Rev 0; 7/04

DALLAS SEMICONDUCTOR MAXIM

Getting Started with TINI

TABLE OF CONTENTS

Hardware Requirements 4

The TINI Board Model 400: DSTINIm400 4
The Socket for the DSTINIM400: DSTINIs400 4
The TINI Board Model 390: DSTINI1-512 or DSTINI1-1MG 7
The E10 Socket for the DSTINI1: DSTINIS-005 7
The E20 Socket DSTINI1: DSTINIS-006 8
Summary of Modules and Sockets Available from Dallas Semiconductor 10
DS80C390 vs. DS80C400 10

Software Requirements 11

Development Platform Requirements 11
Software Required to Load Applications 11
Software Required to Develop Applications in Java 11
Software Required to Develop Applications in C 12
Software Required to Develop Applications in 8051 Assembly Language 12

Software Components 12

Java Development Environment 12
The Java Communications API 12
The TINI SDK 13

Loading the TINI Java Runtime Environment 14

Loading Firmware 1.1x on a DS80C400-Based TINI 14
Loading Firmware 1.0x on a DS80C390-Based TINI 14
Loading Firmware 1.1x on a DS80C390-Based TINI 14
Connecting TINI to Your Host Computer 15
Using MTK to Load Files onto TINI 15
Using JavaKit to Load Files onto TINI 17
Starting the TINI Java Runtime Environment 19

Slush: A Quick Primer 21

Slush Defined 21
Starting a New Session 21
Exploring the File System 22
Getting Help 23

Configuring the Network 24

Some Simple Examples 27

HelloWorld 27
Blinky, Your First TINI I/O 30
HelloWeb—A Trivial Web Server 33

For pricing, delivery, and ordering information, please contact Maxim/Dallas Direct! at 1-888-629-4642, or visit Maxim's website at www.maxim-ic.com.

Debugging Tips 38

Using the DS80C400 Silicon Software 39

Developing Applications in 8051 Assembly 39
Developing Applications in C 39
The C Library Project Web Page 40
Using the Keil Tools for the DS80C400 40

APPENDIX 41

Java Tools 41

Which Version of the Java Software Development Kit Do I Need? 41
What is J2SE? What is Java 2? 41
Downloading the Java Software Development Kit 41
Installing the Java Software Development Kit 41
Using the Compiler and the Virtual Machine 41
What is a Classpath? 42

JavaKit 42

Installing the Java Communications API 43
Running JavaKit 44
The HEX File Format 45
The TBIN File Format 46
Documentation on JavaKit's Macro Feature 46
JavaKit's Command Line Arguments 47
JavaKit's Menu Options 48

TINIConvertor 50

Why Do We Need TINIConvertor? 50
Different Versions of TINIConvertor 51
How to Use TINIConvertor 51
Arguments to TINIConvertor 55
Common Problems Reported when Using TINIConvertor 57

BuildDependency 58

How to Use BuildDependency 58
The Dependency File 59
BuildDependency and TINI Firmware 1.1x 59
BuildDependency's Command Line Arguments 60
Common Problems Reported when Using BuildDependency 60

Macro **60**

Using Macro 62
Macro's Command Line Parameters 62

a390 **62**

Assembly Applications on the DS80C400 63
Instruction Set for the DS80C390 and DS80C400 63
Assembling Files with a390 64
Command Line Arguments to a390 64

MTK **65**

Benefits Over JavaKit 65
Installation of MTK 65
Using MTK 65

Common Problems During TINI Development **66**

Getting Started with TINI

Introduction

When *The TINI Specification and Developer's Guide* (Addison-Wesley, 2001) was first published, the world of TINI® was much smaller than it is today. Your only option was the DSTINI1 module, based on the DS80C390 processor, which could run Java™ programs using the TINI 1.02 firmware.

Today there are many more options. In addition to the 1.0x Java firmware, Dallas Semiconductor has also introduced the 1.1x firmware, which now supports IPv6, dynamic class loading, reflection, and serialization. But that's not all—there is also a new processor in the TINI kingdom.

The DS80C400 processor has a 64kB ROM that contains a boot loader, network stack, memory manager, and process scheduler implemented in highly optimized 8051 assembly language. Using the latest TINI reference design, the DSTINIm400 reference module, developers can choose among using Java, C, or even coding in 8051. It is recommended that new designs choose the DS80C400 processor for reasons discussed in the *DS80C390 vs. DS80C400* section.

More choices often lead to increased confusion. This document covers both the hardware and software environment needed to develop and execute TINI applications written in Java, C, or 8051 assembly language. As part of the discussion of the required hardware, we will also present a comparison of the DS80C390 and the DS80C400 processors. As part of the discussion on software requirements, we will compare the 1.0x and 1.1x TINI Java firmware, as well as discuss coding applications in Java, C, or assembly language.

Hardware Requirements

This section describes the core hardware configurations commonly used to develop and test TINI applications. Other configurations are possible and can be assembled in piecemeal fashion by readers already in possession of, or familiar with, TINI. Dallas Semiconductor currently produces two basic socket/board combinations, one uses the DS80C390 and one uses the DS80C400. If you are just getting started in the TINI universe, it is recommended that you start with the DSTINIm400 and DSTINIs400.

The TINI Board Model 400: DSTINIm400

The TINI reference board based on the DS80C400 processor is known as the DSTINIm400 reference module. Like the DSTINI1, it is also a complete TINI hardware reference design. The DSTINIm400 is only available in one configuration, which has 1MB of nonvolatile, static RAM and 1MB of flash. It is available as a 144-pin SODIMM module and is shown in Figures 1 and 2.

The Socket for the DSTINIm400: DSTINIs400

For application development and prototyping, a DSTINIm400 board, as shown in Figure 1, is not useful without the ability to connect necessities such as serial, Ethernet, and power. The main function of the DSTINIs400 socket board is to provide physical connectors to interface the DSTINIm400 with other equipment such as an Ethernet network, a serial device, or a 1-Wire® network.

TINI and 1-Wire are registered trademarks of Dallas Semiconductor Corp.

Java is a trademark of Sun Microsystems.

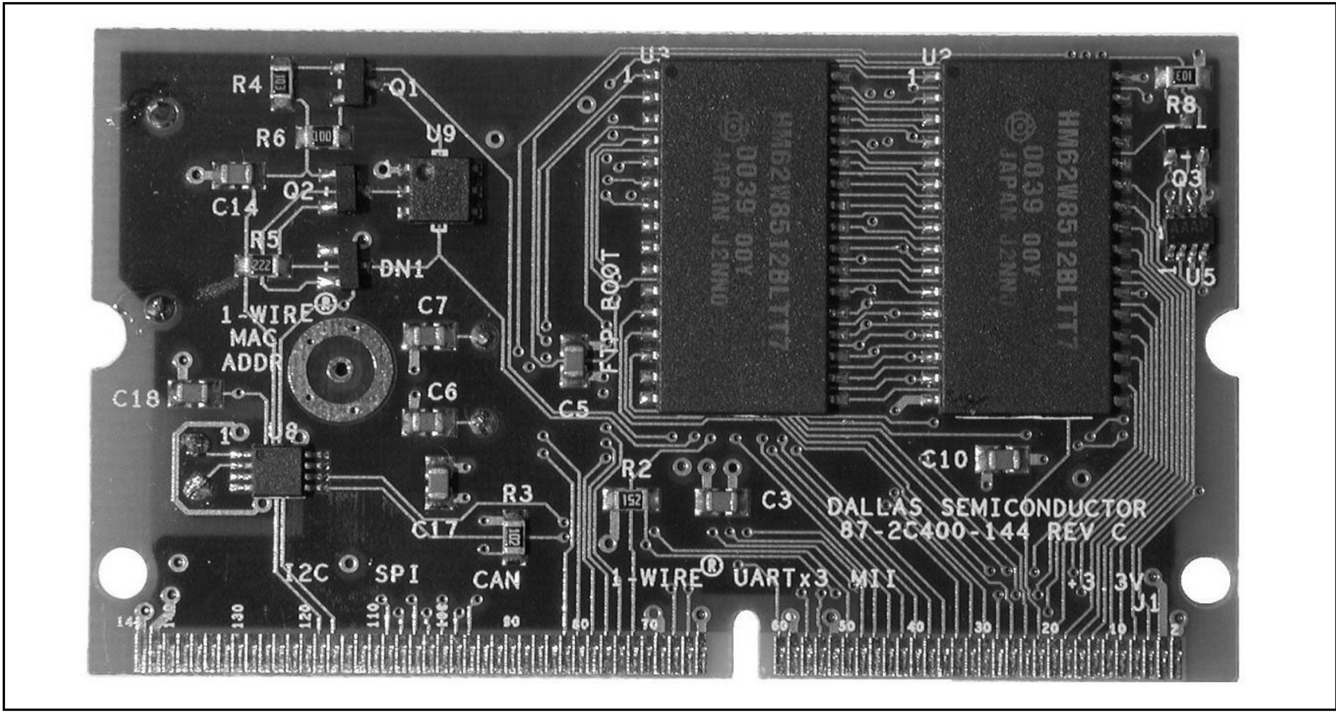

Figure 1. DSTINIm400 Bottom View

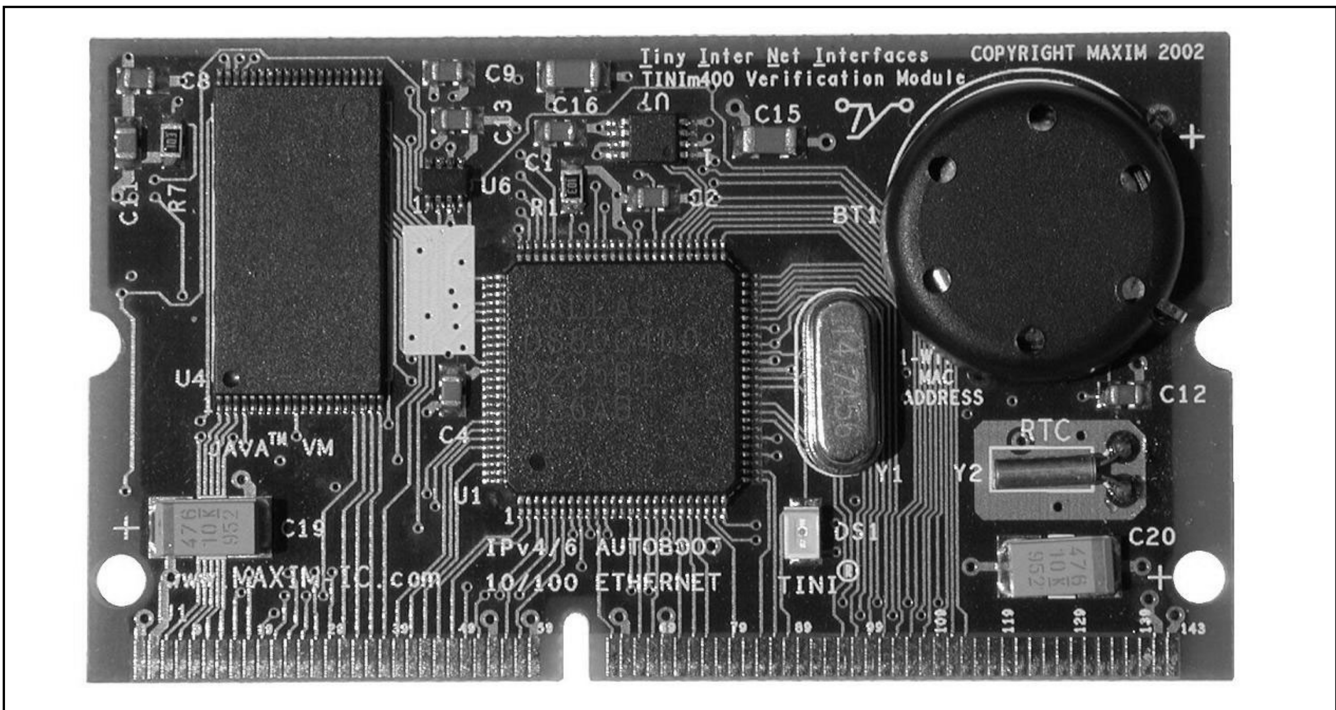

Figure 2. DSTINIm400 Top View

Getting Started with TINI

The DSTINIs400 socket board is aimed at aiding the application development process. It provides the following physical connectors:

- **144-Pin SODIMM Connector**. The SODIMM connector accepts the DSTINIm400 board shown in Figures 1 and 2.
- **9-Pin Female DB9 Connector.** This connector provides a limited DCE- (Data Communications Equipment) type serial port that provides connection to a standard PC DTE (Data Terminal Equipment) serial port using a straight-through serial cable. This port is typically only used for loading the runtime environment and bootstrap application (see the *Software Required to Develop Applications in Java* section). Hardware handshake lines, such as RTS (Request To Send) and CTS (Clear To Send), are not supported by the DCE port.
- **9-Pin Male DB9 Connector.** This connector provides a DTE serial port for straight-through connection to DCE devices such as analog modems. Most TINI applications that control serial devices use the DTE port. In this case TINI is the DTE device, replacing the PC or workstation. The DTE serial port supports all hardware handshake lines except DSR (Data Set Ready) and RI (Ring Indicate).
- **RJ45.** The RJ45 connector accepts a standard 10BASE-T Ethernet cable providing connectivity to an Ethernet network. Use a straight-through cable for connecting TINI to the network or a crossover cable for connecting TINI directly to a PC or workstation.
- **RJ11.** The RJ11 connector provides access to the 1-Wire network using standard telephone cable.
- **Power Jack.** The DSTINIs400 accepts a regulated +5V DC power supply.

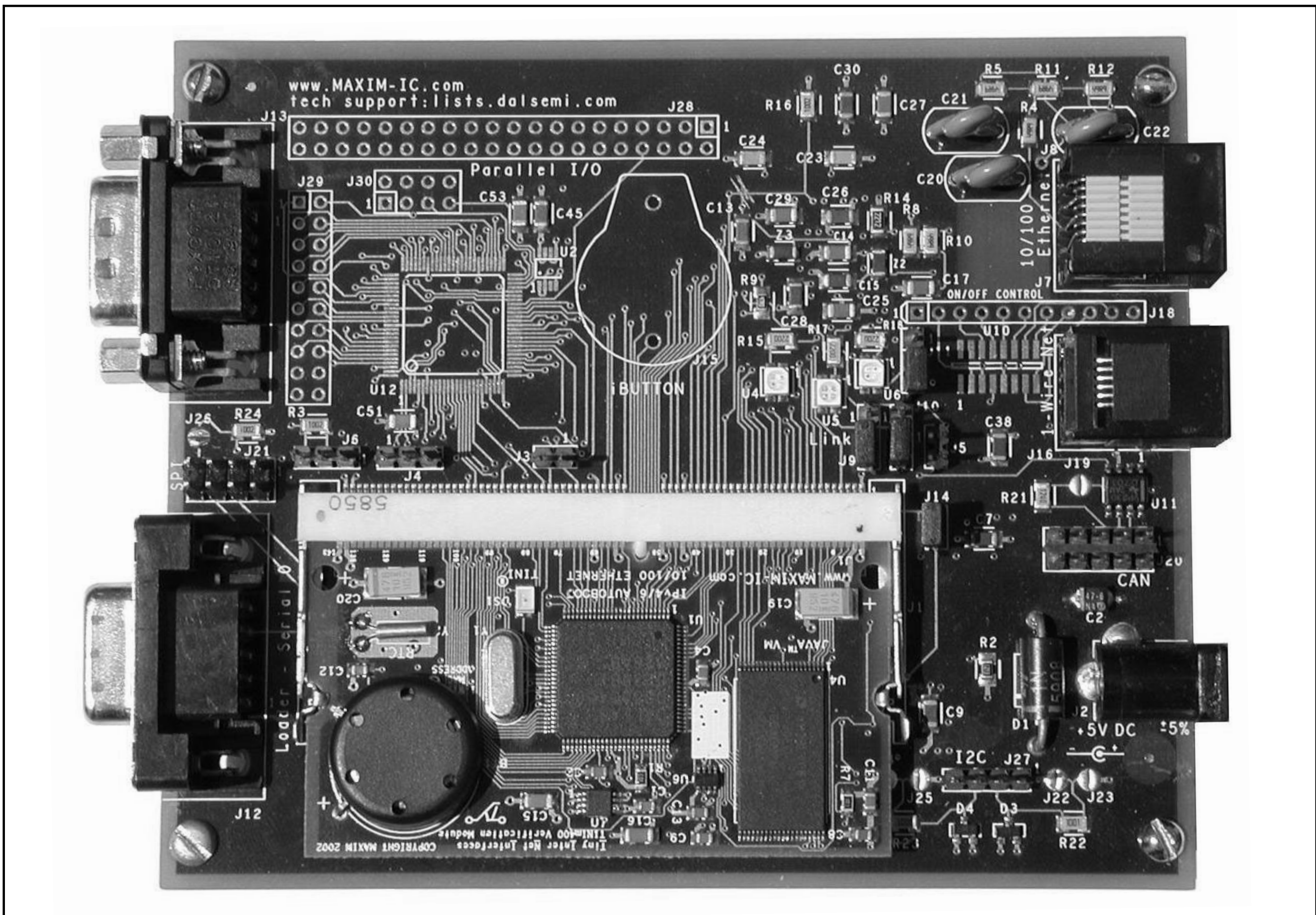

Figure 3. DSTINIs400 Socket (Shown with DSTINIm400 Inserted)

DALLAS SEMICONDUCTOR MAXIM

The TINI Board Model 390: DSTINI1-512 or DSTINI1-1MG

The TINI board model 390, commonly known as the DSTINI1 reference board, is a complete TINI hardware reference design. The DSTINI1 is currently available with either 512kB or 1MB of NV SRAM, in addition to 512kB of flash. It is available as a 72-pin SIMM module and is shown in Figures 4 and 5.

The E10 Socket for the DSTINI1: DSTINIS-005

The E10 socket accepts the DSTINI reference board and allows for complete evaluation of a TINI system based on the DS80C380 processor. The E10 has the same basic features of the DSTINIs400 socket board, the biggest exception being the 72-pin SIMM connector for accepting the DS80C390-based module. Like the DSTINIs400, the E10 requires a 5V supply.

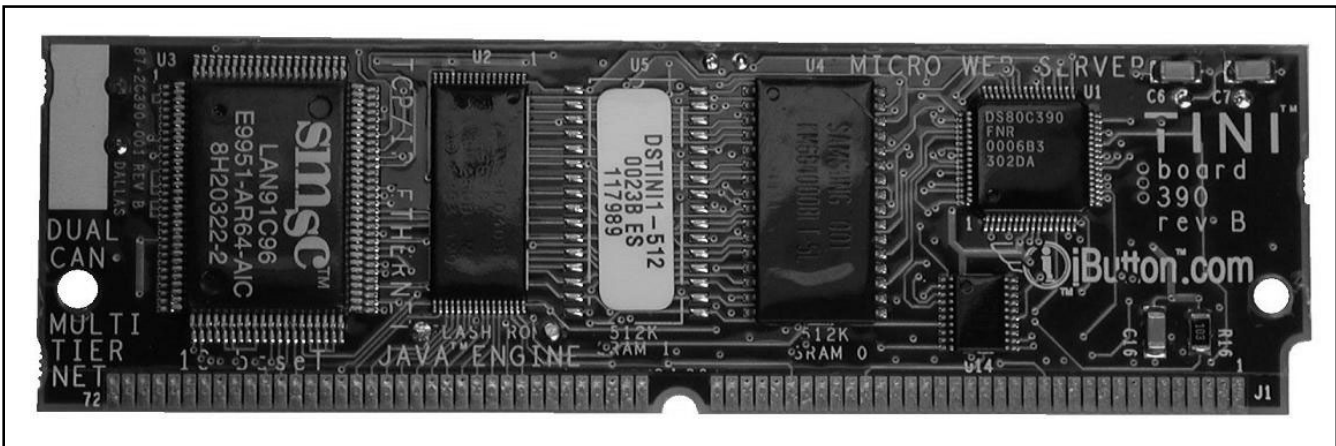

Figure 4. DSTINI1-512 Bottom View

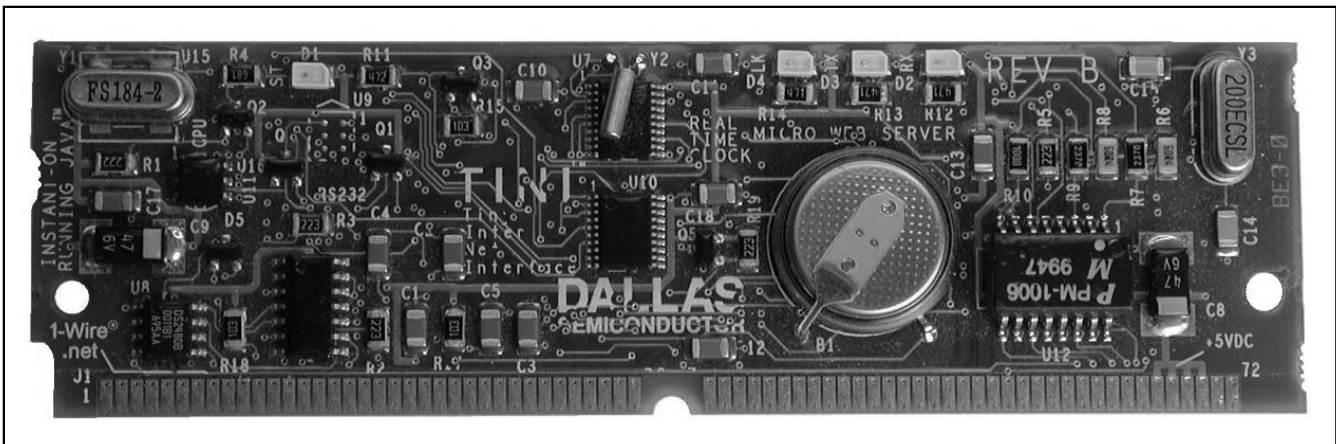

Figure 5. DSTINI1-512 Top View

The “E” in E10 stands for Eurocard and indicates that the size of the socket board is identical to one of the standard Eurocard sizes (160mm x 120mm), allowing it to be placed inside a standard enclosure. Figure 6 shows the E10 socket with a DSTINI1-512 board inserted.

