APPLICATION NOTE 613: Using the Keil C Compiler for the DS80C400

When the ROM for the DS80C400 microprocessor was designed, a suite of functionality was exposed that could be accessed from programs written in 8051 assembly, C, or Java™. The ROM of the DS80C400 is a useful starting block for building C and assembly programs, offering TINI®'s 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.

This application note describes how to get up and running using the Keil uVision2 suite of tools to build applications for the DS80C400 in C, and demonstrates how to make use of the DS80C400's ROM functionality by implementing a simple HTTP server. All development was done using the TINIm400 verification module and Keil uVision2 version 2.37, which includes the C compiler 'C51' version 7.05.

Also See: Using the SDCC Compiler for the DS80C400
Using the IAR Compiler for the DS80C400

Getting Started with Keil's uVision2

This section will help you build a simple HelloWorld-style program written in C using the Keil uVision2 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.

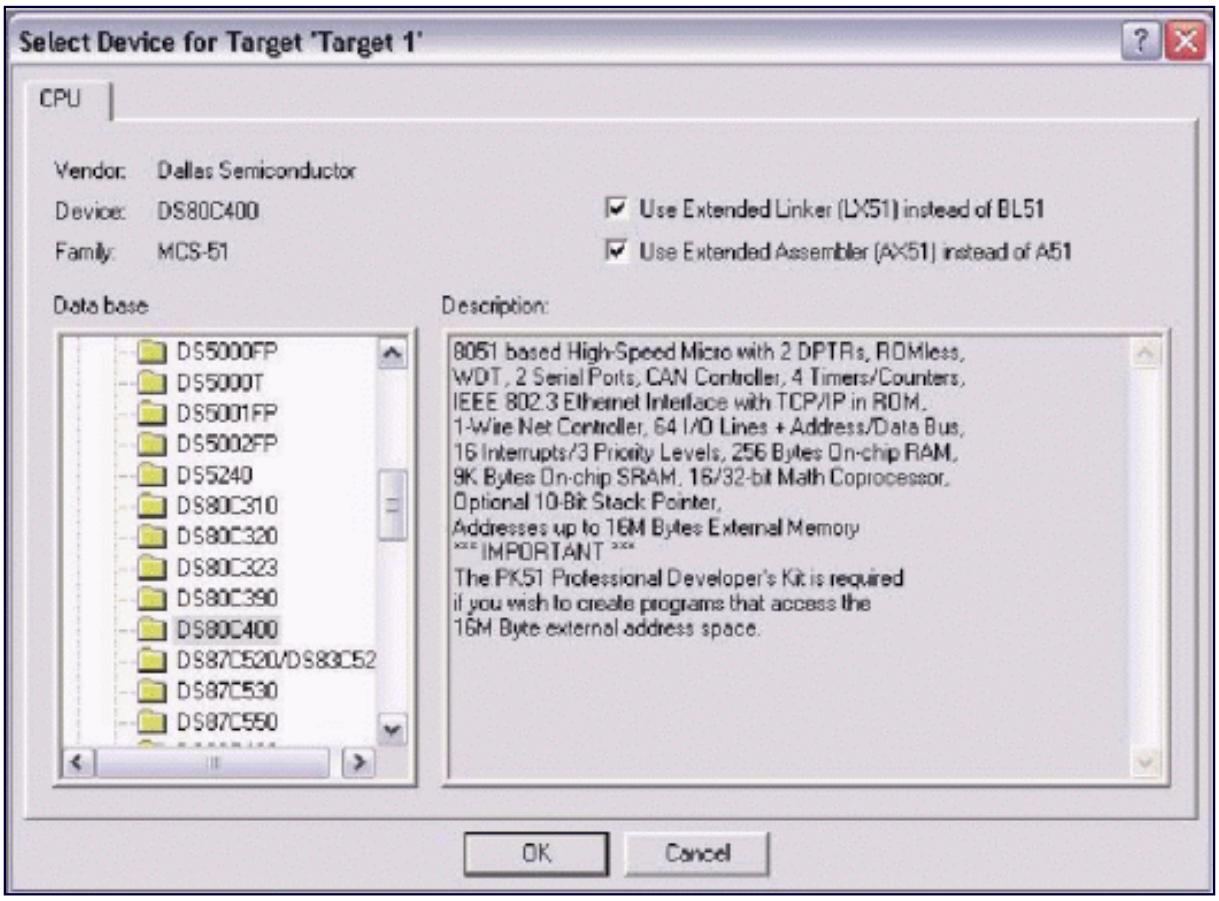

Figure 1. Selecting the device for a new Keil uVision2 project.

