Micriμm

μC/DNSc

Domain Name System (Client)

User's Manual

www.Micrium.com

Table of Contents

1.00 Introduction

DNS (Domain Name Service) is a distributed directory service for internet resources. It enables internet resources (such as servers) to be referenced by names (e.g. "www.micrium.com") instead of their IP addresses, that are meaningless, hard to remember, and sometimes dynamically changes over time.

µC/DNSc is an add-on product to µC/TCP-IP that implements the DNS protocol. The 'c' in µC/DNSc stands for 'client'. The µC/DNSc module implements a part of:

RFC 1035: ftp://ftp.rfc-editor.org/in-notes/rfc1035.txt

Figure 1-1 shows a block diagram that shows the relationship between components needed to run µC/DNSc in a typical embedded target application. You should note that µC/DNSc assumes the presence of µC/TCP-IP but could actually be modified to work with just about any TCP/IP stack.

You may also notice that the µC/DHCPc module is needed to build the example application but is not needed for µC/DNSc. µC/DHCPc is used in the example for convenience. µC/DHCPc automatically obtains a valid IP configuration and DNS server address from a DHCP server (our example had a router connected to the network that provided this service).

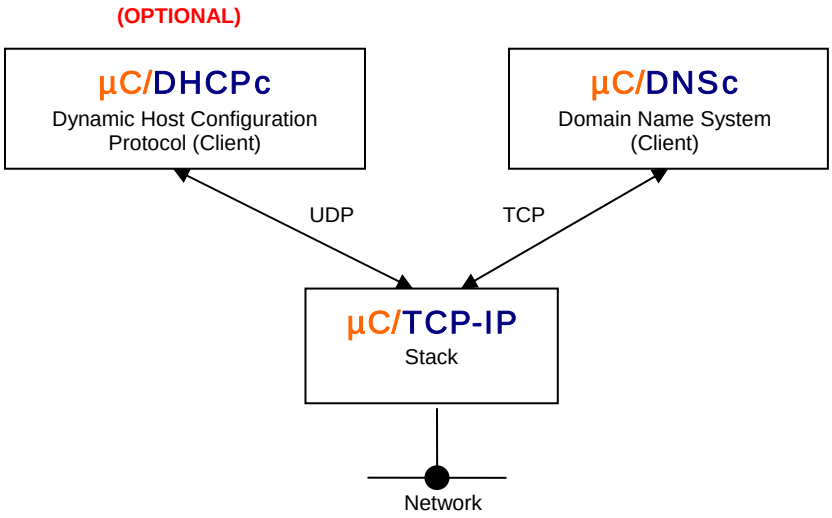

Figure 1-1, Relationship between the different modules.

2.00 Directories and Files

The files for µC/DNSc are placed in multiple directories as described below.

`\Micrium\Software\uC-DNSc`
: This is the main directory for µC/DNSc.

`\Micrium\Software\uC-DNSc\`**`Source`**
: This directory contains the source code for µC/DNSc. DNS is fairly easy to implement and thus, the code is found in just two files:

  `dns-c.c`
  `dns-c.h`

  Note that the 'c' at the end of `dns-c.c` means client and thus it contains 'client' side code. `dns-c.h` is a header file that contains client declarations for DNS.

3.00 Using µC/DNSc

Figure 3-1 shows the test setup to demonstrate the use of µC/DNSc. We used a Cogent Computer CSB337 target (ARM9) connected to a Windows-based PC through a 100-Base-T Hub. The setup also included a 'router' which provided a DHCP server.

The PC's IP address was set to `192.168.0.102` and the target address was `192.168.0.3` (both obtained from the DHCP server). Depending on your router configuration, you may obtain different addresses.

The setup also used an RS-232C serial port on the target to provide a console output. We opened a Hyper Terminal session on the Windows PC to monitor the console output. When the target application is started, the IP address obtained for the target is output onto the Console Port.

