Programming in C for the DS80C400

Since the introduction of the TINI® Runtime Environment for the DS80C390, developers have clamored for a way to use the power of TINI without using the Java™ language. Unfortunately, the network stack and other features of TINI were too intertwined with the Java virtual machine and runtime environment to be used from a C or assembly program. Later, when the ROM for the DS80C400 networked microcontroller was designed, a suite of functionality was exposed that could be accessed from programs written in 8051 assembly, C, or Java. Size constraints limited the functionality in the ROM to a subset of the functionality in the TINI Runtime Environment. The ROM would therefore be a useful starting block for building C and assembly programs because it offers 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 like an HTTP server that interacts with a file system.

This article starts with a C implementation of Hello World and moves on to a simple HTTP server. It describes how to set up the tools to write a simple tutorial program, and then demonstrates how to make use of the DS80C400's ROM functionality. All development was done using the TINIm400 verification module and Keil µVision2™ version 2.37, which includes the C compiler "C51" version 7.05.

Getting started with Keil's µVision2

You can build a simple Hello World-style program written in C using the Keil µVision2 development suite. Follow these instructions to complete your first C application for the DS80C400.

- Select **Project->Create New Project**. Enter the name of the project.
- The *Select Device for Target* dialog will pop up. Under *Data base*, select **Dallas Semiconductor** and the **DS80C400**. Select *Use Extended Linker*, and then select *Use Extended Assembler*. Hit **OK** to continue. **Figure 1** shows the proper configuration for this dialog.
- The dialog will ask, *Copy Dallas 80C390 Startup Code to Project Folder* and *Add File to Project?* Select **No**. We will supply our own startup code.
- When the project window opens on the left, open *Target 1*. Right click on *Source Group 1*, and select *Add files to group 'Source Group 1.'* In the file dialog that pops up, change *files of type* to *Asm Source file*. Add the file **startup400.a51**. This file can be found in the zip file at www.maxim-ic.com/HelloWorld.

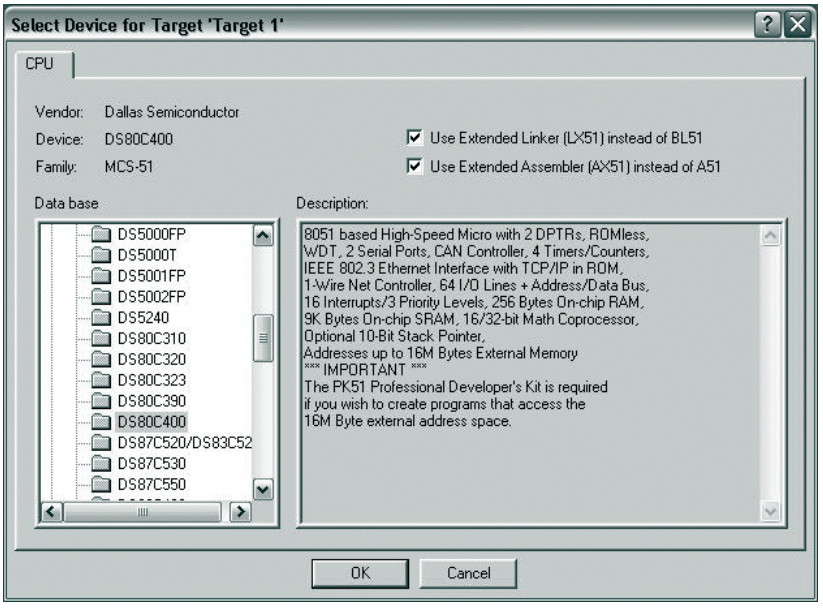

Figure 1. *Select the DS80C400 for a new Keil µVision2 project.*

The DS80C400's ROM is a useful starting block for building C and assembly programs because it offers a proven network stack, process scheduler, and memory manager.