It 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 up *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 ftp://ftp.dalsemi.com/pub/tini/ds80c400/c_libraries/HelloWorld.zip.

Open the file **startup400.a51** by double clicking on it. Find the segment declaration for `?C_CPURESET?0`. Make sure this code segment is declared at `400000h`.

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

Also, there should be a "`DB 'TINI'`" line followed by another single DB, with a comment that says "Target bank". This makes sure 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

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

```
}
```

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

Right click on *Target 1* on the left, and select *Options for target 'Target 1'*. An option dialog will appear. 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 On-Chip Arithmetic Accelerator*, *Use multiple DPTR registers*, *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.

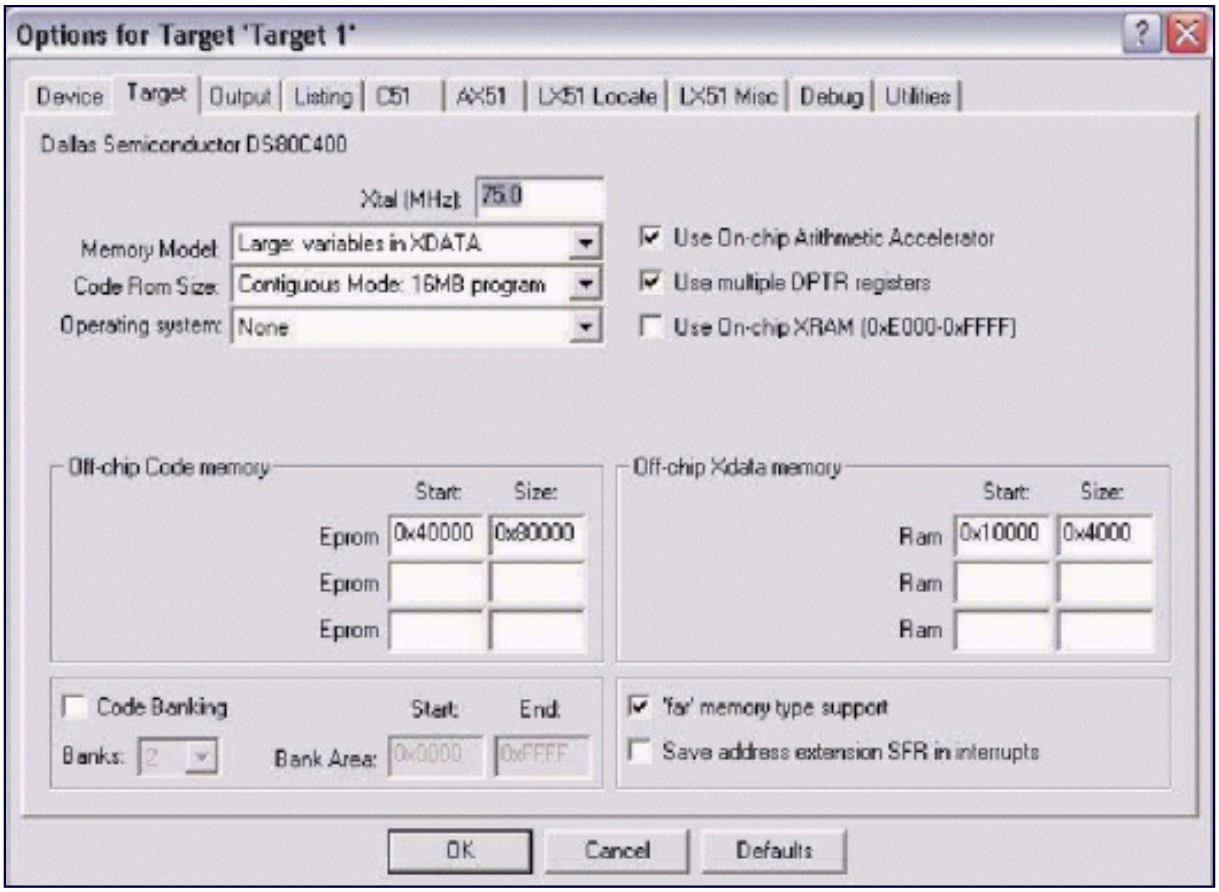

Figure 2. Options for target dialog from setp 7 (note that 'Eprom:Start' does not display the last '0' in 0x400000)

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 everything was done right, it should build with no errors or warnings, and a hex file should have been generated. You are now ready to load the application onto your board.