The E20 Socket DSTINI1: DSTINIS-006

The E20 socket, also known as the DSTINIS-006, is virtually identical to the E10 socket except that it has an on-board power regulator, accepting 9V to 18V AC/DC, and providing the TINI board with 5V. Figure 7 shows an E20 socket, highlighting the power regulator.

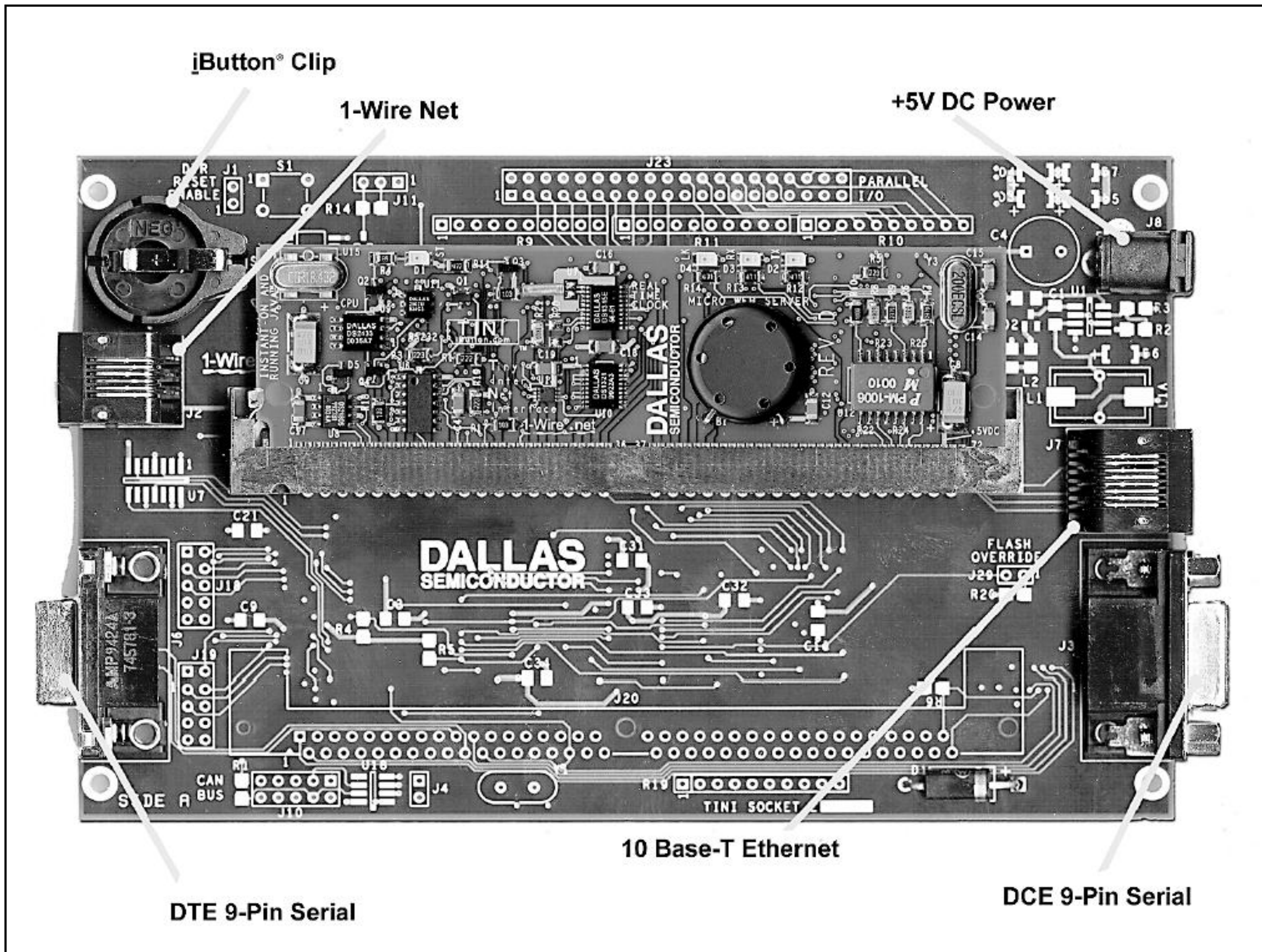

Figure 6. E10 Socket with DSTINI1-512 Inserted and Major Connectors Labeled

DALLAS SEMICONDUCTOR MAXIM

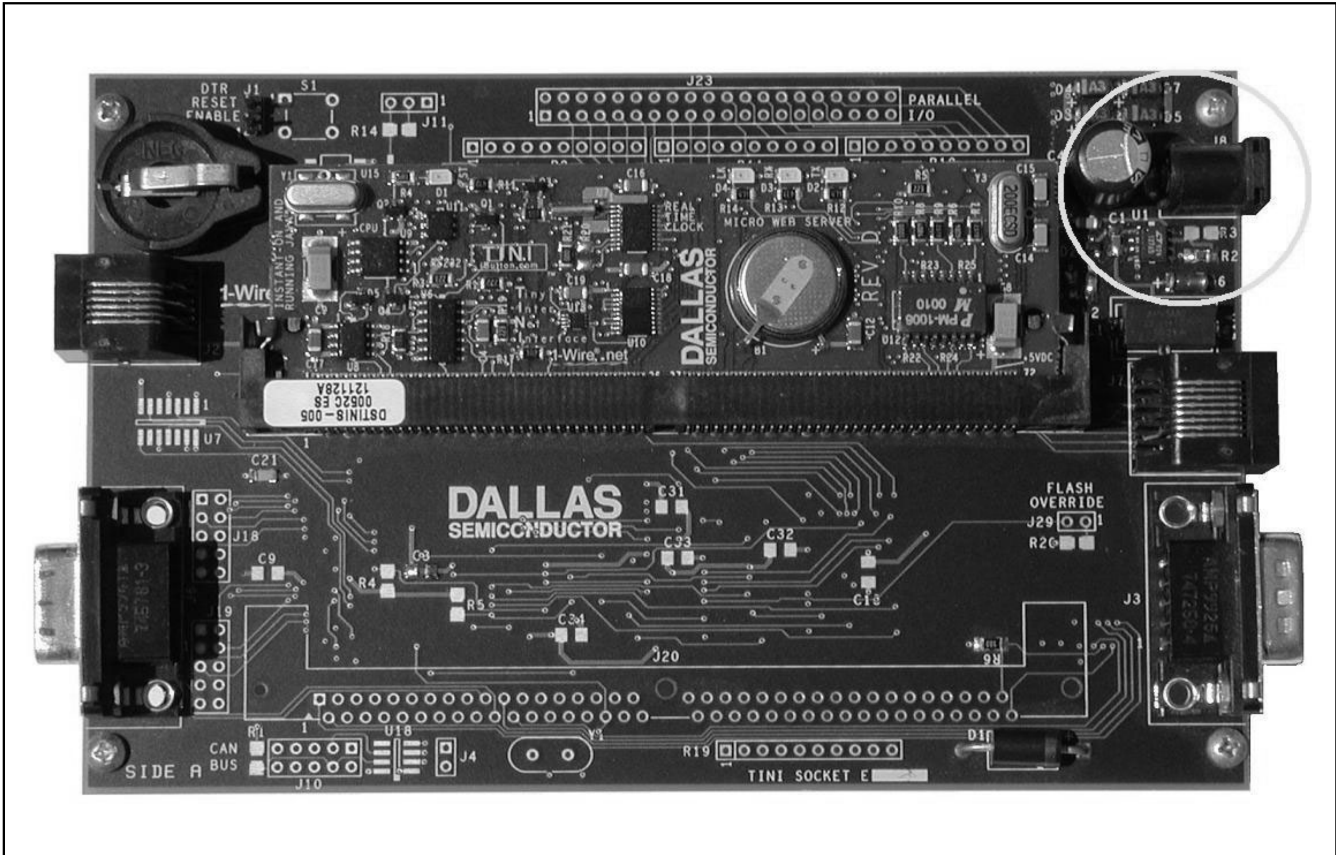

Figure 7. E20 Socket with DSTINI1-512 Inserted (Note: Power Regulator is Circled in the Upper Right.)

Summary of Modules and Sockets Available from Dallas Semiconductor

Table 1 and Table 2 provide a summary of the sockets available from Dallas Semiconductor.

DS80C390 vs. DS80C400

At this point, you may be thinking that there is not much difference between the DS80C390-based TINI boards and the DS80C400-based TINI boards. They have roughly the same features, the sockets for the two are pretty similar, so why has Dallas Semiconductor introduced all these choices except to cause confusion?

A DS80C390-based solution is good for many applications, however, this microcontroller was not designed with Ethernet networking as a goal. With the DS80C400, networking was one of the primary goals from the start. The DS80C400 integrates the Ethernet MAC and as a result increases overall network throughput. In addition, several other enhancements were made to the DS80C400 to increase software performance. These include:

- Increasing maximum operating frequency from 40MHz to 75MHz
- Adding two additional data pointers for a total of four
- Auto-increment and decrement on all data pointers
- Optimized the `inc dptr` instruction for 1 machine cycle
- Hardware TCP/IP checksum generation
- Hardware 1-Wire master
- Additional serial port

In addition to these enhancements, the DS80C400 contains a 64kB ROM rich with features, including:

- A Berkeley-style network stack, supporting IPv4 and IPv6
- A preemptive, priority-based task scheduler
- Memory manager
- DHCP and TFTP algorithms
- 1-Wire operations on the internal 1-Wire

The functions in the ROM form a base for applications written in C or assembly language to harness the power of TINI that was previously reserved for Java.

In short, the DS80C400 contains numerous hardware enhancements that make it much more attractive for many applications.

Table 1. Sockets Available from Dallas Semiconductor

SOCKET	FEATURES	SIZE (mm)	WORKS WITH MODULES	POWER SUPPLY
E10	1-Wire, serial, 10/100 connections	160 x 120	DSTINI1-512, DSTINI1-1MG	5V DC ±10% >150mA
E20	1-Wire, serial, 10/100 connections	160 x 120	DSTINI1-512, DSTINI1-1MG	9V–18V AC/DC >200mA
DSTINIs400	1-Wire, CAN, serial, 10/100 connections	120 x 100	DSTINIm400	5V DC ±10% >150mA

Table 2. Modules Available from Dallas Semiconductor

MODULE	PROCESSOR	FEATURES	SIZE	WORKS WITH MODULES
DSTINI1-512	DS80C390	512k RAM, 512k Flash	72-Pin SIMM	E10, E20
DSTINI1-1MG	DS80C390	1M RAM, 512k Flash	72-Pin SIMM	E10, E20
DSTINIm400	DS80C400	1M RAM, 1M Flash	144-Pin SODIMM	DSTINIs400

Software Requirements

Development Platform Requirements

We use the term “development platform” to refer to the computer used for creating, building, and loading TINI applications. This is the machine that runs the JDK or equivalent Java development and runtime environment and is connected to TINI using Ethernet and/or a serial cable. Typically we will just refer to the host development machine as “the host.”

Since all the required tools have been written in Java, TINI applications can be developed on any of the following operating systems.

- Any Win32 OS (Windows 95, 98, NT, 2000)
- Linux
- Solaris
- MAC OSX

To load the TINI runtime environment (see the section *Loading the TINI Java Runtime Environment*) the host must also have an RS-232 serial port. Nearly every PC and workstation meets this requirement. However, modern Macintosh machines and many laptops no longer have serial ports. In this case, USB-to-serial converters, such as the Keyspan USA-19HS, provide a serial port capable of communicating with a TINI.

Besides one of the operating systems mentioned above and a serial port, the host machine must also have some software installed correctly. The software required depends on the language used for development and the tools used to load programs on the TINI.

Software Required to Load Applications

Dallas Semiconductor provides two tools for loading programs onto a TINI board. The preferred tool for loading TINI is the Microcontroller Tool Kit, commonly called MTK. Currently, the MTK is only available for Windows platforms. However, releases for Linux and MAC OSX are planned for the future. Aside from the MTK executable file, no additional software is needed to run.

The second loader tool is called JavaKit, a Java-based GUI program with a boot loader interface. JavaKit requires the following software to run:

- Java Development Environment
- Java Communications API
- TINI Software Development Kit

Since JavaKit is a Java application, it can be run on any of the listed platforms.

It is recommended that developers new to TINI use the MTK program for interfacing with TINI. The installation and configuration of JavaKit is very error-prone and can be frustrating. Note that either loader tool can be used independent of the tools used to develop your TINI application. More details on using JavaKit and MTK can be found later in this document and in the *Appendix*.

Software Required to Develop Applications in Java

To develop Java applications for TINI, two pieces are required:

- Java Development Environment
- TINI Software Development Kit

There are many Java Integrated Development Environments (IDEs) available for free and for sale, most of which can probably be configured for use with TINI. However, the command line Java Development Kit, available for free from Sun Microsystems, is probably the best tool to start with and is more than capable of handling TINI development. More information on these tools can be found in the following sections and in the *Appendix*.

Software Required to Develop Applications in C

Currently, there are three implementations of C compilers that can build applications for the DS80C400's 24-bit address space. Keil's PK51 Professional Developer's Kit contains the first C compiler to support the DS80C400. The Small Device C Compiler (SDCC), an open source project, also added support for the 24-bit address space. More recently, IAR's Embedded Workbench has added support for the DS80C400.

Cost is one potential issue with development in C. Both the Keil PK51 Development Kit and the IAR Embedded Workbench are for sale, which may make them prohibitive to hobbyists and students. The SDCC compiler is free.

While all these compilers can generate valid code for the DS80C400, most developers will be interested in using the Dallas Semiconductor-provided C libraries, which offer access to the ROM's exported functionality, and also provides services such as DNS, SMTP, flash programming, and much more. Currently, these libraries are only available for the Keil tools, since Keil support has existed for several years. Dallas Semiconductor is currently porting these libraries for use with SDCC and IAR.

The C library project home page[1] is the central resource for developers using C in their TINI applications. As support for these additional compilers is added, the C library website will be updated.

Software Required to Develop Applications in 8051 Assembly Language

The previously mentioned C compilers also provide an assembler that supports the 24-bit addressing mode of the DS80C400, and are adequate for application development. In addition, the TINI Software Development Kit also provides an 8051 assembler that is free and extensively field-tested.

Software Components

Java Development Environment

All the examples from *The TINI Specification and Developer's Guide* were compiled using javac from Sun's JDK, standard edition 1.2.2. Since then, Sun has moved to the current revision, 1.4.1. Any version in between has proven to be adequate, although when using the 1.4 JDK, files for TINI must be compiled with the '-target 1.1' switch due to a change in the Java class file format. The most popular choice for TINI development appears to be 1.3.1.

Sun's JDK is free and available for most platforms that support Java development of any sort. However, you can certainly use your favorite Java IDE such as JBuilder or Visual Cafe to edit and compile your TINI applications. In fact there are Open-Source extensions to JBuilder that allow for a purely graphical development environment for TINI.

For more detailed information on the Java Development Environment, see the *Appendix.*

The Java Communications API

The Java Communications API[2] (comm API, or javax.comm) is also available from Sun Microsystems and provides the infrastructure required to communicate with RS-232 serial ports in a platform-independent fashion. This API is used by JavaKit to manage loading the TINI runtime environment. At the time of this writing, comm API drivers supplied by Sun supported only the Win32 and Solaris platforms. However, the Open-Source project RXTX[3] provides driver support for Linux and MAC OSX.

The installation process for the comm API for Win32 and Solaris is described in a README in the comm API's distribution. Note that problems installing the communications API are frequent. The *Appendix* contains more detailed information on installing and running with the comm API, and tries to address the most common problems.

There is extra work, such as compiling the driver source, involved for those installing the comm API on Linux. Detailed instructions are provided at the RXTX website. For MAC OSX users, a prebuilt binary of the library and archive is available at ftp://ftp.dalsemi.com/pub/tini/unsupported/JavaKit-2.2.8-X.dmg.gz. Please note that the tools for MAC OSX are not officially supported.

[1] *C Library Project is located at www.maxim-ic.com/ds80c400/libraries.*

[2] *Available for Windows and Solaris at http://java.sun.com/products/javacomm/downloads/index.html*

[3] *RXTX home page: www.rxtx.org*

The TINI SDK

The latest release of the TINI software distribution can be freely downloaded from Dallas Semiconductor's website (ftp.dalsemi.com/pub/tini/index.html). At the time of this writing, there are two branches of TINI firmware. Firmware versions 1.0x contain the TINI API as it was originally developed. It implements a good portion of the core Java 1.1.8 API, plus several Dallas Semiconductor provided classes to assist with I/O access and network configuration. The most current firmware version in this family is 1.02g.

Not long after the original version of *The TINI Specification and Developer's Guide* was released, the first TINI 1.1x firmware was released. It included several enhancements to the 1.0x firmware including:

- IPv6 network stack
- Dynamic class loading and correct static initialization behavior
- Reflection
- Serialization

The most current version in this firmware family is 1.13, which replaced 1.12 in June 2004.

So why have two versions? It sounds like the 1.12 firmware is simply superior to the 1.02g firmware, so why even maintain the older 1.0x stuff? There are a few reasons:

- All those enhanced Java features (dynamic class loading and serialization) do not come without a cost. Reflection information is fat, and was previously stripped out of TINI files in the 1.0x firmware. Also, many applications do not need reflection for their embedded applications.
- The 1.0x firmware is a little more field tested than the newer 1.1x.
- In order to make room for the 1.1x enhancements, a few things had to be removed into a supplemental modules file that could be built into applications that needed it. This included the CAN, PPP, and URL functionality. With space at a premium, some applications may not be willing to give up this extra memory cost.

The TINI SDK distributions are very similar for 1.0x and 1.1x firmware. It is distributed as a single compressed tar file (.tgz), which can be opened by WinZip, gunzip, or almost any other common compression utility. After downloading the distribution and extracting its contents, the SDK is installed. There is no setup executable to run, no DLLs to install and no modifications to be made the registry.

These are some of the important files included in the SDK. It is important to understand the contents of these files because we will use them to build the examples later in this document.

- **README.txt**. The README.TXT file is located in the root of the SDK hierarchy. Start by completely reading this document. It contains detailed instructions on how to install the TINI runtime environment, boot the TINI system, and initialize its network settings. It also contains references to other documents in the SDK that further describe the process of creating a full development environment.
- **tini.jar**. This jar (Java ARchive) file is located in the bin directory and includes three important utility programs: JavaKit, TINIConvertor, and BuildDependency. The JavaKit utility manages the firmware-loading process and performs other system maintenance tasks. It can also be used to run Slush user sessions over a serial connection. The TINIConvertor utility takes the class files in your application as input and generates a binary image suitable for execution on TINI. The BuildDependency utility is a wrapper application for TINIConvertor that helps build more applications where the inclusion of one class might mean the inclusion of several support (dependency) classes.
- **tiniclasses.jar**. The tiniclasses.jar file is located in the bin directory and contains all of the class files in TINI's API. In this sense it is similar to the rt.jar file distributed with Sun's JRE and JDK 1.2 and higher. This file must always be included in the boot classpath when compiling applications for TINI. However, it should never be in your classpath when running applications on your host computer.
- **tini.db**. The tini.db file is an ASCII database that contains information about the class files in the TINI API. TINIConvertor uses this file along with the class files in your application to produce a binary image suitable for interpretation by TINI's JVM.

- **TBIN files**. The TBIN extension is short for "TINI binary" and is the default extension used for binary images that are targeted for loading into the flash ROM. The TBIN files distributed with the TINI SDK contain the implementation of the TINI Java Runtime and the Slush shell application. The TBIN files you need to load depend on which firmware and which hardware you are using. Make sure to check the file "installation.txt" in the docs directory for specific installation instructions of any firmware you attempt to load.
  - Firmware 1.0x (DS80C390-based systems only): Load the files tini.tbin and slush.tbin using the JavaKit or MTK utility.
  - Firmware 1.1x (DS80C390-based systems only): Run the LoaderLoader application for your current firmware revision to install the image tini1_1xbank0.tbin. Then use the JavaKit or MTK utility to load the files tini1_1xbanks0to6.tbin and slush.tbin. Make sure to check with the file 'installation.txt' in the docs directory for specific instructions in this configuration, as it is one of the leading causes of headache and confusion in the TINI universe.
  - Firmware 1.1x (DS80C400-based systems only): Use the JavaKit or MTK utility to load the files tini400.tbin and slush400.tbin.

The tinixxx.tbin contains the binary image of TINI's runtime environment, and may sometimes be referred to as 'the firmware.' It is a combination of the native operating system and the Java API. This file must be loaded before any Java applications can be executed. The slushxxx.tbin file is itself an application, and contains the binary image of the user shell known as Slush. A description and a quick tour of slush are included later in this document.

Loading the TINI Java Runtime Environment

At this point, it is assumed that you have successfully installed your favorite Java development environment, the Java Communications API, and the TINI SDK on the host machine. The steps required for installing the TINI Java Runtime Environment depend on the hardware and firmware versions you have chosen.

Loading Firmware 1.1x on a DS80C400-Based TINI

Loading the TINI 1.1x Runtime Environment on the DSTINIm400 or compatible system consists of loading two files with JavaKit: tini400.tbin and slush400.tbin. The tini400.tbin file loads data from address 400000h to 470000h, and the file slush400.tbin loads from address 470100h to 480000h. The 100h bytes in between the end of the tini400.tbin file and the start of the slush400.tbin file is reserved for flash preservation of network configuration information (such as in conjunction with Slush's "ipconfig –C" command).

Loading Firmware 1.0x on a DS80C390-Based TINI

On DS80C390-based TINI systems, the boot loader program occupies all of bank 0 (address range 0 to 10000h). Loading the 1.0x firmware then consists of loading the files tini.tbin and slush.tbin. The file tini.tbin loads the TINI firmware and core API from address 10000h to address 70000h. the file slush.tbin loads the slush application from address 70100h to address 80000h. The 100h bytes in between the end of the tini.tbin file and the start of the slush.tbin file is reserved for flash preservation of network configuration information, such as in conjunction with Slush's "ipconfig –C" command.

