java.net package, the TINI Java runtime also contains an implementation of the javax.comm subsystem. Since both TCP/IP and the serial ports are effortlessly accessible from Java, the TINI system easily lends itself to implementing serial-to-Ethernet bridges.

The following code fragments show how to initialize a serial port on the TINIm390:

```
 public static void main(String args[])
{
  TINIOS.setSerialBootMessagesState(false);
  TINIOS.setDebugMessagesState(false);
  TINIOS.setConsoleOutputEnabled(false);
  System.out.println("Connecting to serial0 at 9600bps, "
                     "listening on TCP port 8001");
  try {
    CommPortIdentifier portId =
    CommPortIdentifier.getPortIdentifier("serial0");
    SerialPort port = (SerialPort) portId.open("VModemTINI",
    10000);

     TINIOS.setRTSCTSFlowControlEnable(1, false);
     TINIOS.setRTSCTSFlowControlEnable(0, true);
    TCPSerialVirtualModem modem = new
    TCPSerialVirtualModem(port,
```

[3]Refer to *Application Note 196: Designing a Virtual Modem Using TINI* at www.maxim-ic.com.

```
                /* Comm speed */ 9600, /*TCP Port */ 8001);
      modem.processInput();
    }
    catch (Exception e) {
      System.out.println("Exception: "+e.toString());
    }
  }
```

The code first disables all TINI OS debug output, standard practice on the TINI. After getting a port identifier, the port is then opened (the second parameter tells how long to wait if the port is currently in use by another application). Next, the state of the hardware flow control is set. Since the TINIm390 has only one set of RTS/CTS lines for serial ports 0 and 1, a program should always disable flow control on the other port before enabling it on the desired port. Next, a Java virtual modem is instantiated.

The virtual modem class consists of an AT command interpreter (not shown here, although by far the largest part of the example) and networking code. The following code sets the serial port bit rate, data and stop bits, as well as parity, and shows how easy it is to handle inbound connections:

```
  /** Creates a new VirtualModem connected to a serial port on
   *  one end and a TCP port on the data side.
   * serial  -- the serial port this VirtualModem talks to.
   * speed   -- the speed the serial port should be set to.
   * tcpport -- the TCP port this VirtualModem listens on.
   * throws IOException when there's a problem with the serial
or TCP port.
   */
  public TCPSerialVirtualModem(SerialPort serial, int speed, int
tcpport)
          throws IOException
  {
    super(serial);

    try {
      serial.setSerialPortParams(speed, SerialPort.DATABITS_8,
                                         SerialPort.STOPBITS_1,
SerialPort.PARITY_NONE);
    }
    catch (UnsupportedCommOperationException e) {
      throw new IOException();
    }
  ...

    serverSock = new ServerSocket(tcpport, 1); // backlog of one
    listenThread = new listenInbound();
    listenThread.start();
  }
```

Finally, the following excerpt of the `listenThread()` accepts an incoming connection request:

```
  public void run()
    {
      int rc;
      Socket s;
      while (running) {
        s = null; // No incoming connection request
        try {
```

```
        answered = false;
        s = serverSock.accept();

        // Discard incoming connection if already connected
        if (connected)
          throw new IOException();

        sock = s; // for answer()
        ...
```

UPS monitor

The second example connects a TINIm390/400 to a serial port of an uninterruptible power supply. The software implements the Network UPS Tools protocol[4], allowing a variety of clients on a variety of platforms to monitor the UPS state and health. This project originated from the need to monitor an existing UPS from a new Macintosh computer without serial ports.

The TINI verification module can be used as a "black box" to connect multiple serial devices to the Ethernet. Depending on the end-equipment requirements, the TINI can either pass the data straight through or parse, interpret, and modify the data stream.

There are two basic kinds of UPS devices: so-called "smart" ones and simple (or "dumb") ones. A simple UPS signals its state on several serial pins; it does not actually send any ASCII data. Because there are not many serial pins, it can only report a limited set of information, for example:

SIGNAL	MEANING
RTS *(from UPS)*	Low Battery
TD *(from UPS)*	On Battery
CTS *(to UPS)*	Kill UPS Power

The `javax.comm.notifyOn…()` methods can be used in Java to easily implement code that reacts to status changes, for example:

```
    ...
    // Listen for DTR changes
    try {
      port.addEventListener(this);
    } catch (TooManyListenersException e) {
      ...
    }
    port.notifyOnDSR(true);
    ...

  public void serialEvent(SerialPortEvent ev)
  {
    try {
      if (ev.getEventType() == SerialPortEvent.DSR)
        ...
    } catch ...
    ...
  }
```

A smart UPS is more interesting, since it implements a serial protocol and can return values like the battery charge percentage or the temperature. Protocols are vastly different between different vendors and often undocumented. The UPSMonitor example on the Dallas ftp site supports an APC SmartUPS, but could be easily modified for other brands.