Loading the Sample Application onto the TINIm400 Module

This section describes loading the hex file produced by the Keil compiler onto the TINIm400 verification module using the tool **JavaKit**. To use **JavaKit**, you must have the Java Runtime Environment[2] (at least version 1.2) and the Java Communications API[3] installed. The **JavaKit** tool is included with the TINI Software Development Kit, available at ftp://ftp.dalsemi.com/pub/tini/tini1_11.tgz. 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 running **JavaKit**, chances are someone has already had a similar problem and it is chronicled in the archives of the TINI Interest List. You can search the archives for this list at http://lists.dalsemi.com/search/search.html.

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

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

Figure 3 shows the **JavaKit** window.

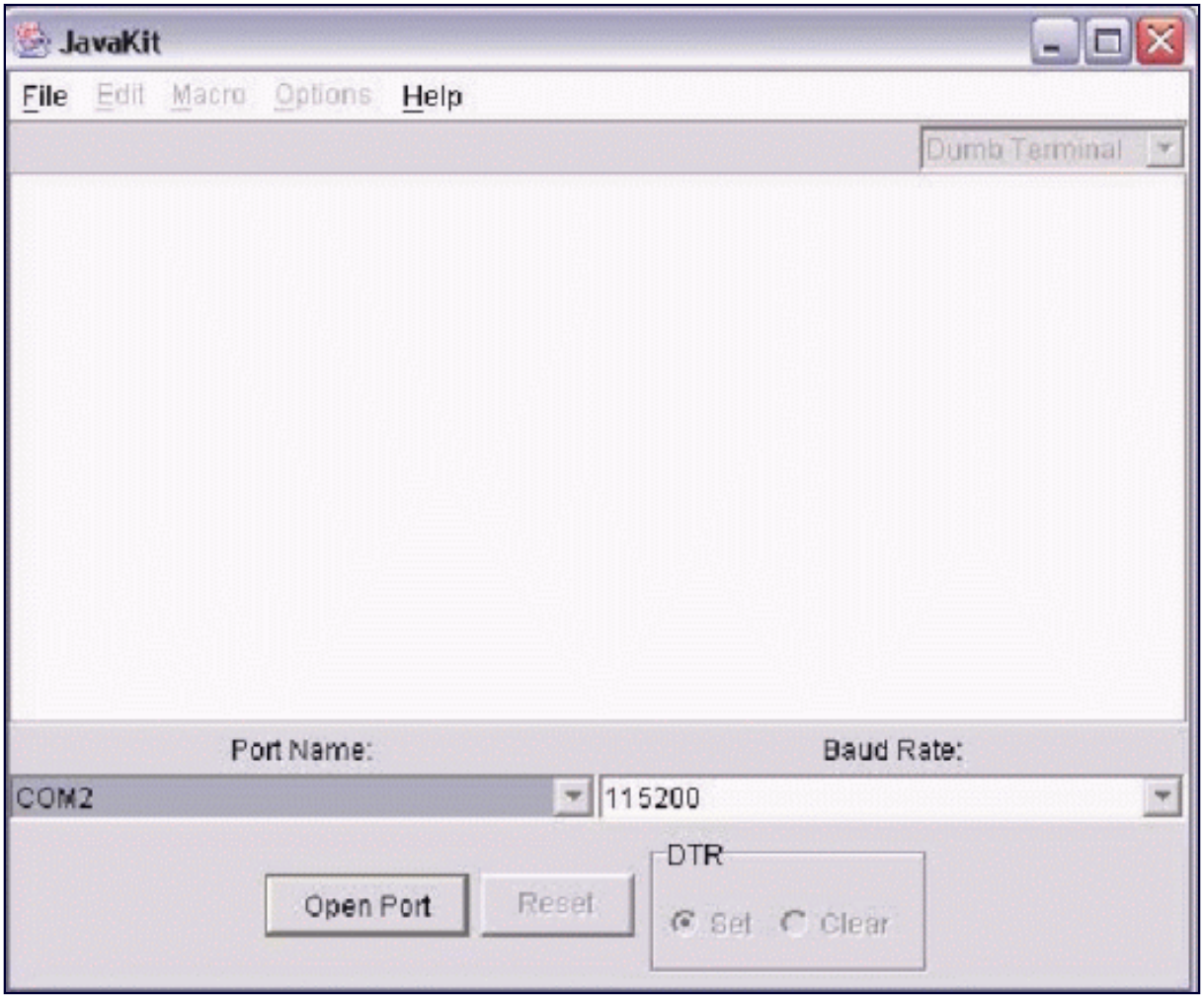

*Figure 3. **JavaKit** interface.*

Once **JavaKit** is running, select the serial port you will use to communicate with the TINIm400 and open the serial port using the 'Open Port' button. Then hit the 'Reset' button. The loader prompt for the DS80C400 should print, and should look something 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
>
```

From the 'File' menu at the top of **JavaKit**, select *Load HEX File as TBIN*. Search for the *helloworld.hex* file that we just created and select it. The *Load HEX File as TBIN* option converts the input hex file into a TBIN file, and then loads it. This 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 set of data.
There are two ways to execute your program once it is loaded. Since we loaded the program into bank 40, you can type:

```
> B40
> X
```

To select bank 40 and execute the code that is there. You can also type:

```
> E
```

This will make the ROM search for executable code. It searches for a special tag that signifies that the current bank has executable code. This tag consists of the text 'TINI' followed by the current bank number, and is located at address 0002 of the current bank. Our *HelloWorld* 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
```

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 starts from bank FEh and searches downward for executable code. If you type 'E' and some other code executes, it means that the ROM has found an executable tag at a higher address than 400000h, where your code was loaded. You may need to find where that tag is, and delete the contents of that bank.

Interfacing to the ROM and the ROM Libraries

