Apr 06, 2004

App Note 3170: Finding the Middle Ground: Developing Applications with High-Speed 8-Bit Microcontrollers

Software developers for personal computers have a number of advantages over embedded developers. Not only do they develop for systems that rival the power and memory of supercomputers of just a few years ago, but they develop for systems that generally already exist. Embedded developers, on the other hand, not only develop on much smaller systems, but they usually have to design the system first.

An approach must be chosen based on the size of the problem. If the problem has little user interaction, controls a small number of devices, and is a relatively simple design, it can be tackled with a lowpower, 8-bit microcontroller, like an 8051, 68HC11, Amtel AVR, or PIC variant. These usually provide adequate power and flexibility. If the problem has large amounts of user interaction, needs to talk over Ethernet, or needs to talk to complex devices like digital cameras, then usually a PC-104, StrongARM, or another type of "one-calorie personal computers" is used. These generally provide abundant processing power, complex operating systems, and large amounts of RAM.

There is a gray zone in between. For example, let's say Joe's Security Service is entering the cutthroat market of network door locks. Due to recent security alerts, companies want to install door locks that log users with more than just an ID number. They want electronic door locks that take a photo of either the person's face or their thumbprint whenever the user wants to open the door. These images are sent over a network to a central server either for logging or, in a complex example, image recognition and validation. If the image is validated, the server sends back a response to the network door lock, and the door opens for the user. Joe would like to have his customers install many doors throughout their facilities, so it is important to keep costs low.

One of Joe's competitors, Alex's Security Central, is developing a network camera door lock using an 8-bit RISC connected to an Ethernet controller. Joe dismisses this solution as underpowered. He knows that there have been a number of projects involving connecting these Harvard-architecture chips to an Ethernet controller. However, many of these projects are in their infancy, none of them are commercially supported, and the TCP/IP stack is limited by the architecture itself. If talking over the network did not disqualify them, talking to a digital camera

would. An adequate image would require 40kB to 60kB of memory, which has to compete with the code memory space as well. Even if he was using something with non-Harvard architecture, there is just too much work to be done and data to be processed with a traditional 8-bit microcontroller.

Joe's arch nemesis, Troy of Troy's X-Treme Security, is also developing a solution. Joe hates Troy because Troy has no respect for the art and finesse of embedded system design. For romance's sake, we will also say that Troy is dating Joe's ex-girlfriend, Amiga, who left Joe because he spent too much time at the computer (also known as an "occupational hazard" among embedded system developers). Troy is developing a solution using a StrongARM running Pocket PC, which has speedy I/O and networking capabilities. Joe sees this solution as overkill. Beyond taking a photo, the processor will sit idle most of the time. The ideal solution does not need a lot of memory or power, so running embedded Linux or Pocket PC would add unnecessary bloat, and too much cost, for such a simple device.

What Joe needs is a microcontroller powerful enough to handle the network and the camera, but less expensive and functional than a 32-bit solution. It would help Joe if it supported a higher level language than pure assembly in order to simplify development. How will Joe be able to beat Alex's power, Troy's price, and win back his true love? Enter TINI® .

A Network Bridge

TINI, or tiny Internet interfaces, is a product of Dallas Semiconductor. The TINI platform is designed to work as a network bridge: a PC can talk to TINI over TCP/IP, and TINI talks to a sensor, legacy hardware, or other device. TINI offers a number of external interfaces, including 1-Wire® , 2-wire, RS-232 serial, CAN, and SPI™. TINI has a robust networking implementation, and offers support for IPv4, IPv6, DNS, DHCP, PPP, Telnet, and FTP.

There are two reference designs available. The most common, the TINIm390 verification module based on the DS80C390, is a 72-pin SIMM that includes 512kB of flash, 512kB or 1MB of battery-backed SRAM, an Ethernet controller, and a real-time clock. The new version, the TINIm400 verification module based on the DS80C400, is a 144 SO DIMM with similar features, except the Ethernet MAC is built into the DS80C400.