[4]See www.exploits.org/nut/.

The following code shows how to receive UDP requests and send out UPS status information over UDP.

```
// Listen to incoming UDP requests
private class listenUDPThread extends Thread
{
  private DatagramSocket sock;
  private byte[] buffer;
  private DatagramPacket dp;

  public listenUDPThread(DatagramSocket s)
  {
    sock = s;
    buffer = new byte[BUF_SIZE];
    dp = new DatagramPacket(buffer, buffer.length);
  }

  public void run()
  {
    while (running) {
      try {
        sock.receive(dp);
        byte[] data = parseCommand(buffer, dp.getLength());
        sock.send(new DatagramPacket(data, data.length,
                  dp.getAddress(), dp.getPort()));
      }
      catch (Exception e) {
      }
    }
    try {
      sock.close();
    }
    catch (Exception e) {
    }
  }
}
```

...serial and TCP ports are abstracted as Input/OutputStreams `dataIn` ***and*** `dataOut`***... to bridge data between the CAN and 1-Wire....***

Because of the powerful networking support built into Java, this example is almost self-explanatory. The code in the `while()` loop waits until it receives a UDP request, parses it, and sends out an answer to the originator of the request using `getAddress()` on the incoming packet.

Generic serial-to-Ethernet application

A complete serial-to-Ethernet example is beyond the scope of this article. (A complete example is shown and explained in the *The TINI Specification and Developer's Guide.*) However, the following code fragment shows how to efficiently use multithreading to transfer data between the serial and networking portions of a serial-to-Ethernet bridge. The serial and TCP ports are abstracted as Input/OutputStreams `dataIn` and `dataOut`, so this layer of the code does not actually need to know anything about the network, and it could also bridge data between the CAN and 1-Wire, for example.

```
public GenericBridge()
{
  ...
  running = true;
  dcThread = new dataCopy();
  dcThread.start();
}
```

```
// Thread that copies everything from dataIn to dataOut
private class dataCopy extends Thread
{
  public void run()
  {
      int r = 0;
      while (running && r >= 0) {
        try {
          synchronized (threadLock) {
            r = dataIn.read(dataBuffer);
            if (r > 0)
              dataOut.write(dataBuffer, 0, r);
          }
        }
        catch (Exception e) {
          r = -1;
          ... // Handle error
        }
      }
    }
  }
}
```

Conclusion

Many legacy devices only support asynchronous serial communications, yet current applications demand Ethernet connectivity and TCP/IP networking. Using the powerful Java runtime and the TINI technology on the DS80C390 and DS80C400 microcontrollers, developing a serial-to-Ethernet converter is easy and can be done in a matter of hours.

Microsoft and Windows are registered trademarks of Microsoft Corp.

Designing a networked On/Off switch using the TINI platform

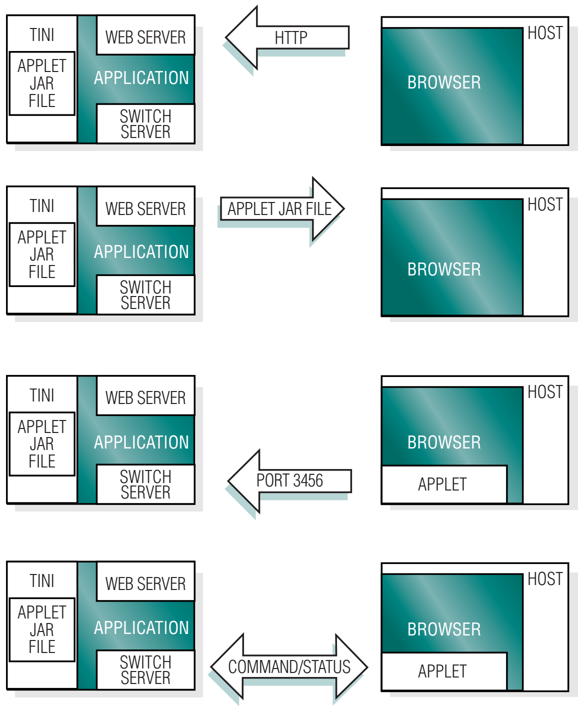


Figure 1. *The web server application software executing on the TINI runtime environment uses an HTTP connection to transfer an applet to the host and sets up a two-way connection for transferring command and status data.*

The TCP/IP network stack and local control capabilities needed to design an IP-networked relay are provided by the TINI platform. Adding a Java runtime environment greatly reduces the complexity with which small sensors and actuators can be accessed and controlled remotely over a network. The following discussion, which presents an IP On/Off switch constructed with a simple relay circuit and a TINIm390/400 verification module, can be extended to other remote monitoring and control applications as well. Familiarity with object-oriented programming such as the Java language is assumed.[1]

The circuit is controlled with an application called TINIWebServer (slightly modified), which is executed directly by the TINI runtime environment. An applet, served to the host workstation, opens two-way communications back to the TINI runtime environment for commands and status, and displays a graphical user interface for simple remote control of the relay.