The procedure for calling ROM functions is described in the High-Speed Microcontroller User's Guide supplement for the DS80C400[4]. However, calling these ROM functions from C is a little more complicated. 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 of the utility functions accept parameters passed in special function registers or direct memory locations. In order to translate from Keil 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. 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, and you can use any of the library functions. The ROM libraries include:
ROM Initialization Routines
DHCP Client
Process Scheduler
Sockets (TCP, UDP, Multicast)
TFTP Client
Utility functions (CRC16, random numbers)

Using the Extension Libraries

In addition to the ROM libraries, other libraries have been written (and are being written) to provide useful functionality that has not been included in the ROM. These libraries include:

- File System, adapted from the TINI file system and implementing methods declared in stdio.h
- DNS client implementation
- 1-Wire®, using the API defined in the Public Domain Kit at www.ibutton.com/software/1wire/wirekit.html
- I2C, implementing a design similar to the one used by TINI
- CAN, implementing a design similar to the one used by TINI

The C Library project (including documentation, sample applications, and release notes) for the DS80C400 can be found at ftp://ftp.dalsemi.com/pub/tini/ds80c400/c_libraries/index.html.

A Simple Application: HTTP Server and SNTP Client

A small application has been written to demonstrate some of 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 to only 'GET' requests. It uses the core Dallas Semiconductor provided libraries to call socket and scheduler functions, and it also uses the file system to store a few web pages. The application consists of two processes. The HTTP server is spawned off into a new process that handles connections on port 80, and the main process sits in a loop, attempting a time synchronization approximately every 60 seconds.

Initializing the File System

Before the HTTP server can be started, the file system must be initialized. There are 2 static files that this demonstration program attempts to make sure are in the file system before the server starts, a home page (*index.html*) and the source to the program (*source.*

html). These files could be installed in the file system in a number of ways. One possibility is to include the text of these files in the code data of the program, and then write the file data to the file system on startup. This is the simplest way, and our demonstration program has code space to spare.

