



APPLICATION NOTE 3550

Using the IAR Compiler for the DS80C400

Abstract: The functionality of the ROM in the DS80C400 microcontroller can be accessed from programs written in 8051 assembly, C, or Java™. By using this ROM and software libraries developed by the Dallas Semiconductor, many applications can be built. This application note describes how to start using the 8051 IAR Embedded Workbench™ to build applications for the DS80C400 in C. The DS80C400's ROM functionality will be demonstrated by implementing a simple HTTP server.

Also See:

- [Using the SDCC Compiler for the DS80C400](#)
- Application note 613, "[Using the Keil C Compiler for the DS80C400.](#)"

Introduction

The ROM for the DS80C400 microcontroller contains a suite of functionality that can be accessed from programs written in 8051 assembly, C, or Java. Offering the proven TINI® network stack, process scheduler, and memory manager, the ROM of the DS80C400 is an excellent starting point for building C and assembly programs. Assembly language is sufficient for simple programs. For more complex programs, however, C can take advantage of the ROM components and the software libraries provided by Dallas Semiconductor. These libraries will help you build applications using Keil µVision2®, SDCC, and the IAR 8051 compiler.

This application note describes how to start using the 8051 IAR Embedded Workbench™ to build applications for the DS80C400 in C. The note also demonstrates how to use the DS80C400's ROM functionality by implementing a simple HTTP server. All the development presented here was done with the TINIm400 verification module and the 8051 IAR Embedded Workbench, which includes the C compiler version 6.11A.

Getting Started with 8051 IAR Embedded Workbench

This section explains how to use the IAR Embedded Workbench tool suite to build a simple Hello World program written in C, your first C application for the DS80C400.

1. Install the IAR Embedded Workbench
2. Select **File**→**New**→**Workspace**. Enter the name of the workspace **appnote** in the workspace window.
3. Select **Project**→**Create New Project**. Enter the name of the project **helloworld** in the dialog box that appears. Make sure that **8051** is selected as the Tool chain.
4. When the project window opens on the left, select **Project**→**Add Files...** In the pop-up dialog file, change **files of type** to **Assembler Files**. Add the file **Cstartup.s51**, which can be found in the zip file: [Download](#).
5. Open the file **Cstartup.s51** by double clicking on it. Find the segment declaration **RSEG CSTART:CODE:ROOT(0)**. This is where the code segment starts. The various segment start addresses are declared in **link51ew_400.xcl**. The beginning of code is declared at 0x400000h in this file. There should also be a **DB 'TINI'** line followed by another **DB, high(?INIT)** with a comment that says **Target Bank**. This ensures that the application corresponds to the beginning of the flash on the TINIm400.
6. Create a new file, **main.c**. Write the following in that file:

```
#include <stdio.h>
#include <printf.c>
void main ()
{
    printf("Test program using IAR compiler");

    while (1)
    {
    }
}
```

Save the contents of this file. Add the file you just created to the project **helloworld** by selecting **Project**→**Add Files** and choosing **main.c** in the file dialog. Make sure that you are adding the same **main.c** that you created; it is quite possible that another file with the same name exists in your default directory.

7. Similarly, add the files **low_level_init.s51** and **putchar.c** to your project. The **low_level_init.s51** file contains low-level DS80C400 initialization routines, and **putchar.c** contains a low-level routine for outputting the characters to the default console.

8. Copy and unpack the **ROM initialization** library files from http://files.dalsemi.com/tini/ds80c400/c_libraries/iar/bin/init.zip to the same directory. Add the **rominit.r51** library file to the project.
9. Before we compile our Hello_World application, we need to configure the IAR tool chain to suit the DS80C400 target.
 1. Select **Project**→**Options**→**General**.
 - Click **Target** tab and browse to select **DS80C400** for the **Derivative**. Change the value for **Extended stack at:** to **0xFFDC00**. This is because the IAR startup code relocates the DS80C400 hardware stack to 0xFFDC00. Refer to **Figure 1** for these settings.
 - Click the **Data Pointer** tab. Select **Number of DPTRs = 1**. This is because the libraries supplied by Dallas Semiconductor are created using this option.