System software overview

The class `com.dalsemi.tininet.http.HTTPServer` enables the switch-control application to implement a simple web server whose only purpose is to transfer an applet to the remote host. Executed within the host's browser, the applet establishes a two-way TCP connection for exchanging commands and status with the TINI application. The applet also provides a graphical user interface for displaying controls and status. **Figure 1** represents the overall software system.

System hardware overview

In **Figure 2**, an On/Off control circuit forms the interface for a TINIm390/400[2] verification module. The TINIm390/400 provides Ethernet network connectivity and controls the switch through port pin P5.0 (other port pins work equally well). An n-channel power MOSFET controls the relay by switching current from the relay to ground. You can accommodate various current and voltage requirements by resizing the relay and FET, and the relay can be omitted altogether if you don't need to isolate the external circuit from the TINI verification module's power supply. The diodes protect against voltage spikes from the relay coil while the switch is changing state. To make possible new services such as networked switch control, the hardware and software components have been integrated in the TINI chipset reference design with standards-based Internet technologies.

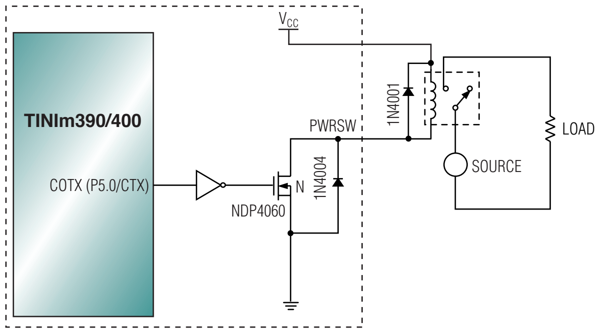


Figure 2. *The TINI chipset reference design controls the switch, which is a simple transistor and relay circuit with protection diodes, using any available port pin.*

The TINI switch-control application

Four classes make up the switch-control and web-server-interface portions of this application. The `PowerSwitch` class interfaces directly to the hardware using the TINI class `com.dalsemi.system.BitPort` API class. The `WebWorker` class comes directly from the TINIWebServer example in our Software Developer's Kit (TINI SDK), and is responsible for servicing the arriving HTTP con-

[1]Readers should particularly understand the terms class, method, object, and constructor in this context.
[2]The TINIm390 data sheet can be viewed at www.ibutton.com/TINI/dstini2.pdf.

nections. The `SwitchWorker` class manages all command and status communications between the applet and the TINI application. The `TINIWebServer` class drives the application by binding together operation of the individual classes.

. . . the applet establishes a two-way TCP connection for exchanging commands and status with the TINI application.

The `PowerSwitch` class is an interface to the hardware, creating a `BitPort` object for pin P5.0 in its constructor. Two methods are implemented in this class. The On method sets the state of the port pin used to apply voltage to the relay's coil; the `Off` method removes voltage from the coil by clearing the state of the port pin. The single pole, double throw (SPDT) relay in Figure 2 can operate in the normally closed or normally open position, or it can switch the external voltage source between those two positions. The `On` and `Off` methods assume the circuit is normally open, and must be modified to accommodate the normally closed circuit. To indicate whether on/off corresponds to the `BitPort` set or `clear` method, an extra boolean variable (`invert`) can be added to this class. Another method (`setInvert`) would be needed to initialize the invert variable. The hardware diagram illustrates a normally open circuit.

The `WebWorker` class provides an interface between the network and the application. It simply sets up and drives an object (`com.dalsemi.tininet. http.HTTPServer`), which in turn spawns a thread that services each incoming HTTP connection. This class is essentially unchanged from the `TINIWebServer` example mentioned above. Accessible from any location on the network, the HTTPServer can prompt for a password, or accept other forms of control that limit access to approved users only.

The `TINIWebServer` allows remote control of the switch by tying together the network and hardware interfaces (**Figure 3**). The `drive()` method, for instance, sets up the web server by creating a `WebWorker` thread and creating the web page "index.html." The primary purpose of the web page is to download and run the applet on the host workstation. The application would not have to create the web page if the index page contained only static information. Rather, the index page could simply be copied to the web server's root along with the jar file containing the applet.