This demonstration program initializes its file system is by finding the files it needs from a TFTP server. This is a little more interesting, and shows more of the DS80C400's built-in functionality. In our example, we have a TFTP server running at a known IP address. The following code requests the files *index.html* and *source.html* from the TFTP server.

```
void initialize_filesystem()
{
	struct sockaddr address;
	unsigned int i;
	unsigned int result;
	void* start = (void*)FS_START;

	// initialize the file system
	int x = finit(FOPEN_MAX, FS_BLOCKS, start);
	printf("Result of FS init: %d \r\n", x);

	if ((x==0) && (fexists("index.html")==0) && (fexists("source.html")==0))
	{
		printf("File system OK, skip TFTP init.\r\n");
		return;
	}

	// lets get the files we want off a TFTP server
	// initialize TFTP server setting
	for (i=0;i<18;i++)
		address.sin_addr[i] = 0;

	// since the DS80C400 supports Ipv6, the address is 16 bytes long
	// however, since we are only using Ipv4 addresses, only the last
	// 4 bytes are meaningful
	address.sin_addr[12] = TFTP_IP_MSB;
	address.sin_addr[13] = TFTP_IP_2;
	address.sin_addr[14] = TFTP_IP_3;
	address.sin_addr[15] = TFTP_IP_LSB;

	result = settftpserver(&address, sizeof(struct sockaddr));
	printf("Set TFTP server to selected server, result: %u\r\n", result);
	result = tftp_init();
	printf("Result of TFTP init: %u \r\n", result);

	get_tftp_file("source.html");
	get_tftp_file("index.html");
}

void get_tftp_file(char* filename)
{
	unsigned int result;
	unsigned char* TFTP_MSG;
	FILE* file;

	printf("Free FS RAM: %ld\r\n", getFreeFSRAM());
	TFTP_MSG = getTFTPData();
	file = fopen(filename, "w");
	result = tftp_first(filename);
	if (result==0xFFFF)
	{
		printf("Error in TFTP_FIRST...\r\n");
		return;
	}
```

```
        printf("Result of first segment: %u\r\n", result);
        fwrite(TFTP_MSG, 1, result, file);

        while (result >= 512)
        {
               result = tftp_next(TFTP_MORE_DATA);
               if (result==0xFFFF)
               {
                      printf("Error in TFTP_NEXT...\r\n");
                      return;
               }
               printf("Result of next segment: %u\r\n", result);
               TFTP_MSG[result] = 0;
               fwrite(TFTP_MSG, 1, result, file);
        }
        tftp_next(TFTP_LAST_SEGMENT);

        fclose(file);
        printf("Done with TFTP server.\r\n");
}
```

Notice that the function `finit` must be called every time the application starts to make sure the file system is installed and functioning properly. If the file system initializes correctly (returns a 0) and the files we want already exist, then the function exits without trying to download the files. Otherwise it tries to read the files from the TFTP server and write them to the file system, as shown in the function `get_tftp_file`.

SolarWinds provides a free TFTP server for Windows platforms that was used in the development of this demonstration. From their website (www.solarwinds.net) follow the **Downloads - Free Software** menu choice and you will find the TFTP server download. After installing, use the **Configure** option under the **File** menu to configure which files are available to the TFTP server for download. 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 very simple version of an HTTP server as 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. This makes it very easy to set up a server socket. 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, but rather 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 we 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 will dynamically generate a response consisting of the latest results from the timeserver, as well as the number of seconds that have passed since the last instance the timeserver was queried. When the file **stats.html** is requested, statistics for server uptime and number of requests are displayed.

If the file is not found, and invalid request method is given, or there is a file system failure, an HTTP error code is reported.

The SNTP Client

The second major portion of the timeserver application is a Simple Network Time Protocol client, as described in RFC 1361. This is a simple version of the Network Time Protocol (RFC 1305). SNTP calls for the use of 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 at the time this application note was written, DNS lookup was not supported, so the IP address for the server is set manually.

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

// set a timeout of about 2 seconds
buffer[0] = 0x0;
buffer[1] = 0x0;
buffer[2] = 0x8;
buffer[3] = 0x0;
setsockopt(socket_handle, 0, SO_TIMEOUT, buffer, 200);

buffer[2] = 0;                          // reset since we used this in call to setsockopt
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);
```

First, a datagram socket is created and given a timeout of about 2 seconds (0x800==2048 milliseconds). This ensures that if the communication fails with our chosen server, we will not wait forever 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 timestamp 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 when the last time update was based on the DS80C400's internal clock. Since the program only updates about once every 60 seconds, the HTML page *time.html* will use this value to report how long it has been since the last time update. Finally, the socket is closed and the SNTP client goes to sleep for another 60 seconds.

A Note About Synchronization

Using the LARGE memory model, the Keil Compiler will pass a limited number of arguments in memory that is safe across process swaps. This means that certain functions must not be called from multiple processes at the same time. While efforts have been taken

to develop the C Libraries for the 400 such that all variables are passed in direct memory that is safe across process swaps, some functions are still dangerous. For instance, adhering to the Berkeley-style socket headers required some longer method signatures that involve passing data in unsafe memory. Therefore, there are 2 libraries for sockets.

One library (**rom_sock.lib**) adheres to the Berkeley-style headers. However, it is unsafe when using this library to call a function from two processes at the same time. This may not be a problem, if one process is using UDP functions and another is using TCP functions. For true protection against concurrent access to unsafe memory, another socket library has been developed (**rom_sock_synch.lib**). The functions in this library are similar to the Berkeley-style functions, but have fewer or rearranged arguments, such that the Keil compiler passes parameters in safe memory areas. In all cases, please consult the documentation to see if functions are multi-process-safe.

A Note About Passing Pointers

The Keil documentation provides ways to write your own methods in 8051 assembly that can be called from your C programs. If you choose to do this, note that pointers as passed from a C program to 8051 assembly are not immediately usable on the DS80C390 and DS80C400. Since the traditional 8051 architecture is 16-bit, Keil's pointers consist of two bytes of pointer, and one byte of memory type. When using Dallas's 24-bit 8051 micros, the memory type byte is used for the high byte of the pointer, but in an altered form. In the current version of the Keil compiler, the high pointer byte has its high bit set and is incremented by one. The following macro from **rom_offsets.inc** is used in the Dallas Semiconductor libraries to correct the altered pointers.

