Keywords: SDCC, DS80C400, MxTNI, sdcc, MxTNI C, ds80c400, mxtni, c compiler, MxTNI c

APPLICATION NOTE 3346

Using the SDCC compiler for the DS80C400

Nov 04, 2004

Abstract: The DS80C400 contains a ROM that provides a network stack, memory manager, and process scheduler that is flexible enough to be used from applications written in Java, C and 8051 assembly. SDCC provides a free, open-source compiler for 8051 devices that is compatible with the 24-bit addressing mode of the DS80C400. Complicated applications written in C can be easily built using the features of the DS80C400 ROM with the assistance of libraries provided by Dallas Semiconductor. These libraries, along with documentation and example code, are available for download.

This application note describes how to use the SDCC tools to build applications for the DS80C400. It begins with a HelloWorld application, and then demonstrates how to make use of the ROM libraries to implement a simple HTTP Server. The applications here are written and built for use with the TINIm400 reference module, and must be modified for designs with other memory configurations.

Also see:

- Using the IAR Compiler for the DS80C400
- App Note 613: Using the Keil C Compiler for the DS80C400

Introduction

The DS80C400 contains a ROM that provides a network stack, memory manager, and process scheduler that is flexible enough to be used from applications written in Java, C and 8051 assembly. SDCC provides a free, open-source compiler for 8051 devices that is compatible with the 24-bit addressing mode of the DS80C400. Complicated applications written in C can be easily built using the features of the DS80C400 ROM with the assistance of libraries provided by Dallas Semiconductor. These libraries, along with documentation and example code, are available for download.

This application note describes how to use the SDCC tools to build applications for the DS80C400. It begins with a HelloWorld application, and then demonstrates how to make use of the ROM libraries to implement a simple HTTP Server. The applications here are written and built for use with the TINIm400 reference module, and must be modified for designs with other memory configurations.

Getting Started with the SDCC Compiler

Follow these instructions to complete your first C application for the DS80C400 using the SDCC Compiler:

1. Install SDCC compiler[1]

- Download the installation file for the latest SDCC compiler version from SDCC website.
- Follow the instructions of setup file (possibly sdcc/doc/INSTALL.txt).

2. Create a new file "main.c" using your favorite text editor. Write the following in that file:

```
#include <stdio.h>
void main ()
{
        printf("Hello Universe!!!!....Welcome to SDCC Tini Test Program");

                        while (1)
                        {
}
                }
```

   Make sure to save the contents of this file.
3. Copy the files startup400.a51 and reg400.inc (included in the startup code download) from the SDCC C Library Website[2] to the same directory where you stored main.c. This file contains the **startup_code** function that will be called on application startup to initialize the DS80C400 chip. The startup code does the following:
   - Configures DS80C400 in 24-bit contiguous address mode
   - Configures timer 2 to generate 115200 baud rate for serial port
   - Initializes the data memory
4. Copy and unpack the **ROM initialization** library files(rominit.lib and rom400.h from the init library download) from SDCC C library website to the same directory. The library distribution is zipped, so use WinZip or gunzip/tar to open the package.
5. Before we compile our "Hello Universe" application, we need to make a small change in one of the SDCC-installed support files to override the default DS80C400 support functions and use Dallas Semiconductor's C library instead. Make the following changes:
   - Rename \SDCC\lib\ds400\libds400.lib file to \SDCC\lib\ds400\libds400.lib.old
   - Create an empty \SDCC\lib\ds400\libds400.lib file (use the **touch** command or create a blank file in your favorite text editor)
6. Build the "Hello Universe" application...
   - To create an object file (.rel) from our startup400.a51 file, execute the following from the command line:

     ```
     asx8051 -losffgp startup400.a51
     ```

     asx8051 is the assembler provided with the SDCC tools. The options provided to the assembler are:

Option	Purpose
l	generates a list file
o	generates an object file
s	generates a symbol file
ff	flag reolcatable references by mode in listing file
g	make undefined symbols be global

p	disables listing pagination

The "los" option is mandatory, as the linker requires list, object and symbol files to generate executables. The "ff" and "p" options generate a readable list file. The "g" option tells the assembler to not generate an error if it finds an undefined symbol that is not declared as external.