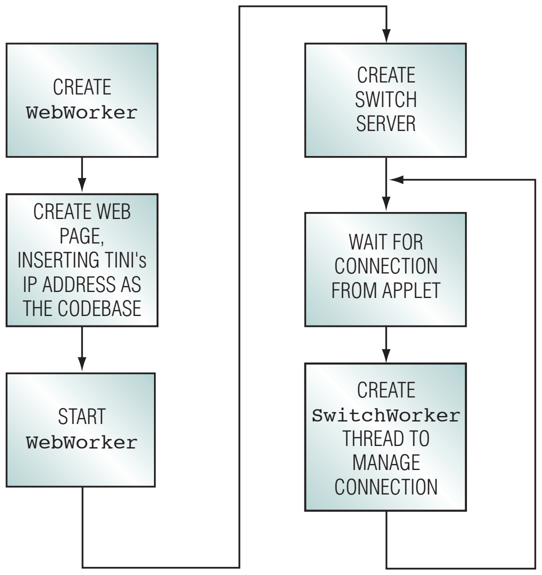

CREATE WebWorker
CREATE WEB PAGE, INSERTING TINI's IP ADDRESS AS THE CODEBASE
START WebWorker
CREATE SWITCH SERVER
WAIT FOR CONNECTION FROM APPLET
CREATE SwitchWorker THREAD TO MANAGE CONNECTION

Figure 3. *The TINIWebServer class creates a webpage and starts a webserver before creating the switch server to service incoming command and status requests.*

The one parameter in the web page that changes on every TINI chipset design is CODEBASE. The applet uses that information to connect back to the TINI application on a separate server socket. A custom web page could be created and uploaded to each TINI chipset reference design that is installed in the field. An easier approach has the page created by the application each time it is run. The `createIndexPage` method creates the file index.html and inserts the IP address into the CODEBASE section using three writes:

1) index.write(indexTop.getBytes(), 0, indexTop.length());

2) index.write(InetAddress.getLocalHost().getHostAddress().getBytes());

3) index.write(indexBottom.getBytes(), 0, indexBottom.length());

The first and third writes copy the static portions of the web page into the file, and the second write copies the IP address to the CODEBASE section of the file. After the application sets up the web server and creates the page, it starts the web server. It creates a server socket to handle incoming connections from the downloaded applets, and calls the `serviceConnection` method each time an applet connects with the TINI application.

The one parameter in the web page that changes with every TINI is CODEBASE.

The `serviceConnection` method creates a new instance of `SwitchWorker` and passes the socket to this class. The `SwitchWorker` constructor creates a new thread to manage the connection between the host applet and the TINI application. The `serviceConnection` method also handles the next incoming connection and then transfers control to the `drive` method.

R/W	SW 6 STATE	SW 5 STATE	SW 4 STATE	SW 3 STATE	SW 2 STATE	SW 1 STATE	SW 0 STATE

Figure 4. The SwitchServer executing on the TINI runtime environment sets up the listener and loops indefinitely, waiting for commands, setting the switch state based on those commands, and sending the updated state back to all listeners.

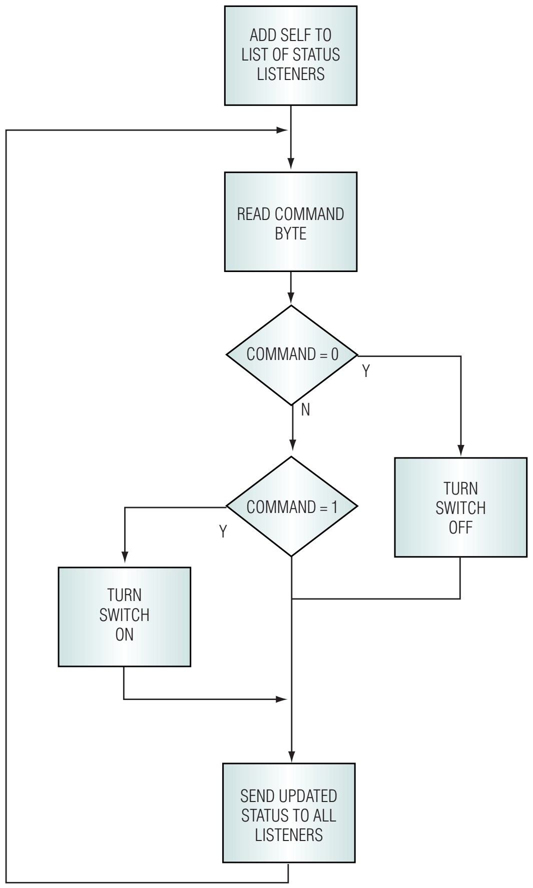

ADD SELF TO LIST OF STATUS LISTENERS
READ COMMAND BYTE
COMMAND = 0
Y
N
TURN SWITCH OFF
COMMAND = 1
Y
TURN SWITCH ON
SEND UPDATED STATUS TO ALL LISTENERS

Figure 5. The command byte simply uses one bit per switch for control of up to seven switches and reserves the last bit to distinguish a status request from a command.

The `SwitchWorker` class manages all communications between the applet and the webserver (**Figure 4**). Until the connection is dropped, it loops continuously, performing the following steps:

- Block on read(), waiting for a command byte from the applet.
- If the command byte is 0, turn the switch off. If the byte is 1, turn the switch on.
- Read the current switch state and send it back to the applet.