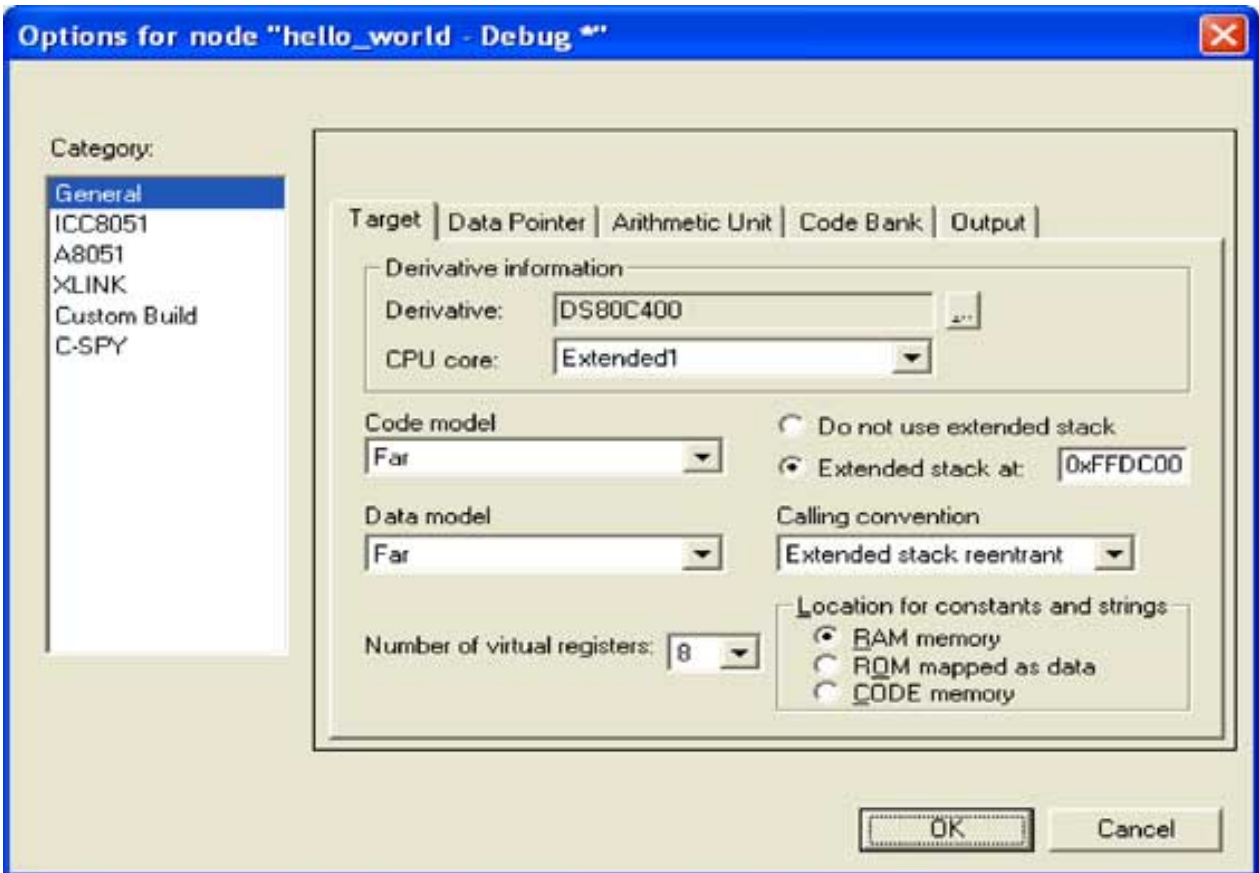


Figure 1. Selecting the Target options for a new IAR 8051 project.

2. Select **Project**→**Options**→**ICC8051**
 - Click **Code** tab. Select **Size** and **None** under **Optimizations**.
 - Click **List** tab. Enable **Output List File** and **Output assembler File**.
 - Click **Preprocessor** tab. Include the following lines for the include path:

```

$TOOLKIT_DIR$\INC\
$TOOLKIT_DIR$\INC\CLIB
$TOOLKIT_DIR$\src\lib
$TOOLKIT_DIR$\src\lib\clib
..\include\

```

The last line above is the include path where library head files (*.h) should be. Make sure that the header files are present in the path specified above. **\$TOOLKIT_DIR\$** refers to the IAR installation path (for example, c:/program files/iar systems/embedded workbench 3.n/8051).

3. Select **Project**→**Options**→**A8051**
 - Click **List** tab. Enable **Output List File**.
 - Click **Preprocessor** tab. Include following lines for the include path:

```

$TOOLKIT_DIR$\INC\
$TOOLKIT_DIR$\src\lib
..\include\

```

The last line above is the include path where library head files (*.h; *.inc) should be. Make sure the header files are present in the path specified above.