- It is essential that the application is built for address 400000h, which corresponds to the beginning of the flash on the TINIm400. Open the file **startup400.a51** by double clicking on it. Find the segment declaration for `?C_CPURESET?0`. Make sure that this code segment is declared at `400000h`:

```
?C_CPURESET?0
SEGMENT CODE AT 400000h
```

- Additionally, there should be a "`DB'TINI'`" line followed by another single DB, with the comment "Target bank." This declaration is part of a tag that tells the DS80C400 ROM to execute the code starting at address 400000h. This ensures that the application is built for address 400000h, which should correspond to the beginning of the flash on the TINIm400. Make sure that line reads:

```
            DB    40h         ; Target bank
```

- Create a new file. Save it as "main.c." Write the following in that file:

```
#include <stdio.h>

void main()
{
     printf("Test 400 Program\r\n");
     while (1) { }
}
```

- Save the contents of this file. Right click on *Source Group 1* and add the source file **main.c**. The source file should now be added to the project.

- Right click on *Target 1* on the left. Select *Options for target 'Target 1'* to view an option dialog. The first tab selected should be *Target*. Change *Memory Model* to **Large: variables in XDATA**. Change *Code Rom Size* to **Contiguous Mode: 16MB program**. Select the check boxes for *Use multiple DPTR registers* and *far memory type support*. Under *Off-chip Code memory*, add the first entry with a *Start* of **0x400000** and *Size* of **0x80000**. For *Off-chip XData memory*, add an entry with a *Start* of **0x10000** and a *Size* of **0x4000**. **Figure 2** shows this dialog after it has been configured. Note that the last '0' in **0x400000** is not displayed in the window.

  These settings are based on the memory configuration of the TINIm400 reference module, which includes 512k of RAM at address 0 and 1M of flash at address 400000h. The starting addresses and sizes in the Keil configuration should be changed for custom DS80C400 designs.

- Select the *Output* tab. Click on *Create HEX File* and select *HEX-386* in the drop-down box.

- Press F7 to build the application. If every task was done correctly, the application should build with no errors or warnings. A hex file should have been generated. You can now load the application onto your board.

Loading the sample application onto the TINIm400 module

This section describes how to load the hex file produced by the Keil compiler onto the TINIm400 verification module by using the tool **JavaKit**.

To use **JavaKit**, you must have the Java Runtime Environment (at least version 1.2) and the Java Communications API installed. The Java Runtime Environment can be downloaded at http://java.sun.com/j2se/downloads.html, and the Java Communications API can be found at http://java.sun.com/products/javacomm/index.html. The **JavaKit** tool is included with the TINI Software Development Kit, available at www.maxim-ic.com/TINIdevkit. Instructions for running **JavaKit** can be found in the file **Running_JavaKit.txt** in the *docs* directory of the TINI Software Development Kit. If you encounter technical issues when running **JavaKit**, it is possible someone already had a similar problem, which is chronicled in the archives of the TINI Interest List. You can search the archives for this list at www.maxim-ic.com/TINI/lists.

Use this command line to have the **JavaKit** talk to the TINIm400 module.

```
java JavaKit -400 -flash 40
```

Once **JavaKit** is running, select the serial port you will use to communicate with the TINIm400. Open the serial port using the **Open Port** button. Then press the **Reset** button. The loader prompt for the DS80C400 should print and look like this:

```
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

>
```

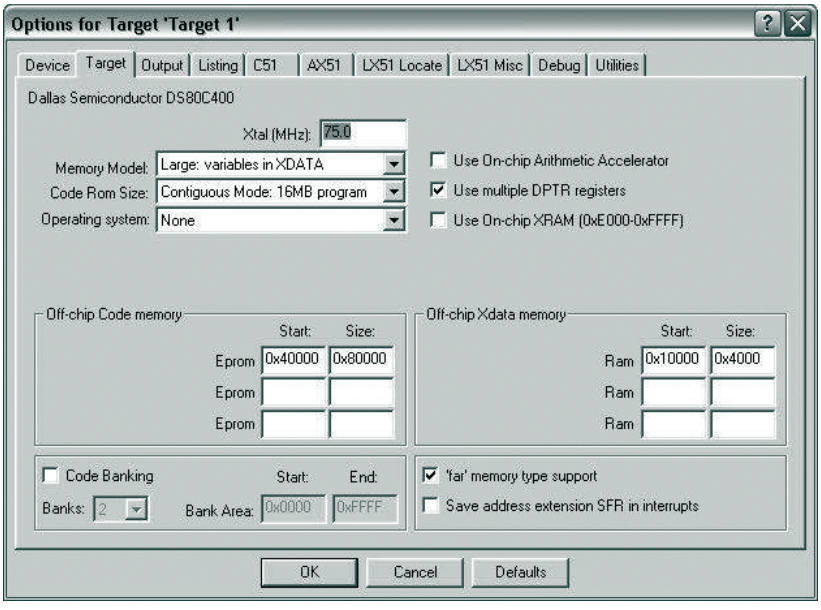

Figure 2. *The Target Options dialog is used to enter configuration information for the target platform. The configuration shown is suitable for use with the TINIm400 module.*