The algorithm can be extended to multiple switches by allocating each of the seven lower bits of the command byte to represent the state of a separate switch (**Figure 5**). The most significant bit is reserved to indicate a read-only operation. The algorithm can also be extended to allow multiple applets to connect to a single TINI webserver application at the same time. The `SwitchWorker` simply keeps a "vector of listeners." Each time an applet sends a command to change a switch's state, the webserver sends the status back to all of the applets currently connected.

The host applet

An applet is used on the host because it provides a rich set of graphical objects for the display of controls and status, and enables (among other things) two-way communications. The TINI class `com.dalsemi. tininet. http. HTTPServer` class is fast and small, but it supports only the HTTP GET operation. The resulting data can flow only in one direction from the TINI application to the host. This application, however, requires data flow in both directions; commands are sent from the host to the TINI webserver and status is sent from the webserver to all connected hosts. Communication between the host and the TINI application incurs no protocol overhead. Its 1-byte commands and 1-byte status allow very quick responses for control and status.

The host applet includes two classes: the primary class (`SwitchControl`) handles host-side network communications and creates all graphical elements displayed on the web page (**Figure 6**); the other class (`ImageButton`) creates a graphical toggle button that displays one of two bitmaps according to the button's status (**Figure 7**). The toggle button should be sufficient for control and status of the switch, but the behavior of applets varies from one browser to another. Accordingly, simple `On/Off` buttons and a text window for status were added to accommodate a wider range of browsers. `ImageButton` and `On/Off` buttons perform the same control function, just as bitmaps and the status window perform the same status function.

After creating the graphical elements, the `SwitchControl` class creates a status listener thread. The thread then sleeps, blocked on read, waiting for status from the TINI application. When the thread unblocks, the `ImageButton` bitmap and the status window are updated and the method loops back to the top of the method to await the next status byte. The applet-event thread drives the `actionPerformed` method, which is called each time a graphical button is pressed. If a call is triggered by the `ImageButton` or the `On/Off` button, it toggles the current state and sends an On or Off command to TINI. If triggered by the `On/Off` button, it sends an On/Off command. The `ImageButton` class is simply an exercise in AWT (Abstract Window Toolkit) component programming.

Conclusion

It is easy to implement an On/Off switch controlled remotely over a network with the TINI runtime environment, a Java application, and a simple relay circuit. The large selection of available circuit components makes possible a variety of applications, with control (from any location with network access) of anything from light bulbs to industrial equipment.

Associated files

Files associated with this article are located at ftp://ftp.dalsemi.com/pub/tini/appnotes/NetSwitch/NetSwitch.tgz.

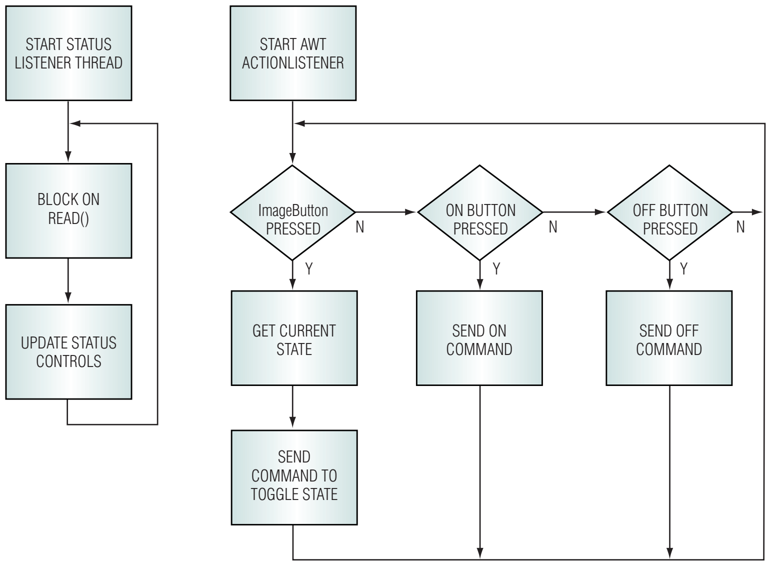
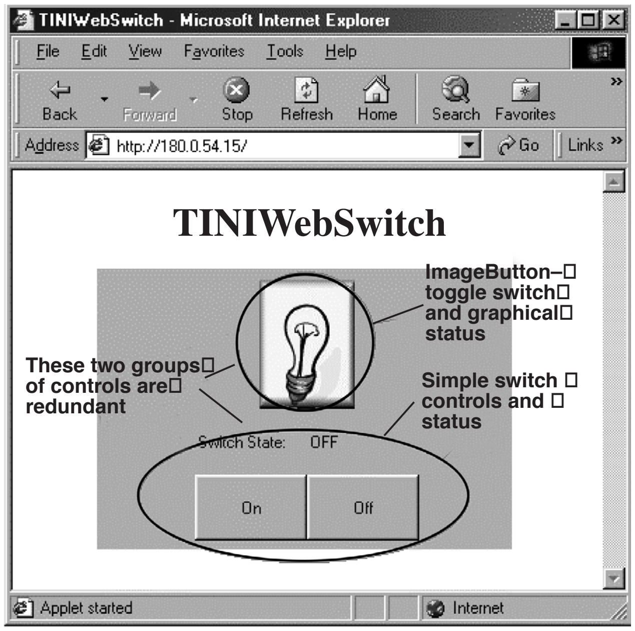