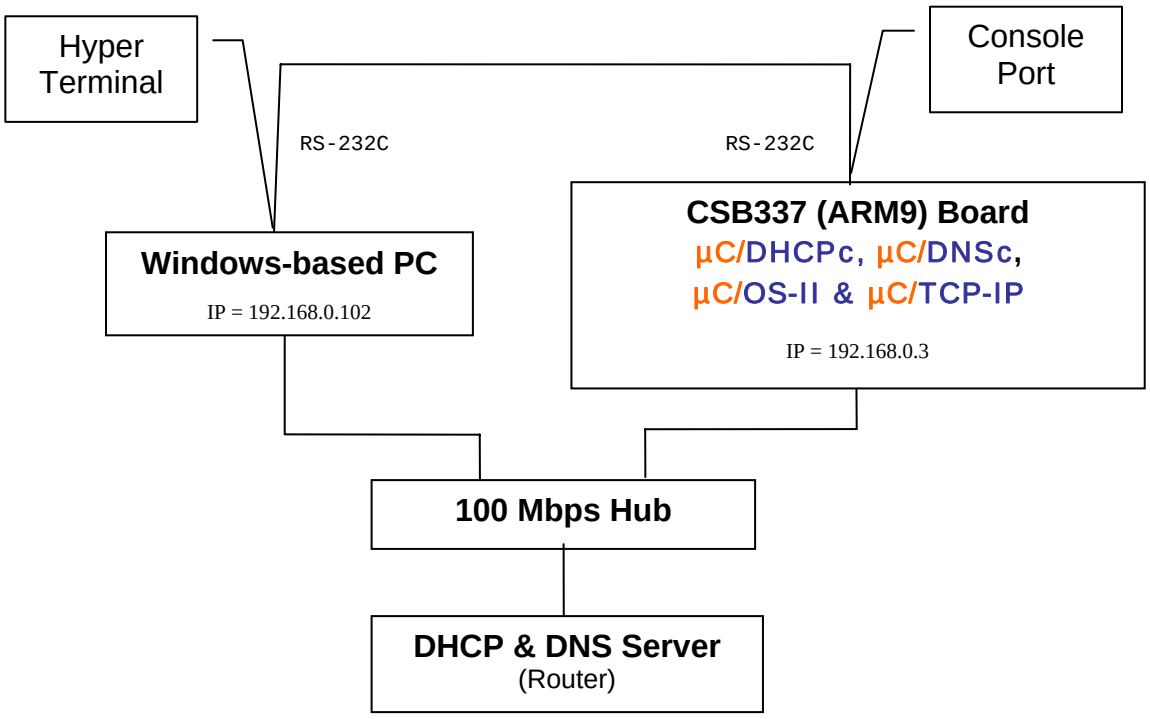

Figure 3-1, Test setup

The sample application was created using the IAR Embedded Workbench V4.31a and debugged with IAR's C-Spy which connects to the target using a J-Tag emulator.

3.01 Modules

As shown in figure 3-1, the target board runs the following Micrium software component:

µC/DHCPc
µC/DNSc
µC/OS-II
µC/TCP-IP

Not shown in figure 3-1 are the following support modules:

µC/CPU
µC/LIB

µC/DHCPc is an implementation of the client portion of the DHCP (dynamic host configuration protocol). This protocol enables dynamic IP configuration to be distributed from one central source (the server) to booting clients.

µC/DNSc is an implementation of the client portion of the Domain Name Service. This service enables fully qualified name addresses, e.g. "www.micrium.com" to be translated into their IP addresses.

µC/OS-II is a real-time, multitasking kernel that allows you to have up to 250 application tasks.

µC/DNSc runs as one of those tasks. You can use just about any other RTOSs with µC/DNSc but you would need to provide RTOS adaptation functions.

µC/TCP-IP is an embedded TCP/IP stack that provides IP v4 networking support.

µC/CPU is a module that removes the dependencies of the target CPU. In other words, µC/CPU defines common data types used by all Micrium software components and thus allows these components to be easily ported to different CPUs simply by changing the µC/CPU implementation files.