- To create an object file from main.c, execute the following:

```
sdcc -c -mds400 --model-flat24 --stack-10bit --no-xinit-opt main.c
```

sdcc is the compiler. The options passed to the compiler are:

Option	Purpose
-c	compiles main.c and creates an object file
-mds400	generates code for the DS80C400 processor
--model-flat24	use the 24-bit contiguous memory model
--stack-10bit	use the 1024-byte extended stack (10 bit stack addresses)
--no-xinit-opt	don't initialize the external RAM memory area
p	disables listing pagination

Note the double dashes on the last three arguments in the table.

- To link the object files and build the executable, run the following:

```
    sdcc -mds400 --model-flat24 --stack-10bit -Wl-r --xram-loc
0x10000 --xram-size 0x3fff --code-loc 0x400000 main.rel startup400.rel
-l
rominit.lib
```

The new options used here are:

Option	Purpose
-Wl	pass options through to the linker
--xram-loc	external RAM start address (only RAM for SDCC variable use!)
--xram-size	external RAM size (only RAM for SDCC variable use!)
--code-loc	code starting address
-l	include the specified libraries
p	disables listing pagination

Note the double dashes on the xram-loc, xram-size, and code-loc arguments. Also note that the RAM specified to the command will be used for SDCC variable storage, and should not conflict with the memory range used in the init_rom function to initialize the DS80C400--that memory is used for the network stack and memory manager.

- To compress the executable file and generate a hex file, execute the following:
  packihx main.ihx>hellouniverse.hex
  The packihx utility compresses the executable file by accumulating contiguous data records up to 16 bytes.

Now that we have an executable file, we need to download the application onto the TINIm400 module and execute it.

Loading the Sample Application onto the TINIm400 Module

This section describes loading the hex file produced by the SDCC compiler onto the TINIm400 verification module using the tool **Microcontroller Tool Kit (MTK)** provided by Maxim/Dallas Semiconductor. The Current version of MTK is available only for Windows®.

If your development environment is not Windows, you will need to use the **JavaKit** application for downloading and executing applications. To use JavaKit, you must have the Java Runtime Environment[3] (at least version 1.2) and the Java Communications API[4] installed. The JavaKit tool is included with the MxTNI™ Software Development Kit. As of this writing, firmware version 1.13 was the most recent firmware released. Instructions for running JavaKit can be found in the file **Running_JavaKit.txt** in the docs directory of the MxTNI SDK. If you encounter technical issues running **MTK** or JavaKit, chances are someone has already had a similar problem and it is chronicled in the Dallas semiconductor discussion board. You can search the existing posts (and create new posts).

Download the most recent version of the MTK application. To install MTK, run the installation file and follow the instructions. After a successful installation, a new menu group will be added: Start->All Programs->Dallas Semiconductor MTK. When MTK is launched, the dialog box shown in **Figure 1** will be displayed.

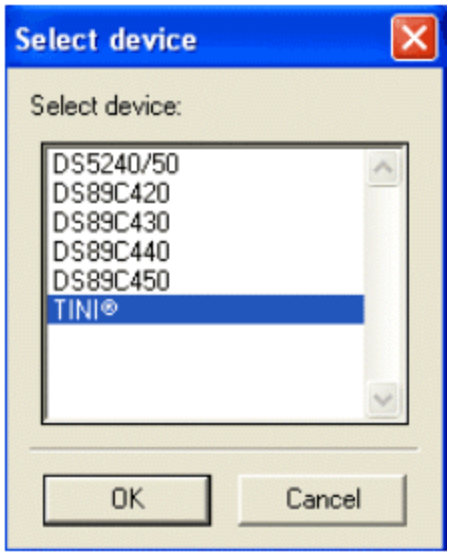


Figure 1. MTK options on startup.

Select the option MxTNI to work with the TINIm400 evaluation board.

After selecting MxTNI, the MTK main window will be opened. Select the serial port you will use to communicate with the TINIm400 from Options->Configure Serial Port menu option. Then, select the Tini->Tini Options menu item, and the following dialog box will be displayed. Select the DSTINIm400 button to configure MTK for communication with the TINIm400 board. **Figure 2** shows this dialog with the DSTINIm400 button.