The 390 and the 400 are both 8-bit microcontrollers. In fact, they are both 8051 microcontrollers at the core. However, they have been heavily expanded from their original roots. First, their cores are 4 cycles-per-instruction instead of the standard 12. This gives a triple-speed increase over standard 8051s at the same clock frequency. Second, they have much higher addressing capabilities. The 390 supports 4MB of program memory and 4MB of data memory, and the 400 supports a flat 16MB address space. Third, they support much higher clock frequencies. The 390 can run up to 40MHz, while the 400 can run up to 75MHz. Lastly, they both have integer math accelerators for multiplication and division. They offer a middle range of power between a traditional 8-bit microcontroller and a 16-/32-bit microcontroller.

A unique feature of the TINI platform is the operating system developed by Dallas. It is a royalty-free, multitasking, multithreaded operating system that boasts a Java™ runtime environment and is available as a free download. The core OS and libraries fit into a 512kB flash with enough room for a 64kB application in the last flash bank. The DS80C400 also contains a ROM library for C and assembly programming.

Network Camera

To show what is possible regarding the problem of the network camera, the TINI is used to implement a streaming web camera and to benchmark performance. Rather than use the TINIm400 reference design, a custom, high-speed DS80C400 design is used (Figure 1). The network camera discussed here will take raw images and send them over UDP to a PC host. It will talk to the PC either using host-side software, or through a Java applet served over HTTP.

The camera chosen is the M4088 module, which uses an OmniVision 5017 CMOS chip. The camera is a noninterlaced, black and white digital camera with a 384 x 288 pixel resolution. The camera exposes 8 data lines, 4 address lines, and the chip select, allowing it to be memory mapped easily. It also exposes a vertical sync line that asserts when an image is being taken, a horizontal sync line that asserts for every scan line, and a pixel clock that informs when a pixel is coming. Once the camera is initialized, accessing it from software is easy. The 5017 has an internal clock divider that allows programmatic control of the frame rate. It also has a single frame mode, which allows the host device to control when an image is taken. This is useful, since this design does not have the processor power to handle the camera's top speed of 50 frames/s.

The 400 version of the camera is designed to run at 73MHz (18.4MHz x 4). The 400 has a built-in Ethernet MAC, but it needs an Ethernet PHY and magnetics. It supports many other PHYs, such as HomePNA and HomePlug PHYs. An Intel LXT972A is used in this example, which connects directly to the 400 using a standard MII interface (Figure 2). The PHY requires its own 25MHz clock.

The camera must have 12ns memory for execution. A Hitachi HM62W8511H is used, which provides the necessary access time. On boot, the processor executes bootstrap code from flash that copies the executable image from flash to SRAM, flags the clock quadrupler to be enabled, and jumps to the TINI starting address. The board does not have the battery backup or nonvolatizer on this configuration that is available on the TINIm400, as the high-speed SRAM would drain the battery too quickly. This means that TINIOS does not have a persistent file system. This is not as much of an issue as it seems since, as can be seen later, TINI will build the file system on startup.

TINIOS requires a DS2502 for MAC address storage. While not required, there also is a DS1672 real-time clock connected over I^2C™. This gives TINI access to the clock in software, but provides an additional bonus feature. On boot, if a real-time clock is detected, TINIOS autocalculates the system bus speed and adjusts its internal timers accordingly, including serial

port baud rates, timer ticks, and network timings.

Connecting the camera is straightforward; A0-A3, D0-D8, and WEB hook to the appropriate lines. The CE4 is connected to the camera's CSB line, which maps the camera at 0 x 800000. The VSYNC line of the camera is connected to the P1.1 line for reading, and PSEN to the OEB camera line. The HREF line must be inverted before connecting it to INT1 so that level-triggered interrupts can be used.