µC/LIB declares functions to replace the standard C `str???()` functions, `mem???()` functions and more. The reason we did this is to make it easier to get our software validated by third parties (e.g. FAA, FDA, etc.) because we provide all the source code to our products. You might argue that most compiler vendors also provide source code for these libraries. However, it's better (from a certification point of vue) to always certify the same source code instead of different source from multiple compiler vendors.

3.02 µC/DNSc Configuration

The µC/DNSc module doesn't require configuration. You just have to pass the IP address of the DNS server (possibly obtained with µC/DHCPc) at module initialization.

3.03 Creating the test application

This section describes the directories and files needed to build the sample application. You will notice that a fair amount of software components are needed to create the executable that runs on the target.

You may also notice also that the packages µC/DHCPc is needed to build the example application. This package has been used for convenience. µC/DHCPc obtains automatically a valid IP configuration from a DHCP server.

In the discussion of directories, we will assume that the directories listed are relative to the install directory:

```
\Micrium\Software\
```

In other words,

```
.\EvalBoards\Cogent\CSB337\IAR\BSP
```

Actually means:

```
\Micrium\Software\EvalBoards\Cogent\CSB337\IAR\BSP
```

3.03.01 Example Application Source Files

The test application was placed in the following directory:

```
.\EvalBoards\Cogent\CSB337\IAR\uC-Apps\Ex1
```

and consist of the following files:

```
app.c
app_cfg.h
Ex1.*
includes.h
net_cfg.h
os_cfg.h
```

app.c

This file contains the application code for example #1. As with most C programs, code execution start at `main()` which is shown in listing 3-1.

Listing 3-1

```
int  main (void)
{
#if (OS_TASK_NAME_SIZE >= 16)
    CPU_INT08U  err;
#endif


    BSP_Init();                       /* (1) Initialize BSP                */

    APP_DEBUG_TRACE ("Initializing: uC/OS-II\n");
    OSInit();                         /* (2) Initialize OS                 */

                                      /* (3) Create start task             */
    OSTaskCreateExt(AppTaskStart,
                    (void *)0,
                    (OS_STK *)&AppStartTaskStk[APP_START_TASK_STK_SIZE - 1],
                    APP_START_TASK_PRIO,
                    APP_START_TASK_PRIO,
                    (OS_STK *)&AppStartTaskStk[0],
                    APP_START_TASK_STK_SIZE,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

                                      /* (4) Give a name to tasks          */
#if (OS_TASK_NAME_SIZE >= 16)
    OSTaskNameSet(OS_TASK_IDLE_PRIO,   "Idle task",  &err);
    OSTaskNameSet(OS_TASK_STAT_PRIO,   "Stat task",  &err);
    OSTaskNameSet(APP_START_TASK_PRIO, "Start task", &err);
#endif

    APP_DEBUG_TRACE("Start OS...\n");
    OSStart();                        /* (5) Start OS                      */
}
```

L3-1(1) `main()` starts off by initializing the I/Os we'll be using on this board.

L3-1(2) The example code assumes the presence of an RTOS called µC/OS-II and `OSInit()` is used to initialize µC/OS-II.

L3-1(3) µC/OS-II requires that we create at least ONE application task. This is done by calling `OSTaskCreateExt()` and specifying the task start address, the top-of-stack to use for this task, the priority of the task and a few other arguments.

L3-1(4) µC/OS-II allows you to assign names to tasks that have been created. We thus assign a name to the application task. These names are used mostly during debug. In fact, task names are displayed during debug when using IAR's C-Spy debugger (or other µC/OS-II aware debuggers).

L3-1(5) In order to start multitasking, your application needs to call `OSStart()`. `OSStart()` determines which task, out of all the tasks created, will get to run on the CPU. In this case, µC/OS-II will run `AppTaskStart()`.

The first, and only 'application' task that µC/OS-II runs is shown in listing 3-2.