Figure 6. The applet's `SwitchControl` *class uses one thread to read incoming status from the webserver and update the controls. It uses a second thread to detect user input and send the commands back to the TINI webserver.*



TINIWebSwitch - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address http://180.0.54.15/ Go Links
TINIWebSwitch
ImageButton– toggle switch and graphical status
These two groups of controls are redundant
Simple switch controls and status
Switch State: OFF
On
Off
Applet started
Internet

Figure 7. The switch-control applet supplies Off/On buttons and status fields to give remote control and feedback on the physical switch state.

Embedded networking with IPv6

The next generation of IP—IPv6—extends the addressing space to 2^{128}, a number far beyond human imagination, making sure all future devices get a unique address of their own.

Address space for IP nodes is getting tight. Although not all of the 2^{32} (roughly 4 billion) IPv4 addresses[1] have been allocated (and in 2001 there was a slight dip in the previous exponential growth)[2], industry expects to run out of addresses in the course of the next few years. The next generation of IP—IPv6—extends the addressing space to 2^{128}, a number far beyond human imagination, about 6.67 x 10^{23} addresses per square meter of our planet[3]. This will ensure all future devices get a unique address of their own.

Having enough addresses eliminates the need for network address translation (NAT)[4], temporary address leases, and other kludges necessary to conserve the strictly rationed IPv4 addresses. Although there will be significantly more desktop and server computers as well as other classic network devices, a tremendous increase is expected in a different area—multitudes of small devices will change internetworks as we know them today. The new network citizens are always on wireless and mobile devices like GPRS and UMTS cell phones or PDAs. Also included are small embedded devices, monitors, sensors, and smart nodes built into almost anything, from cars to water meters.

But IPv6 not only extends the addressing space. It overhauls IP to make configuration easy and automatic (another must for embedded applications); it makes IP more robust, extensible, and mobile, adds security features and quality-of-service support, and simplifies and speeds up routing. A severe problem plaguing IPv4 is the unimpeded growth of the backbone routing tables due to the almost random way IPv4 addresses were originally allocated. IPv6 is a better, reengineered IP and will gradually replace IPv4—there are just too many advantages to pass it by. Dual IPv4/IPv6 network stacks support mixed environments and allow for gradual adoption of IPv6.

Asia (especially Japan) was one of the first to adopt IPv6 since this region was somewhat shortchanged when IPv4 addresses were assigned initially. India and China have the largest expected internetwork growth, both in relative and absolute numbers. Because of its benefits and government-mandated adoption plans in several countries, IPv6 is becoming increasingly important. It long ago left the prototype stage and is now a standard part of most operating systems, for example, Microsoft Windows XP, Sun Solaris™ 8/9, etc.[5]

This article presents a brief introduction to IPv6 and describes how to use IPv6 networking with the silicon software resident in the DS80C400 microcontroller. Basic network literacy and a reasonable level of comfort with IPv4 are required to benefit from this article.

IPv6 Overview

Addresses

An important part of IPv6's auto-configurability is the way addresses are used. A 128-bit IPv6 address is divided into a 64-bit prefix (net bits or subnet) and 64 host bits. The prefix, which also shows the scope of an address, is either assigned by the network provider[6] and broadcast by routers or it can be local to the link or site. On an Ethernet, the host bits are usually derived from the device's unique MAC (media access control) address (in the form of IEEE EUI-64). This means that an IPv6 node is operational with a valid IP address as soon as it is plugged in. To communicate globally, the node needs to solicit or listen to the router broadcasts containing the prefix and combine the prefix with the EUI-64. Unlike the DHCP addition to IPv4, all IPv6 nodes can be self-configuring, even in the absence of a server.

IPv6 addresses are written in hexadecimal notation in groups of 16 bits, for example `3ffe:aaaa:bbbb:cccc:260:8ff:fe8d:6ee9`, which is an address of global scope.

[1]The current IP is *Version 4 (IPv4)*.
[2]Due to the way IPv4 addresses are allocated, only about 160 million addresses are actually available.
[3]The total Earth surface is approximately 509,917,870 square kilometers.
[4] Or "IP masquerading."
[5]Visit www.ipv6.org for more information.
[6]Usually, 48 prefix bits are assigned by the ISP; 16 are at each site's discretion.