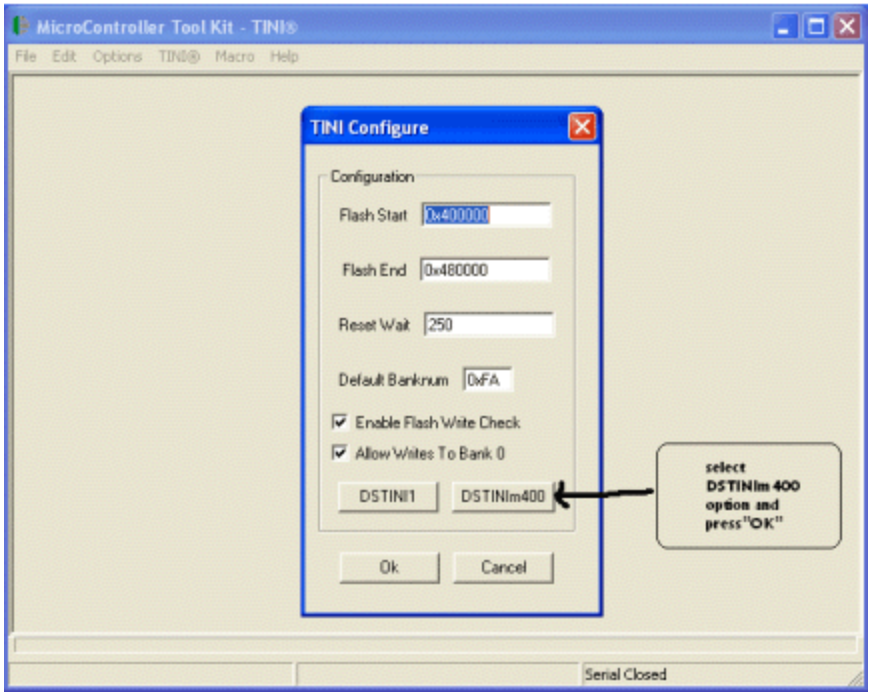


Figure 2. Selecting the TINIm400 configuration option.

Open the serial port by selecting the Tini->Open COMx at xxx baud menu option. Then select the Tini->Reset option to reset the evaluation board. The loader prompt for the DS80C400 should print:

```
DS80C400 Silicon Software - Copyright (C) 2002 Maxim Integrated Products
Detailed product information available at http://www.maximintegrated.com

Welcome to the TINI DS80C400 Auto Boot Loader 1.0.1
>
```

From the File menu, select Load HEX File. Search for the hellouniverse.hex file that we just created and select it. 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 (or zero), and is located at address 0002h of the current bank. The SDCC Compiler inserts this tag in generated assembly code. If you open up the main.asm source code generated for the

hellouniverse project, you will find the following piece of code:

```
   .area CSEG    (CODE)
interrupt_vect:
   ; DS80C400 IVT must be generated at runtime.
        sjmp    __sdcc_400boot
        .ascii  'TINI'  ; required signature for 400 boot loader.
        .db     0       ; selected bank or zero...
__sdcc_400boot:
        ljmp    __sdcc_gsinit_startup
```

Note that the sjmp __sdcc_400boot statement is located at address 0000h of bank 40. It is followed by the executable tag { 'T', 'I', 'N', 'I', 0h}, located at address 0002, since the sjmp statement is two bytes. When you type 'E', the ROM starts from bank C0h 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 SDCC ROM Libraries

The procedure for calling ROM functions from assembly is described in the High-Speed Microcontroller User's Guide supplement for the DS80C400 . However, calling these ROM functions from C is little more complicated. Parameters must be converted from the SDCC C Compiler's conventions to the conventions used by the ROM. The SDCC compiler passes parameters in a combination of the hardware stack, accumulator, and the data pointer. The ROM functions accept parameters in a number of different ways. For example, the socket functions accept parameters stored in a single buffer of external RAM. Conversely, many of the utility functions accept parameters passed in special function registers or direct memory locations. In order to translate from SDCC 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 including a header file and linking with corresponding library file. The ROM libraries for SDCC Compiler include:

- ROM initialization Routines
- DHCP Client
- Process Scheduler
- Sockets (TCP, UDP, Multicast)
- TFTP Client
- Utility functions (CRC16, random numbers)

The extension libraries like the File System, Mail client and HTTP Server are not available for the SDCC compiler at the time of this writing. Watch the SDCC Library Home Page for the DS80C400 for updates as we add libraries supported for SDCC.