4. Select **Project**→**Options**→**XLINK**
 - Click **Output** tab.
 - In the **Output file** group, enable **Override default** and change the text to **hello_world.hex**

- In the **Format** group, enable **Other** and select **Intel Extended** from the choices. Refer to **Figure 2** for the details.
- Click **List** tab. Enable **Generate Linker Listing**.
- Click **include** tab. Refer to **Figure 3**.
 - Enable **Ignore CSTARTUP in Library**.
 - Click on **Override default** and change the file name to **\$TOOLKIT_DIR\$\config\lnk51ew_400.xcl**. **\$TOOLKIT_DIR\$** refers to the IAR installation path (for example, c:/program files/iar systems/embedded workbench 3.n/8051). Make sure that the files **lnk51ew_400.xcl** and **lnk_base_400.xcl** are present in the path specified. These files can be found in the zip file: [Download](#).

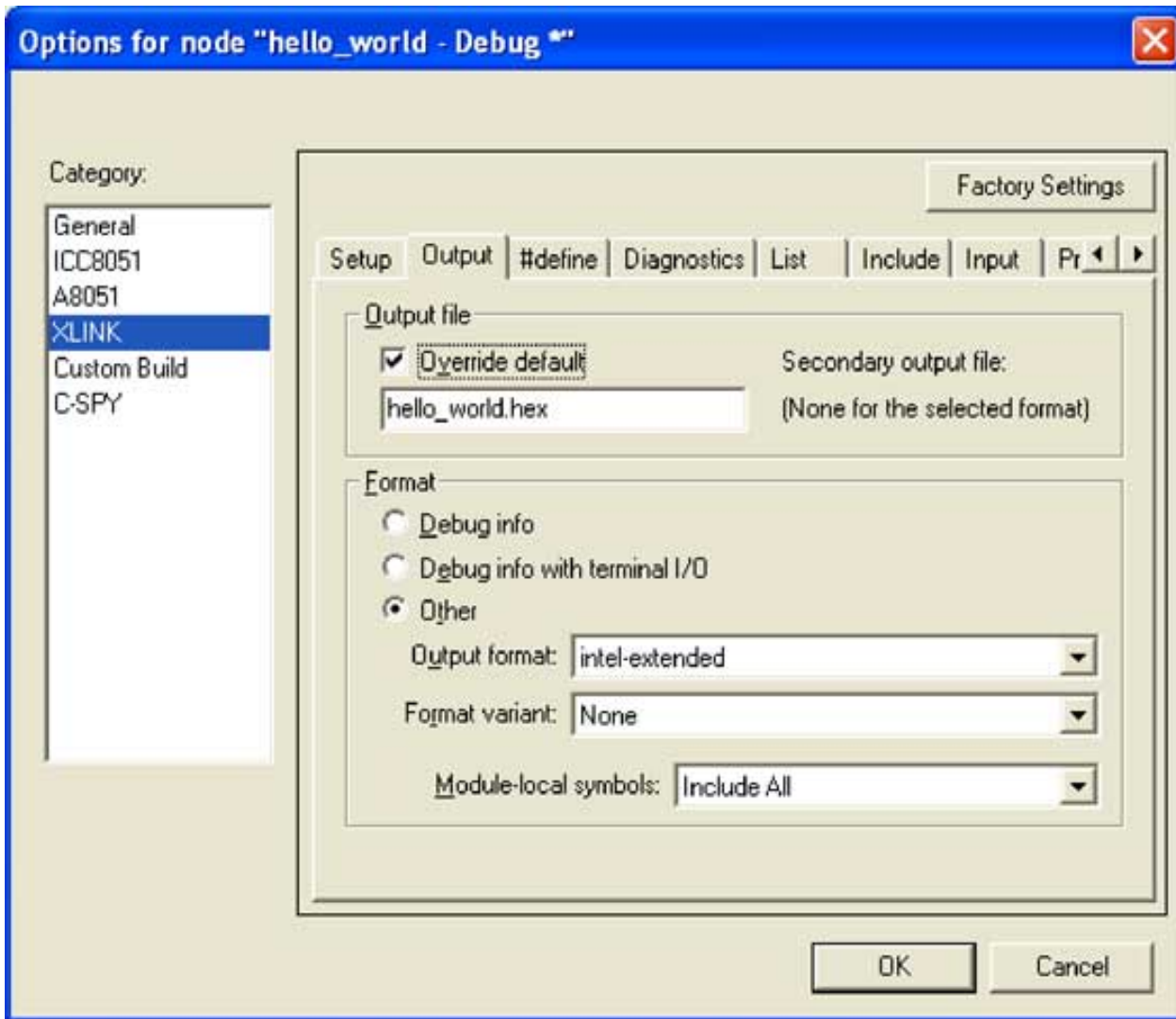


Figure 2. Selecting the XLINK Output options for a new IAR 8051 project.