From the *File* menu at the top of **JavaKit**, select *Load HEX File as TBIN*. Find the *helloworld.hex* file that we just created, and select it. The *Load HEX File as TBIN* option converts the input hex file to a TBIN file, and then loads it. This operation is faster than loading it as a hex file because an ASCII hex file is more than twice as large as a binary file for the same data set.

There are two ways to execute your program once it is loaded. Since the program was loaded into bank 40, you can type:

```
> B40
> X
```

To select bank 40 and execute the code there, you can also type:

```
> E
```

This will make the ROM search for executable code, a special tag signifying that the current bank has executable code. This tag consists of the text "TINI" followed by the current bank number. It is located at address 0002 of the current bank. Our *Hello World* program declares this tag in the *startup400.a51* file with the following lines:

```
?C_STARTUP:    SJMP    STARTUP1
               DB      'TINI'    ; Tag for TINI Environment 1.02c
                                 ; or later (ignored in 1.02b)
               DB      40h       ; Target bank
```

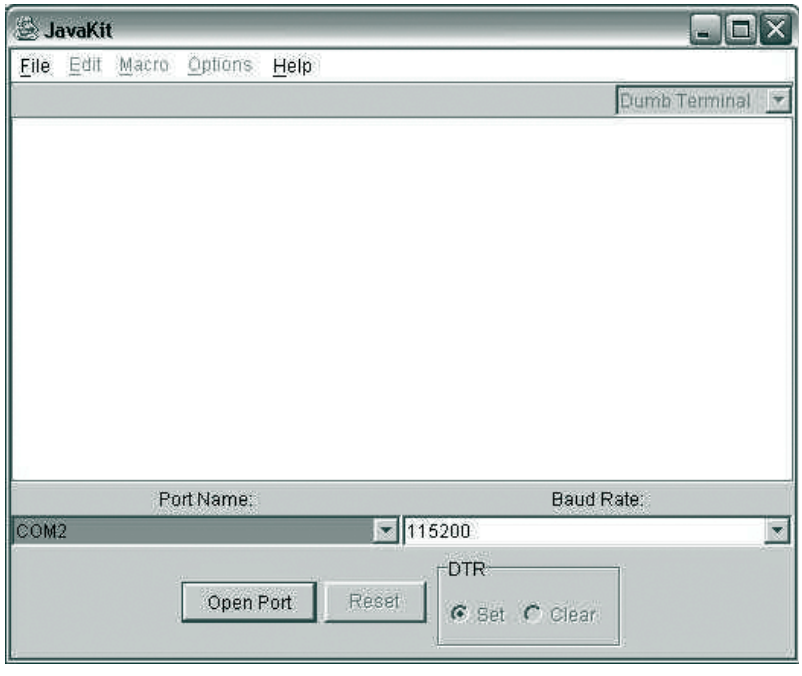

Figure 3. *The JavaKit program is used to load applications and communicate with the serial port of the DS80C400.*

In addition to the ROM libraries, other libraries (more are still being written) provide useful functionality not included in the ROM. Libraries have been developed for file system operations, DNS lookups, I²C communication, and 1-Wire communication.

Note that the `SJMP STARTUP1` statement is located at address 0000 of bank 40. It is followed by the executable tag { 'T', 'I', 'N', 'I', 40h }, located at address 0002, since the `sjmp` statement is two bytes.

When you type "E," the ROM searches downward through the memory banks for executable code. If you type "E" and some other code executes, it means that the ROM has found an executable tag at an address higher than 400000h, where your code was loaded. You may need to find that tag and delete the contents of that bank. You can erase a flash bank by using the **Z** loader command:

```
> Z41
You sure? Y
```

To erase all banks of flash, you need to zap from bank 40h to bank 4Fh.