Listing 3-2

```
static  void  AppTaskStart (void *p_arg)
{
    (void)p_arg;                    /* Prevent compiler warning                 */

    APP_DEBUG_TRACE("Initialize interrupt controller...\n");
    BSP_InitIntCtrl();          (1) /* Initialize the interrupt controller      */

    APP_DEBUG_TRACE("Initialize OS timer...\n");
    Tmr_Init();                 (2) /* Start timers                             */

#if (OS_TASK_STAT_EN > 0)
    APP_DEBUG_TRACE("Initialize OS statistic task...\n");
    OSStatInit();               (3) /* Start OS statistics task                 */
#endif

    AppInit_TCPIP();            (4) /* Initialize uC/TCP-IP                     */

    AppInit_DHCPc();            (5) /* Initialize DHCP client (if present)      */

                                (6)
    DNSc_Init(AppIPDnsSrv);          /* Initialize DNS client                    */

                                (7)
    APP_DEBUG_TRACE ("Creating Application Tasks.\n");
    AppTaskCreate();                 /* Create application tasks                 */

    App_DNS_Test();             (8) /* Do some basic DNS tests                  */

    LED_Off(1);
    LED_Off(2);
    LED_Off(3);

    while (TRUE) {              (9) /* Task body, always written as an infinite loop.  */
        OSTimeDlyHMSM(0, 0, 0, 100);
    }
}
```

L3-2(1) Initializes µC/OS-II tick interrupt and the AT91RM9200's interrupt controller.

L3-2(2) Initialize and start kernel timer tick. This timer determines how many times per seconds the kernel will be informed that the time advances.

L3-2(3) If you set `OS_TASK_STAT_EN` to `1` in `os_cfg.h` then the µC/OS-II statistic task is initialized by calling `OSStatInit()`. `OSStatInit()` basically figures out how fast the CPU is running in order to determine how much CPU usage your application will be consuming. Details as to how this is done can be found in the µC/OS-II book (see References).

L3-2(4) We then initialize the TCP/IP stack by calling `NetInit()`. `NetInit()` initializes all of µC/TCP-IP's data structures and creates two tasks. One task waits for packets to be received and the other task manages timers. Because in this example we are using the µC/DHCPc service that will request an IP configuration from an external server and the µC/DHCPc module MUST be initialized after µC/TCP-IP, we MUST configure the µC/TCP-IP stack with generic values. This is needed for proper execution of the DHCP client.

The MAC address is obtained from the monitor.

L3-2(5) Configure and start the DHCP protocol. The µC/DHCPc package will try to contact a DHCP server to obtain IP configuration. The IP configuration obtained from the DHCP server is applied to the IP stack. The DNS server is also obtained from DHCP server and saved for use by the DNS client. See µC/DHCPc documentation for more details.

L3-2(6) Initialize µC/DNSc module. The DNS server address is passed as parameter.

L3-2(7) Initialize LED task. You can start your own tasks here!

L3-2(8) Do some DNS basic tests. With the help of a DNS server, DNS addresses are resolved to their IP addresses and the result is printed to the console.

L3-2(9) At this point the TCP/IP stack is initialized with the desired settings and the task enters an infinite loop and blinks one of the LEDs. You should note that this task doesn't do anything other than blink the LED. We could have just as well terminated this task but, blinking the LED on the CSB337 board gives us an indication that the application is running.

`app_cfg.h`

This file is used to establish the task priorities of each of the tasks in your application as well as the stack size for those tasks. The reason this is done here is to make it easier to configure task priorities for your entire application. In other words, you can set the task priorities of all your tasks in one place.

`Ex1.*`

These files are IAR embedded workbench project files.

`includes.h`

`includes.h` is a 'master' header file that contains `#include` directives to include other header files. This is done to make the code cleaner to read and easier to maintain.

net_cfg.h

This file is used to configure µC/TCP-IP and defines the number of timers used in µC/TCP-IP, the number of buffers for packets reception and transmission, the number of ARP (Address Resolution Protocol) cache entries, the number of Sockets that your application can open and more. In all, there are about `50` or so `#define` to set in this file.

os_cfg.h

This file is used to configure µC/OS-II and defines the number maximum number of tasks that your application can have, which services will be enabled (semaphores, mailboxes, queues, etc.), the size of the idle and statistic task and more. In all, there are about `60` or so `#define` that you can set in this file. Each entry is commented and additional information about the purpose of each `#define` can be found in the µC/OS-II book.