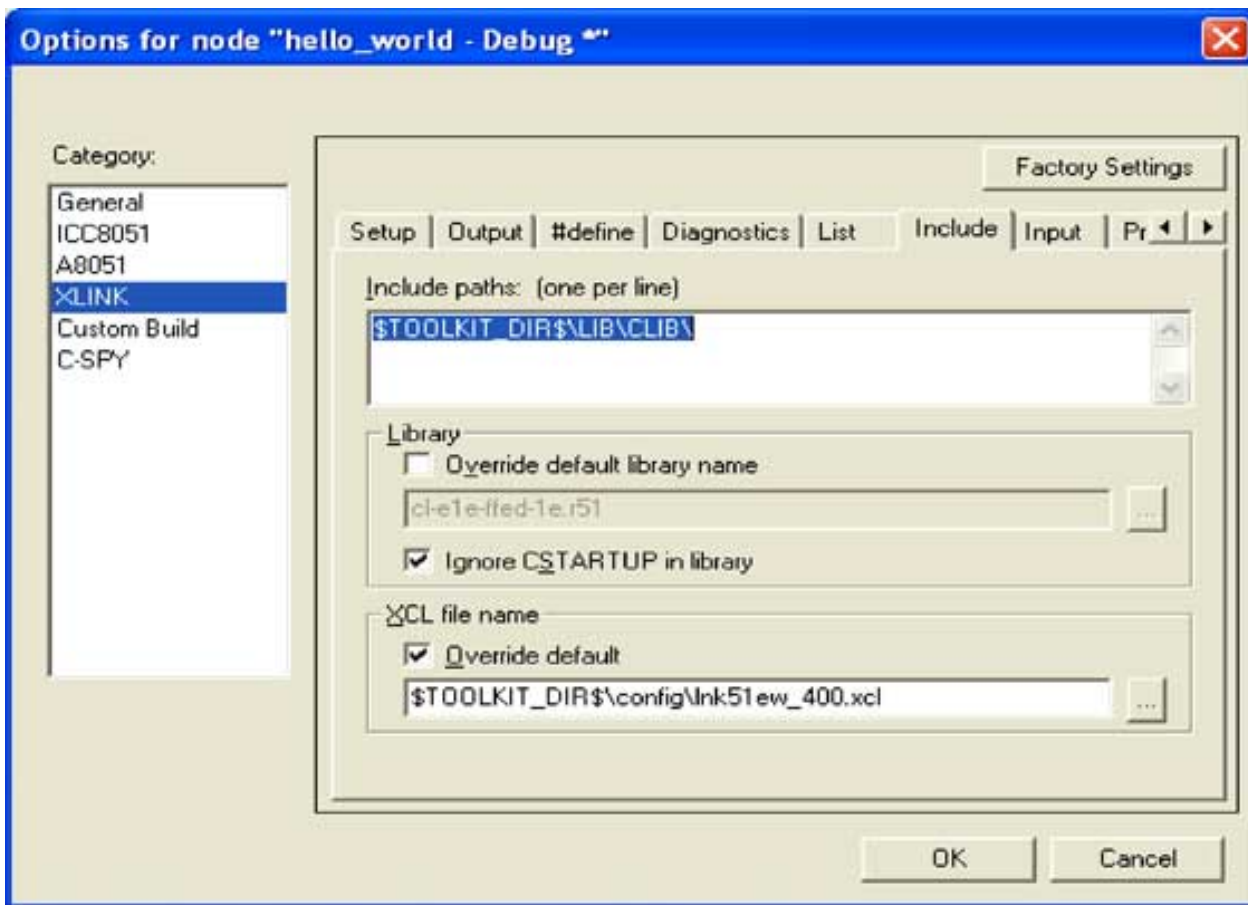


Figure 3. Selecting the XLINK Include options for a new IAR 8051 project.

Build the Hello_World application. If everything was done correctly, then the build should succeed without errors or warnings, and **hello_world.hex** should be generated in the `<project_dir>/debug/exe` directory. If your project configuration is Release, the hex file will be created in the `<project_dir>/release/exe` directory.