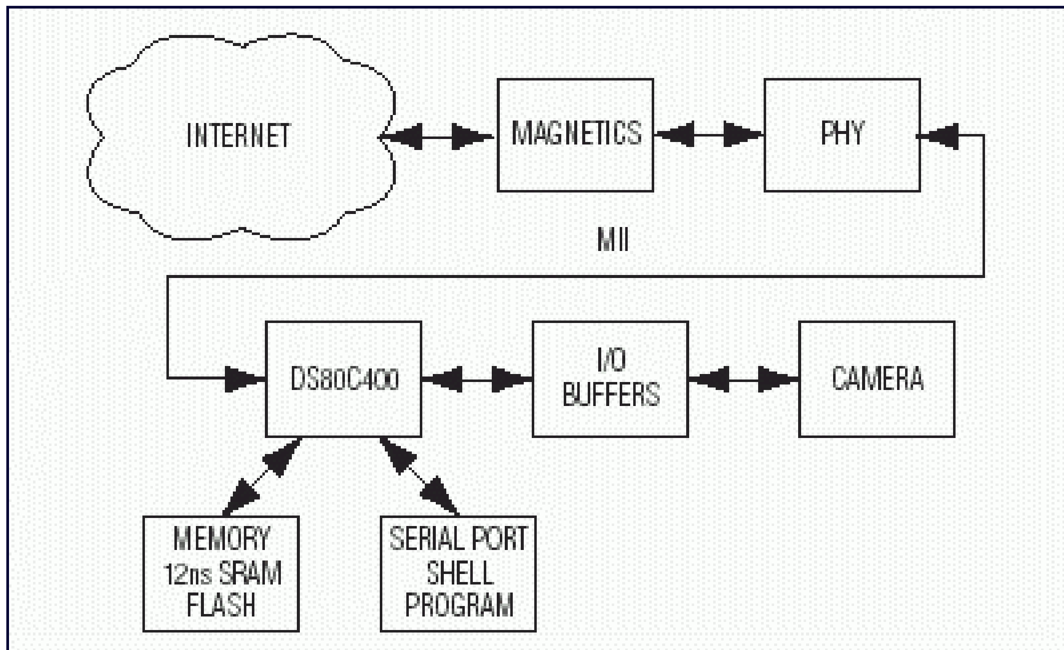


Figure 1. This diagram illustrates the design of the high-speed DS80C400.

Software Implementation

For development, the TINI SDK is used, which is available as a free download from the Dallas Semiconductor website. One might assume that a Java virtual machine would not have sufficient power to capture the camera data fast enough. However, TINI allows for interrupt service routines to be installed under the operating system that communicates with the application using native methods. This shows one of the strengths of the TINI platform; high-level protocols (like HTTP) can be implemented in pure Java, while high-speed portions of code can be implemented with native 8051 assembly. In many ways, it is the best of both worlds.

TINI supports the following packages from JDK 1.1:

java.io
java.lang
java.net
java.util
javax.comm

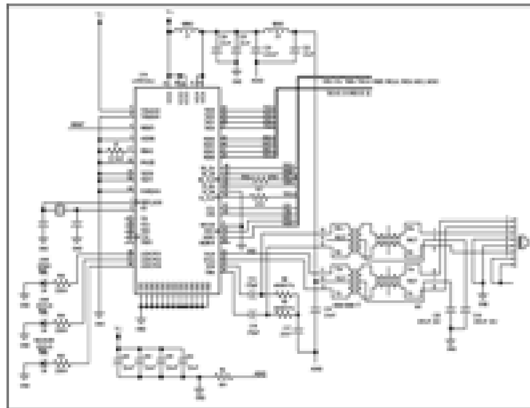
Some differences do exist between TINI's JVM and a PC's JVM. First, while TINI supports

garbage collection, it does not support finalizers. This means the programmer is required to close their resources explicitly instead of having it done by the system, but this kind of resource management is a good idea in embedded development anyhow. TINI also has some size limitations on classes: any individual class file cannot be larger than 64kB, a method can only have 63 locals, and arrays can only be 64kB in length. The system is also limited to 8 processes, and 32 threads per process. Not all the differences are limitations, though. For example, the 1.1 JDK does not have any support for IPv6, so the TINI implementation comes from the 1.4 JDK.