Interfacing to the ROM and the ROM libraries

Calling the ROM functions from C is complicated. (The procedure for calling ROM functions is described in the *High-Speed Microcontroller User's Guide: DS80C400 Supplement*.[1]) Parameters must be converted from the Keil C Compiler's conventions to the conventions used by the ROM. The Keil compiler passes parameters in a combination of XDATA locations and registers. The ROM functions accept parameters in different ways. For example, the socket functions accept parameters stored in a single parameter buffer, and many utility functions accept parameters passed in special function registers or direct memory locations. Dallas Semiconductor wrote libraries for accessing the ROM functions to translate from Keil calling conventions to the ROM's parameter conventions.

Using ROM functions in your C programs requires only importing the library and including a header file. To import a library in your project, right click on *Source Group 1* in your Keil project window and select *Add Files to Group 'Source Group 1.'* Change the file filter to '*.lib' and select the library you need to include. Then include the header file at the top of your source. You can use any of the library functions. There are ROM libraries to support ROM initialization, DHCP client operations, process management, socket functions, TFTP client operations, and utility functions such as CRC and pseudo-random number generation.

Using the extension libraries

In addition to the ROM libraries, other libraries (more are still being written) provide useful functionality not included in the ROM. Libraries have been developed for file system operations, DNS lookups, I²C™ communication, and 1-Wire® communication.

The C Library project (including documentation, sample applications, and release notes) for the DS80C400 can be found at www.maxim-ic.com/ds80C400/libraries.

A simple HTTP server and SNTP client application

Dallas Semiconductor wrote a small application to demonstrate the functionality of these libraries, specifically the file system, sockets, process scheduler, and TFTP libraries. The sample application consists of an SNTP client and an HTTP server that responds only to 'GET' requests. It uses the core Dallas Semiconductor-provided libraries to call socket and scheduler functions. It also uses the file system to store a few web pages. The application consists of two processes: (1) the HTTP server is spawned as a new process that handles connections on port 80, and (2) the main process sits in a loop, attempting a time synchronization approximately every 60 seconds. The source code and project files for this application are available at www.maxim-ic.com/timeserver.

[1] Available online at www.maxim-ic.com/DS80C400UG.

Initializing the file system

Compared to the TINI Runtime Environment, applications written in C allow more space for user code and data.

Before the HTTP server can be started, the file system must be initialized. The demonstration program ensures that two static files, a home page (*index.html*) and the source to the program (*source.html*), are in the file system before the server starts.

The program initializes its file system by downloading the files it needs from a TFTP server. In our example, a TFTP server is running at a known IP address. The files *index.html* and *source.html* are requested from the TFTP server, then written to the file system.

SolarWinds provides a free TFTP server for Windows® platforms that was used in the development of this demonstration. From SolarWinds' website (www.solarwinds.net), follow the **Downloads—Free Software** menu to find the TFTP server download. After installing, use the **Configure** option under the **File** menu to configure the available files. Make sure to change the program to use your TFTP server's IP address (`TFTP_IP_MSB, TFTP_IP_2, TFTP_IP_3`, and `TFTP_IP_LSB`).

The simple HTTP server

The HTTP server in this application is implemented as a simple version of an HTTP server described by RFC 2068. In this version, only the 'GET' method is supported. Input headers are ignored, and few output headers are given.

The server socket is created by calling Berkley-style socket functions, which make the server socket easy to set up. The following code shows how our simple HTTP server creates, binds, and accepts new connections.

```
struct sockaddr local;
unsigned int socket_handle, new_socket_handle, temp;

socket_handle = socket(0, SOCKET_TYPE_STREAM, 0);
local.sin_port = 80;
bind(socket_handle, &local, sizeof(local));
listen(socket_handle, 5);

printf("Ready to accept HTTP connections...\r\n");

// here is the main loop of the HTTP server
while (1)
{
     new_socket_handle = accept(socket_handle,
&address, sizeof(address));
     handleRequest(new_socket_handle);
     closesocket(new_socket_handle);
}
```

Note that when a new socket is accepted, this simple application does not start a new thread or process to handle the request. Rather it handles the request in the same process. Any HTTP server of more-than-demonstration quality would handle the incoming request in a new thread, allowing multiple connections to occur and be handled simultaneously. After the request is handled, close the socket and wait for another incoming connection.