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 IAR 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 can use the **JavaKit** application for downloading and executing applications. To use JavaKit, you must have the Java Runtime Environment¹ (at least version 1.2) and the Java Communications API² installed. The **JavaKit** tool is included with the TINIm Software Development Kit. Download the [TINI SDK](#). As of this writing, firmware version 1.15 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 TINIm SDK. If you encounter technical issues running **MTK** or **JavaKit**, chances are someone already had a similar problem which is chronicled in the [Dallas Semiconductor discussion board](#). You can search the existing posts (and create new posts).

The most recent version of the MTK application can be [downloaded](#). 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 similar to the one shown in **Figure 4** will be displayed.



Figure 4. MTK options on startup.

Select the option **TINI** to work with the TINIm400 evaluation board.

After selecting **TINI**, 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 **DSTINI m400** button to configure MTK for communication with the TINIm400 board. **Figure 5** shows this dialog with the **DSTINI m400** button.

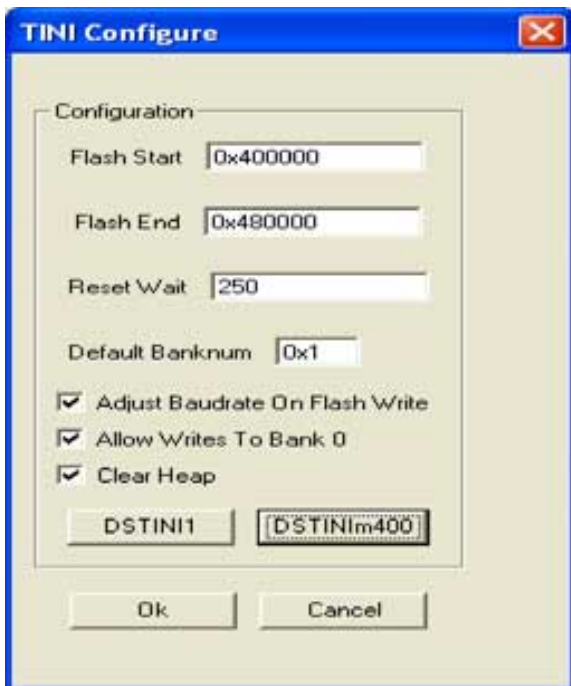


Figure 5. Selecting the TINIm400 configuration option.

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, 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, select **Load HEX File**. Search for the **hello_world.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, and is located at address 0x0002 of the current bank. The startup code of the application declares this tag with the following lines:

```
?VECTOR_TABLE:
    sjmp ?INIT
    DB    'TINI'          ; Tag for TINI Environment 1.02c
                                ; or later (ignored in 1.02b)
    DB    high(?INIT)    ; Target bank
```

Note that the `sjmp ?INIT` statement is located at address 0x0000 of bank 0x40. It is followed by the executable tag { **'T', 'I', 'N', 'I', 'O'h** }, located at address 0x0002, because 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 0x400000, where your code was loaded. If this happens, you may need to find where that tag is, and delete the contents of that area of memory.

Interfacing to the ROM and IAR ROM Libraries

The procedure for calling ROM functions from assembly is described in the [High-Speed Microcontroller User's Guide](#) supplement for the DS80C400³. However, calling these ROM functions from C is somewhat more complicated. Parameters must be converted from the IAR C compiler's conventions to the conventions used by the ROM. The IAR compiler passes parameters in a combination of hardware stack locations and registers. The ROM functions accept parameters in a number of different ways. For example, the socket functions accept parameters stored in a single parameter buffer. Conversely, many of the utility functions accept parameters passed in special function registers or stack memory locations. To translate from IAR calling conventions to the ROM's parameter conventions, Dallas Semiconductor wrote ROM libraries for accessing the ROM functions.

Using ROM functions in your C programs involves only including a header file and linking with corresponding library file. The ROM libraries for IAR 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 IAR compiler at the time of this writing. Watch the IAR Library Home Page for the DS80C400⁴ for updates as Dallas Semiconductor adds more library support for IAR.

A Simple Application: HTTP Server

A simple http server was written to demonstrate how to use some of the ROM libraries' functionality, specifically the socket and task scheduler libraries. 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 a time server once every 60 seconds.

The SNTP Client Module

The following code covers core functionality of SNTP client module.