In addition to the standard Java classes, TINI provides the following packages:

com.dalsemi.comm
com.dalsemi.fs
com.dalsemi.system
com.dalsemi.tininet

These packages provide both low-level system access and support for other protocols, like CAN and I^2C. TINI also has support for a number of high-level protocols. For example, one of the examples in the TINI SDK is a lightweight (3kB .class file) HTTP server.



For Larger Image

Figure 2. The DS80C400 connects to the PHY using a standard MII interface.

The software follows the design outlined in Figure 3. At the bottom layer is the camera interrupt service handler, which holds a persistent pointer to the camera buffer in indirect memory. The camera is set to single-frame mode, where it waits for a command before transmitting the image. Once flagged, the image is transmitted synchronously at a rate controlled by the FRCTL register. On the 400, the camera has been set to a rate of 10 frames/s, which transmits a 384 x 288 image in 1/10th of a second at a transfer rate of 1080kB/s.

HTTP SERVER	CAMERA SERVER	CASH
NATIVE METHODS	JVM	
CAMERA ISR	NETWORK	

Figure 3. TINI streaming camera software supports both low-level and high-level protocols.

Talking to the camera using an 8051 assembly is relatively easy. Each pixel of the image is read synchronously from one camera register. Since the camera is memory mapped, reading and writing from its registers involves pointing the data pointer at the camera address and executing a movx opcode. In a traditional 8051 assembly, moving from one address to another would involve loading the data pointer, reading memory into the accumulator, setting the address to have it copied to the data pointer, and writing the accumulator to memory.

```
  mov   R0,#LOW(MEMORY_LOW)
  mov   R1,#HIGH(MEMORY_HIGH)
camera_loop:
;
; Move the camera address into the data pointer.
;
  mov dptr,#CAMERA_ADDRESS
;
; Move the data into the accumulator.
;
  movx a,@dptr
;
; Move to the address we will be writing to. Since
; this will increment every time, we will keep
it stored
; in registers. We will also need to move it one
byte at
; a time using the DPL and DPH SFRs.
;
  mov dpl,R0
  mov dph,R1
;
; Write the accumulator to the address
  movx @dptr,a
;
; Increment the data pointer and store back in
R0 and R1.
;
  inc dptr
```

```
  mov R0, dpl
  mov R1, dph
; Do the loop again...
```

The DS80C400 has four data pointers, allowing for data to be quickly copied from one address to another. This removes all the address swapping, making a copy run much faster.

```
;
; Set up the data pointers. We use the DPS
register to select
; what data pointer we want to use. A data
pointer move allows
; for a 24-bit address to be loaded directly.
;
  mov dps,#0
  mov dptr,#CAMERA_ADDRESS

  mov dps,#1
  mov dptr,#MEM_ADDRES

;
; Set data pointer 0 as the current data pointer.
  mov dps,#0
camera_loop:
;
; Read from data pointer zero.
;
  movx a,@dptr
;
; Switch to the next data pointer. Note that doing
; an inc on this register only affects the data
; pointer-selection bit. This allows one cycle
toggling
; from one data pointer to another.
;
  inc dps
;
; Store the data and increment the address.
;
  movx @dptr,a
  inc dptr
;
; Switch back to the first data pointer.
;
  inc dps
```

```
;
; Do the loop again...
;
```