```
 FIXKEILPOINTER  MACRO DIRECT_DPX
        LOCAL  must_be_null
        mov    a, DIRECT_DPX
        jz     must_be_null
        dec    a
        anl    a, #7Fh
        mov    DIRECT_DPX, a
must_be_null:
        ENDM
```

The Keil compiler passes pointers either in registers `r3:r2:r1` (`r3` being the memory type byte) or in the XDATA memory area. This macro will work for any register or other direct memory value by passing it the memory type byte, and returning in the same location the high pointer byte. The following code demonstrates its use:

```
;
; Keil passes pointers as r3:r2:r1...
;---- Variable 'buffer1?972' assigned to Register 'R1/R2/R3' ----
;
FIXKEILPOINTER r3
;
; r3:r2:r1 is now usable as a pointer value.
;

;
; ...or in XDATA.
;---- use dpx1:dph1:dpl1 for buffer pointer ----
;
mov     dptr, #buffer2?1078
GETX
mov     dpx1, a
inc     dptr
GETX
mov     dph1, a
inc     dptr
GETX
mov     dpl1, a
FIXKEILPOINTER   dpx1
;
; Data pointer 1 is now usable as a pointer.
;
```

Note that there is also an opposite to the FIXKEILPOINTER macro that allows functions to convert pointers they need to return into a form that code generated by the Keil compiler can understand. In this case, use the macro UNFIXKEILPOINTER. This macro is used in the same way as the FIXKEILPOINTER macro. One difference is that when you return a pointer from a method written in assembly, the pointer should be stored in registers r3, r2, and r1, with the high pointer byte in r3. Therefore, just before a function exits that should return a pointer, it should call the macro:

```
UNFIXKEILPOINTER r3
ret
; End of the assembly function
```

Keep Your Keil Installation Current

From time to time, Keil releases updates for its uVision2 tool suite. The web site http://www.keil.com/update/ contains the latest information on the most current versions of both the C51 compiler and the uVision2 IDE. From this page you can select which downloads you would like and view what changes have been made.

The update should be an InstallShield executable. After you run it, the application will display a window titled **Setup uVision2**. Choose the **Update Current Installation** option to perform the update. The program should detect your current installation directory, click **Next** to continue. On the next screen, select if you want to keep your previous uVision2 configuration, and click **Next** again. Finally, verify the options you selected and begin installing.

Conclusion

The Keil 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 access the network stack, memory manager, process scheduler, file system, and many other features of the DS80C400. Moreover, applications written in C can choose which libraries to use and include, allowing more space for user code and data as compared to the TINI Runtime Environment. Developers using the C language for the DS80C400 will be able to write lean applications, giving them plenty of speed, power, and code space to tackle any problem.

References

[1] App Note 609: Internet Speaker with the DS80C400 Silicon Software

[2] Download at http://java.sun.com/j2se/downloads.html

[3] Download at http://java.sun.com/products/javacomm/

[4] The High-Speed Micro User's Guide Supplement for the DS80C400 can be found at http://pdfserv.maxim-ic.com/arpdf/Design/DS80C400UG.pdf

Relevant Links:
- C Library Home
- Keil Software Development Tools
- Java Development Kit Download Page
- Java Communications API
- Ethernet speaker application note
- 1-Wire Public Domain Kit
- DS80C400 User's Guide
- SolarWinds free TFTP server
- TINI Software Development Kit

More Information

DS80C390:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples
DS80C400:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples
DSTINIm400:	QuickView	-- Full (PDF) Data Sheet	
DSTINIs400:	QuickView	-- Full (PDF) Data Sheet	