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

```

for (i=0;i<256;i++)
    buffer[i] = 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 = htons(NTP_PORT) // port number

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

//IAR 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, "London", last_time_reading_1);
formatTimeString(timeStamp - (6 * SECONDS_PER_HOUR), "Dallas", last_time_reading_2);

formatTimeString(timeStamp + (5 * SECONDS_PER_HOUR) + (30 * SECONDS_PER_MINUTE), "Bangalore", last_time_reading_3);

formatTimeString(timeStamp - (10 * SECONDS_PER_HOUR), "Honolulu", last_time_reading_4);

last_reading_seconds = getTimeSeconds();
closesocket(socket_handle);

```

The SNTP client module was implemented following 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 lookup support was not available when this application note 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 London it is 05:33:19 on May 11, 2005.**"

The function **getTimeSeconds** is used to determine when the last time update was based on the DS80C400's internal clock. As 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 time-server application is a web server. The HTTP server in this application is implemented as a very simple version of an HTTP server, as described by RFC 2068. In our version, 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 = htons(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, 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 the IAR Compiler

The IAR documentation provides ways to write your own methods in 8051 assembly that can be called from your C programs. The following points are important to remember while writing 8051 assembly functions to be called from C programs written using the IAR compiler. In case the registers are not available for passing the arguments, they are pushed onto the stack in little Endian order.

1. Function parameter passing convention

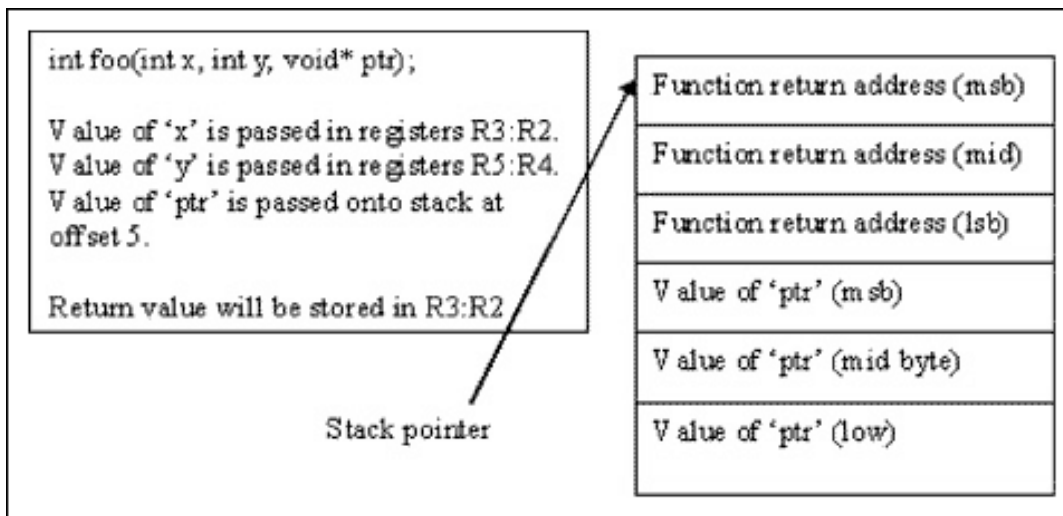
The following table shows how arguments are passed.

Arguments	Character
8-bit values	R1,R2,R3,R4,R5
16-bit values	R3:R2 or R5:R4
24-bit (pointer) values	R3:R2:R1
32-bit values	R5:R4:R3:R2

The following table shows convention for function returns values.

Arguments	Character
8-bit values	R1
16-bit values	R3:R2
24-bit (pointer) values	R3:R2:R1
32-bit values	R5:R4:R3:R2

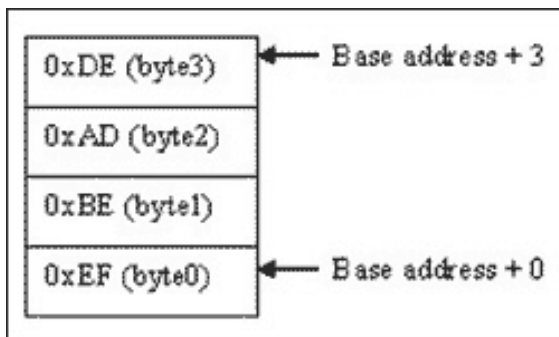
The arguments and returns values for the function `int foo(int x, int y, void* ptr);` will be passed as follows:



2. Data-type storing convention

IAR follows the Little Endian storing convention. Restated, IAR uses the format for storage of binary data in which the least significant byte of the number is stored at the lowest address.

For example, a 4-byte long value, 0xDEADBEEF, will be stored in memory as follows:



3. Interfacing a simple assembly program with 'C'

This section demonstrates how to write an assembly program and interface with the 'C' program using the IAR Embedded Workbench. The application swaps the bytes of the 16-bit and 32-bit values and prints the swapped bytes on to the default console. The prototype of the C callable function is `int ltob(int *shortptr , long *longptr)`.