As can be seen, the loop runs faster because most of the address handling is done outside of the loop. For memory copies, the DS80C400 also has optimizations for even faster copies. First, all data pointer increments are performed in a single cycle. An autoincrement mode, when enabled, can automatically increment the address in the data pointer after a read or write is performed. Also, an autoselection feature can be enabled to toggle between two data pointers with every read or write operation. This makes a memory copy incredibly easy.

```
;
; Set the base address of data pointer zero.
;
  mov dps,#0
  mov dptr,#ADDRESS1
;
; Set the base address of data pointer one.

  mov dps, #1
  mov dptr,#ADDRESS2

;
; Enable autoselection and autoincrement.
;
  mov dps, #(DPS_AID | DPS_TSL)
memory_loop:
;
; Read from data pointer 0, increment the
; data pointer, and toggle the selection bit
; in one instruction.
;
  movx a,@dptr
;
; Write to data pointer 1, increment the data pointer,
; and toggle the selection bit in one instruction.
;
  movx @dptr,a

; Loop...
```

Back to the example, the HSYNC line is used to generate the interrupt. When the camera asserts the HSYNC line on each scan line, the ISR fires and begins to capture data. In order to improve performance, all other interrupts have been moved to low priority, including the scheduler. This has a noticeable effect on image quality.

The camera has a programmable window which allows the user to configure how much of the 384 x 288 image to use. Finding an appropriate image size is difficult; a large image has increased quality, but takes more time to transmit and reduces frame rates. For this application, Joe sets the camera to a 240 x 180 resolution. This is a standard resolution for Internet video, and has an additional hardware advantage. Looking at Figure 4, when the camera is transmitting the image, it actually enumerates every pixel the image array, but only asserts the HSYNC line for pixels inside the specified window. This means that an image capture consumes about 3/5 of the CPU during the 100ms period.

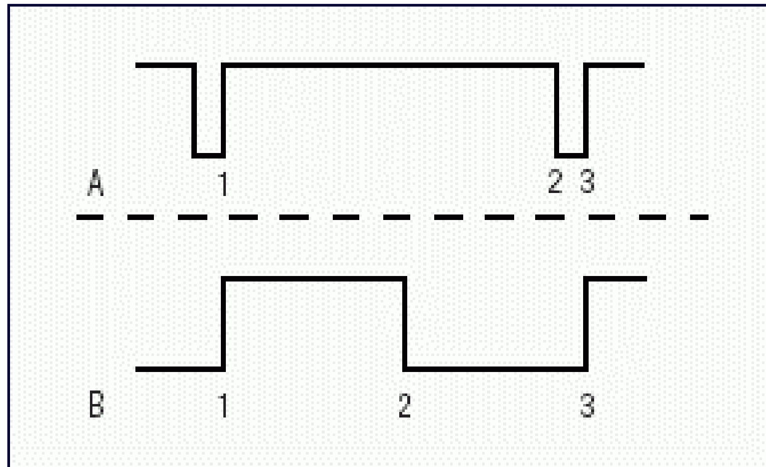


Figure 4. The HSYNC line is asserted for each individual scan line. In A, the pixel window has been set to the full 384 pixels, while in B it has been set to 192 pixels, beginning at the left edge with both set to the same frame rate. Marker 1 is where the scan line is asserted by the camera, marker 2 is where it finishes with the scan line, and marker 3 is the time when the camera starts the next scan line. Though B takes half as long as A, the period per scan line is the same.

On initialization, the camera software allocates a 95kB contiguous block of memory from the memory manager, which is used for double buffering. In Java, one thread handles all of the image capture, while another thread handles transmitting the image over the network. Java simplifies this by providing all of the threading and locking mechanisms necessary.

Above the ISR in the software design are the native methods that provide the camera driver for the Java virtual machine. The TINI platform uses the TINI native interface (TNI) to allow developers access to low-level code from Java. Native methods can call into the TINIOS native API, allowing them access to the memory manager, scheduler, and other operating system internals. They can have parameters passed to them, and even throw exceptions like other Java methods. These are linked to the Java executable through a TINI dynamically linked library (TLIB), which can be built using the TINI developers kit.