The `handleRequest` method consists of parsing the incoming request for a file name and verifying that the method is 'GET.' No other method (not even 'POST,' 'HEAD,' or 'OPTIONS') is allowed. Two file names are handled as a special case. When the file **time.html** is requested, the server dynamically generates a response consisting of the latest results from the timeserver, and the number of seconds that passed since the last instance the timeserver was queried. When the file **stats.html** is requested, statistics for server uptime and the number of requests are displayed.

If the file is not found or an invalid request method is given, an HTTP error code is reported.

A datagram socket is first created and given a timeout of about 2 seconds (0x800 = 2048ms). This ensures that if the communication fails with our chosen server, we will not wait indefinitely for a response.

The SNTP client

The second major portion of the timeserver application is a Simple Network Time Protocol (SNTP) client, as described in RFC 1361. This is a version of the Network Time Protocol (RFC 1305). SNTP requires UDP communication to request a time stamp from a server listening on port 123. Our timeserver uses the following code to periodically synchronize with the server time.nist.gov. Note that when this article was written, DNS lookup was not supported, so the IP address for the server is set manually. DNS has since been added to the C library website, and the following code can be updated to perform a lookup for the IP address.

```
    socket_handle = socket(0, SOCKET_TYPE_DATAGRAM, 0);

    // set a timeout of about 2 seconds.
    //'timeout' is unsigned long
    timeout = 2000;
    setsockopt(socket_handle, 0, SO_TIMEOUT, &timeout, 4);

    // assume 'buffer' has already been cleared out
    buffer[0] = 0x23;  // No warning/NTP Ver 4/Client

    address.sin_addr[12] = TIME_NIST_GOV_IP_MSB;
    address.sin_addr[13] = TIME_NIST_GOV_IP_2;
    address.sin_addr[14] = TIME_NIST_GOV_IP_3;
    address.sin_addr[15] = TIME_NIST_GOV_IP_LSB;
    address.sin_port = NTP_PORT;
    sendto(socket_handle, buffer, 48, 0, &address,
sizeof(struct sockaddr));
    recvfrom(socket_handle, buffer, 256, 0, &address,
sizeof(struct sockaddr));
    timeStamp = *(unsigned long*)(&buffer[40]);
      timeStamp = timeStamp - NTP_UNIX_TIME_OFFSET;
      // now we have time since Jan 1 1970
      formatTimeString(timeStamp, "London",
last_time_reading_1);
      last_reading_seconds = getTimeSeconds();
      closesocket(socket_handle);
```

A datagram socket is first created and given a timeout of about 2 seconds (0x800 = 2048ms). This ensures that if the communication fails with our chosen server, we will not wait indefinitely for a response.

The next line sets the options for the request. These bits are described in Section 3 of RFC 1361. The value 0x23 requests no warning in case of a leap second, requests that NTP version 4 be used, and states that the mode is 'Client.' After we send the request and receive the reply using the common datagram functions `sendto` and `recvfrom`, the seconds portion of the time stamp value is assigned to the variable `timeStamp`, and then adjusted to the reference epoch January 1, 1970. The function `formatTimeString` is used to convert the time stamp into a readable string, such as **"In London it is 15:37:37 on March 31, 2003."**

The function `getTimeSeconds` is used to determine the last time update, based on the DS80C400's internal clock. Since the program only updates about once every 60 seconds, the HTML page *time.html* uses this value to report the interval since the last time update. Finally, the socket is closed and the SNTP client goes to sleep for another 60 seconds.

Conclusion

The Keil C Compiler and libraries provided by Dallas Semiconductor allow applications written in C to access the power and functionality formerly only accessible through TINI's Java environment. Programs written in C can now access the network stack, memory manager, process scheduler, file system, and many other features of the DS80C400 networked microcontroller. Additionally, applications written in C allow more space for user code and data, compared to the TINI Runtime Environment. Developers using the C language for the DS80C400 can write lean applications with plenty of speed, power, and code space to tackle any problem.

Developers using the C language for the DS80C400 can write lean applications with plenty of speed, power, and code space to tackle any problem.