The sample application consists of two files, `main.c` and `eswap.s51`. The `main.c` calls our example function `ltob()` written in assembly language. Create a new project, namely `endian`; add `cstartup.s51`, `low_level_init.s51`, `putchar.c` files and the Dallas Semiconductor ROM initialization library `rominit.r51`, as detailed in the section above, *Getting Started with 8051 IAR Embedded Workbench*.

Create a new `main.c` file with the following contents and add it to the project `endian`. In C, a function must be declared so the compiler knows how to call it. The `ltob()` function is declared before the `main()`. Note that the function `ltob()` returns '0' on success and non-zero if any of the pointers are NULL. The program should output following results onto a console:

```
-----
Program output:
Set values:          int= 0xdead   long = 0x12345678
Converted values:   int= 0xedde   long = 0x78563412
-----
```

```
// program main.c
#include <stdio.h>

#include <printf.c>
#include <frmwri.c>

int ltob(unsigned int *intptr,unsigned long *longptr);

void main()
{
```

```

unsigned long i = 0x12345678;
unsigned int k = 0xdead;
int err;

printf("set values:          int=0x%x long=0x%lx\n",k,i);

err = ltob(&k,&i);

if(err)
    printf("Error: One of the pointers is NULL\n");
else
    printf("converted values:    int=0x%x long=0x%lx\n",k,i);

while(1)
    ;
}

```

Create a new file, **eswap.s51**, enter the following assembly code, and add it to the project **endian**. This assembly program declares our function **ltob()** as PUBLIC, since it will be called by 'C' program. The first parameter to the **ltob()** is a pointer and is passed in registers r3:r2:r1 of DS80C400 controller. The second parameter, also a pointer, is pushed onto the stack at offset 3 to 5 (offset 3 contains least significant byte, and offset 5 contains most significant byte) by the IAR compiler. First, the function retrieves the pointer (pointing to a 32-bit value) stored in the stack, swaps the bytes it points to, and stores the swapped bytes back at the same memory location. Similarly, the 16-bit value is also byte-swapped and stored back at the same location where it was stored prior to conversion. Note that the registers r6 and r7 are preserved through the assembly function. This is because the IAR compiler treats these registers as permanent registers, meaning that these registers should not be altered by any of the function calls.

```

#include "reg400.inc"

r0_b0    equ 0                ; Register bank 0 equates.
r1_b0    equ 1
r2_b0    equ 2
r3_b0    equ 3
r4_b0    equ 4
r5_b0    equ 5
r6_b0    equ 6
r7_b0    equ 7

PROGRAM ENDIAN_SWAP

PUBLIC ltob

RSEG FAR_CODE:CODE:NOROOT(0)

; *****
;
; int ltob(unsigned int* shortptr, unsigned long* longptr)
;
; *****

ltob:

    // shortptr is in r3:r2:r1
    // longptr is in stack at offset 5

    ; get the longptr stored in the stack
    mov a,SP
    clr c
    subb a,#5
    mov b,a
    mov a,esp
    anl a,#0x3
    orl a,#0xDC    ; extended stack is at 0xff dc00
    subb a,#00    ; subtract 0x0005 to point to MSB of 2 nd argument
    mov DPX,#0xFF
    mov DPH,a
    mov DPL,b

    push r6_b0    ; save r6:r7 for the compiler

```

```

push r7_b0

movx a,@DPTR
mov r4,a           ;store least significant byte of 'longptr' in r4
inc DPTR

movx a,@DPTR
mov r5,a           ;store middle byte of 'longptr' in r5
inc DPTR

movx a,@DPTR
mov r6,a           ;store most significant byte of 'longptr' in r6

mov a,r4_b0
orl a,r5_b0
orl a,r6_b0
jz ltob_err       ; is (longptr == NULL)?

mov dpx,r6_b0     ; point to the memory where 'longptr' is pointing to
mov dph,r5_b0
mov dpl,r4_b0

pop r6_b0         ; restore r6:r7 for the compiler
pop r7_b0

push dpx
push dph
push dpl

movx a,@dptr      ; get the long value (in r4:r3:r2:r1) from the memory
mov r4,a
inc dptr
movx a,@dptr
mov r5,a
inc dptr
movx a,@dptr
mov r6,a
inc dptr
movx a,@dptr
mov r7,a
inc dptr

pop dpl
pop dph
pop dpx

mov a,r7_b0       ; swap the long value bytes and store it in memory
movx @dptr,a
inc dptr
mov a,r6_b0
movx @dptr,a
inc dptr
mov a,r5_b0
movx @dptr,a
inc dptr
mov a,r4_b0
movx @dptr,a

mov a,r1_b0       ; is (shortptr == NULL)?
orl a,r2_b0
orl a,r3_b0
jz ltob_err

mov dpx,r3_b0     ; point to a memory where the 'shortptr' is pointing to
mov dph,r2_b0
mov dpl,r1_b0

push dpx
push dph
push dpl

movx a,@DPTR     ; get the integer value from memory
mov r2,a