The native methods allow Java to send commands to the camera. The camera driver class has the following definition:

```
public static final int IMAGE_BUFFER_0 = 0;
```

```
public static final int IMAGE_BUFFER_1 = 1;
/**
takePhoto takes a photograph and stores it in
the memory.
@param buffer Species what image buffer to use.
Use IMAGE_BUFFER_0 or IMAGE_BUFFER_1
*/

static native void takePhoto(int buffer);

/**
getScanlines pulls a fixed number of
scanlines out of the memory buffer. This
allows the Java application
the ability to work with fixed pieces of the
image.

  @param start first scanline to copy from.
  @param end last scanline to copy from
  @param offset offset into the data array to
copy to
  @param data array to copy into
  @param buffer selects the image buffer to read
from. Useful IMAGE_BUFFER_0 or IMAGE_BUFFER_1
*/
  public static native int getScanlines(int
start, int end , int offset, byte []data, int
buffer);
```

The main method, takePhoto, captures one image into the image buffer. First, it enables the interrupt, and sends the camera a command to take a single image. There is a small impasse here. It is preferred that the Java thread is put to sleep at this point; however, it cannot be woken up from the ISR because the TINIOS functions are not reentrant. For this purpose, TINIOS allows developers to register poll routines that are called every 4ms by the system. A poll routine does a quick check to see if the photo is finished, and wakes up the thread if it is. This poll routine is registered right before putting the thread to sleep. Upon waking up, the thread returns to Java. As mentioned previously, the camera is double buffered, so the image buffer to overwrite must be specified. In order to let Java access the underlying image buffer by allowing blocks of scan lines to be copied into Java byte arrays, getScanlines is also implemented.

A storage problem is also present. The TINI runtime takes 7 banks of a 512kB flash, leaving one bank for a user application. As mentioned before, there is not a nonvolatile file system on the high-speed design, so the file system must be created from scratch. It is desirable to have everything necessary to run from power-up in the flash bank, including the Java applet that the HTTP server will serve to the web browser. In order to include the applet in the executable, the

applet binary is converted into a format compatible with the assembler and included in the data segment of the library. Then there is a native method to copy it from the library into a Java byte array. On startup, the Java code reads the size of the applet, creates an array, copies the applet into the array, and writes the contents to the file system. It is a little clunky, but it means one can start from a clean boot and have everything necessary to start. The following methods perform this task:

```
  /**
Extracts the sample jar file from the native
library. The demo application had a jar file
embedded inside the native library.
This allowed the jar file, the application, and
the native library all to be embedded in flash
format.

  @param dummy Array to copy jar image into.
Must be of greater size than that specified in
getJarFileSize()
  */
  static native void getJarFile(byte []dummy);
  /**
  Gets the size of the embedded jar file
  */
  static native int getJarFileSize();
```

The Java on top of the camera driver is much simpler. A number of threads are running, most of them independent from one another. First, there is the HTTP server. The code for this comes from the HTTP server example included with the TINI SDK. It is very lightweight, and is not designed for servelets, cgi-bin processing, or other features. Files are read from the TINI file system, which is a hierarchical file system implemented in TINI's nonvolatile memory. On startup, the camera applet is read from flash memory and written to the file system, and the index.html page is generated.

Next, there is the camera image server. The camera server has two main threads. The first thread opens a TCP server socket on port 42877 and awaits connections from an applet. How is server socket opened on an embedded system? Actually, it is much like how it would be done on a PC.

```
  sockpuppet = new ServerSocket(42877);
```

Something to note is that a server socket binds to a port on both the IPv4 and IPv6 interfaces. This means that no changes are necessary to make the camera IPv6 compliant. IPv6's much larger address space will help network appliances in the future, since they currently need to fight for addresses with PCs, cell phones, and other devices on a network.