Loading Firmware 1.1x on a DS80C390-Based TINI

The additional features of the 1.1x firmware (dynamic classloading, reflection, IPv6, etc.) require the TINI firmware to share the space occupied by the bootloader. This makes the installation procedure for 1.1x firmware a bit more complicated on a DS80C390-based TINI. The file 'Installation_390.txt' in the 'docs' directory of a TINI 1.1x firmware distribution will have specific instructions for this procedure. However, the basic procedure is described here.

Loading any 1.1x firmware requires running the program 'LoaderLoader.tini' from some other version of the TINI firmware. Each TINI 1.1x distribution includes a pre-built executable version of the 'LoaderLoader.tini' program for several of the latest TINI firmware versions. For example, the TINI 1.12 firmware comes with pre-built 'LoaderLoader.tini' executables for the 1.11, 1.10, and 1.02f firmware versions. In order to run the 'LoaderLoader.tini' application, you will need to FTP three files to the TINI:

1) The LoaderLoader.tini application file.
2) The loaderloader.tlib native library file, required by the LoaderLoader.tini application.
3) The bank 0 image. For the 1.12 firmware, the bank 0 image is stored in a file called tini112-bank0.tbin2.

DALLAS SEMICONDUCTOR MAXIM

All three files are located in a subdirectory of the bin folder, based on the firmware revision they were meant to upgrade. For instance, if you are running firmware 1.02f and are installing 1.12, you will load the files listed above from the 'bin/tini1.02f' directory of the TINI SDK.

The 'LoaderLoader.tini' program copies a new bootloader image from address 0 to 10000h. This procedure must not be interrupted or you may be left with a loader-less TINI. Once the 'LoaderLoader.tini' program is complete, it will prompt you to reset the TINI. After resetting, you should load the remainder of the TINI firmware and the slush application. For the 1.12 firmware, the remainder of the firmware is contained in the file tini112-bank1to6.tbin. The slush application is always contained in the file 'slush.tbin' for the DS80C390-based TINIs.

Connecting TINI to Your Host Computer

Before starting either the JavaKit or MTK applications, make sure you are connected properly to your TINI socket. Use a straight-through (not a null-modem) serial cable to connect to the TINI socket.

For the 390 sockets (E10 and E20), connect your serial cable to the connector labeled J6. Also, make sure the jumper at J1 (labeled 'DTR Reset Enable') is placed.

For the 400 socket (DSTINIs400), connect your serial cable to the connector at J12 (labeled 'Loader – Serial 0'). Make sure the jumper at J14 is placed—this is the DTR reset enable jumper for the 400 socket, though it is not so clearly labeled. Also, make sure the module is seated well into the socket. If you have problems communicating with your TINI, removing and reinserting the module might ensure a proper connection.

Using MTK to Load Files Onto TINI

There are two commonly used tools for loading files onto TINI. The older interface for loading files is JavaKit, which is difficult to get started with due to the need to install and configure the Java communications API. The Microcontroller Tool Kit (MTK)[4] is the answer to the installation problems of JavaKit. It is an executable and therefore does not require the Java Development Kit or the Java Communications API. To install MTK, run the executable install program. It will give you choices for where to install MTK. Normally, the defaults for all of these options are just fine.

Start the MTK program by double clicking on its icon. You may be presented with a list of supported devices. Select the 'TINI' option. At this point, the MTK application window should appear.

The startup procedure is the same as for JavaKit—we will need to choose our communication port and baud rate, open the serial port, and then issue a reset. Choose the "Configure Serial Port" from the 'Options' menu. Select your communication port and baud rate from the drop down boxes. If your communication port does not appear on the list, type it in. Press 'OK' to close the port configuration dialog.

To open the serial port, select the first item under the 'TINI' menu. If you have chosen to use COM1 at 115,200 bps, this selection will read "Open COM1 at 115200 baud". If the port opened, the gray background should turn white. Otherwise, an error message will appear. If you cannot open the serial port, make sure that no other software currently has control of the port you are trying to use.

Next we issue a reset to the TINI. Select the menu entry "Reset" under the "TINI" menu. You should see a loader prompt appear in the text area. If a loader prompt does not appear, check to make sure you are connected to the correct serial connector with a straight-through cable, and that the DTR jumper is placed. Figure 8 shows MTK after a successful TINI reset.

[4] *The MTK is available at ftp://ftp.dalsemi.com/pub/microcontroller/dev_tool_software/mtk/*

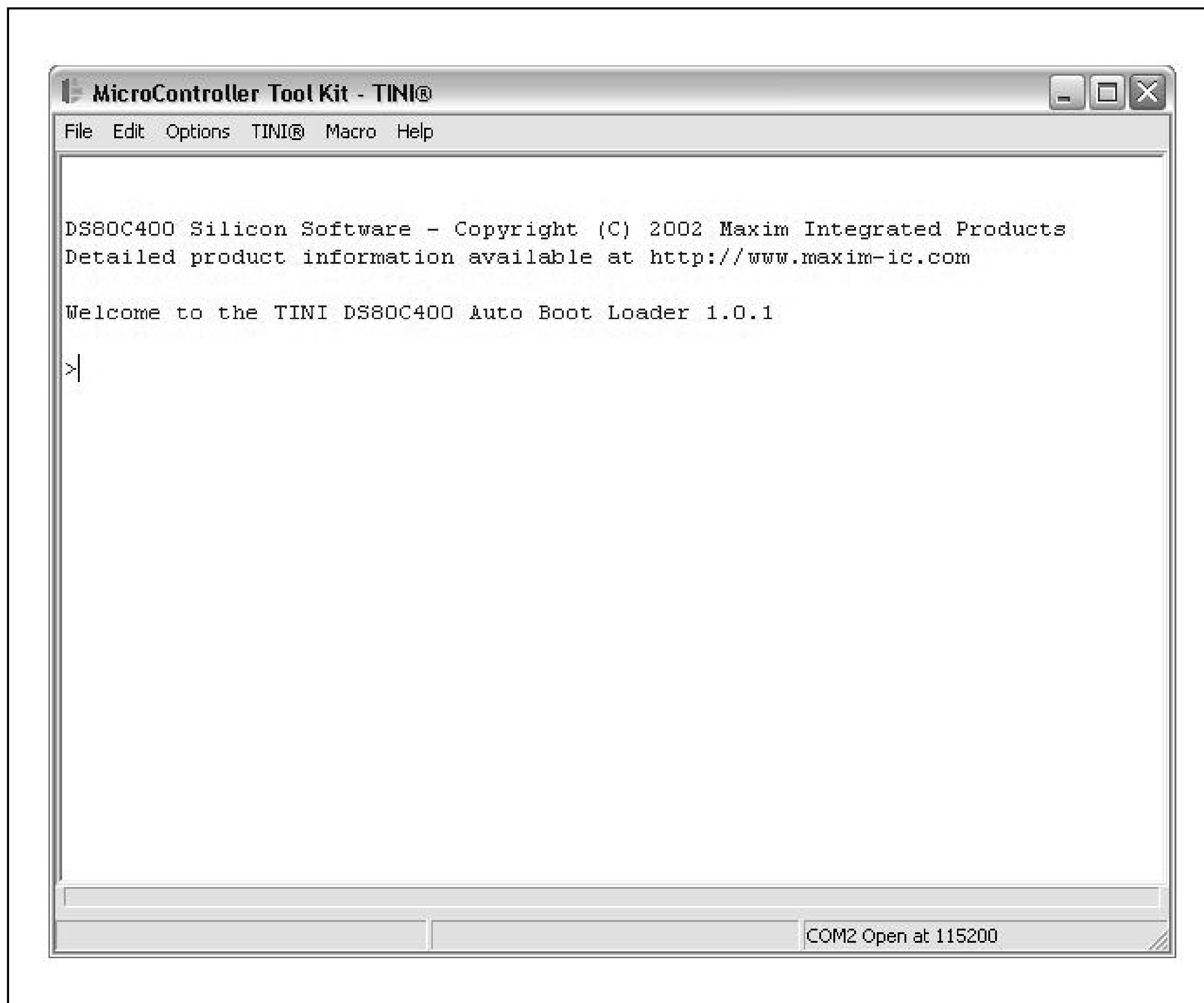

Figure 8. MTK After Issuing a Successful TINI Reset

DALLAS SEMICONDUCTOR MAXIM

Now that we have the loader's attention, we can load the runtime binaries. Select "Load TBIN" from the File menu. Navigate to the bin directory of the TINI SDK and select the proper TBIN files to load (detailed in the section above). In our case, we have a DS80C400-based system, so we are loading tini400.tbin and slush400.tbin. We see the following output from MTK:

```
>
Loading File C:\TINI\tini1.12\bin\Slush_400.tbin
Loading Bank 47...
................................................................
Load complete.
Loading File C:\TINI\tini1.12\bin\tini_400.tbin
Loading Bank 40...
................................................................
Loading Bank 41...
................................................................
Loading Bank 42...
................................................................
Loading Bank 43...
................................................................
Loading Bank 44...
................................................................
Loading Bank 45...
................................................................
Loading Bank 46...
.........................................................
Load complete.
>
```

These files are rather large and take a couple of minutes to load. At this point, the firmware is loaded, and we can start the TINI Java Runtime Environment.

Using JavaKit to Load Files onto TINI

JavaKit is the most commonly used interface for loading file onto TINI because it has been around since nearly the beginning of TINI itself. It is recommended that new users use the MTK as their interface for loading files onto TINI, as classpath and communication API installation make JavaKit an often frustrating way to start working with TINI.

JavaKit is a Swing-based GUI utility, which should be compatible with any JDK version 1.2 or later. It is included with the TINI firmware releases in the file **tini.jar**. JavaKit requires that the Java Communications API be properly installed. See the *Appendix* for more detailed installation instructions and common problems encountered with the Java Communications API.

To start JavaKit, your command line should look something like this:

```
java –classpath c:/tini1.12/bin/tini.jar JavaKit
```

Note that you will need to replace the section 'c:/tini1.12/bin/tini.jar' with the path to the 'tini.jar' file for the firmware version that you downloaded and installed. Also note that you may need to use the full path to the java executable as well, such as 'c:/jdk1.2.2/bin/java'.

If you are using a DS80C400-based TINI, your command will need to look a little different:

```
java -classpath c:/tini1.12/bin/tini.jar JavaKit -flash 40 -400
```

Note the two additional arguments '-flash 40' and '-400' at the end. The first tells JavaKit that this TINI system contains flash starting at address 400000h. The default for JavaKit is assuming that 512k flash lives at address 0, since this is the configuration for the original DS80C390 modules. The second argument tells JavaKit to use some slightly different timing parameters for the reset sequence used to enter the boot loader, which behaves just a little differently from the bootloaders written for the DS80C390.

Once you have JavaKit running, you should see something like in Figure 9.

First, select the port that you will be using to communicate with TINI from the 'Port Name' drop-down list. Then, make sure '115200' is selected for the baud rate. The TINI boot loaders will perform an autobaud sequence, so any baud rate in this box is acceptable, but 115200 will provide you with the best response time.

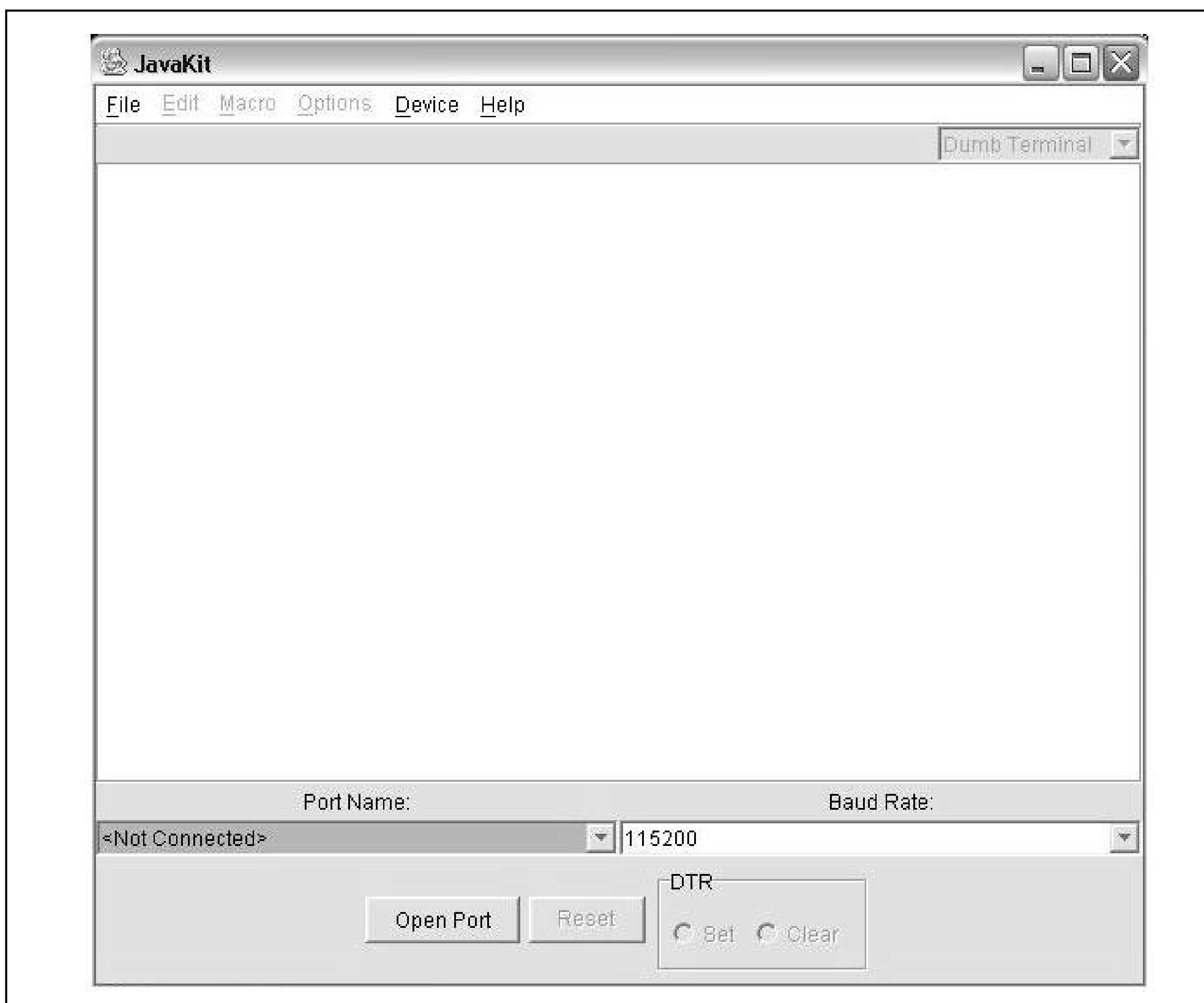

Figure 9. The JavaKit Window

DALLAS SEMICONDUCTOR MAXIM

Once the port and baud rate are selected, click the 'Open Port' button. If the program can open the serial port, you will see the Port Name gray out, and the text on 'Open Port' will change to say 'Close Port'. Hit the 'Reset' button and the TINI prompt should appear. The TINI prompt for the DS80C400 looks like:

```
DS80C400 Silicon Software - Copyright (C) 2002 Maxim Integrated Products
Detailed product information available at http://www.maxim-ic.com
Welcome to the TINI DS80C400 Auto Boot Loader 1.0.1
>
```

If you could not open the serial port, make sure no other resource is using it. PDA software is often guilty of opening serial ports in the background. If you did not see the boot loader prompt, make sure the DTR jumper is placed, and you started JavaKit with the correct command line (used the '-400' switch for the DSTINIm400 module). See the section "Common Problems when Using JavaKit" in the *Appendix* for more troubleshooting details.

Now that we have the loader's attention, we can load the runtime binaries. Select "Load File" from the File menu. Navigate to the bin directory of the TINI SDK and select the proper TBIN files to load (detailed in the previous section). In our case, we have a DS80C400-based system, so we are loading tini400.tbin and slush400.tbin. We see the following output from JavaKit:

```
Loading file: /Users/kardis/TINI/tini1.12/bin/Slush_400.tbin.
Please wait... (ESC to abort.)
Load complete.
Loading file: /Users/kardis/TINI/tini1.12/bin/tini_400.tbin.
Please wait... (ESC to abort.)
Load complete.
>
```

These files are rather large and take a couple of minutes to load. At this point, the firmware is loaded, and we can start the TINI Java Runtime Environment.

Starting the TINI Java Runtime Environment

Now we have loaded the binary images that comprise TINI's runtime. Before the system is booted for the first time, the heap must be initialized. On DS80C390-based TINIs type "BANK 18" at the boot loader prompt and hit Enter. This selects the lowest 64k portion of TINI's heap. Next type "FILL 0" and hit Enter.

```
> BANK 18
> FILL 0
```

On DS80C400-based TINIs, the sequence is the same except that the heap starts in bank 0.

```
> BANK 0
> FILL 0
```

This cryptic two-step sequence fills the low 64k of heap with 0s, forcing the TINI OS to initialize the heap, file system, and all other persistent settings on the next boot sequence.

Now we're ready to boot the system for the first time. To exit the serial loader and boot the TINI runtime, type "EXIT" at the prompt. The following is the text generated by the OS in the boot process for the 1.12 firmware on a DSTINIm400. There will be differences in the exact text across hardware and firmware revisions, but the basic text shown here is representative of all TINI firmware versions.

```
——> TINI Boot <——
TINI OS 1.12 - Copyright (C) 1999 - 2003 Maxim Integrated Products
API Version: 9004
Crystal: 14.7396MHz x 2
Doing First Birthday
```

```
Probing for memory config
Memory Size: 0FD600
Addresses: 182A00,280000
Skip List MM
POR Count: 00000001
Running POR Code
Memory POR Routines
000028
Transient blocks freed: 0000, size: 000000
CPersistant blocks freed: 0000, size: 000000
Memory Available: 0F1240
Creating Task:
0100
01
Loading application: 470100
Creating Task:
0200
02
Application load complete
[-= slush Version 1.12 =-]
[  System coming up.  ]
[ Beginning initialization... ]
[ Not generating log file. ]      [Info]
[ Initializing shell commands... ] [Done]

[ Checking system files... ]      [Done]
[ Initializing and parsing .startup... ]
[ Initializing network... ]
[ Starting up Telnet server... ]  [Done]
[ Starting up FTP server... ]     [Done]
[ Network configuration ]         [Done]
[ System init routines ]          [Done]
[ slush initialization complete. ]
Hit any key to login.
```

After a few seconds, the boot process completes, and slush prompts you for a login.

After pressing a key, slush prompts the user for a login name.

```
Welcome to slush. (Version 1.12)
TINI login:
```

The next section provides a brief introduction to slush that will cover, among other things, the login process.

Slush: A Quick Primer

This section provides a brief overview of slush and a look at just enough of the commands and features we need to load and run the example applications in this document. A more complete description of slush is provided in the 'Slush.txt' file included in the 'docs' directory of the TINI SDK. Note that while there may be some minor differences between slush provided for TINI 1.1x and TINI 1.10x firmwares, the information in this section applies to both.

Slush Defined

Slush is a small command shell intended to provide a UNIX®-like interface to TINI's runtime environment by providing Serial (TTY), Telnet, and FTP servers. Slush is a Java application that is interpreted by TINI's JVM. Slush is less than a full operating system but more than a simple shell. It provides a way to view and manipulate the file system, run other Java applications, and control system functions such as the watchdog timer and network configuration.

Slush is designed to be a multithreaded, multiuser system allowing simultaneous user sessions. It is typically used in the development phase. Slush provides conveniences such as network accessibility using the ubiquitous networking client application Telnet for user interaction and FTP for transferring applications and data files to and from the file system. After an application has been developed and debugged, it is typically built and targeted for installation in the flash ROM, replacing slush. Transitioning an application from the development phase to production deployment is discussed in Chapter 11 of *The TINI Specification and Developer's Guide*.

Starting a New Session

Slush uses a user name and password to authenticate a login request and start a new user session. When slush is booted for the first time (as in the previous section), it creates two new default accounts: a root account with "super user" or administration privileges and a guest account with more limited access to system resources. Additional users can be added or removed by a user with administrative privileges. Table 3 shows the user names and password for the default accounts.

Table 3. Default User Accounts for Slush

ACCOUNT NAME	USER NAME	INITIAL PASSWORD
root	root	tini
guest	guest	guest

When we left the previous section we had booted slush for the first time and left it at the login prompt. It is important to note that both slush and TINI's file system are case sensitive. All characters in the user name and password for both default accounts are lower case. Log on to the system to establish a user session with slush. Use the root account by typing "root<CR>" at the login prompt and "tini<CR>" at the password prompt.

```
TINI login: root
TINI password:
```

The password characters typed at the password prompt are not echoed by the system. After successfully logging on to the system, slush returns a prompt comprised of the host name, TINI in this case, and the login session's current working directory in the file system. Immediately after logging on to the system, the current working directory is the root directory of the file system.

```
TINI />
```

UNIX is a registered trademark of The Open Group.

Exploring the File System

Using slush, we can explore the file system in its initial state, just after the first slush boot. A detailed listing of the files in a directory can be displayed using the 'ls' command with the '-l' option.

```
TINI /> ls -l
total 2
drwxr-x 1 root admin 1 Jan 27 15:13 .
drwxr-x 1 root admin 3 Jan 27 15:14 etc
TINI>
```

The first line after the prompt displays the total number of files and directories contained within the current directory. In the preceding sample listing, the second file is a directory named "etc." This directory is created automatically by slush the first time it boots and contains several system files. Changing to the "etc" directory using the cd (change directory) command and displaying its contents using "ls -l" produces the following listing.

```
TINI /> cd etc
TINI /etc> ls -l
total 5
drwxr-x 1 root admin 3 Jan 27 15:14 .
drwxr-x 1 root admin 1 Jan 27 15:13 ..
-rwxr— 1 root admin 28 Jan 27 15:14 .tininet
-rwx—-- 1 root admin 225 Jan 27 15:14 .startup
-rwxr— 1 root admin 101 Jan 27 15:14 passwd
TINI /etc>
```

This detailed listing displays, from left to right, the following information about each file or directory contained within the current working directory.