```

```

inc dptr
movx a,@dptr
mov r1,a
inc dptr

pop dpl
pop dph
pop dpx

mov a,r1_b0      ; swap the integer bytes
movx @dptr,a
inc dptr
mov a,r2_b0
movx @dptr,a      ; bytes of an integer are swapped and stored in memory

mov r3,#00      ; return 'success'
mov r2,#00
sjmp ltob_exit

ltob_err:
mov r3,#00      ; return 'error'
mov r2,#01

ltob_exit:

ret

END              ; end of assembly program

```

Download: [source code of the above application](#).

Limitations and Development Issues

The following limitations were observed while working with IAR compiler 6.11A.

1. IAR compiler uses the stack for storing the local variables. The stack is limited to 1024 bytes in the DS80C400. The default stack swap size (**ROM_SAVESIZE**) for the DS80C400 library is 384 bytes. Make sure that this limit is changed appropriately if your application declares many stack variables. To change the default task swap size, use Dallas Semiconductor's library called **task_genesis(unsigned int savesize)** or **task_fork(unsigned char priority, unsigned int savesize)** defined in **rom400_task.h**, and supply **savesize** parameter with the correct value.
2. Functions like **printf**, **sprintf** do not work correctly unless 'lowest optimization level' is selected. To select the level of optimization, go to **project**→**options**→**ICC8051** and select '**None**' in the **Code** tab.
3. The default libraries for **printf**, **sprintf** by IAR do not work properly. For them to work correctly, your C program should include the C files provided by IAR (like **#include <printf.c>**).

Conclusion

The IAR compiler and libraries provided by Dallas Semiconductor allow applications written in C to access the power and functionality of DS80C400 ROM software. Programs written in C can access the network stack, memory manager, process scheduler, and many other DS80C400 features. Developers using the C language for the DS80C400 microcontroller 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 that are presently available for the Keil compiler over to IAR. Please visit the DS80C400 IAR Library home page regularly for updates.

Relevant Links

[IAR ROM Libraries Project Home](#)

[IAR home page](#)

[Java Development Kit Download Page](#)

[Java Communications API](#)

Application note 609, "[Internet Speaker with the DS80C400 Silicon Software](#)."

[1-Wire Public Domain Kit](#)

[DS80C400 User's Guide](#)

[TINI Software Development Kit](#)

Notes

- ¹ [Java runtime environment](#)
- ² [Java communications API](#)
- ³ [The High-Speed Micro User's Guide Supplement for the DS80C400](#)
- ⁴ http://files.dalsemi.com/tini/ds80c400/c_libraries/iar/index.html

µVision2 is a registered trademark of Keil Corporation
IAR Embedded Workbench is a registered trademark of IAR Systems AB.
Java is a trademark of Sun Microsystems, Inc.
TINI is a registered trademark of Maxim Integrated Products, Inc.
Windows is a registered trademark of Microsoft Corp.

Application Note 3550: www.maxim-ic.com/an3550

More Information

For technical support: www.maxim-ic.com/support

For samples: www.maxim-ic.com/samples

Other questions and comments: www.maxim-ic.com/contact

Automatic Updates

Would you like to be automatically notified when new application notes are published in your areas of interest? [Sign up for EE-Mail™](#).

Related Parts

DS80C320: [QuickView](#) -- [Full \(PDF\) Data Sheet](#) -- [Free Samples](#)

DS80C400: [QuickView](#) -- [Full \(PDF\) Data Sheet](#) -- [Free Samples](#)

DS80C410: [QuickView](#) -- [Full \(PDF\) Data Sheet](#) -- [Free Samples](#)

DS80C411: [QuickView](#) -- [Full \(PDF\) Data Sheet](#) -- [Free Samples](#)

AN3550, AN 3550, APP3550, Appnote3550, Appnote 3550

Copyright © by Maxim Integrated Products

Additional legal notices: www.maxim-ic.com/legal