The same machine would have the "link-local" address `fe80::260:8ff:fe8d:6ee9`, where `fe80::/64` is the prefix for link-local addresses; /64 shows the length of the prefix and :: is short for 0s. The loopback host (127.0.0.1 in IPv4 parlance) is simply ::1. Site local addresses have a prefix of fec0::/10. Since there is no direct equivalent to site local addresses in IPv4, these addresses are rarely used now.

. . . IPv6 not only extends the addressing space, it overhauls IP to make configuration easy and automatic.

From a user's view, these long addresses are, of course, normally hidden behind DNS names like `www.maxim-ic.com`. To serve IPv6 addresses, an IPv6-capable DNS server is required[7]. There are no fundamentally new concepts; an IPv6 address entry in the DNS would be created as `example   IN AAAA   3ffe:aaaa:bbbb:cccc:260:8ff:fe8d:6ee9` instead of the `IN A` record used for IPv4. DNS use is strongly encouraged, since IPv6 address prefixes are expected to change more frequently. Network renumbering is much easier than with IPv4 and it can even be automatic.

There are both unicast and multicast addresses in IPv6. In addition, a new *anycast* address destination type was defined. A packet addressed to an anycast IP is delivered to the closest or best host from several hosts. Anycast helps achieve load balancing through routing[8].

Protocols

Although IPv6 keeps the higher layer protocols UDP and TCP without any changes, the IP packet header had to be modified to accommodate the larger addresses. It was also cleaned up and streamlined to be 64-bit aligned and to always have a fixed length for the benefit of routers; the IP header checksum was removed since the higher layer protocols already have a checksum that encompasses parts of the IP header.

An interesting modification is the replacement of ARP with the neighbor discovery protocol (NDP), part of the new ICMPv6. Instead of broadcasting address resolution requests all over the campus, IPv6 maps multicast groups and IPv6 addresses in a way that eliminates these broadcasts and ensures that nodes (almost) only receive traffic that really interests them.

The details of ICMPv6 and multicasting are beyond the scope of this article. Refer to www.ipv6.org for pointers on this and many other interesting IPv6 features.

TCP/IP on the DS80C400

The on-chip DS80C400 silicon software (ROM) contains the latest revision of the field-proven Dallas TCP/IP stack. The silicon software also includes a small operating system and all utility functions needed to develop small C or assembly language TCP/IP network client and server applications with as little as 128kB of external memory. The DS80C400 can also be used with the TINI Java runtime environment when easier and more rapid application development is desired, or when any of the extended Java features like object serialization are required.

The resident C and assembly language support is implemented in the form of a BSD and industry-standard, cross-platform socket interface, i.e., functions like socket(), bind(), listen(), accept(), connect(), send() etc.[9]

The TINI Java environment closely follows JDK 1.1.8 and supports the entire `java.net` package; any Java compliant compiler can be used. The TINI executes standard Java programs and byte codes. An overview and detailed documentation for the TINI runtime can be found on our website (www.maxim-ic.com/TINI).

In addition to network application support, the DS80C400 silicon software also implements network boot functionality, which can load applications over TFTP, supporting both DHCP on IPv4 and, even easier, TFTP over self-configuring IPv6. **Figure 1** shows the DS80C400 network boot over IPv4 and IPv6, respectively. The network bootloader is invoked either by a hardware pin

[7]For example, BIND9 from www.isc.org/products/BIND/.
[8]On IPv4, DNS load balancing (e.g., round-robin) is commonly used, which does not take routing issues into consideration.
[9]These and all other supported functions are documented in the *High-Speed Microcontroller User's Guide: DS80C400 Supplement*. Visit www.maxim-ic.com/MicroUserGuides.htm.