3.03.02 BSP (Board Support Package) Source Files

The concept of a BSP (Board Support Package) is to hide the hardware details from the application code. It is important that function names in a BSP reflect the function and does not make references to any CPU specifics. For example, the code to turn on an LED is called `LED_On()` and not `CSB337_led()`. If you use `LED_On()` in your code, you can easily port your code to another processor (or board) simply by rewriting `LED_On()` to control the LEDs on a different board. The same is true for other services. You will also notice that BSP functions are prefixed with the function's group. LED services start with `LED_`, Timer services start with `Tmr_`, etc. In other words, BSP functions don't need to be prefixed by `BSP_`.

The CSB337 BSP is found in the following directory:

```
.\EvalBoards\Cogent\CSB337\IAR\BSP
```

The BSP directory contains the following files:

```
bsp.c
bsp.h
net_bsp.c
net_bsp.h
net_isr.c
CSB33x_lnk_ram.xcl
```

`bsp.c` and `bsp.h`

`bsp.c` contains I/O, timer initialization code, LED control code, and more. The I/Os that we use on the board are initialized when `BSP_Init()` is called. `bsp.c` also contains functions to turn ON and OFF LEDs, toggle LEDs, configure CPU interrupts and more.

`net_bsp.*`

This file contains code specific to the NIC (Network Interface Controller) used and other functions that are dependent of the hardware. Specifically, this file contains code to read data from and write data to the NIC, provide delay functions, control power to the NIC, get a time stamp and more.

`net_isr.c`

This file contains code to initialize interruptions from the NIC and clear them after they are handled.

`CSB33x_lnk_ram.xcl`

This file contains the linker command file for the IAR toolchain. This file specifies where code and data is placed in memory. In this case, all the code is placed in RAM to make it easier to debug. When you are ready to deploy your product, you will most likely need to create a `CSB33x_lnk_flash.xcl` to locate your code in flash instead of RAM.

3.03.03 CPU Source Files

CPU manufacturers typically provide you with C header files that define their CPUs and the I/Os found on some of these chips. The CSB337 board contains a Atmel AT91RM9200 and the I/O definitions are found in the following directory:

```
.\CPU\Atmel\AT91RM9200\*.*
```

You should note that our directory structure architecture allows us to support multiple CPUs and thus, we would create CPU directories using the following scheme:

```
.\CPU\<manufacturer>\<CPU>\*.*
```

Where:

<manufacturer>	is the name of the CPU manufacturer.
<CPU>	is the name of the CPU.

3.03.04 µC/CPU Source Files

Some support functions are needed to adapt the software to different CPUs and compilers. This is different from the files provided by the CPU manufacturer as described in the previous section. Specifically, we need to specify what C data type is needed for a 16-bit unsigned integer, a 16-bit signed integer, a 32-bit signed integer, etc. Some compilers might assume that an `int` is 16 bits while others might assume that an `int` is 32 bits. To avoid the confusion, we defined data types that remove this confusion. In other words, we declare the following data types:

```
CPU_VOID
CPU_BOOLEAN
CPU_CHAR
CPU_INT08U
CPU_INT08S
CPU_INT16U
CPU_INT16S
CPU_INT32U
CPU_INT32S
CPU_FNCT_PTR
```

We also declared two functions that are used to disable and enable interrupts, `CPU_SR_Save()` and `CPU_SR_Restore()`, respectively.

The CPU/compiler specific files are placed in the following directory:

```
.\uC-CPU\<CPU>\<compiler>\*.*
```

Where:

<CPU>	is the name of the CPU or, a generic name that represents a family of CPUs.
<compiler>	is the name of the compiler manufacturer.