A Simple Application: HTTP Server

A simple http server has been written to demonstrate how to use some of the ROM libraries functionality, specifically the socket and process scheduler libraries. This sample occasionally updates its time from a network timeserver, and serves that information through its web server.

The sample application consists of two modules, an HTTP server and an SNTP client. The main program

creates a new subtask for running the http server that handles client connections on port 80. The parent task will be trying to synchronize the current time from the time server once every 60 seconds.

The SNTP Client Module

The following piece of code covers core functionality of SNTP client module.

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

// set a timeout of about 2 seconds
for (i=0;i<256;i++)
    buffer[i] = 0;
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_high = (NTP_PORT/0x100); //higher byte of port number
address.sin_port_low = (NTP_PORT%0x100); //lower byte of port number

sendto(socket_handle, buffer, 48, 0, &address, sizeof(struct sockaddr));
recvfrom(socket_handle, buffer, 256, 0, &address, sizeof(struct sockaddr));

//SDCC uses little Endian for storing data, so reorganize the data before
converting it to long
buffer[0]=buffer[43];
buffer[1]=buffer[42];
buffer[2]=buffer[41];
buffer[3]=buffer[40];

timeStamp = *(unsigned long *)(&buffer[0]);

formatTimeString(timestamp - (5 * SECONDS_PER_HOUR), "Tampa, USA",
                 last_time_reading_1);
formatTimeString(timeStamp - (3 * SECONDS_PER_HOUR), "Sao Paulo, Brazil",
                 last_time_reading_2);
formatTimeString(timeStamp + (1 * SECONDS_PER_HOUR),"Marseille, France",
                 last_time_reading_3);
formatTimeString(timeStamp + (5 * SECONDS_PER_HOUR) + (30 *
                 SECONDS_PER_MINUTE), "Bangalore, India",
                 last_time_reading_4);
formatTimeString(timeStamp +  (8 * SECONDS_PER_HOUR), "Hsinchu, Taiwan",
                 last_time_reading_5);
last_reading_seconds = getTimeSeconds();
closesocket(socket_handle);
```

The SNTP client module was implemented as per RFC 1361. The SNTP module communicates with time.nist.gov using the UDP protocol to request a time stamp. Note that the IP address for time.nist.gov is set manually as DNS support was not available for the SDCC compiler when this appnote was written.

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 Marseille, France it is 9:37:37 on September 3, 2000."

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.

The Simple HTTP Server

Another sub module of the timeserver application is a web server. The server in this application is implemented as a simple version of an HTTP server as described by RFC 2068. Only the "GET" method is supported--input headers are ignored, and few output headers are given. The File System library was not available when this application note was written, so the sample application dynamically generates HTML pages.

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
");

// 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 parses the incoming request for a file name and verifies that the request method is 'GET'. No other method (not even 'POST', 'HEAD' or 'OPTIONS') is allowed.

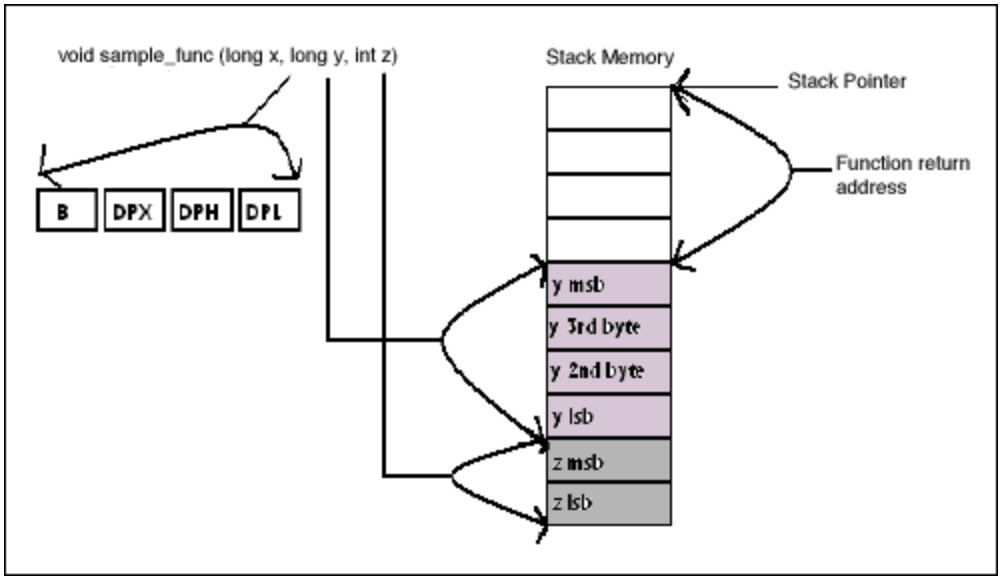
A Note About Writing DS80C400 Assembly Functions for SDCC Compiler

Even though SDCC provides a rich set of library functions, sometimes we will want to write optimized modules in assembly language, or port existing 8051 assembly modules into our application. The following are some important points to keep in mind while writing 8051 assembly functions to be called from C programs written using the SDCC compiler:

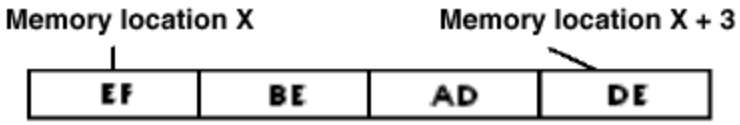
1. Function Parameter passing convention: The following table shows how arguments are passed for reentrant functions

Argument position	Character	Integer	Long	Address
First argument	Dpl	Dph:dpl	B:dpx:dph:dpl	B:dpx:dph:dpl
Second argument onwards	the values will be passed through hardware stack from right to left			

   The arguments for the function void sample_func(long x, long y,int z) reentrant; will be passed as follows.



2. Data type storing convention:
   SDCC follows the Little Endian storing convention. In other words, SDCC uses the format for storage of binary data in which the least significant byte appears first. For example, a 32-bit long value 0xDEADBEEF will be stored as follows:



3. Address pointer size
   SDCC uses four bytes for storing memory addresses. The following table shows the format of memory address:

Most significant byte	3rd byte	2nd Byte	Least significant byte
address type (possible values for ds80c400: 0-near, 1-far, 2-code)	MSB of address	2nd byte of address	LSB of address

Near address pointers use indirectly addressable, internal RAM memory (idata) for storage and its address size is only one byte. The upper 16 bits of the raw address are unused.

Far address pointers are for access to external memory and are 24 bits.

For more information about SDCC ASx8051 assembler, see the ASxxxx assembler reference manual. All SDCC documentation can be downloaded from http://sdcc.sourceforge.net/snap.php#Docs

Limitations and Development Issues

The following are the limitations that we observed while working with SDCC version 4.0 compiler:

1. The compiler does not support recursive functions
2. The library routines are not optimized.
3. The assembler does not support macros
4. Functions like printf and sprintf have some issues and would not work properly for some parameter combinations. For example, the following code causes the application to hang:

   ```
   char temp[50];
   sprinf(temp,"%d",234234);
   ```

5. Arithmetic expression with long constants are not working properly
6. The assembly code generated for array initialization (‘int[] values={1, 2, 3, 4, 5};') does not initialize the correct memory area.

As SDCC is always under active development, please download the latest release if you found any bug in your current version of SDCC tools or your version is very older than the current release.

Conclusion

The DS80C400 ROM libraries for the SDCC compiler provided by Dallas Semiconductor expand the options for embedded network application designers seeking low-cost network microcontroller solutions. Compared to the MxTNI Java 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. Dallas Semiconductor is working on porting all the DS80C400 libraries to SDCC that currently work for the Keil compiler. Please frequently visit the DS80C400 SDCC Library home page for updates.

Relevant Links

- SDCC ROM Libraries Project Home
- SDCC Software Development Tools

- Java Development Kit Download Page
- Java Communications API
- Ethernet speaker application note
- 1-Wire Public Domain Kit
- DS80C400 User's Guide
- MxTNI Software Development Kit
- Using the Keil Compiler for the DS80C400 appnote

Notes

[1] Download the SDCC compiler
[2] The home page for SDCC ROM libraries
[3] Java runtime environment
[4] Java communications API

Related Parts		
DS80C400	Network Microcontroller	Free Samples

More Information
For Technical Support: http://www.maximintegrated.com/support
For Samples: http://www.maximintegrated.com/samples
Other Questions and Comments: http://www.maximintegrated.com/contact

Application Note 3346: http://www.maximintegrated.com/an3346
APPLICATION NOTE 3346, AN3346, AN 3346, APP3346, Appnote3346, Appnote 3346