- Permissions
- Number of links
- Owner
- Group
- File count/size
- Last modification date
- Name

Let's look at the listing for the .startup file in detail. The permissions for the .startup file, from left to right, indicate that it is not a directory(-). The owner (root in this case) has read (r), write (w), and execute (x) privileges, while others have no read, write, or execute privileges. The file system does not support different groups, but this entry is present for UNIX-listing compatibility when using the FTP server. The link count is also purely for compatibility, since the file system doesn't support links.

All three of the files in the "etc" directory are created by slush during the initial boot sequence. The .tininet file stores the host and domain names. By default the host name is "TINI." The passwd file stores the user name along with the SHA1 (Secure Hash Algorithm) hash of the password for every account on the system. The most interesting of the autogenerated files is .startup. This file is parsed and interpreted by slush on every reboot. It allows a user with administrative privilege to set environment variables and automatically launch applications on system boot. We can view the contents of .startup, or any other ASCII text file, using the cat command.

```
TINI /etc> cat .startup
########
#Autogen'd slush startup file
```

```
setenv FTPServer enable
setenv TelnetServer enable
setenv SerialServer enable
##
#Add user calls to setenv here:
##
initializeNetwork
########
#Add other user additions here:
```

Each line of the file is either a command to be interpreted by slush or a comment that begins with the “#” character. The three lines that begin with “setenv” enable the FTP, Telnet, and serial servers, respectively. So, for example, if an application needed to use the same serial port that slush uses for the serial server, a user with administrative privilege could comment out the “setenv” line that enabled the serial server. The next time the system is booted, slush will only start the FTP and Telnet servers. This allows another application to claim exclusive ownership of the serial port.

Applications can be launched on system boot by adding the appropriate commands to the .startup. For example, adding this command

```
java /bin/MyApp.tini > /log/debug.out
```

causes slush to run MyApp.tini from the bin directory and redirect all output from java.lang.System.out and java.lang.System.err to a log file named debug.out. All applications launched from the .startup file are forced to run in the background. Type “cd /” at the command prompt to return to the root directory.

```
TINI /etc> cd /
TINI />
```

Getting Help

The help command provides a hands-on approach to exploring slush as well as some insight into the capabilities of TINI’s runtime environment. Type ‘help’ at the prompt at any time to obtain a complete list of all commands supported by slush.

```
TINI /> help
Available Commands:
addc          append        arp           cat
cd            chmod         chown         copy
cp            date          del           df
dir           downserver    echo          ftp
gc            genlog        help          history
hostname      ipconfig      java          kill
ls            md            mkdir         move
mv            netstat       nslookup      passwd
ping          ps            pwd           rd
reboot        rm            rmdir         sendmail
setenv        source        startserver   stats
stopserver    su            touch         useradd
userdel       wall          wd            who
whoami
TINI />
```

A command's description and usage is obtained by typing help followed by the name of the command at the prompt. Typing "help java" at the prompt displays the usage message for the java command. Note that this is the usage statement for the java command associated with Slush for TINI 1.0x. The Slush for TINI 1.1x has a couple extra options that we won't discuss here.

```
TINI /> help java
java FILE [&]
Executes the given Java application.
'&' indicates a background process.
```

The java command is used to launch new Java processes. The usage message specifies the required and optional parameters. In this case, the java command requires the name of the application binary file to be executed and optionally allows the user to launch the application as a background process using the & parameter. We'll use the java command later to run the example programs.

At this point we can start a user session, navigate the file system, and get help with unfamiliar commands. We will continue interacting with our "slush user session" in the next couple of sections to configure the network as well as load and run some small example applications. The sections that follow describe new slush commands and functionality as they are encountered.

Configuring the Network

Use the slush command "ipconfig" to set network configuration information. The "ipconfig" command provides several options that allow for complete control of all important network parameters. Executing "ipconfig" with no parameters displays the current network settings. For this section we are using a DSTINIm400 running firmware version 1.12. The main difference between this and running a 1.0x firmware is that IPv6 is supported in 1.12. Other than that, everything in this section applies to both firmware versions.

```
TINI /> ipconfig
Hostname             : tini00e5f5.
Current IPv4 addr.   : 0.0.0.0/0 (0.0.0.0) (active)
Current IPv6 addr.   : fe80:0:0:0:260:
                       35ff:fe00:e5f5/64
                       (active)
Default IPv4 GW      :
Ethernet Address     : 00:60:35:00:e5:f5
Primary DNS          :
Secondary DNS        :
DNS Timeout          : 0 (ms)
DHCP Server          :
DHCP Enabled         : false
Mailhost             :
Restore From Flash   : Not Committed
```

Since we have just installed the runtime and cleared the heap, almost nothing has been configured. The Ethernet address is an IEEE-registered MAC ID to avoid any possible collision on an Ethernet network. It is read from the read-only memory of a 1-Wire chip on the TINI board. This implies that it is always available and always the same, allowing it to serve as a general-purpose unique identification for the TINI board as well as the Ethernet address.

Although we have done nothing to configure IP networking, we have an IPv6 address. This is because IPv6 supports automatic address configuration, which runs immediately once the TINI starts. Also note that the last portion of the IPv6 address is used to generate a semi-unique hostname.

Use the help command to obtain the list of options supported by ipconfig.

```
TINI /> help ipconfig ipconfig [options]
Configure or display the network settings.
[-a IP -m mask]     Set IPv4 address and subnet mask.
[-n domainname]     Set domain name
[-g IP]             Set gateway address
[-p IP]             Set primary DNS address
[-s IP]             Set secondary DNS address
[-t dnstimeout]     Set DNS timeout (set to 0 for backoff/retry)
[-d]                Use DHCP to lease an IP address
[-r]                Release currently held DHCP IP address
[-x]                Show all Interface data
[- h mailhost]      Set mailhost
[-C]                Commit current network configuration to flash
[-D]                Disable restoration of configuration from
                    flash
[-f]                Don't prompt for confirmation
TINI />
```

As you can see from the preceding usage message, “ipconfig” provides fine grain configuration and control of network settings and parameters. We won’t cover all of them here, just enough to get TINI up and running on the network. If there is a DHCP (Dynamic Host Configuration Protocol) server available on your network, you can use the -d option to dynamically obtain an IP address and subnet mask as well as several other network parameters, depending on the configuration of the DHCP server. Usually, if TINI is to be used as a server, you will use a static IP address, making it easy for network clients to access the service(s) TINI is providing. For static network configuration we need to set the IP address and subnet mask at a minimum. The following command sets the IP address and subnet mask.

```
TINI /> ipconfig -a 192.168.0.15 -m 255.255.255.0
Warning: This will disconnect any connected network users and reset all network
servers.
OK to proceed? (Y/N): y
[ Sun Jan 28 14:52:46 GMT 2001 ] Message from System: Telnet server started.
[ Sun Jan 28 14:52:46 GMT 2001 ] Message from System: FTP serverstarted.
```

You will, of course, substitute the IP address and subnet mask used here with values provided by your network administrator. We can test our new settings by “pinging” the TINI board from the host machine, using the ping command. Also, you can see from this command that slush automatically starts Telnet and FTP servers after setting the network information. At this point you should be able to establish a Telnet session with TINI, using the host’s Telnet client. Win32, Solaris, MACOSX, and Linux all provide command line Telnet client programs. There are also graphical Telnet clients available for most platforms that should work fine with TINI.

```
C:\>telnet 192.168.0.15
Connecting To 192.168.0.15...
Welcome to slush. (Version 1.02)
TINI login: root
TINI password:
TINI />
```

Once connected, slush prompts the user for a user name and password. Use the same name and password (root, tini) that we used to log in to the serial session from JavaKit in the previous section. We can kill the Telnet session by using the exit command.

Now TINI is on the network and ready for action. However, with only the IP address and subnet mask set, network messages intended for machines on different physical networks can't reach their destination. To extend TINI's reach beyond its physical network, we will need to set at least one more network parameter: the IP address of the default gateway (or router). The default gateway address is set using the -g option. The other network parameter we would like to set now is the IP address of the DNS (Domain Name System) server using the -p option. This allows us to use host names rather than raw IP addresses when communicating with other hosts. Running the following command from our serial session adds the default gateway and primary DNS server's IP addresses to the current network configuration.

```
TINI /> ipconfig -g 192.168.0.1
-p 192.168.0.2
Warning: This will disconnect any connected network users
and reset all network servers.
OK to proceed? (Y/N): y
[ Sun Jan 28 15:02:53 GMT 2001 ] Message from System: FTP server stopped.
[ Sun Jan 28 15:03:00 GMT 2001 ] Message from System: Telnet server stopped.
[ Sun Jan 28 15:03:00 GMT 2001 ] Message from System: Telnet server started.
[ Sun Jan 28 15:03:01 GMT 2001 ] Message from System: FTP server started.
```

Note that if the FTP and Telnet servers are running, slush stops them before changing the requested network settings. After aborting any active FTP or Telnet sessions, the new network parameters are set and the servers are restarted. We can test both of the new settings by pinging a host machine on another network, using that host's name as opposed to its IP address.

```
TINI /> ping www.ibutton.com
Got a reply from node www.ibutton.com/198.3.123.121
Sent 1 request(s), got 1 reply(s)
```

At this point we'll want to log out of the serial session and close JavaKit. Now we can interact with TINI and run our examples over the network using the host's Telnet and FTP clients. From this point forward in the book nearly all examples will be run from a Telnet client. Start a new Telnet session and run 'ipconfig' with no parameters.

```
TINI /> ipconfig
Hostname            : tini00e5f5.
Current IPv4 addr.  : 192.168.0.15/24 (255.255.255.0) (active)
Current IPv6 addr.  : fe80:0:0:0:260: 35ff:fe00:e5f5/64 (active)
Default IPv4 GW     : 192.168.0.1
Ethernet Address    : 00:60:35:00:e5:f5
Primary DNS         : 192.168.0.2
Secondary DNS       :
DNS Timeout         : 0 (ms)
DHCP Server         :
DHCP Enabled        : false
Mailhost            :
Restore From Flash  : Not Committed
```

Allow this session to remain active because it will be used to run the examples in the following section.

Some Simple Examples

At this point we have loaded the runtime environment and configured TINI for network operation, and now we can interact with the runtime environment, using slush. Now we'll create three very small applications from scratch and detail the process of building, loading, and running the examples. We'll use slush via a Telnet session to run the applications, display any output, interact with the file system, and control processes.

Some of these examples are slightly different when running on a DS80C400-based TINI than when running on a DS80C390-based TINI, but generally the procedure is similar. Where differences exist, they will be specifically pointed out.

HelloWorld

Naturally, we simply must begin with the canonical HelloWorld program. While it won't exactly enhance our skills as Java coders, it does provide a necessary vehicle for describing the application development process in a step-by-step fashion. Typically, to develop and test your application requires these five steps.

1) Create the source file.
2) Compile the source file.
3) Convert the class file.
4) Load the converted image.
5) Run the converted image.

The remainder of this section will detail all five steps. We'll recycle this experience for the remaining examples, allowing us to focus on other details. For the sake of becoming familiar with the development process, we'll perform all of these steps manually. Since this quickly becomes tedious for real-world application development, the process of building and loading applications should be automated using a reasonable combination of make files and shell scripts (or batch files for those running Windows).

Step 1: Create the source file. Create and save a file named “HelloWorld.java” containing the source code in Listing 1.

Listing 1. HelloWorld

```
class HelloWorld
{
    public static void main
    (String[] args)
  {
    System.out.println(“Hello World”);
  }
}
```

Step 2: Compile the source file. Compile “HelloWorld.java” to a class file, using your favorite Java compiler. If you’re using Sun’s JDK and the JDK’s bin directory is in your path, change to the directory that contains the file we just created and execute the following command.

```
javac HelloWorld.java
```

If the compile completes successfully, you should have a new file named “HelloWorld.class” in the current working directory. If you are using Java 1.4 or later, you will need to change this command just a little.

```
javac –target 1.1 HelloWorld.java
```

Step 3: Convert the class file. The utility program TINIConvertor performs a conversion on input, specifically one or more Java class files, and outputs a binary image suitable for execution on TINI. TINIConvertor’s function is described in the *Appendix*. However, it is worth mentioning that TINIConvertor is performing a portion of the class loading process. For the 1.0x firmware, the binary file produced by TINIConvertor is typically about 25 to 35 percent the size of the sum of the original class files. The savings is considerably less for the 1.1x firmware, since space is consumed by information that supports reflection. TINIConvertor does not generate code native to TINI’s microcontroller; rather, it generates a binary file containing Java byte codes that are interpreted by TINI’s JVM.

TINIConvertor is a Java application that lives in the tini.jar file and is run from a command shell on the host. It is controlled by a series of command line parameters that specify the converter’s input and output. A list of all required and extended parameters can be obtained by running TINIConvertor with no parameters.

To convert HelloWorld.class to a binary image that we can execute on TINI, run TINIConvertor supplying the three mandatory command line parameters: input file or directory (-f), API database (-d), and output file (-o). In the following command line examples, line breaks are inserted only for clarity—the entire command should be on one line.

```
java -classpath c:\tini1.12\bin\tini.jar TINIConvertor
  -f HelloWorld.class
  -d c:\tini1.12\bin\tini.db
  -o HelloWorld.tini
```

In this example, our application consists of only one class file, HelloWorld.class, so we can specify the class file’s name with the -f parameter. In general, our applications will consist of several classes in one or more packages. In this case, supply the directory name of the root of the package structure hierarchy. This causes TINIConvertor to include all class files in and below the specified directory when creating the application binary.

The other input required by TINIConvertor is the name of the API database distributed in the SDK. This file is named tini.db and must be supplied with the -d parameter. TINIConvertor uses this file to resolve information between your application and the API. The tini.db file is specific to a version of the TINI SDK. If you have multiple versions of the SDK installed on the host, be sure to use the correct tini.db file.

TINIConvertor produces an output file with the name provided with the –o parameter. Other than being a legal name, as determined by the host file system, there are no specific rules that restrict the name of the final application binary. By convention, the name of the class that contains the main method is used for the file name with an extension of ".tini." The extension is used to indicate that this file is a TINI executable. Following this convention produces a binary output file named HelloWorld.tini.

Step 4: Load the converted image. Use the FTP client provided with your operating system to connect to TINI and transfer the binary image, generated in the previous step, to the TINI file system.

```
C:\tini1.02\HelloWorld>ftp 192.168.0.15
Connected to 192.168.0.15.
220 Welcome to slush. (Version 1.02) Ready for user login.
User (192.168.0.15:(none)): root
331 root login allowed. Password required.
Password:
230 User root logged in.
ftp>
```

After successfully establishing a connection and logging in to slush, we can transfer HelloWorld.tini to TINI's file system. First, type "bin" at the FTP prompt to ensure that our binary image is not altered during the actual file transfer.

```
ftp> bin
200 Type set to Binary
```

Transfer HelloWorld.tini, using this put command.

```
ftp> put HelloWorld.tini
200 PORT Command successful.
150 BINARY connection open, putting HelloWorld.tini
226 Closing data connection.
ftp: 171 bytes sent in 0.00Seconds 171000.00Kbytes/sec.
```

Finally, close the FTP session by typing bye or quit at the prompt.

```
ftp> bye
221 Goodbye.
```

We can check that our file transfer completed successfully by using the ls command at the slush prompt in our Telnet session.

```
TINI /> ls -l
total 3
drwxr-x 1 root admin 2 Jan 28 14:45 .
-rwxr— 1 root admin 171 Jan 28 15:46 HelloWorld.tini
drwxr-x 1 root admin 3 Jan 28 14:45 etc
```

The file HelloWorld.tini now appears in the root directory of the file system and has the same size that was listed during the FTP transfer. Note that all operating systems that are capable of hosting TINI application development have an FTP client that works nearly identically to the preceding session. There exist several graphical FTP clients for various platforms. These are useful for developers that prefer not to work from a command shell. For some, a command line FTP client is preferable because it allows for easy automation of the file transfer process. For example, using the Windows FTP client, we can create a file with the following contents.

```
root
tini
bin
put HelloWorld.tini
bye
```

If we call this file load.cmd, we can use the following command to transfer HelloWorld.tini without any interaction with the FTP client command prompt.

```
C:\tini1.12\myapps\HelloWorld>ftp
-s:load.cmd 192.168.0.15
```

Using the -s option causes the FTP client to read the specified file and execute each line as if it were typed in manually in response to a prompt.

Step 5: Run the converted image. Now we're ready to run the application using the java command at the slush prompt.

```
TINI /> java HelloWorld.tini
Hello World
TINI />
```

HelloWorld.tini executes and produces the output we expect. After the program terminates, control of the user session returns to the command prompt.

Blinky, Your First TINI I/O

Now that we know how to build, load, and execute a Java application, let's try an example that performs the most basic form of I/O by controlling a single microcontroller port pin. There is a status Light Emitting Diode (LED) on the DSTINI512 and DSTINI1 boards that is connected to p3.5 (port 3, bit 5) of the microcontroller. This pin is also shared with the internal 1-Wire network but since we're not doing any 1-Wire at the moment, we're free to play with it. On the DSTINIm400 module, there is a status LED connected to p5.2 (port 5, bit 2). Therefore, our application code will need to behave a little differently on a DS80C390-based TINI than on a DS80C400-based TINI.

The relevant portion of the DS80C390 board's schematic is shown in Figure 10. The anode side of the LED is connected to V_{CC} (the power-supply voltage). A 680Ω current-limiting resistor separates the LED's cathode and the source of transistor Q2. In this circuit, Q2 is just used as a saturation switch to ground. So we can think of it as either being off (nonconducting) or on (conducting). The port pin drives the gate of Q2. Setting the pin high (a logic 1) forces Q2 into a conducting state, causing current to flow through the LED and turning it on. Setting the pin low (a logic 0), forces Q2 to a nonconducting state, stopping the flow of current through the diode, thereby turning it off.

The DSTINIm400 module has a similar configuration, but without the transistor (Figure 11).

The Blinky program, shown in Listing 2, uses the class BitPort from the com.dalsemi.system package to access p3.5. Once we have an instance of BitPort, we can invoke the methods set and clear, to turn the LED on and off, respectively.

```
public void set()
public void clear()
```

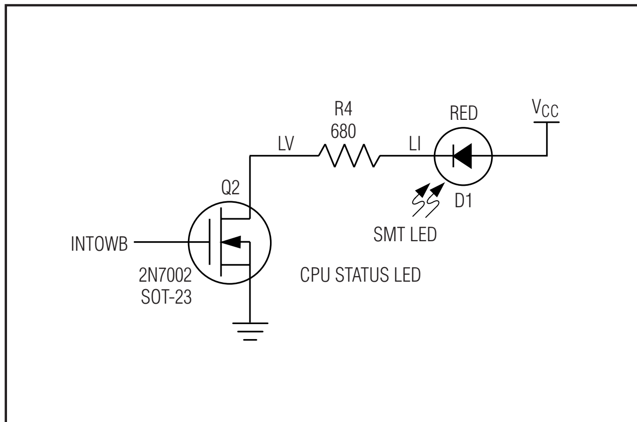

Figure 10. LED on the DSTINI Module

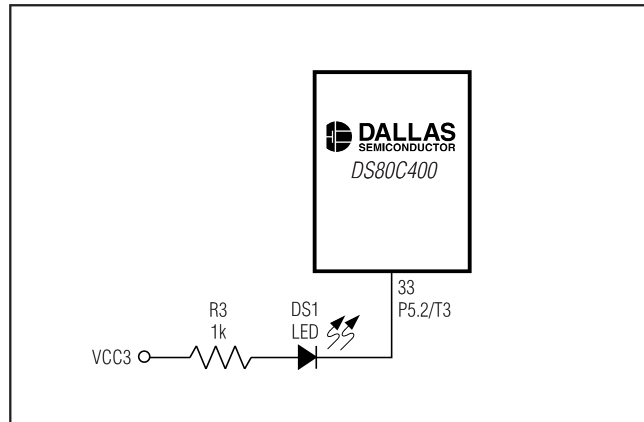

Figure 11. LED on the DSTINIm400 Module

Note that if we use a DS80C400-based TINI (with the status LED connected to p5.2), we would need to change the declaration line for the BitPort instance bp to.

```
BitPort bp = new BitPort(BitPort.Port5Bit2);
```

We can compile, convert, and load Blinky following the steps we used for the HelloWorld example with one exception. The BitPort class is not part of the normal Java API, so simply compiling with “javac Blinky.java” will cause an error. Therefore, you will need to use the ‘-bootclasspath’ tag to tell the compiler where TINI’s API classes are defined.

```
javac –bootclasspath c:\tini1.12\bin\tiniclasses.jar Blinky.java
```

You should never put the ‘tiniclasses.jar’ file in your classpath (or in the jre/lib/ext folder) or you will experience problems running any Java program. You should, however, include the ‘bootclasspath’ argument when compiling any application for TINI. Also, remember the ‘-target 1.1’ switch if you are using Java JDK 1.4 or later.

Now we are ready to run the program. We will make one small change to the way in which we run this program from what we did with the HelloWorld example. Blinky runs forever just brainlessly blinking the status LED at 2Hz. If we run it in the same fashion that we ran HelloWorld (as a foreground process), we would never get our command prompt back in the Telnet session. We would either have to start a new Telnet session to halt Blinky with the kill command or remove power and force the system to reboot. Instead, we can eliminate these problems by just executing Blinky in the background.

```
TINI /> java Blinky.tini &
TINI />
```

Now if you take a look at your TINI board, you should see the status LED (D1 on the DS80C390 board, DS1 on the DSTINIm400) blinking about twice per second. It will continue to blink until you kill the process. To kill a process from slush, you use the *kill* command specifying the process id on the command line. To learn the process id, use the *ps* command.

```
TINI /> ps
3 processes
1: Java GC (Owner root)
2: init (Owner root)
4: Blinky.tini (Owner root)
```

Listing 2. Blinky Listing for DS80C390-Based TINI

```
import com.dalsemi.system.BitPort;
class Blinky
{
      public static void main(String[] args)
      {
              BitPort bp = new BitPort(BitPort.Port3Bit5);
              for (;;)
              {
                       // Turn on LED
                       bp.clear();
                       // Leave it on for 1/4 second
                       try
                       {
                              Thread.sleep(250);
                       }
                       catch (InterruptedException ie)
                       {
                       }
                       // Turn off LED
                       bp.set();
                       // Leave it off for 1/4 second
                       try
                       {
                              Thread.sleep(250);
                       }
                       catch (InterruptedException ie)
                       {
                       }
              }
      }
}
```

The ps command shows us the total number of processes and lists each process ID followed by its name. Now let's kill Blinky, since the thrill of a blinking light is probably starting to wane.

```
TINI /> kill 4
TINI /> ps
2 processes
1: Java GC (Owner root)
2: init (Owner root)
```

After killing process 4 and examining the process list, we see that the process count has gone from three to two, and only the background garbage collector and command shell are running. Notice that the first Java process started during the bootup phase. Slush in this case is always named "init." Even if you kill and immediately restart the same process, it will not get the same process id. Process ids are always incrementing and are not recycled. So, if you were to run Blinky again and do a *ps*, the process ID would be 5. The process ID is an unsigned 16-bit value and therefore rolls to the lowest available value after 65,535.