3.03.05 µC/DHCPc Source Files

The µC/DHCPc package enables the test application to run anywhere without IP configuration, assuming a DHCP server is present on the local network. At runtime, the test application is provided an IP address, a network mask, the IP address of the network gateway and the IP address of a DNS server by the DHCP server. The µC/DHCPc package assumes infinite IP address leases only.

To use the µC/DHCPc package, the following files need to be included in your application:

```
.\uC-DHCPc\Source\*.*
```

3.03.07 µC/LIB Source Files

µC/DHCPc, µC/DNSc and µC/TCP-IP doesn't make use of any of the standard C library functions `strcpy()`, `strcat()`, `memcpy()`, `memset()` etc. Instead, these and other functions have been re-written from scratch to provide similar functionality. The reason we did this is to make it easier to get our software validated by third parties (e.g. FAA, FDA, etc.) because we provide all the source code to our products. You might argue that most compiler vendors also provide source code for these libraries. However, it's better (from a certification point of vue) to always certify the same source code instead of different source from multiple compiler vendors. The libraries used by µC/DHCPc, µC/DNSc and µC/TCP-IP are found in the following directory:

```
.\uC-LIB\*.*
```

3.03.08 µC/OS-II Source Files

µC/DHCPc and µC/DNSc doesn't need any task nor use any kernel resources. However, the sample application needs to create some tasks using µC/OS-II. Thus, the following files need to be included in your application:

```
.\uCOS-II\Source\*.*
.\uCOS-II\Ports\ARM\Generic\IAR\*.*
```

The CSB337 board contains a Atmel AT91RM9200 (ARM9) CPU. The µC/OS-II port for this CPU is found in the directory shown above. In fact, the µC/OS-II ARM port is generic and can thus be used with other ARM CPUs.

3.03.09 µC/TCP-IP Source Files

µC/DHCPc and µC/DNSc assumes the presence of µC/TCP-IP, a TCP/IP stack designed specifically for embedded systems. The following files need to be included in your application:

```
.\uC-TCPIP\IF\*.*
.\uC-TCPIP\IF\Ether\*.*
.\uC-TCPIP\NIC\Ether\AT91RM9200\*.*
.\uC-TCPIP\OS\uCOS-II\*.*
.\uC-TCPIP\Source\*.*
```

You should note that the CSB337 use a Atmel AT91RM9200 microcontroller which contains its own ethernet controller and thus, we need to include the driver for that chip in our build (as shown in the directory above).

References

µC/OS-II, The Real-Time Kernel, 2nd Edition
Jean J. Labrosse
CMP Books, 2002
ISBN 1-57820-103-9

Embedded Systems Building Blocks
Jean J. Labrosse
CMP Books, 2000
ISBN 0-87930-604-1

Contacts

CMP Books, Inc.
1601 W. 23rd St., Suite 200
Lawrence, KS 66046-9950
USA
+1 785 841 1631
+1 785 841 2624 (FAX)
e-mail: rushorders@cmpbooks.com
WEB: http://www.cmpbooks.com

Cogent Computer Systems, Inc.
1130 Ten Rod Road, Suite A-201
North Kingstown, RI 02852 USA
USA
+1 401 295 6505
+1 401 295 6507 (Fax)
WEB: www.CogComp.com

IAR Systems
Century Plaza
1065 E. Hillsdale Blvd
Foster City, CA 94404
USA
+1 650 287 4250
+1 650 287 4253 (FAX)
e-mail: Info@IAR.com
WEB : www.IAR.com

Micriμm
949 Crestview Circle
Weston, FL 33327
USA
+1 954 217 2036
+1 954 217 2037 (FAX)
e-mail: Sales@Micrium.com
WEB: www.Micrium.com

Validated Software
Lafayette Business Park
2590 Trailridge Drive East, Suite 102
Lafayette, CO 80026
USA
+1 303 531 5290
+1 720 890 4700 (FAX)
e-mail: Sales@ValidatedSoftware.com
WEB: www.ValidatedSoftware.com