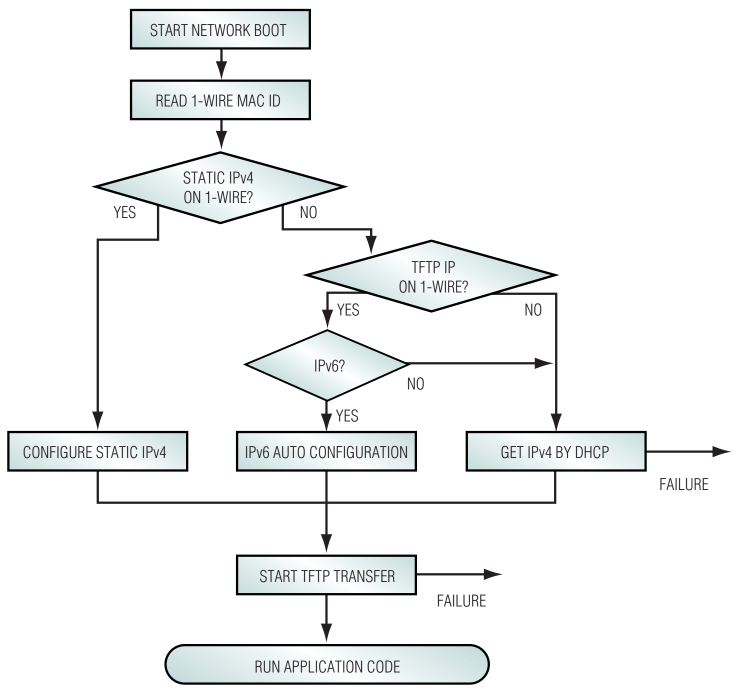


Figure 1. The DS80C400 takes advantage of IPv6 autoconfiguation when booting from the network.

on the DS80C400 or by a user command in the serial bootloader.

IPv6 on the DS80C400

The DS80C400 silicon software supports the IPv6 features needed[10] to participate on the network and follows the "Minimum Requirements of IPv6 for Low-Cost Network Appliances" draft[11]. Because embedded-device resources are constrained, we do not anticipate embedded devices implementing the full IPv6 feature set including security, mobile IP, and routing.

The adoption of IPv6 will be phased in over several years. The DS80C400 network stack, therefore, is an integrated dual stack for both IPv4 and IPv6. There are ways to tunnel IPv6 over existing IPv4 networks (6over4); since the DS80C400 supports both protocol families, it expects routers to tunnel packets if necessary and does not perform protocol conversions itself.

Example 1 runs on the TINI 1.1 Java environment for the DS80C400 and shows fragments of a simple multithreaded network server handling both IPv4 and IPv6 requests. You will not see the IPv6 specific code in this example, because there is none. Applications can usually be ported from IPv4 to IPv4&6 with zero effort; only the printing of IP addresses has to be checked and possibly replaced by a call to TINI 1.1 utility functions provided for that purpose. The TINI 1.1 Java environment adds the Java 2 SE 1.4 Inet6Address class to support IPv6. No other user-visible changes are required, and all other changes are behind the scenes.

The DS80C400 can also be used with the TINI Java runtime environment when easier and more rapid application development is desired or when any of the extended Java features, like object serialization, are required.

Example 2 is the core of a network client written in C that relies solely on the DS80C400 silicon software. Again, there is no IPv6-specific code except for the target address. In the DS80C400 network stack implementation, all network addresses are 128-bit long. Internally, IPv4 addresses are right-aligned and the first 96 bits are set to 0. The example assigns an IPv6 target address and a target TCP port, creates a socket, and then connects to the target.

Example 1. TINI Java Network Server

```
// Listen to inbound TCP connections
private class listenTCPThread extends Thread
{
  private ServerSocket serverSock;
  public void run()
  {
    while (running) {
      try {
        // Create new thread for each client
       Thread client = new clientTCPThread(serverSock.accept());
        client.start();
      }
      catch (Exception e) {}
    }
    ...
```

[10]The IPv6 portion of the DS80C400 Silicon Software was developed in close collaboration with InternetNode, Inc., a joint venture of the Japanese company Yokogawa and Wide Research Institute Co. Ltd.
[11]See www.tahi.org/lcna/.

```
private class clientTCPThread extends Thread
{
  private Socket sock;
  private InputStream  is; private OutputStream os;
  BufferedReader br;

  public clientTCPThread(Socket s) throws IOException
  {
    sock = s;
    is = s.getInputStream(); os = s.getOutputStream();
    br = new BufferedReader(new InputStreamReader(is));
  }

  public void run()
  {
    // Loop while socket is open
    try {
      while (running) {
       os.write(parseCommand(br.readLine().getBytes(), 0));
      }
    }
    ...
```

IPv6 is a better, reengineered IP and will gradually replace IPv4—there are just too many advantages to pass it by.

Example 2. C Network Client

```
{
  struct sockaddr target;
  unsigned int s;
  ...
  /* Fill sockaddr with valid IPv6 target address and port */
  target.sin_addr[0] = 0x3f;
  target.sin_addr[1] = 0xfe;
  ...
  target.sin_addr[15] = 0xe9;
  target.sin_port = 34000;

  /* Open socket and connect to target address */
  s = socket(0, SOCKET_TYPE_STREAM, 0);
  result = connect(s, &target, sizeof(struct sockaddr));

  ... /* Send/receive data here */

  closesocket(s);
}
```

Conclusion

As an evolution and fine-tuning of the IP protocol, IPv6 is becoming more important and crucial for the success of networked embedded devices. IPv6 provides an unlimited number of IP addresses, auto-configuration, and a general streamlining of the IP protocol.

The DS80C400 makes writing an application that supports both IPv4 and IPv6 networking easy. IPv6 offers compelling benefits for all new applications.

Solaris is a trademark of Sun Microsystems.