When a process connects, it sends either an •eA' for connect or •eD' for disconnect. On a connect command, the IP address is added to a shared vector of connected addresses, while the disconnect command removes it from the vector.

```
sock = sockpuppet.accept();
ch = (char)sock.getInputStream().read();
switch (ch)
{
case 'A':
cwt.addAddress(sock.getInetAddress());
break;
case 'D':
cwt.removeAddress(sock.getInetAddress());
break;
}
sock.close();
```

The other thread is the image transmitter. If there is an address in the camera vector, it transmits the captured image to it in UDP packets. The camera capture and transmission have been optimized to run in parallel. The capture has been double buffered to allow transmission from one buffer while capture occurs in another. This is possible because the camera only uses about 50% of the CPU while transmitting. The asynchronous locking is all done in Java.

```
//
// Notify the camera thread we are ready for
// the next frame.
//
synchronized(stopper)
{
    stopper.notify();
}
```

No sort of image compression hardware is utilized, so the image is transmitted raw. The packet layout is extremely simple. Each packet has a 2-byte header, followed by the data for five scan lines. The first byte is the frame number, which is a rolling counter that increments for every frame transmitted. The second byte is the vertical offset divided by five.

Finally, a configuration tool is run on the serial port. This application, the CAmera SHell or CASH, is a menudriven utility that allows the IP address to be set, and the connected users to be seen. A lot of this functionality is taken from the Slush shell that comes with the TINI SDK. To configure the camera, the user powers it up and communicates with it over the serial port. The CASH presents a simple user interface for setting the IP address. Once configured, it can simply sit on the network, waiting for a user. When someone connects with a web browser, the camera

serves the applet, which makes the connection to the camera server and displays the images. While TINIOS supports up to 24 simultaneous open sockets at a time, for speed reasons we limit the camera to four users at a time. Using multicast would alleviate this problem, but Java applets do not support it.

The network camera captures and transmits 4.5 frames per second, with an average network transmit rate of 200kB/s. Something to remember is that very little support hardware has been connected to the camera. There is no image capture or encoding hardware in the system, which means the DS80C400 is juggling image capture, image transmitting, network traffic, web serving, and user interaction over the serial port at the same time.

Conclusion

Let's return to Joe. Seeing that TINI is the ideal solution for his product, he continues his development of the network door lock. With his low-cost, high-power solution, Joe becomes the reigning king of the network door lock market. He easily beats Troy's solution that, despite bearing the logo of a international software monopoly, just costs too much. Alex's solution, having to overcome both camera and network issues, has difficulties making it to market. Amiga leaves Troy and returns to Joe, who promises her that he will never again let the computer come between the two of them. With the success of his business, Joe and Amiga retire to a small cabin in Minnesota that, of course, has a network door lock on every door.

Embedded devices need to be designed to solve specific problems. It is a challenge to find the right balance between power and cost. This becomes even more complicated when adding network capability to a device. One route is to try to extend an 8-bit microcontroller into the networking world. While it is feasible, it will inevitably be slow. Another approach is to use an embedded Linux, PC- 104, or Pocket PC device. While this will be fast and responsive, it also adds a lot of unnecessary bloat. One could build a smaller 32-bit solution, but that requires licensing an operating system and TCP/IP stack.

The DS80C400 with TINI is a good in-between solution. It has a robust, well-rounded TCP/IP stack that has been hardened over time. It has an operating system with support for multiple processes, Java, threading, and synchronization. The processors can handle heavyweight tasks like talking to a digital camera without the bloat of a heavyweight operating system. If it works for the average Joe, it might work for you.

TINI and 1-Wire are registered trademarks of Dallas Semiconductor.
SPI is a trademark of Motorola, Inc.
Java is a trademark of Sun Microsystems.
Purchase of I^{2}C components of Maxim Integrated Products, Inc., or one of its sublicensed Associated Companies, conveys a license under the Philips I^{2}C Patent Rights to use these components in an I^{2}C system, provided that the system conforms to the I^{2}C Standard Specification as defined by Philips.

More Information

DS1672: QuickView -- Full (PDF) Data Sheet -- Free Samples

DS2502: QuickView -- Full (PDF) Data Sheet -- Free Samples

DS80C390: QuickView -- Full (PDF) Data Sheet -- Free Samples

DS80C400: QuickView -- Full (PDF) Data Sheet -- Free Samples