HelloWeb—A Trivial Web Server

Finally, we'll upgrade the HelloWorld example, taking it to the World Wide Web. The HelloWeb program, shown in Listing 3, is a very small web server. The "built-in" HTTPServer class, provided in the com.dalsemi.tininet.http package, does the bulk of the work. HelloWeb creates an instance of HTTPServer that listens for client HTTP requests on server port 80. It also logs all requests to a file named web.log in the "/log" directory. The main loop simply spins forever, invoking the serviceRequests method on the HTTPServer instance.

The build process for this example is a little trickier than for the previous two examples. The reason is that the HTTPServer class is a part of the built-in firmware for the 1.0x TINI firmware, but is not part of the default firmware package for TINI 1.1x. When the new features for 1.1x (reflection, serialization, dynamic class loading, IPv6) were added, some things had to be removed from the TINI firmware to make room. These portions that were removed became modules that could be built in to any application that needed them. One of those modules is the HTTPServer. Therefore, the HTTPServer is a module in the 1.1x firmware, but part of the core API for the 1.0x firmware. This means we will have different compiling and building steps for the different firmware versions.

To compile for the 1.0x firmware, the command is identical to the one we used for the Blinky example:

```
javac –bootclasspath c:\tini1.02g\bin\tiniclasses.jar HelloWeb.java
```

To compile for the 1.1x firmware, we also need to include the modules archive in our classpath.

```
javac –bootclasspath c:\tini1.12\bin\tiniclasses.jar
–classpath c:\tini1.12\bin\modules.jar;. HelloWeb.java
```

Now that we have our class file HelloWeb.class, we can convert to a TINI file. For the 1.0x firmware, this again looks identical to the Blinky example:

```
java –classpath c:\tini1.02g\bin\tini.jar TINIConvertor
–f HelloWeb.class –o HelloWeb.tini
–d c:\tini1.02g\bin\tini.db
```

As you might expect, though, the conversion process for the 1.1x firmware is a little more involved. Since the HTTPServer class is not a part of the firmware, we need to add it to our application. We could use TINIConvertor for this and tell it to build in the HTTPServer class file that is stored in the modules.jar file, but the conversion would fail. An error would tell us that the HTTPServer class references another class that we need to build into our application, the HTTPServerException class. TINIConvertor isn't smart enough to go find these classes on its own, so we go ahead and tell it to use the HTTPServerException from the modules.jar file and try again. This time it fails and tells us we need the class HTTPWorker.

[5] *Slush does not support the use of <ctrl>C to terminate foreground processes.*

Listing 3. HelloWeb

```
import com.dalsemi.tininet.http.HTTPServer;
import com.dalsemi.tininet.http.HTTPServerException;
class HelloWeb
{
    public static void main(String[] args)
    {
        // Constuct an instance of HTTPServer that listens for
        // requests port 80
        HTTPServer httpd = new HTTPServer(80);
        httpd.setHTTPRoot("/html");
        httpd.setIndexPage("index.html");
        // Specify a name for the log file and turn on logging
        httpd.setLogFilename("/log/web.log");
        httpd.setLogging(true);
        // Spin around forever servicing inbound requests
        for (;;)
        {
            try
            {
                // Wait for a new request
                httpd.serviceRequests();
            }
            catch (HTTPServerException e)
            {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

We could keep adding classes to the command line like this, but our command line is getting difficult to manage and you're probably getting sick of the runaround. This is where the BuildDependency tool comes in. BuildDependency is another Java application that acts as a smart wrapper for the TINIConvertor application. By using a dependency file that declares which classes are required to build a target (for example, the HTTPSERVER target requires the classes HTTPServer, HTTPServerException, HTTPWorker, PostScript, and PostElement from the com.dalsemi.tininet.http package), we can easily build in the classes we need for our application.

```
java –classpath c:\tini1.12\bin\tini.jar BuildDependency
–f HelloWeb.class –o HelloWeb.tini
–d c:\tini1.12\bin\tini.db
–p c:\tini1.12\bin\modules.jar
–x c\tini1.12\bin\owapi_dep.txt
–add HTTPSERVER
```

You'll notice that the first few arguments here look like TINIConvertor arguments. Sure enough, BuildDepend-ency ignores these and passes them along to TINIConvertor. However, it intercepts the last three arguments and generates the arguments needed for TINIConvertor to build your application.

The '-p' argument tells BuildDependency to look in the file 'modules.jar' for the dependent classes that will be required for this application. The '-x' argument tells BuildDependency which file to use for its list of class dependencies. If you open that file, you can see how dependency lists are declared and how you might use this to create your own dependency lists. The '-add' argument tells BuildDependency to add in the class files declared in the 'HTTPSERVER' dependency list. See the *Appendix* for a detailed discussion of BuildDependency.

Now we are ready to load the HelloWeb.tini application on our TINI. Like Blinky, HelloWeb runs forever and should therefore be executed as a background process. But there's a little more work to do before we can run HelloWeb. Unlike the first two examples, HelloWeb requires some application data in the form of an ASCII file, namely index.html. On the host, create and save a file named index.html with the following contents.

```
<html>
<head><title>Hello Web!</title></head>
<body>
<h1>Hello from TINI!</h1>
</body>
</html>
```

Now let's return to our slush Telnet session to make directories for our web root (the directory our web server will serve files from) and log file.

```
TINI /> mkdir html
TINI /> mkdir log
TINI /> ls -l
total 7
drwxr-x 1 root admin 6 Jan 28 14:45 .
drwxr– 1 root admin 0 Jan 28 18:06 log
drwxr– 1 root admin 1 Jan 28 18:06 html
-rwxr– 1 root admin 297 Jan 28 18:05 HelloWeb.tini
-rwxr– 1 root admin 220 Jan 28 17:58 Blinky.tini
-rwxr– 1 root admin 171 Jan 28 15:46 HelloWorld.tini
drwxr-x 1 root admin 3 Jan 28 14:45 etc
```

Use FTP again to “put” index.html into the “/html” directory we just created. To make sure we transferred the file successfully, we can, from the slush prompt, change to the “/html” directory and display the contents of index.html, using the ‘cat’ command.

```
TINI /> cd html
TINI /html> cat index.html
<html>
<head>
<title>Hello Web!</title>
</head>
<body>
<h1>Hello from TINI!</h1>
</body>
</html>
```

Just to summarize, we should have done the following to our system:

- Loaded the application HelloWeb.tini
- Created the directory ‘/html’ and stored the file ‘index.html’ in it
- Created the directory ‘/log’

Now that we have the web server application binary (HelloWeb.tini) and the HTML file that it will serve in the web root, we can return to the root directory and start the program as a background process.

```
TINI /html> cd ..
TINI /> java HelloWeb.tini &
TINI />
```

Typing the ‘ps’ command at the slush prompt shows that our server is indeed up and ready to receive and process client HTTP requests.

```
TINI /> ps
3 processes
1: Java GC (Owner root)
2: init (Owner root)
5: HelloWeb.tini (Owner root)
```

Now we can test our simple server, using any browser and typing TINI's IP address or DNS name in the URL line. Figure 12 shows the results of browsing the elaborate website served by HelloWeb, using the Netscape browser.

Recall that when we created the instance of HTTPServer, we specified that it generate a log file. Let's take a look at its contents, using the 'cat' command from the slush prompt.

```
TINI /> cd log
TINI /log> cat web.log
192.168.0.3, GET, index.html
TINI /log>
```

The log file shows us that the server has processed one "GET" request for the file 'index.html' from a client with the IP address 192.168.0.3. You can hit the reload (or refresh if you're using Internet Explorer) button on your browser several times and watch the log file grow by one entry for each new request.

If this were a real application serving real web pages, we would need to handle the log file a little differently. One option is that we don't enable logging, since we are working with a relatively small memory footprint. Eventually, the log file would grow too large to fit in TINI's memory. Another option is to somehow manage the size of the log file, either by occasionally deleting the log file or shrinking its contents. Some applications periodically off-load the log file. The point is that blindly logging can be dangerous, since you're likely to run out of memory.

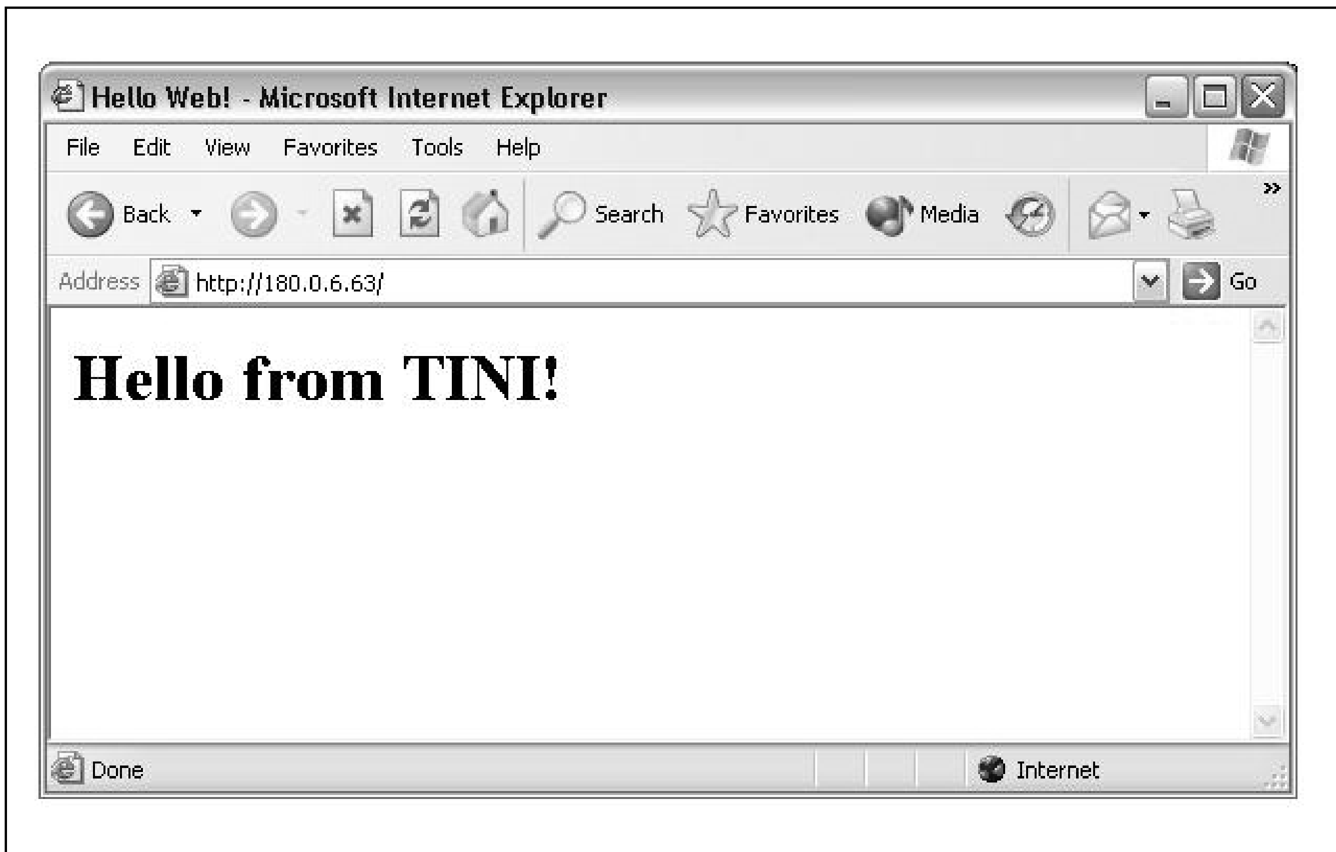

Figure 12. Page Served by the HelloWeb Application

Debugging Tips

Trivial applications like those in the previous section require little in the way of debugging. So the development cycle of building the application on the host and loading and running it on TINI isn't really much of a burden. Real-world applications are much more complicated and involve significant debugging.

This is one of the more complex areas of TINI application development, and any embedded device development for that matter. As a general rule, do all of the debugging you possibly can on your development host. On a host machine the use of a full-featured IDE that provides a runtime environment with integrated source-level debugging can further aid in the development and debugging cycle.

There are broad classes of applications that can be developed for the TINI platform that will also run on larger, more traditional Java platforms. These classes include applications that use the following mechanisms for monitoring and controlling external devices and communicating with other networked machines.

- Serial communication
- TCP/IP networking
- 1-Wire networking

If your application uses only the standard Java packages supported by TINI (see the file docs/API_Diffs.txt in the TINI SDK) and extensions available on most Java platforms—namely, the Java Communications API and the 1-Wire API—then all debugging can be accomplished using just your host's development environment.

Since TINI's main purpose is interacting with physical devices, it also provides I/O capabilities above and beyond those supported by any other Java platform. Once you're writing applications that make use of the APIs that expose these expanded I/O capabilities, your applications will only run on TINI and therefore must be debugged on TINI. Here are some examples of TINI's expanded I/O capabilities.

- Parallel I/O
- Port pin I/O
- Controller Area Network (CAN)
- I^2C^{TM}, SPI^{TM}

If your application makes use of APIs that support any of the above, you lose source level debugging capabilities and are relegated to using exceptions with informative detail messages and old-fashioned console (System.out.println) debug output.

Also, there's a pretty good chance that if you're using expanded I/O capabilities, TINI is connected to specialized hardware. This often brings traditional hardware diagnostic and debug equipment into the picture—anywhere from expensive DSOs (Digital Storage Oscilloscopes) and logic analyzers to very inexpensive tools like logic probes and DMMs (Digital Multi-Meters).

I^2C is a trademark of Philips Corp. Purchase of I^2C components from Maxim Integrated Products, Inc., or one of its sublicensed Associated Companies, conveys a license under the Philips I^2C Patent Rights to use these components in an I^2C system, provided that the system conforms to the I^2C Standard Specification as defined by Philips.

SPI is a trademark of Motorola, Inc.

Using the DS80C400 Silicon Software

When the DS80C400 was designed, it was decided that a suite of functionality should be exposed in a ROM that could be accessed from programs written in 8051 assembly, C, or Java. Size constraints meant the functionality in the ROM could only be a subset of that included in the TINI Runtime Environment. The ROM would therefore be a useful starting block for building C and assembly programs, offering a proven network stack, process scheduler, and memory manager. Simple programs like a networked speaker could easily be implemented in assembly language, while C could be used for more complex programs, such as an HTTP server that interacts with a file system.

The ROM contains a useful set of functionality, including:

- Network stack with Berkeley-style interface
- Priority-based process scheduler
- Memory manager
- DHCP Client
- TFTP Client
- Utility functions such as CRC16 and random number generation
- Access to the internal 1-Wire bus

Developing Applications in 8051 Assembly

At its core, the ROM functionality of the DS80C400 is coded in 8051 assembly language. The natural interface to these functions is therefore in assembly. Section 23 of the *High-Speed Microcontroller User's Guide: DS80C400 Supplement*[6] describes the functions available from the ROM and the interface for calling those functions.

Coding the DS80C400 in assembly is not a very popular option. Most developers would rather use C for their applications. However, assembly can be a useful option for small applications with strict performance requirements. *Application Note 609: Internet Speaker with the DS80C400 Silicon Software* describes a highly optimized Internet speaker written in 8051 assembly language. It reads a stream of 16-bit audio samples from a TCP connection and writes the samples to a digital-to-analog converter at 44.1kHz.

It is also possible to only develop critical portions of an application in 8051 assembly, while the rest of the application is written in some other language like C or Java. This is often the ideal solution, as most developers find it easier to implement top-level logic in a high level language, and squeeze extra performance out of the timing sensitive portions of their application with raw assembly code.

For more information on adding assembly language routines to Java programs, see the documentation in the TINI SDK in the files *docs/Native_Methods.txt* and *docs/Native_API.txt*. For more information on adding assembly routines to a C program, see the documentation from the C compiler vendor.

Developing Applications in C

You can develop applications in C for both the DS80C390 and the DS80C400. However, if you want to use network functions on the DS80C390, you need to use a third-party network stack. Although the TINI Java Runtime for the DS80C390 has a network stack, it is too highly integrated with the Java Virtual Machine to be separated and used with a C program.

The DS80C400 solves this problem with its Silicon Software. The functionality described in the User's Guide Supplement for the DS80C400 can be exposed to applications written in C with the use of some library code.

[6] *Available at www.maxim-ic.com/DS80C400UG*

The basic procedure for calling ROM functions is described in the *High-Speed Microcontroller User's Guide: DS80C400 Supplement*. However, calling these functions from C is a little more complicated, since the interface is described in assembly language. Parameters must be converted from the C compiler's conventions to the conventions used by the ROM. Code written by C compilers can pass parameters in a variety of ways—in static XDATA locations, in registers, on the hardware stack or on a software stack. The ROM functions accept parameters in different ways. For example, the socket functions accept parameters stored in a single parameter buffer, and many of the utility functions accept parameters passed in special function registers or direct memory locations. To translate from the C compiler's calling conventions to the ROM's parameter conventions, Dallas Semiconductor has written libraries for accessing the functions of the ROM. Using ROM functions in your C programs involves only importing the library and including a header file.

In addition to the ROM libraries, several extension libraries have been written to provide useful functionality that is not included in the ROM due to size constraints. These libraries include:

- File System
- DNS Client
- A port of the 1-Wire public domain kit
- I^2C
- SPI

Providing extension libraries is an ongoing process.

At the time of this writing, ROM libraries and extension libraries are only supported for the Keil tools, because support from IAR and SDCC is relatively new. However, an early release of the ROM libraries for SDCC is available from our FTP site, and an early release of the ROM libraries for IAR is currently underway. Eventually, the range of libraries supported for the Keil, IAR, and SDCC libraries will be the same.

The C Library Project Web Page

The C Library Project web page[7] is the main resource for developing applications in C for all development tools. This page provides links to binary distributions, sample applications, history information, and documentation for each library. There are also links to relevant applications notes and some release notes that attempt to address frequently asked questions about C library development.

Using the Keil Tools for the DS80C400

Application Note 613: Using the Keil C Compiler for the DS80C400 describes how to get started developing for the DS80C400 using the Keil development tools. It provides step-by-step instructions for producing a HelloWorld application, and describes a simple network application that uses some ROM and extension libraries.

[7]*Library Project is located at www.maxim-ic.com/ds80c400/libraries.*

APPENDIX

This appendix provides documentation on the tools generally used during TINI development.

Java Tools

To work with the TINI Java Runtime Environment, you need a Java compiler and a Java virtual machine, both of which come with the Java Software Development Kit. The Java compiler (javac) is going to be used to generate class files that can be built and executed on TINI. The Java virtual machine can be used to run compiled Java classes, and is needed to run the JavaKit, TINIConvertor, and BuildDependency applications.

Which Version of the Java Software Development Kit Do I Need?

There are several versions of the Java SDK available. The latest stable version as of this writing is 1.4.2, but updated revisions are released pretty regularly. Even though 1.4.2 is the latest version released, older versions of the Software Development Kit are suitable for TINI development use. In particular, many developers (and Dallas Semiconductor engineers) successfully use Java SDK versions 1.2.2, 1.3.1, and 1.4.2.

Note that the Java classes on TINI are implemented to match the Java classes from the Java 1.1.8 SDK. This older version of the Java SDK is not suitable for use in building TINI applications, but it may be useful to download this version for a handy copy of the documentation. Since many of the classes implemented on TINI have added functions in the 1.2.2 and later Java API's, the documentation for the 1.1.8 API is a more accurate source for what methods are implemented on TINI.

What is J2SE? What is Java 2?

After Sun released Java 1.2.0, they began calling it Java 2. This is confusing, as Java SDK 1.3.1 is also part of Java 2, and so is Java SDK 1.4.2. Presumably, Java SDK 1.5 will also be Java 2. Clear? In reality, Java 2 refers to the current Java platform, which is a little more than just the compiler and virtual machine and SDK. However, for the purposes of TINI development, it is just these tools that we are interested in, which carry the more formal version-numbering scheme. For this reason, we will talk about version number 1.3.1 and 1.4.2, and ignore things like Java 2.

The J2SE is the Java 2 Standard Edition, which contains the compiler, documentation, virtual machine, and other useful Java tools. You won't find a Java SDK download on the Sun website, because it is part of the J2SE package.

Downloading the Java Software Development Kit

Unfortunately, the link for the Java SDK download page has been prone to change in the past. However, the current link for the J2SE download page is http://java.sun.com/j2se/downloads/index.html. This page contains links for recent releases of the J2SE package. You can also find older versions of the SDK through an archive link, where you can download version 1.1.8 if you like.

If the above link does not work, you can start your search for the J2SE download at Java's home page: http://java.sun.com. Look for the *Downloads* link and then find a link for J2SE or Java 2 Standard Edition.

Installing the Java Software Development Kit

Once you have downloaded the JDK package, installing is straightforward. Follow Sun's instructions and let the install process put all the files in their default locations. Once installed, make sure the path to the virtual machine and compiler are part of your system's PATH variable. Generally, the path to the virtual machine will look something like c:\jdk1.3.1\bin\java.exe, with the compiler in the same directory. Therefore, make sure 'c:\jdk1.3.1\bin' is in your PATH.

Using the Compiler and the Virtual Machine

This document cannot hope to cover using 'javac' and 'java.' Sun provides a good tutorial for getting started with Java at http://java.sun.com/docs/books/tutorial/index.html. If you go into a bookstore, you will be faced with more choices for Java programming books that you will know how to handle. Some sources recommended by the TINI community include:

- Bruce Eckel's *Thinking in Java*: www.mindview.net/Books/TIJ/
- A JavaWorld article on beginning Java books: www.javaworld.com/javaworld/jw-02-1999/jw-02-bookreview.html
- O'Reilly's "Java in a Nutshell"

What is a Classpath?

The classpath is the set of directories and archives that the Java virtual machine searches when loading class files. In simple terms, it is where Java looks for the classes and class libraries it needs to run. Classpath issues are some of the most frustrating of working in Java, especially when just getting started.

The classpath comes into play when working with TINI because the utilities such as JavaKit, TINIConvertor, and BuildDependency are all located in the Java archive (jar file) tini.jar, included in the TINI SDK. You can specify your classpath in several different ways.

You can specify the classpath when invoking the java virtual machine:

```
java –classpath c:\tini1.12\bin\tini.jar TINIConvertor
```

You can set the classpath environment variable on your system:

```
set classpath=c:\tini1.12\bin\tini.jar;.
java TINIConvertor
```

Note that in the previous example, we also set the current directory (‘.’) to be part of our classpath. Otherwise, we will not be able to load class files from the directory in which we are running.

You can also place class files and jar files (Java ARchives) into a special directory where the virtual machine will find them. If you are running your virtual machine out of the [JDK]/bin directory, then you can place class files and jar files in the [JDK]/jre/lib/ext directory. These class files will then be found before anything else in your classpath.

There are a few special considerations to note when using the /jre/lib/ext directory as a ‘super-classpath’:

1) Do not ever put the file ‘tiniclasses.jar’ into the /jre/lib/ext directory. This file contains classes as they are implemented in the TINI firmware. This generally results in problems that can only be described as ‘weird,’ because they usually don’t manifest themselves in the same way twice. Note that putting the ‘tini.jar’ file in the /jre/lib/ext directory is OK, although if you ever want to use a different firmware version, you should replace this file with the one from the firmware version of TINI that you would like to use.
2) Sometimes, the /jre/lib/ext path will appear to not work. This shows up when trying to run a program that uses the Java Communications API. You could swear that you’ve placed all the files in the right places (such as the comm.jar file to the /jre/lib/ext directory), but you are still having classpath problems. The likely problem here is that you are not running the Java executable that you think you are! The Java installation usually puts a copy of java.exe somewhere in the ‘c:\windows’ directory, which generally appears first in the system PATH. To fix this problem, either delete the copy of ‘java.exe’ in the windows directory (only for the brave) or use the full path to the java executable you mean to use (as in ‘c:\jdk1.3.1\bin\java MyClass’).
3) Class files in the /jre/lib/ext directory are loaded before anything in your real classpath or current directory. You might think that you are using an archive or class file somewhere else, and are pulling your hair out trying to figure out why you have changed the class, but the new behavior is not showing up. The reason is that the class file is being ‘overridden’ by the higher priority one in /jre/lib/ext.
4) Using the /jre/lib/ext directory isn’t very portable. Everyone is likely to have configured the ext directory in a different way. It is useful for doing development, but as a solution to installing on a customer’s system, it is prone to difficulties and frustration.

JavaKit

JavaKit is a graphical program (written in Java) that is used to load programs onto a TINI system. It also acts as a serial terminal, and can be used to transfer data over the serial port to and from TINI. It is programmed to communicate with the serial bootloaders for the DS80C390 and DS80C400, and can load both HEX files and TBIN files.

Before we run JavaKit, we must install the Java Communications API. The classes that are a standard part of the Java Development Kit download do not include classes for communicating over the serial port, which is why we must install an optional Java package.

Installing the Java Communications API

Installing the Java Communications API (also known as Javax.comm) is undoubtedly the number one source of frustration and confusion for those just starting out with TINI. Problems with installing javax.comm (and thus problems running JavaKit) are the main motivation for the development of the Microcontroller Tool Kit (MTK), a C-based loader program that can be used for the same purposes as JavaKit. However, MTK is new and JavaKit is still the interactive loader of choice for TINI developers, so it won't be killed anytime soon.

The Java Communications API can currently be downloaded from http://java.sun.com/products/javacomm/ for both Windows and Solaris platforms. A readme file is included in the distributions with installation instructions, but the basic process is this:

1) Copy the file win32com.dll to the [JDK]/bin directory, where [JDK] is the directory that you installed the Java Development Kit in (for instance, copy win32com.dll to the directory c:\jdk1.3.1\bin).
2) Copy the file javax.comm.properties to the directory [JDK]\jre\lib.
3) Copy the file comm.jar to the directory [JDK]\jre\lib\ext

A 'jar' file is a Java ARchive, a collection of class files bound together, and is usable by the Java virtual machine, somewhat like a library of Java classes. Jar files and class files that are places in the [JDK]\jre\lib\ext directory are found automatically by the Java virtual machine before those you declare in your classpath. For this reason, you should never put the file tiniclasses.jar in the jre\lib\ext directory. This file contains the core Java classes as they are implemented on TINI. By placing this file in your jre\lib\ext directory, the Java Virtual Machine for your computer will try to execute the Java classes as they are implemented on TINI, which will never work.

At this point, try running JavaKit with the command line:

```
Java –classpath [TINI]\bin\tini.jar JavaKit
```

In Windows, it is likely that the response will be

```
Java.lang.ClassNotFoundException: Could not find the class
javax.comm.UnsupportedCommException
```

This error means that the Java Virtual Machine could not load the Java Communications API, even though it was installed as directed. The problem is typically because the Java Development Kit installation leaves a copy of the files java.exe and javaw.exe in the windows\system directory, which is generally included in the system's path prior to the [JDK]\bin path. There are a few solutions to this:

1) Remove the files java.exe and javaw.exe from the windows\system directory. This is generally perfectly safe and won't affect any other applications.
2) Rearrange the path such that [JDK]\bin is listed before the windows\system directory.
3) When executing the Java Virtual Machine, use the absolute path, such as 'c:\jdk1.3.1\bin\java –classpath [TINI]\bin\tini.jar JavaKit'.

If the GUI window shows up after entering the command for JavaKit, then you have successfully installed the Java Communications API and can run JavaKit.

Note that if you are using the TINIm400 reference board, you should start JavaKit with the command line 'java –classpath [TINI]\bin\tini.jar JavaKit –400 –flash 40.' The '-400' tag tells JavaKit that it needs to alter some reset timing parameters, because the DS80C400 bootloader's serial prompt does not respond to a reset as quickly as the DS80C390 does. Also, JavaKit needs to know something about the flash configuration of the device it is communication with, so it can clear an area of flash if necessary. JavaKit defaults to the flash configuration of the DS80C390-based TINI boards (512k flash starting at address 0). The '-flash 40' tag tells JavaKit that the flash begins at address 400000h. See the following for more on the command line options of JavaKit.

Running JavaKit

Now that we have JavaKit running, let's communicate with a TINI. If everything is working right, we should be able to connect TINI to your PC with a straight through cable, open the port in JavaKit, hit the RESET button and see a loader prompt that looks something like:

```
DS80C400 Silicon Software - Copyright (C) 2002 Maxim Integrated Products
Detailed product information available at http://www.maxim-ic.com
Welcome to the TINI DS80C400 Auto Boot Loader 1.0.1
>
```

To perform this sequence, make sure the TINI is powered up and you have the cable connected to the right serial connector on the TINI board. In the JavaKit panel, under 'Port Name,' select the port on your computer that the TINI is connected to (such as 'COM1'). Make sure the baud rate is selected at 115200, which is the default. Then press the OPEN PORT button. If no other device is currently using that port of your computer, the port should be open. Then press the RESET button, and the load banner should appear.

Following are instructions for some of the most common JavaKit operations:

- Load a File: Under the FILE menu option at the top, select "Load File" or "Load HEX File as TBIN." The boot loaders written for the DS80C390 and DS80C400 understand both the HEX file format and the TBIN file format. Using the "Load File" option, you can select either type of file and JavaKit will send the file correctly. However, loading a TBIN file is about twice as fast as loading a HEX file, so it may be convenient to use JavaKit to convert a HEX file to a TBIN file on they fly, using the "Load HEX File as TBIN" option. A prompt will appear in the JavaKit window when the load operation is complete.
- Verify File: Occasionally, it is useful to check what file is loaded into TINI, to see if it is different from what you think should be loaded. The "Verify File" option under the FILE menu will have the boot loader compare an input HEX or TBIN file to what is currently in memory. If no errors are output, the file matches what is in memory.
- Dump Memory: To dump memory values in the current bank (a 64kB range), type the range in at the loader prompt: D START END. For example, type `D 00 FF` to dump the memory from address 0 up to address FFh.
- Change banks: To change the selected 64kB bank that is selected, use the 'B' command at the loader prompt. For example: `B 41` selects the bank with address range 410000h to 41FFFFh.
- Clear memory: Use the fill command ('F') at the loader prompt to clear a range of memory. To clear the entire bank, just use `F 0`. To clear a selected range, specify like with the dump command: `F 0 100 150` will clear the memory from address 100h to 150h.
- Start executing the application: At the loader prompt, type 'E' to exit the loader and start executing your application.

Common Problems when Using JavaKit

As this is a getting started sequence, it is prone to several errors. The most common symptoms and their possible solutions are listed here for troubleshooting:

The HEX File Format

The HEX file format is an industry-standard data format used for loading files and otherwise representing data. Several good references can be found online by searching for the HEX file format. One good source of information is www.8052.com.

The boot loaders for the DS80C400 and DS80C390 support the HEX file record types 0 (data record), 1 (end of file record), and 4 (extended linear address record). Files with record types 2 (extended segment address), 3 (start segment address), and 6 (start linear address) will need to be altered to remove those records.

The HEX file format, while widely understood and used, is highly inefficient for loading data quickly. It is an ASCII file format, representing bytes in their hexadecimal ASII equivalents. Therefore, at least two characters are used to represent 8 bits of real data (not including address and checksum overhead in each record).

SYMPTOM	POSSIBLE SOLUTION
JavaKit won't start, complaining with a ClassNotFoundException on UnsupportedCommException.	Java can't find the archive 'comm.jar'. Make sure this file is in the [JDK]/jre/lib/ext directory. Also, you may need to invoke 'java' with the full path, since there may be other 'java' executables in your path trying to execute. Do this by typing 'c:\jdk1.3.1\bin\java …'.
Port will not open.	Make sure no other software has control of the port. PDA software usually likes to take control of a serial port even when inactive, so you may need to exit those applications.
No port names appear under the 'Port Name' listing.	This is typically caused when Java can't find the 'javax.comm.properties' file. Remember that this file belongs in the [JDK]/jre/lib directory where you have installed the COMM API.
When I hit RESET, I get 'No Response From TINI'.	This is the most common problem. Check the following possible problems. 1. Make sure you have the correct power supply for your board. 2. Make sure you are connected to the right serial connector on TINI. On the TINIs400, this is labeled "Loader – Serial 0." On the E10/E20 boards, this is the connector at J6. 3. Make sure the DTR jumper is placed. On the TINIs400, this is the jumper J14. On the E10/E20 board, this is jumper J1. 4. Make sure you started JavaKit with the '-400' tag if you are using a DS80C400-based board (such as the TINIm400), 5. Make sure you chose the correct serial port on your computer to open. 6. Make sure the TINI board is seated properly in the connected. The TINIm400 boards are a tight fit into their socket, and this has been a frequent problem.
I can't load TBIN files on my TINIm400.	Make sure to start JavaKit with both the '-flash 40' and the '-400' tags.
I can't load a file into banks 48, 49, 4A…4F.	JavaKit only assumes that a board has 512k flash. If you would like to load beyond this range (the TINIm400 has 1MG flash), use the '-ROMSize 1048576' tag.

The TBIN File Format

The TBIN file format is a binary file format specifically used by the DS80C390 and DS80C400 boot loaders. A TBIN file can have one or more TBIN records. The format for a TBIN file is:

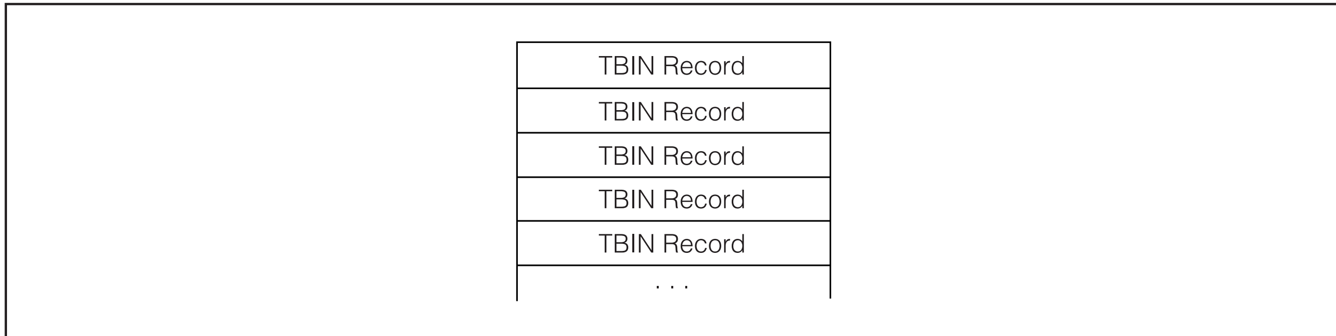

Figure 13. TBIN File Format

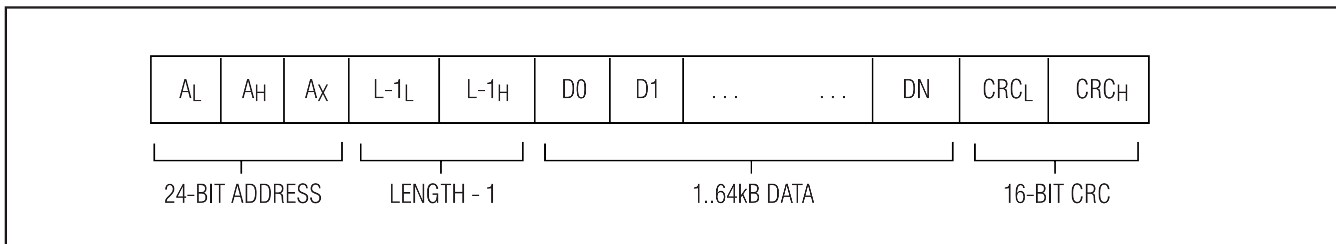

Figure 14. TBIN Record Format

A TBIN record has 3 bytes of starting address, 2 bytes of length, [length] bytes of data, and a 2-byte CRC16. Therefore, a TBIN record can contain a maximum data payload of 64kB plus 7 bytes of overhead.

As a few examples:

- If a TBIN record is to be loaded at address 000000h, A_{low} = 00h, A_{high} = 00h, and A_{xhigh} = 00h
- If a TBIN record is to be loaded at address 010203h, A_{low} = 03h, A_{high} = 02h, and A_{xhigh} = 01h
- If a TBIN record contains 5 bytes of data to load, $Length_{low}$ = 04h and $Length_{high}$ = 00h
- If a TBIN record contains 508h bytes of data to load, $Length_{low}$ = 07h and $Length_{high}$ = 05h
- If a TBIN record contains 65,536 bytes of data to load, $Length_{low}$ = FFh and $Length_{high}$ = FFh
- If a TBIN record 1 byte of data to load, $Length_{low}$ = 00h and $Length_{high}$ = 00h

Note that a TBIN record with 0 bytes of data to load is invalid.

Documentation on JavaKit's Macro Feature

You can use a simple text file to create a macro—a collection of commands—that you would like JavaKit to run for you. Normal text in the macro file will be passed over the serial port as if you had typed it, but there are a few higher-level commands you can include in your macro as well:

`<JavaKit LOAD>FILENAME`

Loads the file FILENAME as if you had used the menu command "FILE->Load File." For instance, to load the file 'D:\tini\bin\Slush.tbin', insert the following line into your macro:

```
<JavaKit LOAD>D:\tini\bin\Slush.tbin
```

`<JavaKit VERIFY>D:\tini\bin\Slush.tbin`

Performs a verify using the file FILENAME as if you had used the menu command “FILE->Verify File”. For instance, to verify the contents of memory against the file ‘D:\tini\bin\Slush.tbin’, insert the following line into your macro:

```
<JavaKit VERIFY>D:\tini\bin\Slush.tbin
```

<JavaKit SET_DTR>

Sets DTR. If the DTR jumper is placed, setting DTR holds the TINI in reset.

<JavaKit CLEAR_DTR>

Clears DTR. If the DTR jumper is placed, clearing DTR releases reset on the TINI. If the TINI sees an end-of-line character over the serial port, it will try to autobaud and enter the loader. Otherwise, it will time out waiting for the end-of-line character, and attempt to execute the application in memory.

<JavaKit RESET>

The same as hitting JavaKit’s RESET button. Sets DTR, clears DTR, and waits for the loader prompt.

<JavaKit PROMPT?

Sets the prompt that commands wait for to signify a finish. Replace the ‘?’ with the expected prompt. Usually in the JavaKit macro file you see “<JavaKit PROMPT>” since the normal prompt is a greater-than sign.

JavaKit’s Command Line Arguments

JavaKit contains many possible command line arguments of varying usefulness. Only a few are useful to most developers, but all are described here. You will find that most options default to values relevant to interacting with the DS80C390 boot loader, since JavaKit was originally written to handle only that processor. It has grown over the years to handle the DS80C400 as well.

If you type ‘java JavaKit –help’, you will see short descriptions for a few command line arguments, which are offered here in more detail. JavaKit options are generally not case sensitive. However, you must still get the capitalization right on ‘JavaKit,’ and probably on the PORTNAME argument to the ‘-port’ tag since that value is the name of a system resource.

- **-port PORTNAME:** You can specify to JavaKit that you would like it to automatically open an port for you. The ‘PORTNAME’ argument must match one of the ports listed in the ‘Port Name’ list.
- -**baud BAUDRATE:** You can specify to JavaKit that you would like it to use a different baud rate than the default 115200 when opening the port you specified with the ‘-port’ argument.
- **-macro:** Specifies a comma separated list of macros that should be run by JavaKit after automatically opening the port selected with ‘-port.’
- **-exitAfterRun:** Specifies that once the macros run from the ‘-macro’ tag have completed, JavaKit should close.
- **-log:** Generates a log file called ‘JavaKit.log’ that creates a list of all activity performed by JavaKit.
- **-advanced:** JavaKit contains several options beyond those listed with the ‘-help’ command. Use the ‘-advanced’ tag to see them all.
- **-400:** Tells JavaKit to use timing parameters tuned for the DS80C400 boot loader. The boot loader in the DS80C400 requires a little more time
- **-flash BANK:** Specifies that the device we are communicating with has flash starting at bank ‘BANK.’ Since loading to the flash requires first zapping the flash (using the boot loader’s ‘Z’ command), JavaKit needs to know if the sector it is loading to needs to be zapped or not. Optionally, you could manually zap each bank by hand from the boot loader prompt. The starting bank for flash defaults 0 (for the DS80C390-based TINI boards). For use with the TINIm400, use the switch ‘-flash 40.’

Note that the first several options are intended for use in automating the loading of TINI boards, and aren’t really practical for general development cycles.

Now if you type ‘java JavaKit –advanced’, you unleash all of JavaKit’s options. Following is a list of options we have not yet covered:

- **-padSize PADSIZE:** Indicates a number of white space characters to send to the TINI when programming a HEX file. The white space characters serve to delay the processing of a HEX record, giving the TINI time to finish programming the last record into the flash.
- **-binPause TIME:** Indicates a number of milliseconds to wait between loading binary segments. This serves to give the TINI time to finish processing the last TBIN record it received.
- **-bankSize SIZE:** Currently unimplemented. Meant to specify the size for each memory bank.
- **-ROMSize SIZE:** Specifies the size of the flash in bytes. This defaults to 524,288 (512kB) for the DS80C390 TINI boards. Usually, the default value is fine for most TINIm400 operations. However, if you would like to load into a full 1MB flash, use '-ROMSize 1048576.' Note you will need to use the '-flash' argument when using a TINIm400.
- **-flushWait TIME:** This argument is a duplicate of the '-binPause' argument.
- **-resetWait TIME:** Specifies the amount of time to wait between toggling DTR to reset the TINI and writing an end-line-character to autobaud the boot loader. The default reset wait time is tuned for the DS80C390 boot loaders. However, you should generally not need to use this switch even for the DS80C400, because the '-400' tag implicitly sets the reset wait to a known good value for that boot loader.
- **-debug:** When 'debug' is turned on (using this switch or the menu option), data sent and received is displayed in the console (i.e., the DOS prompt or terminal window) that started JavaKit. This can generate a huge amount of debug information, so it is best to only enable debug when troubleshooting specific problems, and not in general.
- **-verbose:** When 'verbose' mode is enabled (using this switch or the menu option), additional information is printed to the JavaKit screen during file loading and verifying, tracking the progress of the operation and providing information about the target destination address.
- **-noDTRTest:** Tells JavaKit not to check for a boot loader after toggling DTR. Generally, after JavaKit toggles DTR it tries to raise the boot loader and checks for a loader prompt, printing out an error message if it could not find a loader prompt. This switch tells JavaKit to skip that step.
- **-allow:** Allows loading files to a bank higher than 7. Using this switch on the DS80C390 based TINI's is dangerous because if your board is not configured right, an attempt to load to bank 8 is indistinguishable from an attempt to load bank 0, which can destroy your boot loader. Only use this switch if you have a DS80C390 based TINI with a flash larger than 512kB on chip enable 0.
- **-loaderdanger:** Allow updating of the boot loader. As the command line description for this states, "You should probably not do this." In previous version of TINI, we experimented with boot loader updates that ended up with lots of loader-less TINI's due to a process that was prone to serial communication problems and lots of user error. It is not recommended to use this tag unless a Dallas Semiconductor engineer is holding your hand.
- **-movebank0 BANK:** Specifies a bank that should be targeted for loading instead of bank 0 when a TBIN file targets bank 0.
- **-m400:** Convenience argument that is the equivalent of '-flash 40 –400.'
- **-ymodem:** Deprecated—no longer a supported option.
- **-noAutoZap:** Disables the automatic zapping of flash banks before loading.

-noBank0Protect: JavaKit goes to great lengths to protect the accidental overwriting of bank 0 by default, because this is where the boot loader for DS80C390 systems is stored. Using this tag omits all the extra protection and checks. If you are using a DS80C390-based system, only use this tag if you are sure that you want to overwrite your boot loader.

JavaKit's Menu Options

We've already talked about the more useful JavaKit menu options. Here we'll discuss each menu option, for those curious about what else JavaKit can do.

- **FILE: Load File.** Sends a selected HEX or TBIN file to the boot loader for loading into the memory space of TINI.
- **FILE: Verify File.** Compares a selected input HEX or TBIN file to the current memory contents. If any discrepancies are found, the verify process outputs an error statement. Otherwise, the verify operation terminates quietly.

- **FILE: Load HEX File as TBIN.** Transforms a HEX file to a TBIN file on the fly for faster loading.
- **FILE: Enable Loader Update.** The ability to update your boot loader is normally disabled, and should almost never be enabled. Most people will never need to enable the loader update as performed by JavaKit. Loader updates for the 1.1x TINI firmware are now performed by application code, which is much more fail-safe than using JavaKit to update a loader.
- **FILE: Update Loader.** This option is normally disabled, and should generally not be used. It is available for updating boot loaders on very old DS80C390-based TINI boards. This option should not be enabled or performed without assistance from a Dallas Semiconductor engineer.
- **FILE: Load Macro.** Opens a load file dialog to load a macro from a file for execution with the 'MACRO: Run Macro' menu option.
- **FILE: Save Macro.** Opens a save file dialog to save the current macro being recorded, which was started with the 'MACRO: Record Macro' menu option.
- **FILE: Exit** Quits the JavaKit program. Quitting JavaKit nicely (rather than with a Ctrl-C) allows JavaKit to preserve some useful configuration information such as the last known window size and location.
- **EDIT: Select All.** Selects all the text in the JavaKit window area.
- **EDIT: Copy.** Copies the selected text into the copy buffer.
- **EDIT: Paste.** Inserts the current text from the copy buffer to the current cursor location.
- **MACRO: Record Macro.** Begins to record JavaKit's actions in a macro that can be saved, loaded and run as a group of commands. See the documentation on JavaKit's macro feature for more information.
- **MACRO: Run Macro.** Runs the currently selected macro of JavaKit commands. Only one macro is selected/loaded at a time. To select a different macro, load it from a file or record a new one. See the documentation on JavaKit's macro feature for more information.
- **OPTIONS: Enable Console Output.** Sends output that would normally be sent to the JavaKit window to the console (DOS prompt or terminal) that started JavaKit.
- **OPTIONS: Send CTRL-C.** Immediately sends the CTRL-C command byte (0x03) over the serial port to the microprocessor.
- **OPTIONS: Enable Line Wrap.** JavaKit normally does not wrap text at the edge of the JavaKit window, meaning that text can appear far to the right of the text area. Enable line wrapping to see lines of text wrap to the next line when it would otherwise go beyond the edge of the window.
- **OPTIONS: Don't Trap Control Keys.** JavaKit allows some hotkeys to frequently used functions such as 'Load File.' This menu option allows you to disable JavaKit trapping those sequences, in which case the control sequence will be passed on through the serial port (and thus to the microprocessor).
- **OPTIONS: Generate Log File.** Allows you to enable and disable logging of your JavaKit session to the file 'JavaKit.log.'
- **OPTIONS: Disable Autozap Mode.** Performs the same function as the command line option '-noAutoZap.' However, note that autozapping can be disabled and enabled with this menu option. Use of the menu options overrides any settings set on the command line.
- **OPTIONS: Disable Bank 0 Protection.** Performs the same function as the command line option '-noBank0Protect.' However, note that bank 0 protection can be disabled and enabled with this menu option. Use of the menu options overrides any settings set on the command line.
- **OPTIONS: Enable Debug Mode.** Performs same function as the command line option '-debug.' However, note that debug mode can be disabled and enabled with this menu option. Use of the menu options overrides any settings set on the command line.
- **OPTIONS: Enable Verbose Mode** Performs same function as the command line option '-verbose'. However, note that verbose mode can be disabled and enabled with this menu option. Use of the menu options overrides any settings set on the command line.

- **DEVICE: TINIm390.** Sets timing configuration and flash address/size information to the defaults for the DS80C390-based TINI boards. This includes a 100ms DTR reset wait time and 512kB flash assumed to start at address 000000h.
- **DEVICE: TINIm400.** Sets timing configuration and flash address/size information to the defaults for the TINIm400 TINI board. This includes a 250ms DTR reset wait time and 1MB flash assumed to start at address 400000h.

TINIConvertor

TINIConvertor is a command line tool provided by Dallas Semiconductor to convert Java class files into TINI applications. It is a Java tool, and is provided with the TINI Software Development Kit. It can be found in the file tini.jar in the bin directory of the TINI SDK. TINIConvertor is used to output '.tini' files, which can then be loaded (typically by ftp) into TINI's file system and executed with slush's 'java' command.

Why Do We Need TINIConvertor?

The Java class file format (documented in the Java Virtual Machine Specification for the curious: http://java.sun.com/docs/books/vmspec/index.html) is the core of Java's strength as a programming language. Without going into too much detail, the Java class file format contains all the information that a virtual machine needs to execute Java byte codes from that class. The Java class file format uses strings extensively to reference fields, methods, and other classes. For example, consider the following Java code:

```
StringBuffer sb = new StringBuffer();
sb.append("this is a test");
```

This might look something like this when compiled into Java byte codes (note that this is a gross misrepresentation of what Java byte codes actually look like, and is only meant to convey the general idea):

```
Create new object "java.lang.StringBuffer"
Push reference to "this is a test" on the stack
Invoke method "java.lang.StringBuffer.append(String)"
```

To complete these operations, the Java virtual machine must find the class "java.lang.StringBuffer" by name, find and run its "<init>()" method, push a string onto the Java stack, and then find and run the StringBuffer's "append(String)" method. Each of these operations involves some kind of String search to match the class names to the actual loaded class object or match the method names to the actual Java byte codes as stored in memory. While the class file format is what gives Java its "Write Once, Run Anywhere" claim, all this string comparison is a little much for an 8-bit processor to handle. Therefore, we have the TINI class file format, which turns the above code into something more like this:

```
Create new object with class number 0x54
Push reference to "this is a test" on the stack
Invoke method 0x12 of class 0x54
```

In reality, the Java byte codes are not changed, and the TINI class file format is pretty similar to the Java class file format. The major difference is in the constant pool, where references to classes, methods, and fields are altered to contain indexes into class tables, method tables, and field tables. This conversion is what makes it possible for TINI to handle Java.

Different Versions of TINIConvertor

There are two basic versions of TINIConvertor, one for each version of the firmware. TINIConvertor 0.75 is the current version in use for the TINI 1.0x firmwares. Note that this version number may change with future releases of 1.0x firmware, but should remain below 1.0. The banner for TINIConvertor 0.75 starts with the following text:

```
TINIConvertor + ZIP
Version 0.75 for TINI
built on or around May 3, 2001
Disassembler/Builder Version 0.15 TINI, October 30, 2000
JiBDB 0.60 for TINI, November 10, 2000
Copyright (C) 1996 - 2001 Dallas Semiconductor Corporation.
```

TINIConvertor 1.31 is the current version in use for the TINI 1.1x firmwares. An update of TINIConvertor was required with the introduction of reflection in TINI 1.10. The banner for TINIConvertor 1.31 starts with the following text:

```
TINIConvertor (KLA)
Version 1.31 for TINI 1.1
Built on or around August 27, 2003
Copyright (C) 1996 - 2003 Dallas Semiconductor Corporation.
```

Since TINIConvertor versions are prone to change, you won't see their version numbers printed very often. Instead, you will often see "TINIConvertor for 1.0 firmware" or "TINIConvertor for 1.1 firmware."

How to Use TINIConvertor

TINIConvertor has lots of options, which can be a little confusing. First, let's start with a simple example of its use, and we'll add complexity as we go.

Even for the simplest of applications, you must supply three arguments to TINIConvertor. The first is the name of the class file that you would like to convert. The second is the name of the output file. The third is the name of the database that corresponds to the version of firmware you are using. Note that these arguments can be delivered in any order, as long as the class file name follows the '-f' tag, the database name follows the '-d' tag, and the output file name follows the '-o' tag.

Let's say you've compiled HelloWorld.java and now have HelloWorld.class. Let's also say that you are using TINI firmware 1.12, which is installed into the C:\TINI\ directory. The following command line will build your HelloWorld application:

```
java –classpath
  c:\TINI\tini1.12\bin\tini.jar TINIConvertor
      –f HelloWorld.class
      –o HelloWorld.tini
      –d c:\TINI\tini1.12\bin\tini.db
```

Note that the line feed is added for clarity—the entire command should be on the same line. Also note that this is the same command as if you were using the 1.02g firmware, except that you would need to change the directories for 'tini.jar' and 'tini.db'.

After you execute this command you should have a file HelloWorld.tini, which you can FTP to TINI and execute with Slush's 'java' command. Now let's consider a slightly more complex example, where we have two class files:

```
public class HelloWorld2
{
      public static void main(String[] args)
      {
             System.out.println("Hello TINI World!");
             System.out.println("Helper has this to say: " +
                                Helper.getMessage());
      }
}
```

HelloWorld2.java

```
public class Helper
{
      public static String getMessage()
      {
             return "Help! Get me outta here!";
      }
}
```

Helper.java

If we compile these classes and run the same command line we ran earlier (but replacing 'HelloWorld' with 'HelloWorld2'), we will see this error message (with the 1.1x TINIConvertor):

```
Exception caught: java.lang.RuntimeException: Could not finish
loading: java.lang.ClassNotFoundException: Could not find the class
Helper
```

Or this error message with the 1.0x TINIConvertor:

```
Uncaught exception: Class "Helper" not found.
```

TINIConvertor is not smart about what it does: it does not go and look in the current directory or your classpath for the class files that it needs to load. It only converts exactly what you tell it to convert. If you didn't tell it to load a class file that it needs to load, it will choke with an error message like the one listed above. Note that you don't need to tell TINIConvertor where to look for classes that are part of the firmware (such as java.lang.String, java.util.Calendar, etc). These are taken care of with the 'tini.db' file.

In order to build our two-class example, we need to add another '-f' tag to the command line:

```
java -classpath
  c:\TINI\tini1.12\bin\tini.jar TINIConvertor
        -f HelloWorld2.class
        -f Helper.class
        -o HelloWorld.tini
        -d c:\TINI\tini1.12\bin\tini.db
```

Now consider that we may have 10 classes in the current directory that we want to build into our application (Helper1.class through Helper10.class, for example). We could list each of these classes with their own '-f' tag for an excruciatingly long command line:

```
java -classpath
  c:\TINI\tini1.12\bin\tini.jar TINIConvertor
        -f HelloWorld2.class
        -f Helper1.class
        -f Helper2.class
        -f Helper3.class
        -f Helper4.class
        -f Helper5.class
        -f Helper6.class
        -f Helper7.class
        -f Helper8.class
        -f Helper9.class
        -f Helper10.class
        -o HelloWorld.tini
        -d c:\TINI\tini1.12\bin\tini.db
```

Instead of this, we can also tell TINIConvertor to build a directory with the same '-f' switch we've been using all along. If the argument to a '-f' is a directory, TINIConvertor will build in every class file it finds in that directory and all of its subdirectories.

```
java -classpath
  c:\TINI\tini1.12\bin\tini.jar TINIConvertor
        -f .
        -o HelloWorld.tini
        -d c:\TINI\tini1.12\bin\tini.db
```

In this case, we have told TINIConvertor to build in everything it finds in and below the current directory.

Now that we have introduced the ability to add lots of classes into our applications, we have also introduced the ability for new headaches. Suppose that somewhere in one of our directories that we are building into our application is a class file that contains a 'public static void main(String[])' function, which is not the one we intend for our TINI application to execute. Perhaps it was used to debug that class independently of the rest of the application. Let's return to our two-class example for this:

```
public class Helper
{
      public static String getMessage()
      {
              return "Help! Get me outta here!";
      }
      public static void main(String[] args)
      {
              System.out.println("Debugging Helper class…");
              System.out.println("getMessage returns: "+getMessage());
      }
}
```

Helper.java

```
public class HelloWorld2
{
      public static void main(String[] args)
      {
              System.out.println("Hello TINI World!");
              System.out.println("Helper has this to say: " +
                                    Helper.getMessage());
      }
}
```

HelloWorld2.java

When we build this application, TINIConvertor will take the first 'static void main(String[])' method it finds for your application entry point. Maybe in this case it will be from HelloWorld2, maybe it will be from Helper. This can be very frustrating, because developers often forget when they have littered their code with 'public static void main(String[])' methods used for debugging. To tell TINIConvertor which class contains your application entry point, that is, the 'public static void main(String[])' that you really want to run, use the '-m' argument and the class name that contains our main function:

```
java –classpath c:\TINI\tini1.12\bin\tini.jar TINIConvertor
  –f HelloWorld2.class
  –f Helper.class
  –o HelloWorld.tini
  –d c:\TINI\tini1.12\bin\tini.db
  –m HelloWorld2
```

Note that when we start introducing packages this becomes a little more complex. Suppose that our classes Helper and HelloWorld2 are declared to be in package com.dalsemi, and the source files live in the directory com/dalsemi. In this case the command line above becomes:

```
java –classpath c:\TINI\tini1.12\bin\tini.jar TINIConvertor
   –f com/dalsemi/HelloWorld2.class
   –f com/dalsemi/Helper.class
   –o HelloWorld.tini
   –d c:\TINI\tini1.12\bin\tini.db
   –m com.dalsemi.HelloWorld2
```

Note that the arguments to '-f' refer to file names, and the argument to '-m' is a Java class name.

Arguments to TINIConvertor

- **-f FILENAME:** Build in the file FILENAME into the application. If FILENAME refers to a directory, then all the class files in that directory and all subdirectories are built into the application. If FILENAME refers to a JAR file, then all the class files in that archive are built into the application. Class files that have been compiled with the Java Development Kit 1.4 or later should have been built with the "-target 1.1" switch, otherwise JavaKit will print the error message "Error: Minor Version not 3." The '-f' switch may be used multiple times to add as many class files as required to the application. If no class contains the method 'static void main(String[])', the application will not convert.
- **-o OUTPUTFILENAME:** Specifies the name of the application file that will be built by TINIConvertor. If a file by the name of OUTPUTFILENAME already exists, it will be overwritten. Note that TINIConvertor is not smart about the file extension of the output file—whatever name is given for OUTPUTFILENAME will be the name of the output file.
- **-d DATABASEFILE:** Specifies the database file that contains information on the TINI firmware that this application is being built for. The TINI database files contain information that converts class names, method names, and field names into class numbers, method numbers, and field numbers for the classes that are part of the TINI API. Applications need to be rebuilt for different firmware revisions, since class numbers, etc., are likely to change from firmware version to firmware version.
- **-v:** Tells TINIConvertor to produce verbose output when converting the class files into an application. This information can be useful for debugging when you might be having problems building an application.
- **-m MAINCLASSNAME:** Specifies which class contains the intended application entry point. The method 'static void main(String[])' in the class MAINCLASSNAME will be the first Java code executed when the application starts.
- **-n NATIVELIBNAME:** Build the native library NATIVELIBNAME into the application. Applications can find native libraries both in the TINI file system and when they are built as part of the application file.
- **-l:** Builds the application in the TBIN file format. Files in the TBIN format can be loaded directly by the TINI boot loader. TBIN files built with the '-l' switch (and targeted for the correct address) will be the default startup application executed when the TINI OS boots.
- **-t HEXADDRESS:** When used in conjunction with the '-l' switch, specifies the target address where the TBIN file will be loaded. If an address is not specified, the starting address will be 70100h for TBIN files (which is where firmware for DS80C390 based TINI's begins application execution). To build a startup application for a DS80C400 based TINI, use the argument '-t 470100.'
- **-path ARG (*Option only available on TINIConvertor for 1.1x firmwares*):** Specifies path(s) to look in for class files that were specified with the '-f' option. The paths can be separated by commas, colons, or semi-colons. Native libraries specified with the '-n' option can also be found in the specified path.

- **-i (*Option only available on TINIConvertor for 1.1x firmwares*):** Lets TINIConvertor 'find' files it needs that were not specified on the command line by looking in the classpath. This can be useful for small applications that may contain inner classes or depend on a small number of local classes, but is not recommended for building final applications. This is because it is often difficult to know exactly what is in your classpath, and it is possible that some class files might be built into your application that were not intended to be included.
- **-noreflect (*Option only available on TINIConvertor for 1.1x firmwares*)**: Build the application without reflection information. Reflection information can add a significant amount of size of to a TINI application, since it means adding string data for every method and field name of every class. If an application does not need reflection functionality, using this option will save a considerable amount of space, though it won't make the application faster or more efficient. Note that excluding reflection information from an application will cause functions like *Class.forName(String)* and *Class.newInstance()* to fail. This has been known to cause applications such as TINIHttpServer and some applications that use the 1-Wire API to fail.
- **-ref CLASSNAME (*Option only available on TINIConvertor for 1.1x firmwares*):** Used in conjunction with the '-noreflect' argument, the '-ref' argument allows you to specify a class that you WOULD like to maintain reflection information on. This is useful if you know there are only a few classes in your application for which you will need reflection information.
- **-fake (*Option only available on TINIConvertor for 1.1x firmwares*):** Build a class that can provide a wrapper TINI application to load a class dynamically and run it's main method. This tag should only be used when building Slush, and even then its usefulness is debatable. It allows you to execute a Java class file (instead of a TINI application), such as with the command line 'java MyClass.class.' The inserted class actually loads the specified class file and invokes its main method. It is useful as a demonstration of dynamic class loading, but would never be used in a real application that cared about performance.
- **-b FILENAME (*Option only available on TINIConvertor for 1.1x firmwares*):** Builds a database file for this application, writing it to a file named FILENAME. This can be useful for running a faster dynamic class loader. Dynamic class loading is a slow process on TINI (it is basically the same process as TINIConvertor, but running on TINI instead of on your PC), but doing some 'preconverting' can help speed the process up. An example is included in the TINI SDK called 'FastClassLoader' that makes use of this idea, and uses this TINIConvertor tag.

There are a number of "hidden" options in TINIConvertor beyond those documented here. For instance, TINIConvertor for 1.1x firmware contains options **–r**, **-ver**, **-db**, **-exc**, **-api**, **-nat**, **-bytes**, and -**offs**. TINIConvertor for 1.0x firmware contains a similar set of 'hidden' options. These options are not hidden for any devious purpose and are absolutely useless to TINI developers other than those at Dallas Semiconductor, where the TINIConvertor tool is also used to build the firmware image (the core API including network stack, virtual machine, etc., implemented in 8051 assembly language). These options facilitate that build process. Other TINI developers cannot alter the TINI firmware, so these options are not documented because they have no reason to be used outside of Dallas.

Common Problems Reported When Using TINIConvertor

SYMPTOM	POSSIBLE SOLUTION
Java reports “Exception in thread "main" java.lang.NoClassDefFoundError: TINIConvertor”.	The file ‘tini.jar’ is not in your classpath. Make sure to point to it with the ‘-classpath’ option to TINIConvertor… java –classpath [TINI]\bin\tini.jar …
TINIConvertor reports an error “Minor Version not 3”.	The class file you are trying to convert was probably built using JDK 1.4 or later. These versions of Java contain a virtual machine change that TINIConvertor does not support. Recompile the class file with the ‘-target 1.1’ tag.
TINIConvertor starts, but reports “Could not finish loading: java.lang.ClassNotFoundException: Could not find the class CLASSNAME” (1.1x) or “Uncaught exception: Class "CLASSNAME" not found.” (1.0x)	The class file that contains the class CLASSNAME must be included with an additional ‘-f’ switch. Remember that TINIConvertor will not find items in your classpath. You must somehow specify every file that TINIConvertor needs to build into your application.
TINIConvertor starts, but reports “Serious Error! Method does not exist in this class!!!: add:(Ljava/lang/Object;)Z in java/util/Vector from CLASSNAME” (1.1x) or “getMethodRef() failed for add(Ljava/lang/Object;)Z” (1.0x).	This is usually caused when an application uses a function that is not part of the 1.1.8 API, but was added to Java in later API’s. The most common occurrence of this is the ‘add’ method in the Vector class, which was not added until JDK 1.2. The application needs to be recompiled against the class files from TINI (using the ‘-bootclasspath tiniclasses.jar’ switch to javac). The unimplemented functions will need to be altered to something that TINI supports.
TINIConvertor starts, but reports “java.lang.NoSuchMethodError at com.dalsemi.system.classloader.reflection.DatabaseResolver.<init>”	This usually occurs when you have ‘tiniclasses.jar’ in your [JDK[/jre/lib/ext directory. This file should never show up in your jre/lib/ext, because then Java will try to run using it. These classes are meant for TINI, and will cause odd errors like this one when Java tries to use ‘tiniclasses.jar.’
TINIConvertor starts, but reports “Could not find a class with a 'main' entry point.” (1.1x) or “No Application with main() found.” (1.0x)	None of the classes included for conversion contain the method ‘static void main(String[])’, which is the application entry point. Note that the ‘main’ function cannot be ‘public void main(String[])’ (*not static*) nor can it be ‘public static void main()’ (*no String[] parameter*).
TINIConvertor starts, but reports “Could not find 'public static void main' anywhere, but I found a 'public void main'. Please check this.” (1.1x)	The application entry point needs to be ‘public static void main(String[])’. A non-static function is not acceptable as the entry point, since no objects have been created when the application begins running.
TINIConvertor completes, but when I run my application, either (1) the wrong application starts, or (2) no application starts and I am kicked back out to the command line.	There might be multiple ‘public static void main(String[])’ functions declared in your application. This is common when certain classes or modules have ‘main’ functions used for debug purposes, and are not removed for building the application. Use the ‘-m’ argument to specify which class contains the main function you would like to be the application entry point.

BuildDependency

Imagine that you have a large number of classes that are parts of programming modules. For instance, Dallas Semiconductor makes iButton® and 1-Wire devices, and provides a Java API for communicating with those devices. If you want to communicate with an iButton with family code 10, you need the class OneWireContainer10. However, OneWireContainer10 also depends on the TemperatureContainer class, plus a few others. We have seen that you can specify all of these files individually to TINIConvertor with different '-f' switches, and that you can also build an entire directory or archive into your application. Neither is a good idea in this case—there are too many classes to use multiple '-f' switches and stay sane, and building the entire Java 1-Wire API will make your application far too large.

The solution is BuildDependency. Using a set of dependency rules, BuildDependency figures out which classes are required for a specified module and formats the input to TINIConvertor to build in the right classes. Therefore, BuildDependency is a wrapper for TINIConvertor. It can do everything TINIConvertor can, plus do a little figuring of things out as well.

BuildDependency is never absolutely necessary to use, but it makes life easier. You won't find too many people willing to build 1-Wire applications with just TINIConvertor because figuring out what the dependencies of a class file are can be a frustrating process.

How to Use BuildDependency

BuildDependency lets any option it does not recognize pass through to TINIConvertor. You will still need to specify the input files (-f), output file name (-o), and TINI database file (-d). In fact, you could take any valid TINIConvertor command and just replace 'TINIConvertor' with 'BuildDependency' and get the same result.

In order to build modules into your application, there are a few other things you need to tell BuildDependency:

- Which modules you want to add to your application
- Where the list of dependency rules is located
- Where the class files that are part of the module are located (a directory or an archive name)

As an example, consider the application in MyApp.class that uses iButtons with family codes 04 and 21. The BuildDependency command line might look like:

```
java –classpath c:\tini1.12\bin\tini.jar BuildDependency
  -f MyApp.class
  -o MyApp.tini
  -d c:\tini1.12\bin\tini.db -add OneWireContainer04; OneWireContainer21
  -x c:\tini1.12\bin\owapi_dep.txt
  -p c:\tini1.12\bin\owapi_dependencies_TINI.jar
```

This tells BuildDependency to build the classes required for using iButtons with family codes 04 and 21 (by 'adding' dependencies for OneWireContainer04 and OneWireContainer21). The rules for the dependencies 'OneWireContainer04' and 'OneWireContainer21' are located in the file "c:\tini1.12\bin\owapi_dep.txt." Once BuildDependency figures out which classes need to be built into the application, it searches for this classes in the file "c:\tini1.12\bin\owapi_dependencies_TINI.jar."

Multiple entries can be input for dependencies, dependency rule files, and archives by separating the entries with a semi-colon or comma. Alternately, multiple entries can also be added with multiple '-add', '-x', and '-p' switches.

BuildDependency will search for class files specified by dependency definitions first in the list of locations given by the '-p' switch, and then in the current directory. For instance, if our command line looks like:

```
java –classpath [...] BuildDependency [...]
  -p c:\somedirectory;c:\archives\jarfile.jar
```

iButton is a registered trademark of Dallas Semiconductor Corp.

Suppose BuildDependency decides it needs to find the class `com.dalsemi.onewire.container.TempContainer.` The search then looks for:

1) The file c:/somedirectory/com/dalsemi/onewire/container/TempContainer.class
2) A file com/dalsemi/onewire/container/TempContainer.class stored in the archive file c:\archives\jarfile.jar
3) The file ./com/dalsemi/onewire/container/TempContainer.class

The Dependency File

The dependency file contains the definitions for classes that should be built for specified modules. A default dependency file is included in the TINI SDK in the file [TINI]/bin/owapi_dep.txt. However, there is nothing to stop you from altering that dependency file, writing a replacement, or writing your own new dependency file.

The basic format for defining a dependency group is:

```
GROUPNAME=DEP_CLASS_1;DEP_CLASS_2;DEP_CLASS_3;DEP_CLASS_4; ...
```

So to define 'OneWireContainer04,' the full line would be (note that the entry is listed on one line, but displayed here spread over several lines, see the default dependency file in the TINI SDK for examples):

```
OneWireContainer04=com.dalsemi.onewire.container.OneWireContainer04;com.dalsemi.onew
ire.utils.Bit;com.dalsemi.onewire.container.ClockContainer;com.dalsemi.onewire.con-
tainer.MemoryBank;com.dalsemi.onewire.container.MemoryBankNV;com.dalsemi.onewire.con
tainer.MemoryBankScratch;com.dalsemi.onewire.container.OneWireSensor;com.dalsemi.one
wire.container.PagedMemoryBank;com.dalsemi.onewire.container.ScratchPad
```

Now you see why you would not want to specify all those files to TINIConvertor. Note how long that dependency listing is. To shorten it, you can also use groups like properties.

```
CONT_ROOT=com.dalsemi.onewire.container

UTILS_ROOT=com.dalsemi.onewire.utils

OneWireContainer04=%CONT_ROOT%.OneWireContainer04;%UTILS_ROOT%.Bit;%CONT_ROOT%.Clock
Container;%CONT_ROOT%.MemoryBank;%CONT_ROOT%.MemoryBankNV;%CONT_ROOT%.MemoryBankScra
tch;%CONT_ROOT%.OneWireSensor;%CONT_ROOT%.PagedMemoryBank;%CONT_ROOT%.ScratchPad
```

These can also be nested. Say we want to create a group that will build in containers for all the parts that implement the ClockContainer interface (family codes 04, 21, and 26):

```
Clocks=%OneWireContainer04%;%OneWireContainer21%;%OneWireContainer26%
```

Circular references in the dependency definition are protected against.

White space between tokens in the dependency file is ignored. Here we use that to help readability of the dependency file:

```
ADAPTER_ROOT = com.dalsemi.onewire.adapter

CONT_ROOT = com.dalsemi.onewire.container

UTILS_ROOT = com.dalsemi.onewire.utils
```

It is possible that part of a module might be a native library. In that case, the native library should be wrapped with parenthesis:

```
I2C com.dalsemi.system.I2CPort;(mod_i2c.tlib);
```

BuildDependency and TINI Firmware 1.1x

With the introduction in TINI 1.1x of advanced features such as IPv6, dynamic classloading, reflection, and serialization, several modules had to be removed from the firmware and turned into modules. These include:

- PPP
- I^2C
- CAN
- URL protocol implementers

To use any of these modules with TINI firmware 1.1x, you should use BuildDependency and '-add' the dependency for the required module. This usually is only a minor inconvenience for applications using PPP, I^2C, and CAN because the applications will fail to build. When using the URL classes, the problem is trickier because the URL classes find the protocol implementation classes dynamically, so the problem only shows up at runtime. The symptom is an unexplainable `java.net.MalformedURLException`. The solution is to '-add' the proper URL classes that your application requires:

- Add the HTTP dependency for applications using URLs that start with 'http:'.
- Add the FILE dependency for applications using URLs that start with 'file:'.
- Add the FTP dependency for applications using URLs that start with 'ftp:'.
- Add the MAILTO dependency for applications using URLs that start with 'mailto:'.

BuildDependency's Command Line Arguments

Remember that any option that BuildDependency does not recognize is passed unaltered to TINIConvertor. Here are the options that BuildDependency will process:

- **-add NAMES:** Specifies the name(s) of the dependencies to add to this project. This is a semi-colon or comma separated list of dependency names.
- **-p PATH:** Specifies the path(s) to your dependency classes. This is a semi-colon or comma separated list that can include multiple jar files and directories.
- **-x DEP_FILE:** Specifies a semi-colon or comma separated list of filenames of dependency text dependency files. Note that multiple files can be specified, although if a key is redefined, only the last definition found will be used by BuildDependency. BuildDependency has a default set of dependencies used for 1-Wire programs.
- **-debug:** BuildDependency normally suppresses the output of TINIConvertor. Use this tag to see the entire output of TINIConvertor if you are troubleshooting a problem.
- **-dep:** Prints out the entire dependency list using the specified dependency file or the default. This option is for information only. TINIConvertor will not be invoked if you use this option.
- **-depNAME:** Prints out the dependency list for dependency named NAME. This option is for information only. TINIConvertor will not be invoked if you use this option.

Common Problems Reported when Using BuildDependency

Note that since BuildDependency is just a wrapper for TINIConvertor, the same common problems that apply to TINIConvertor also apply to BuildDependency. Problems rarely arise from BuildDependency that are not actually TINIConvertor problems.

Macro

The 'macro' tool is a macro preprocessor that takes an assembly language source file and converts equates to numeric values and macros to inline code. The macro preprocessor is single pass. This means that equates and macros must be declared before they are used. The output of the preprocessor is a file with an .mpp extension. Macro is case insensitive.

Equates are of the form:

```
<Identifier> EQU <expression>
```

where expression can be a value (e.g., 0a0h, 160, or 10100000b) or a compound expression such as "EQUATE_2 EQU (EQUATE_1 + 1)", assuming EQUATE_1 has already been defined. Compound expressions are NOT allowed in macros.

Macros are of the form:

```
<Identifier> MACRO [PARAM <param1> [<param2> [...<paramn>]]]
[LOCAL <local1>[, <local2>[...,<localm>]]]
[statement 1]
[statement 2]
...
[statement s]
ENDM
```

Macros may be nested but inner macros may NOT have parameters.

A statment that uses a macro is of the form:

```
<Identifier>[,<parameter1>[,<parameter2>[...,<parametern>]]]
```

Macro also supports conditional assembly. Conditional statements are of the form:

```
IF (<boolean statement>)
[conditional statement 1]
;...
[conditional statement c]
ENDIF
IF (<boolean statement>)
[include statement 1]
;...
[include statement i]
ENDIF
```

Only code and include statements are supported within conditional blocks, which means no macros or equates can be directly defined within a conditional block. Equates and macros that need to be conditional must be defined in separate include files and conditional include statements must be used to select the equates and macros for a particular build.

The boolean operations supported by macro are =, !=, <, <=, >, and >=. Following are some examples of conditional assembly:

```
DEBUG equ 1
;...
IF (DEBUG != 0)
;...
   mov a, #'D'
   lcall Info_Send1152
;...
ENDIF
DEBUG_LEVEL equ 5
;...
IF (DEBUG_LEVEL > 1)
   lcall dump_extra_spew
ENDIF
```

```
IF (DEBUG_LEVEL > 2)
   lcall dump_verbose_spew
ENDIF
DS400 EQU 0
;...
IF (DS400 != 0)
$include(ds400.inc)
ENDIF
IF (DS400 == 0)
$include(ds390.inc)
ENDIF
```

Using Macro

Macro is typically used before the assembler to create an 'mpp' file. The command line for this looks like:

```
macro -Ic:\tini\tini1.12\native\lib\ somefile.a51
```

The output of this process, if there are no failures reported, is the file 'somefile.mpp.' The '-I' switch specifies an alternate include path. Only one alternate directory can be specified and the alternate directory is searched after the current directory.

Macro's Command Line Parameters

The general usage for 'macro' is:

```
macro [options] inputfilename
```

Supported options for macro are:

- **-q:** Enables quiet mode. In this mode macro does not produce any output to the console.
- **-p:** Pseudo-PIC compatibility mode. Not to be used for TINI applications.
- **-c:** Reformats immediate values (such as 055h) to C format (0x55). This option should not be used with TINI.
- **-d:** Tells macro to forget about built-in 8051 symbols such as ACC. This option should not be used with TINI.
- **-e:** Strips the END statement from the resulting 'mpp' file. The assembler can only work on one assembly file to produce one application. By stripping end statements, multiple mpp files can be concatenated and assembled at once. Note that the last file to be concatenated should not have the end statement stripped.
- **-Ipath:** Specifies an alternate include path to be searched after the current directory.

a390

The TINI SDK includes a390, an 8051 assembler that generates correct code for the 24-bit addressing mode of the DS80C390 and DS80C400. a390 takes an .mpp file (or assembly source if no equates or macros are used) and produces a tlib (relocatable tini library) file, or optionally a non-relocatable tbin or extended hex file.

The assembler is multi-pass. The assembler does not open "include" files. Include files are processed by "macro" and may only contain equates, macros, and DB statements. All relevant information, including DB statements, is placed into the .mpp file. a390 is case insensitive.

The TINIOS and a390 support version control in native tlib files. This allows a vendor of a TINI-based product to support multiple firmware revisions without fear of a library executing random code when installed on the wrong hardware/firmware. The vendor can simply place multiple versions of the tlib file on an FTP site and allow remote downloads without fear of the wrong library being executed. Legacy (unversioned) tlib files are supported but cannot be verified at load time. Two switches have been added to support version control. To tie a library to a specific processor use the -p <390,400> switch. To tie a library to a specific firmware revision use the -f <firmware version

string> switch. The firmware version string must match exactly the string that is displayed at boot time (minus the "TINI OS " prefix). The -f switch cannot be used without the -p switch.

TINIm390/TINIOS 1.02e EXAMPLE:

```
;
; for use with OS displaying banner "TINI OS 1.02e"
;
a390 -p 390 -f 1.02e example1.mpp
```

TINIm390/TINIOS 1.10 EXAMPLE:

```
;
; for use with OS displaying banner "TINI OS 1.10"
;
a390 -p 390 -f 1.10 example1.mpp
```

TINIm400/TINIOS 1.10 EXAMPLE:

```
;
; for use with OS displaying banner "TINI OS 1.10"
;
a390 -p 400 -f 1.10 example1.mpp
```

The tlib file is modified by a390 (when the -p or -f switches are used) with a prefix that is similiar to:

```
sjmp Example1_Init
db "TINI400"
db 0
db "1.10" ; this is optional with the -f switch
db 0
Example1_Init: ; your init routine starts here
```

If the hardware/firmware string compare is unsuccessful, a `java.lang.UnsatisfiedLinkError` will be thrown when trying to load the library.

Assembly Applications on the DS80C400

Assembly code is not only useful for writing native libraries for the TINI Java Runtime, it can also be used to write applications that use the DS80C400 ROM. The network stack, memory manager, process scheduler, and all the other features of the ROM have entry points that can be used from assembly code.

The *High-Speed Microcontroller User's Guide: DS80C400 Supplement* (www.maxim-ic.com/DS80C400UG) has a lengthy section on the assembly interface to the ROM functions. The Ethernet speaker application (*Application Note 609: Internet Speaker with the DS80C400 Silicon Software*, www.maxim-ic.com/AN1858) was entirely implemented in 8051 assembly language, and built using the a390 assembler.

Instruction Set for the DS80C390 and DS80C400

The DS80C390 and DS80C400 support the standard 8051 instruction set. However, when in 24-bit contiguous addressing mode (default when an application starts because the ROM boot loader enters this mode), instructions such as call's and 'mov dptr' instructions require an extra byte of instruction beyond that required by traditional 16-bit 8051. The assembly language, however, is the same.

Use the *High-Speed Microcontroller User's Guide* and the user's guide supplements for the DS80C390 and DS80C400 for documentation on the 8051 instruction set and special function registers of those microcontrollers, all available on our website at www.maxim-ic.com/user_guides.

- *High-Speed Microcontroller User's Guide*
- *High-Speed Microcontroller User's Guide: DS80C400 Supplement*
- *High-Speed Microcontroller User's Guide: DS80C390 Supplement*

Assembling Files with a390

To simply assemble your source file (likely an mpp file, see the description of the macro tool), use this command line:

```
a390 samplefile.mpp
```

To generate a list file when assembling, use:

```
a390 -l samplefile.mpp
```

To tie the generated TLIB (TINI native library) to a specific version of the hardware, use the '-p' tag:

```
a390 -p 400 -l samplefile.mpp
```

To tie the generated TLIB to a specific version of the firmware, use the '-f' tag:

```
a390 -p 400 -f 1.12 -l samplefile.mpp
```

To generate a HEX file instead of a TLIB (if you were building an application in assembly, for instance):

```
a390 -l -Fhex samplefile.mpp
```

Command Line Arguments to a390

The usage for a390 is 'a390 [options] <inputfilename>'. Options are:

- **-l:** Specifies that a390 should generate a list file when assembling.
- **-s:** Prints statistical information to the list file for this assembly. Lists how many bytes are produced for each module, in addition to information on the frequency of a few instructions.
- **-d:** Allows direct data space declarations (such as "mydirect ds 1") and org statements. Default is to not allow either for TINI native library development, since native libraries are relocatable, and direct declarations are likely to conflict with the TINI OS's direct RAM usage. If you do any assembly development, you will likely need to use this option.
- **-Ffmt:** Specifies the output format of the file to a390. Supported output file formats are 'hex', 'tlib', and 'tbin'. A 'tlib' file is the default output. Specify a different format with '-Fhex' or '-Ftbin'.
- **-p processor:** Includes a signature in the TLIB output file that ties the library to a specific version of the processor. For example, '-p 390' ties the library to the DS80C390. The default is to not include any signature.
- **-f firmware:** Includes a signature in the TLIB output file that ties the library to a specific version of the TINI firmware. For example, '-f 1.12' ties the library to the TINI firmware version 1.12. The default is to not include any signature.

MTK

The MTK (Microcontroller Tool Kit) is used to interact with the boot loader of several different microcontrollers, including the DS80C390 and DS80C400. It looks a lot like JavaKit and performs the same functions. The big difference is that MTK is not written in Java, and does not need the JDK to be installed to run.

Benefits Over JavaKit

If you are already using JavaKit, you may not see any benefits in moving to MTK. In fact, the basic functionality is the same.

The benefit is really for those who are just starting out. Using the MTK instead of JavaKit means:

- No need to install the Java Communications API, which is one of the most troublesome parts of getting started. Problems installing the Communications API make up the largest percentage of technical support inquiries for TINI.
- No classpath issues. The MTK is an executable.
- Java is not needed at all for the MTK. This can be useful for developers working in assembly language or in C and do not want to have to install the Java Development Kit.

Currently, the MTK is only supported on Windows platforms. However, it was written with on a cross-platform windowing library. We plan to provide MTK releases for Linux and Mac OS in the future.

Installation of MTK

The MTK is installed by an InstallShield application. Let the installer run and place everything in the default directories and process should be done in a few seconds. In the Start Menu, there should be an entry for 'Dallas Semiconductor MTK,' under which you have a link to the MTK program.

Using MTK

Start the MTK program, and you will see a window asking you to select a device. Select the 'TINI' entry, and the MTK window will appear. There are a couple configuration options we need to setup before we start talking to our TINI. First we need to configure the serial port by selecting OPTIONS->Configure Serial Port. Select your serial port's name from the drop-down box and make sure the baud rate is set to 115200. If your serial port's name is not in the list, you can type it into the port name space.

Next, we need to set some TINI-specific options. We used command line parameters in JavaKit for this, but here we can go to TINI->TINI Options. Here we can set the addresses for the flash, the reset wait time, and set a few other options. There are also buttons to set the configuration options to the defaults for the DSTINI1 (DS80C390-based TINI) or the DSTINIm400 (DS80C400-based TINI). Generally, you can just use these buttons for configuration unless you have developed your own custom board. Click OK once you have selected your target board's options.

At this point we can open the serial port and start talking. Select TINI->Open COM1 at 115,200 (text will differ depending on your port and baud rate selections) and the MTK text area should become white. You will notice that there is no reset button as we have in JavaKit. The menu item TINI->Reset (or pressing Ctrl-R) serves this purpose in MTK.

At this point, MTK is pretty much the same as JavaKit. Under the File menu, you have options to load and verify files. You can interact with the boot loader by typing into the text area. There is also an online help file included with the MTK that details all of the menu commands and gives some basic information on connecting to microcontroller boot loaders.

Common Problems During TINI Development

A few common problems with JavaKit and TINIConvertor are duplicated here. If you don't find the issue you are looking for here, see the previous sections for a more detailed review of common problems with those tools.

SYMPTOM	POSSIBLE SOLUTION
My 1.1x firmware application throws a MalformedURLException.	The URL protocol implementation classes are modules in the 1.1x firmware and must be built into your application. Use the BuildDependency tool and the '-add' tag to add the URL implementation classes.
I can't reset my TINIm400 board. JavaKit says "No response from TINI".	Start JavaKit with the command line "java JavaKit –flash 40 –400."
TINIConvertor says "Minor version not 3".	Recompile all classes with the '-target 1.1' switch.
I can't load to the upper half of my 1M flash.	JavaKit assumes a flash size of 512k, and must use the 'Z' command to zap segments of flash before programming. Use the '-ROMSize' switch to tell it your flash is larger.
I tried turning on the clock quadrupler on my DSTINIm400 but everything stopped. I thought the DS80C400 could run up to 75MHz?	It is not the processor that is limiting your speed. The flash on the DSTINIm400 is not fast enough to support running at crystal x 4. The typical solution is to store the TINI firmware in the flash and copy it to high-speed RAM on bootup before starting the clock quadrupler.
JavaKit won't start.	Make sure that: • comm.jar is in your classpath • You are executing the virtual machine that you think you are, and not the one in c:\windows. Try specifying the full command line for "java.exe".
JavaKit doesn't list any ports.	Make sure that "javax.comm.properties" is in your [JDK]/jre/lib directory.
What does "Could not execute file: Insufficient heap" mean and what can I do about it?	This error message occurs when an attempt to run a .tini file fails with an OutOfMemoryError. This can occur for two reasons: 1. Insufficient memory to run the application 2. Insufficient contiguous memory to run the application Possible solutions include: 1. Delete any files that are not required for running of application. 2. Make sure to kill a task before overwriting the corresponding task binary (.tini file). 3. Reduce usage of files in filesystem. Use RandomAccessFile on existing files instead of creating new files. 4. Run the slush command gc to free up memory and coalesce adjacent free blocks.
When I first start up my TINI it says "Testside Application running" and there is no prompt.	This is a test application that is loaded on the TINI by Dallas Semiconductor before the boards are shipped out. Load the TINI firmware and the slush application.

Maxim cannot assume responsibility for use of any circuitry other than circuitry entirely embodied in a Maxim product. No circuit patent licenses are implied. Maxim reserves the right to change the circuitry and specifications without notice at any time.

© 2004 Maxim Integrated Products Printed USA MAXIM is a registered trademark of Maxim Integrated Products.

DALLAS SEMICONDUCTOR is a registered trademark of Dallas Semiconductor Corporation.